File and Folder Management in Python

Introduction

So far, we have learned how to:

- read files
- write files
- work with CSV files
- work with JSON files

But real programs often need to do more than read and write data.

They may need to:

- create folders
- find files
- check whether a file exists
- rename files
- move files
- copy files
- delete files
- list files inside a folder
- build file paths safely

For example, imagine a program that generates reports:

```
project/
□□□ reports/
□   □□□ january.txt
□   □□□ february.txt
```

□ □□□ march.txt
□□□ app.py

Python can create and manage this structure programmatically.

This is called **file and folder management**.

Python provides several tools for this, but one of the most useful modern approaches is the built-in `pathlib` module.

---

1. What Is File and Folder Management?

File and folder management means controlling files and directories through code.

A program might need to:

```
Create
Read
Write
Copy
Move
Rename
Search
Delete
```

For example:

```
Python program
    □
Find a file
    □
Read its contents
```

-
Process the data
-
Save a new file
-
Move it into a folder

This is common in automation, data processing, applications, scripts, and developer tools.

---

2. Files vs Folders

A **file** stores data.

Examples:

```
notes.txt
students.csv
config.json
report.pdf
```

A **folder** stores files and other folders.

Example:

```
project/
  app.py
  data/
    students.csv
  reports/
    report.txt
```

Think of:

File → contains data
Folder → organizes files

---

3. The `pathlib` Module

Python's modern tool for working with file-system paths is:

`pathlib`

Import `Path`:

```
from pathlib import Path
```

Now you can create path objects:

```
file_path = Path("notes.txt")
```

Instead of treating the path as just a string, Python gives you an object with useful methods.

For example:

```
file_path.exists()
```

checks whether the path exists.

---

4. Creating a Path

A simple relative path:

```
from pathlib import Path

path = Path("notes.txt")
```

A folder:

```
folder = Path("documents")
```

A file inside a folder:

```
file_path = Path("documents") / "notes.txt"
```

The / operator combines path parts.

```
documents / notes.txt
      
documents/notes.txt
```

This is cleaner and safer than manually building paths with strings.

---

5. Relative vs Absolute Paths

There are two common types of paths.

Relative Path

A relative path starts from the current working directory.

```
Path("data/students.csv")
```

It means:

```
data folder
  
students.csv
```

relative to where the program is running.

Absolute Path

An absolute path describes the complete location.

For example, on Windows:

```
C:\Users\Hafsa\Documents\data.csv
```

On Linux or macOS:

```
/home/hafsa/Documents/data.csv
```

In most projects, relative paths are often easier to manage.

---

6. Checking Whether a Path Exists

Use:

```
from pathlib import Path

path = Path("notes.txt")

print(path.exists())
```

Output:

```
True
```

if the file or folder exists.

Otherwise:

```
False
```

This is useful before performing operations.

Example:

```
if path.exists():
    print("Path exists.")
else:
    print("Path does not exist.")
```

---

7. Checking Whether Something Is a File

Use:

```
path.is_file()
```

Example:

```
from pathlib import Path

path = Path("notes.txt")
```

```
if path.is_file():
    print("This is a file.")
```

This is different from:

```
path.exists()
```

because `exists()` only tells you that something exists.

`is_file()` specifically checks whether it is a file.

---

8. Checking Whether Something Is a Folder

Use:

```
path.is_dir()
```

Example:

```
from pathlib import Path

path = Path("documents")

if path.is_dir():
    print("This is a folder.")
```

So:

```
exists()  → Does it exist?
is_file() → Is it a file?
is_dir()  → Is it a folder?
```

---

9. Creating a Folder

Use:

`mkdir()`

Example:

```
from pathlib import Path

folder = Path("reports")
folder.mkdir()
```

Python creates:

`reports/`

If the folder already exists, Python normally raises an error.

---

10. Avoiding Errors When Creating Folders

You can use:

`exist_ok=True`

Example:

```
from pathlib import Path

folder = Path("reports")
```

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

Now:

- if the folder doesn't exist create it
- if it already exists don't raise an error

This is useful when your program may run multiple times.

---

11. Creating Nested Folders

Suppose you want:

```
project/
    data/
        raw/
```

You can create the structure using:

```
from pathlib import Path

folder = Path("project/data/raw")

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

`parents=True` tells Python to create missing parent folders.

Without it, the parent folders may need to exist first.

---

12. Listing Files in a Folder

Use:

`iterdir()`

Example:

```
from pathlib import Path

folder = Path("documents")

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

If the folder contains:

```
documents/
    notes.txt
    tasks.txt
    report.pdf
```

the program can display the paths.

---

13. Listing Only Files

You can check each item:

```
from pathlib import Path

folder = Path("documents")
```

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

This ignores folders.

---

14. Listing Only Folders

Similarly:

```
from pathlib import Path

folder = Path("project")

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

This prints only directories.

---

15. Finding Files With `glob()`

Suppose a folder contains:

```
reports/
    january.txt
    february.txt
    march.pdf
    april.txt
```

You can find all .txt files:

```
from pathlib import Path

folder = Path("reports")

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

Output:

```
reports/january.txt
reports/february.txt
reports/april.txt
```

The pattern:

```
*.txt
```

means:

```
anything ending with .txt
```

---

16. Finding Specific File Types

Find Python files:

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

Find JSON files:

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

Find CSV files:

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

This is useful when building automation scripts.

---

17. Searching Subfolders With `rglob()`

`glob()` searches within the specified folder.

If you want to search recursively through subfolders, use:

```
rglob()
```

Example:

```
from pathlib import Path

project = Path("project")

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

This can find Python files inside:

```
project/
    app.py
```

```
  src/
    main.py
  tools/
    helper.py
```

The search goes through the entire directory tree.

---

18. Getting a File Name

Suppose:

```
from pathlib import Path

path = Path("documents/report.txt")
```

Get the file name:

```
print(path.name)
```

Output:

```
report.txt
```

---

19. Getting the File Extension

Use:

```
path.suffix
```

Example:

```
print(path.suffix)
```

Output:

```
.txt
```

This is useful when you need to identify a file type.

---

20. Getting the File Stem

The stem is the filename without its extension.

```
print(path.stem)
```

Output:

```
report
```

For:

```
report.pdf
```

we get:

```
name   report.pdf
stem   report
suffix  .pdf
```

---

21. Getting the Parent Folder

Use:

```
path.parent
```

Example:

```
path = Path("documents/report.txt")

print(path.parent)
```

Output:

```
documents
```

This is useful when navigating a directory structure.

---

22. Combining Path Information

You can inspect a file like this:

```
from pathlib import Path

path = Path("reports/january.pdf")

print("Name:", path.name)
print("Stem:", path.stem)
print("Extension:", path.suffix)
print("Parent:", path.parent)
```

Output:

```
Name: january.pdf
Stem: january
Extension: .pdf
Parent: reports
```

---

23. Renaming a File

Use:

```
rename()
```

Example:

```
from pathlib import Path

old_name = Path("old_notes.txt")
new_name = Path("notes.txt")

old_name.rename(new_name)
```

The file changes from:

```
old_notes.txt
```

to:

```
notes.txt
```

---

24. Renaming a Folder

The same method works with folders.

```
from pathlib import Path

old_folder = Path("old_reports")
new_folder = Path("reports")

old_folder.rename(new_folder)
```

Now:

```
old_reports/
```

becomes:

```
reports/
```

---

25. Moving a File

A path can also be used to move a file.

For example:

```
notes.txt
```

Move it into:

```
documents/
```

using:

```
from pathlib import Path

source = Path("notes.txt")
destination = Path("documents/notes.txt")

source.rename(destination)
```

The file is now located at:

```
documents/notes.txt
```

For more advanced copying and moving operations, Python's `shutil` module is also useful.

---

26. Copying Files With `shutil`

`pathlib` handles paths and many file-system operations, but `shutil` provides convenient file-copying tools.

Import it:

```
import shutil
```

Copy a file:

```
shutil.copy("notes.txt", "backup.txt")
```

Now both files exist:

```
notes.txt
backup.txt
```

The original remains unchanged.

---

27. Copying a File to a Folder

```
import shutil

shutil.copy(
    "notes.txt",
    "backup/notes.txt"
)
```

This creates a copy in the backup folder.

The original remains where it was.

---

28. Copying an Entire Folder

Use:

```
shutil.copytree()
```

Example:

```
import shutil

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

This copies the folder and its contents.

For example:

```
project/
   app.py
   data/
      users.json
```

becomes:

```
project/
   app.py
   data/
      users.json

project_backup/
   app.py
   data/
      users.json
```

---

29. Deleting a File

Using `Path`:

```
from pathlib import Path

file = Path("notes.txt")

file.unlink()
```

This deletes the file.

Because deletion is destructive, check first when appropriate:

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

---

30. Deleting an Empty Folder

Use:

```
folder.rmdir()
```

Example:

```
from pathlib import Path

folder = Path("old_reports")

folder.rmdir()
```

The folder must be empty.

If it contains files, the operation will fail.

---

31. Deleting a Folder and Its Contents

For a directory containing files and subfolders, use `shutil.rmtree()`:

```
import shutil
```

```
shutil.rmtree("old_project")
```

This removes the entire directory tree.

Because this is destructive, use it carefully.

Do not run recursive deletion on a path unless you are certain it is the intended directory.

---

32. Getting File Information

`Path.stat()` provides file-system information.

Example:

```
from pathlib import Path

file = Path("notes.txt")

info = file.stat()

print(info.st_size)
```

`st_size` gives the file size in bytes.

You can use it to inspect files programmatically.

---

33. Getting the Current Working Directory

Use:

```
from pathlib import Path

print(Path.cwd())
```

`cwd` means:

current working directory

It tells you where relative paths are being resolved.

For example:

```
Current folder
    
project/
```

Then:

```
Path("data/users.json")
```

refers to:

```
project/data/users.json
```

---

34. Changing the Working Directory

Python can change the current working directory using:

```
import os

os.chdir("project")
```

However, changing the global working directory is often unnecessary in well-structured programs.

It is usually better to build explicit paths with `Path`.

For example:

```
from pathlib import Path

data_file = Path("project") / "data" / "users.json"
```

This makes the path relationship clear.

---

35. Creating a File With `touch()`

`Path.touch()` can create an empty file.

```
from pathlib import Path

file = Path("notes.txt")
file.touch()
```

If the file does not exist, Python creates it.

You can then write to it:

```
file.write_text("Learn Python", encoding="utf-8")
```

---

36. Writing Text With `Path.write_text()`

Instead of:

```
with open("notes.txt", "w", encoding="utf-8") as file:
    file.write("Learn Python")
```

you can use:

```
from pathlib import Path

path = Path("notes.txt")

path.write_text(
    "Learn Python",
    encoding="utf-8"
)
```

This is convenient for simple text-file operations.

---

37. Reading Text With `Path.read_text()`

Similarly:

```
from pathlib import Path

path = Path("notes.txt")

content = path.read_text(encoding="utf-8")

print(content)
```

This is a convenient alternative to:

```
with open("notes.txt", "r", encoding="utf-8") as file:
    content = file.read()
```

For more complex file handling, `open()` still provides more control.

---

38. Creating a Project Folder Structure

Suppose you want:

```
haas_project/
□□□ data/
□□□ reports/
□□□ backups/
□□□ output/
```

You can create it programmatically:

```
from pathlib import Path

project = Path("haas_project")

for folder_name in [
    "data",
    "reports",
    "backups",
    "output"
]:
    (project / folder_name).mkdir(
        parents=True,
        exist_ok=True
    )
```

This is useful when setting up projects automatically.

---

39. Organizing Files by Extension

Imagine a folder contains:

```
project/
  notes.txt
  data.csv
  users.json
  app.py
  config.json
```

You could create separate folders and move files based on their extension.

Conceptually:

```
project/
□□□ text/
□   □□□ notes.txt
□□□ csv/
□   □□□ data.csv
□□□ json/
□   □□□ users.json
□□□ python/
  □□□ app.py
```

A simple example:

```
from pathlib import Path
```

```
project = Path("project")

for file in project.iterdir():
    if file.is_file():
        print(file.suffix)
```

Once you understand paths, loops, conditions, and file operations, you can build full automation scripts around this idea.

---

40. A Practical File Organizer

Here is a simple organizer:

```
from pathlib import Path
import shutil

folder = Path("downloads")

for file in folder.iterdir():
    if not file.is_file():
        continue

    extension = file.suffix.lower().replace(".", "")

    if not extension:
        continue

    target_folder = folder / extension
    target_folder.mkdir(exist_ok=True)

    destination = target_folder / file.name

    shutil.move(str(file), str(destination))
```

If `downloads/` contains:

```
photo.jpg
report.pdf
data.csv
notes.txt
```

the program can organize them into:

```
downloads/
□□□ jpg/
□   □□□ photo.jpg
□□□ pdf/
□   □□□ report.pdf
□□□ csv/
□   □□□ data.csv
□□□ txt/
    □□□ notes.txt
```

This is a practical example of Python automation.

---

41. File Paths and Operating Systems

Different operating systems represent paths differently.

Windows often uses:

```
C:\Users\Hafsa\Documents\file.txt
```

Linux/macOS often use:

`/home/hafsa/Documents/file.txt`

Hard-coding separators can create portability problems.

Instead of manually writing:

```
path = "documents\\notes.txt"
```

prefer:

```
from pathlib import Path

path = Path("documents") / "notes.txt"
```

`pathlib` handles platform-specific path behavior for you.

---

42. Why `pathlib` Is Preferred

Older Python code often uses:

```
import os

os.path.exists("notes.txt")
```

Modern Python can use:

```
from pathlib import Path

Path("notes.txt").exists()
```

`pathlib` provides an object-oriented interface for paths.

It makes operations easier to read:

`path.name`
`path.parent`
`path.suffix`
`path.exists()`
`path.is_file()`
`path.is_dir()`

For new Python projects, `pathlib` is generally a strong default for path management.

---

43. `os` vs `pathlib` vs `shutil`

These modules overlap, but they have different strengths.

`pathlib`

Best for:

`Paths`
`Files`
`Directories`
`Path information`
`Basic file operations`

Example:

```
Path("data/file.json").exists()
```

`os`

Provides broader operating-system functionality.

Example:

```
import os

print(os.getcwd())
```

shutil

Useful for:

```
Copying
Moving
Removing directory trees
High-level file operations
```

Example:

```
shutil.copy("a.txt", "b.txt")
```

A practical mental model:

```
pathlib  paths and common file operations
os       operating-system interaction
shutil   high-level file copying/moving
```

---

44. Common Mistakes

Mistake 1: Building paths manually

Avoid:

```
path = "data\\" + "users.json"
```

Prefer:

```
path = Path("data") / "users.json"
```

Mistake 2: Assuming a folder exists

This can fail:

```
Path("reports/output").mkdir()
```

if `reports/` does not exist.

Use:

```
Path("reports/output").mkdir(
    parents=True,
    exist_ok=True
)
```

Mistake 3: Deleting without checking

Instead of:

```
Path("important.txt").unlink()
```

consider:

```
path = Path("important.txt")

if path.exists():
```

```
path.unlink()
```

More importantly, make sure the path is the one you actually intend to delete.

Mistake 4: Confusing `exists()` with `is_file()`

```
path.exists()
```

does not tell you whether the path is specifically a file.

Use:

```
path.is_file()
```

when you need that distinction.

Mistake 5: Using recursive deletion carelessly

This:

```
shutil.rmtree("project")
```

can delete an entire directory tree.

Always verify the target path before using destructive operations.

Mistake 6: Forgetting case differences in extensions

A file may be:

`PHOTO.JPG`

instead of:

`photo.jpg`

Using:

`file.suffix.lower()`

can normalize the extension.

---

45. Best Practices

1. Prefer `pathlib` for paths

`from pathlib import Path`

2. Build paths with `/`

`path = Path("data") / "users.json"`

3. Use `exist_ok=True` when repeated folder creation is expected

`folder.mkdir(exist_ok=True)`

4. Use `parents=True` for nested directories

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

5. Check paths before destructive operations

```
if path.exists():
    ...
```

6. Use `is_file()` and `is_dir()` when type matters

7. Use `shutil` for copying and moving when appropriate

8. Avoid hard-coded operating-system-specific path separators

9. Keep project data organized

For example:

```
project/
├── src/
├── data/
├── reports/
└── backups/
```

10. Use meaningful folder and file names

Prefer:

```
student_reports/
```

over:

```
stuff2/
```

---

46. Practical Problem-Solving Framework

When your program needs to manage files, ask:

Step 1: What path am I working with?

```
path = Path("data/users.json")
```

Step 2: Does it exist?

```
path.exists()
```

Step 3: Is it a file or directory?

```
path.is_file()
path.is_dir()
```

Step 4: What operation is required?

```
Create
Read
Write
Copy
Move
Rename
Delete
Search
```

Step 5: Could the operation fail?

Consider:

```
Missing path
Permission problems
Existing destination
Invalid file type
Unexpected directory structure
```

Step 6: Is the operation destructive?

Be especially careful with:

delete
overwrite
move
recursive removal

The overall decision process is:

Path

Check

Choose operation

Validate

Perform operation

Handle errors

---

47. Mini Project: Automatic Project Setup

Create a Python script that creates this structure:

my_project/
□□□ data/
□□□ reports/
□□□ backups/
□□□ src/
□□□ tests/

Solution:

```
from pathlib import Path

project = Path("my_project")

folders = [
    "data",
    "reports",
    "backups",
    "src",
    "tests"
]

for folder in folders:
    (project / folder).mkdir(
        parents=True,
        exist_ok=True
    )

print("Project structure created.")
```

This is useful when setting up projects automatically.

---

48. Mini Project: File Explorer

Create a simple script that displays everything inside a folder.

```
from pathlib import Path

folder = Path("project")
```

```
if not folder.exists():
    print("Folder does not exist.")
else:
    for item in folder.iterdir():
        if item.is_file():
            print("FILE:", item.name)
        elif item.is_dir():
            print("FOLDER:", item.name)
```

Example output:

```
FILE: app.py
FILE: README.md
FOLDER: data
FOLDER: tests
```

This gives you a basic understanding of how file explorers work.

---

49. Mini Project: Find All Python Files

```
from pathlib import Path

project = Path("project")

python_files = project.rglob("*.py")

for file in python_files:
    print(file)
```

If the project contains:

```
project/
```

```
□□□ app.py
□□□ src/
□   □□□ main.py
□□□ tests/
    □□□ test_app.py
```

the program finds all three Python files.

---

50. Mini Project: Backup a File

```
from pathlib import Path
import shutil

source = Path("data.json")
backup = Path("backups/data.json")

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

shutil.copy(source, backup)

print("Backup created.")
```

The program:

1. identifies the source file
2. creates the backup folder if needed
3. copies the file
4. reports success

This is a simple example of file automation.

---

51. 5 Minute Challenge

Create a program that manages a folder called:

`resources/`

Your program should:

1. create the folder if it does not exist
2. create these subfolders:

```
resources/
□□□ python/
□□□ javascript/
□□□ web/
□□□ tools/
```

3. list all subfolders
4. find all `.pdf` files inside `resources/`
5. print each PDF's filename and extension

Extra challenge

Create a `backups/` folder and copy one selected PDF into it.

---

52. Exercises

Exercise 1

Create a folder called:

`practice/`

using `Path.mkdir()`.

Exercise 2

Check whether the folder exists.

Exercise 3

Create:

`practice/data/`

using `parents=True`.

Exercise 4

Create an empty file using `touch()`.

Exercise 5

Write text to the file using `write_text()`.

Exercise 6

Read the file using `read_text()`.

Exercise 7

Rename the file.

Exercise 8

Create three text files and use `glob("*.txt")` to find them.

Exercise 9

Copy one file into a backup folder using `shutil.copy()`.

Exercise 10

Create a script that searches an entire project for `.py` files using `rglob()`.

Exercise 11

Create a simple file organizer that separates files into folders based on their extensions.

---

53. Mini Quiz

Question 1

Which module provides the `Path` class?

A. `file`

B. `pathlib`

C. `folder`

D. `system`

Answer: B

Question 2

Which method checks whether a path exists?

A. `exists()`

B. `available()`

C. `find()`

D. `check()`

Answer: A

Question 3

Which method checks whether a path is a file?

A. `is_file()`

B. `is_text()`

C. `file()`

D. `check_file()`

Answer: A

Question 4

Which method creates a directory?

A. `create()`

B. `mkdir()`

C. `folder()`

D. `make_dir()`

Answer: B

Question 5

What does `rglob()` do?

A. Deletes files

B. Searches recursively

C. Renames folders

D. Copies directories

Answer: B

Question 6

Which attribute gives the file extension?

A. `name`

B. `parent`

C. `suffix`

D. `stem`

Answer: C

Question 7

Which module is commonly used to copy files?

A. `shutil`

B. `copyfile`

C. `filetools`

D. `pathcopy`

Answer: A

Question 8

Which method deletes a file?

A. `delete()`

B. `remove_file()`

C. `unlink()`

D. `erase()`

Answer: C

---

54. Interview Questions

1. What is `pathlib`?

`pathlib` is a Python standard-library module that provides an object-oriented way to work with file-system paths.

2. What is the difference between `exists()`, `is_file()`, and `is_dir()`?

- `exists()` checks whether the path exists.
- `is_file()` checks whether it is a file.
- `is_dir()` checks whether it is a directory.

3. What does `mkdir(parents=True, exist_ok=True)` do?

It creates the directory and any missing parent directories while avoiding an error if the directory already exists.

4. What is the difference between `glob()` and `rglob()`?

`glob()` searches according to a pattern in a directory, while `rglob()` searches recursively through subdirectories.

5. What are `name`, `stem`, and `suffix`?

For:

```
report.pdf
```

they are:

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

6. What is `shutil` used for?

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

7. Why is `pathlib` preferred over manually constructing paths?

It provides cleaner, more readable, and cross-platform path handling.

8. What is the difference between `unlink()` and `rmdir()`?

`unlink()` removes a file. `rmdir()` removes an empty directory.

9. What does `shutil.rmtree()` do?

It recursively removes a directory and its contents.

10. Why should file deletion be handled carefully?

Deletion is destructive and can permanently remove data, especially when recursive deletion is used.

---

55. Cheat Sheet

Import

```
from pathlib import Path
```

Create path

```
path = Path("data/file.txt")
```

Combine paths

```
path = Path("data") / "file.txt"
```

Check existence

```
path.exists()
```

Check file

```
path.is_file()
```

Check folder

```
path.is_dir()
```

Create folder

```
Path("data").mkdir(
    exist_ok=True
```

```
)
```

Create nested folders

```
Path("project/data/raw").mkdir(
    parents=True,
    exist_ok=True
)
```

List directory

```
for item in Path("data").iterdir():
    print(item)
```

Find files

```
Path("data").glob("*.txt")
```

Recursive search

```
Path("project").rglob("*.py")
```

File name

```
path.name
```

File extension

```
path.suffix
```

File name without extension

```
path.stem
```

Parent folder

```
path.parent
```

Rename / move

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

Delete file

```
path.unlink()
```

Delete empty folder

```
path.rmdir()
```

Copy file

```
import shutil

shutil.copy(source, destination)
```

Move file

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

Copy folder

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

Delete folder tree

```
shutil.rmtree(folder)
```

Create empty file

```
path.touch()
```

Write text

```
path.write_text(
    "Hello",
    encoding="utf-8"
)
```

Read text

```
content = path.read_text(
    encoding="utf-8"
)
```

---

56. Key Takeaways

- Files store data; folders organize files.
- Python can manage both through code.
- `pathlib.Path` is the modern tool for working with paths.
- `exists()` checks whether something exists.
- `is_file()` checks whether it is a file.
- `is_dir()` checks whether it is a directory.
- `mkdir()` creates directories.
- `iterdir()` lists directory contents.
- `glob()` finds files matching a pattern.
- `rglob()` searches recursively.
- `name`, `stem`, `suffix`, and `parent` help inspect paths.
- `rename()` can rename or move paths.
- `unlink()` removes files.
- `rmdir()` removes empty directories.
- `shutil` provides convenient copying and moving operations.

- `shutil.rmtree()` can remove an entire directory tree and should be used carefully.
- `pathlib` makes path handling more readable and portable across operating systems.
- File management becomes especially powerful when combined with loops, conditions, functions, and automation.

The core mental model is:

`Path`
`□`
`Check`
`□`
`Create / Find / Read / Write`
`□`
`Copy / Move / Rename`
`□`
`Delete when necessary`

Once you understand file and folder management, Python can move beyond simply processing individual files and start **automating entire file systems and project workflows**.

Visit haas.dev for more resources and guides.

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

haas.dev

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