Python Project Structure

Introduction

When you first learn Python, you can write everything in one file:

`app.py`

That works for small programs.

But as a project grows, putting everything into one file becomes difficult.

Imagine a project with:

- 20 Python files
- 5,000 lines of code
- Tests
- Configuration
- Dependencies
- Documentation
- Images or other assets

Without a clear structure, finding and maintaining code becomes difficult.

A **Python project structure** is the organized arrangement of your project's files and folders so that each part has a clear responsibility.

A good structure helps you:

- Find code quickly
- Separate responsibilities
- Write tests

- Manage dependencies
- Collaborate with others
- Package and distribute your project
- Maintain the project as it grows

---

1. Small Python Project

For a very small script, you may only need:

```
hello_project/
□□□ main.py
```

`main.py`:

```
print("Hello, Python!")
```

There is nothing wrong with this structure.

The important principle is:

> **Use the simplest structure that fits the project's size.**

Do not create 20 folders for a 30-line script.

---

2. A Growing Project

As the project grows, you might have:

```
calculator/
□□□ main.py
□□□ calculator.py
□□□ utils.py
```

Now responsibilities are separated.

```
main.py
  □
Program entry point

calculator.py
  □
Calculation logic

utils.py
  □
Reusable helper functions
```

This is already easier to understand.

---

3. A Real Python Application

A larger application might look like:

```
task_manager/
□
□□□ pyproject.toml
□□□ README.md
□□□ LICENSE
□□□ .gitignore
□
```

```
□□□ src/
□   □□□ task_manager/
□       □□□ __init__.py
□       □□□ models.py
□       □□□ services.py
□       □□□ repository.py
□       □□□ main.py
□
□□□ tests/
    □□□ test_models.py
    □□□ test_services.py
    □□□ test_repository.py
```

This may look complicated at first.

But each part has a purpose.

---

4. The Project Root

The top-level folder is called the **project root**.

Example:

```
task_manager/
```

Everything related to the project generally lives inside this folder.

Think of it as:

```
task_manager/
    □
```

`Everything belonging to this project`

---

5. pyproject.toml

A modern Python project commonly contains:

`pyproject.toml`

It can define:

- Project metadata
- Python version requirements
- Dependencies
- Build configuration
- Optional dependencies
- Tool configuration

Example:

```
[project]
name = "task-manager"
version = "1.0.0"
requires-python = ">=3.11"

dependencies = [
    "requests>=2.31"
]
```

`pyproject.toml` is the project's central Python configuration file.

---

6. README.md

The README explains the project to humans.

Example:

`README.md`

It might contain:

````
# Task Manager

A simple Python task management application.

## Installation

```bash
python -m pip install -e .
````

Usage

python -m task_manager

A good README should help a new developer understand the project without reading every source file.

---

7. `LICENSE`

If your project is distributed publicly, you may include:

```text

`LICENSE`

It specifies the legal terms under which others can use, modify, or distribute the project.

For example, a project might use:

```
MIT
Apache 2.0
GPL
```

The license should match your actual intentions and legal requirements.

---

8. `.gitignore`

The `.gitignore` file tells Git which files or folders should not be tracked.

Example:

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

For example, your virtual environment:

```
.venv/
```

usually should not be committed to Git.

Likewise, secrets such as:

`.env`

should generally not be committed when they contain credentials.

---

9. The `src` Directory

A common structure for reusable Python projects is:

```
project/
├── src/
│   └── project_name/
```

For example:

```
task_manager/
├── src/
│   └── task_manager/
```

This is called the **src layout**.

It separates:

`project files`

from:

`Python package source code`

---

10. Why Use `src`?

Imagine:

```
project/
□□□ setup.py
□□□ task_manager/
□□□ tests/
```

Depending on how you run the project, Python may accidentally import the local source directory instead of the actually installed package.

The `src` layout helps make packaging mistakes more visible.

The structure becomes:

```
project/
□□□ pyproject.toml
□□□ src/
□   □□□ task_manager/
□□□ tests/
```

The source code has a clearly defined location.

---

11. Package Directory

Inside:

```
src/
```

you might have:

```
src/
└── task_manager/
```

This is your Python package.

It contains the application's Python code.

For example:

```
src/
└── task_manager/
    ├── __init__.py
    ├── models.py
    ├── services.py
    └── repository.py
```

---

12. What Is `__init__.py`?

You may see:

```
__init__.py
```

inside a Python package.

Example:

```
task_manager/
└── src/
    └── task_manager/
        └── __init__.py
```

It can be used to:

- Mark/package a directory in traditional Python packaging
- Initialize package-level behavior
- Expose selected package members

It can also be completely minimal:

```
# src/task_manager/__init__.py
```

A modern Python namespace package does not always require `__init__.py`, but regular packages commonly include it because it provides an explicit package boundary and works naturally with conventional tooling.

---

13. Separating Responsibilities

One of the most important reasons for project structure is **separation of responsibilities**.

Instead of:

```
main.py
```

containing everything:

```
database code
business logic
validation
models
API calls
printing
configuration
```

you can separate them:

models.py
services.py
repository.py
main.py

For example:

models.py

Data representation

services.py

Business logic

repository.py

Data storage/access

main.py

Application entry point

---

14. models.py

Models represent the data your application works with.

Example:

```
from dataclasses import dataclass


@dataclass
class Task:
    title: str
    completed: bool = False
```

Now:

models.py

contains the definition of a task.

It does not need to know how the application starts.

---

15. services.py

A service can contain business logic.

Example:

```
from .models import Task


def complete_task(task: Task) -> None:
    task.completed = True
```

The service performs an operation.

This keeps business logic separate from the user interface or entry point.

---

16. repository.py

A repository can handle data storage or retrieval.

For example:

```
class TaskRepository:

    def save(self, task):
        print(f"Saving: {task.title}")

    def get_all(self):
        return []
```

In a real application, this could communicate with:

- SQLite
- PostgreSQL
- MongoDB
- Files
- APIs

The important idea is:

> The repository handles data access rather than mixing database code into every part of the application.

---

17. main.py

`main.py` can act as an entry point.

Example:

```
from .models import Task
from .services import complete_task


def main():
    task = Task("Learn Python")
    complete_task(task)

    print(task)


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

The entry point coordinates the application instead of containing every implementation detail.

---

18. Importing Between Files

Suppose we have:

```
task_manager/
□□□ src/
    □□□ task_manager/
        □□□ models.py
        □□□ services.py
        □□□ main.py
```

Inside `services.py`:

```
from .models import Task
```

The `.` means:

> Import from the current package.

This is a **relative import**.

---

19. Absolute Imports

You can also use an absolute package import:

```
from task_manager.models import Task
```

This explicitly names the package.

For larger projects, consistent import style matters because it makes dependencies easier to understand.

---

20. The `tests` Directory

Tests should usually be separated from application code.

Example:

```
project/
□□□ src/
□   □□□ task_manager/
□       □□□ models.py
□       □□□ services.py
□
□□□ tests/
    □□□ test_models.py
    □□□ test_services.py
```

The `tests` directory contains automated checks.

---

21. Test File Naming

A common convention is:

```
test_models.py
test_services.py
test_repository.py
```

or:

```
models_test.py
services_test.py
```

The first convention is especially common with pytest.

For example:

```
from task_manager.models import Task
```

```
def test_task_starts_incomplete():
    task = Task("Learn Python")

    assert task.completed is False
```

---

22. Why Tests Are Separate

Imagine your project contains:

```
100 Python files
```

You do not want testing code mixed into every production file.

Instead:

```
src/
    application code

tests/
    testing code
```

This makes the project easier to navigate.

---

23. Test Structure for Larger Projects

A larger project might use:

```
tests/
```

```
□□□ unit/
□   □□□ test_models.py
□   □□□ test_services.py
□
□□□ integration/
□   □□□ test_repository.py
□
□□□ conftest.py
```

The exact structure depends on the project.

Unit tests

Test small pieces independently.

Integration tests

Test multiple components working together.

---

24. Configuration Files

Projects often need configuration.

For example:

```
config/
```

or:

```
src/task_manager/config.py
```

However, do not create a configuration folder just because it sounds organized.

For a small project:

```
config.py
```

may be enough.

For a larger project:

```
config/
□□□ settings.py
□□□ logging.py
```

may make sense.

The structure should follow actual complexity.

---

25. Environment Variables

Secrets and environment-specific values should generally not be hardcoded.

For example:

```
DATABASE_URL
API_KEY
DEBUG
```

may be provided through environment variables.

You might have:

`.env`

locally, but:

`.env`

should generally be excluded from Git when it contains secrets.

Your source code might read:

```
import os

api_key = os.getenv("API_KEY")
```

Project structure should make it clear where configuration comes from without storing secrets in source code.

---

26. Assets

Some applications need non-Python files.

For example:

```
assets/
□□□ images/
□□□ templates/
□□□ data/
```

But whether these belong inside the package or elsewhere depends on how they are used and distributed.

For example:

```
project/
├── src/
│   └── app/
├── assets/
└── tests/
```

could be appropriate for an application.

---

27. CLI Projects

A command-line application might look like:

```
task_cli/
├── pyproject.toml
├── README.md
├── src/
│   └── task_cli/
│       ├── __init__.py
│       ├── cli.py
│       ├── models.py
│       └── services.py
└── tests/
```

The CLI-specific code can live in:

```
cli.py
```

while the actual business logic stays elsewhere.

---

28. Web Application Structure

A larger web application may need additional layers.

For example:

```
web_app/
□□□ pyproject.toml
□□□ README.md
□□□ src/
□   □□□ web_app/
□       □□□ __init__.py
□       □□□ main.py
□       □□□ models/
□       □□□ routes/
□       □□□ services/
□       □□□ repositories/
□       □□□ config/
□
□□□ tests/
```

The exact structure depends on the framework and architecture.

The important principle remains:

```
Each part should have a clear responsibility.
```

---

29. Avoid the Giant `utils.py`

A common beginner pattern is:

```
utils.py
```

and then putting every unrelated helper function inside it.

Eventually:

```
utils.py
```

becomes:

```
2,000 lines
```

containing:

```
date functions
string functions
API helpers
database helpers
validation
formatting
random functions
```

This makes the file difficult to understand.

Instead, organize helpers by responsibility.

For example:

```
utils/
□□□ dates.py
□□□ formatting.py
□□□ validation.py
```

But only split when there is enough complexity to justify it.

---

30. Avoid Overengineering

This is equally important.

Do not turn:

```
hello.py
```

into:

```
hello/
□□□ src/
□□□ domain/
□□□ infrastructure/
□□□ application/
□□□ repositories/
□□□ services/
□□□ adapters/
□□□ factories/
```

when the project contains 20 lines.

A good rule:

> **Structure should solve complexity, not create complexity.**

---

31. Small vs Medium vs Large Projects

Small

```
calculator/
□□□ main.py
□□□ calculator.py
```

Medium

```
task_manager/
□□□ pyproject.toml
□□□ README.md
□□□ src/
□   □□□ task_manager/
□       □□□ __init__.py
□       □□□ models.py
□       □□□ services.py
□       □□□ repository.py
□□□ tests/
```

Larger

```
platform/
□□□ pyproject.toml
□□□ README.md
□□□ LICENSE
□□□ docs/
□□□ src/
□   □□□ platform/
□       □□□ api/
□       □□□ models/
□       □□□ services/
□       □□□ repositories/
□       □□□ config/
```

```
□      □□□ main.py
□□□ tests/
□   □□□ unit/
□   □□□ integration/
□□□ scripts/
```

The structure evolves with the project.

---

32. Project Structure Is Not One Universal Template

There is no single structure that every Python project must use.

A:

`CLI tool`

will have different needs from a:

`web API`

which will have different needs from:

`data science project`

which will have different needs from:

`reusable Python library`

So do not memorize one giant folder tree.

Learn the principles.

---

33. Python Library Structure

A reusable library might look like:

```
my_library/
□□□ pyproject.toml
□□□ README.md
□□□ LICENSE
□□□ src/
□   □□□ my_library/
□       □□□ __init__.py
□       □□□ client.py
□       □□□ models.py
□       □□□ exceptions.py
□□□ tests/
    □□□ test_client.py
    □□□ test_models.py
```

The focus is on reusable functionality.

---

34. Data Science Project Structure

A data science project may have:

```
sales_prediction/
□□□ pyproject.toml
□□□ README.md
□□□ data/
□   □□□ raw/
□   □□□ processed/
```

```
├── notebooks/
├── src/
│   ├── sales_prediction/
│   │   ├── preprocessing.py
│   │   ├── features.py
│   │   └── model.py
└── tests/
```

The `notebooks` directory is common for exploratory analysis.

The `data` directory separates datasets from source code.

The exact organization depends on the project.

---

35. Django or Framework Projects

Frameworks may impose or encourage their own structures.

For example, a web framework may create:

```
project/
├── manage.py
├── project/
│   ├── settings.py
│   ├── urls.py
│   └── ...
└── app/
    ├── models.py
    ├── views.py
    └── ...
```

Do not force a generic structure onto a framework that already has strong conventions.

Learn the framework's conventions first.

---

36. Package vs Project

These terms are related but not identical.

A **project** is the complete development environment.

It may contain:

```
README
tests
source code
configuration
documentation
packaging files
```

A **package** is a Python importable unit containing Python modules.

For example:

```
project/
□□□ src/
   □□□ task_manager/
```

The project contains the package.

Think:

```
Project
  □
  □□□ Documentation
  □□□ Tests
  □□□ Configuration
  □□□ Python Package
```

---

37. Module vs Package vs Project

This distinction is important.

Module

Usually a Python file:

```
models.py
```

Package

A collection of related Python modules:

```
task_manager/
□□□ __init__.py
□□□ models.py
□□□ services.py
```

Project

The complete application/repository:

```
task_manager/
□□□ pyproject.toml
```

□□□ README.md
□□□ src/
□□□ tests/
□□□ ...

Mental model:

Module
  □
Package
  □
Project

---

38. __pycache__

When Python runs code, it may create:

__pycache__/

For example:

task_manager/
□□□ __pycache__/
  □□□ models.cpython-313.pyc

These are generated bytecode cache files.

You normally do not manually manage them.

They are commonly excluded through .gitignore.

---

39. Virtual Environment Location

A common project structure is:

```
project/
□□□ .venv/
□□□ pyproject.toml
□□□ src/
□□□ tests/
```

The virtual environment is local to the project.

However:

`.venv/`

should normally not be committed to Git.

The project configuration describes what the environment needs.

---

40. `scripts/`

Some projects contain:

`scripts/`

for development or maintenance scripts.

Example:

```
scripts/
□□□ seed_database.py
□□□ generate_data.py
□□□ cleanup.py
```

These are useful when scripts are part of the project's workflow.

Again, do not create the folder unless there are scripts that justify it.

---

41. `docs/`

Documentation can live in:

```
docs/
```

Example:

```
docs/
□□□ architecture.md
□□□ installation.md
□□□ api.md
```

This becomes useful when the README is no longer enough.

---

42. Imports and Project Structure

Good structure helps create predictable imports.

Example:

```
src/
□□□ task_manager/
   □□□ models.py
   □□□ services.py
```

Then:

```
from task_manager.models import Task
```

and:

```
from task_manager.services import complete_task
```

The relationship between folders and imports becomes understandable.

---

43. Circular Imports

Poor project structure can lead to circular imports.

For example:

```
models.py
  □ imports
services.py
  □ imports
models.py
```

This creates a dependency cycle.

Instead, reconsider the responsibilities.

For example:

```
models.py
   
simple data definitions

services.py
   
business logic using models
```

A clear dependency direction can prevent many problems.

---

44. Think in Dependencies

When organizing files, ask:

> Which component depends on which?

For example:

```
main
 
services
 
repository
 
database
```

This is easier to reason about than:

`everything imports everything`

A good project structure makes dependency relationships easier to see.

---

45. Cohesion

A useful design principle is **cohesion**.

Cohesion asks:

> Do the things inside this module belong together?

For example:

`date_utils.py`

containing date-related functions has good cohesion.

But:

`utils.py`

containing:

```
database code
date functions
API calls
password validation
image processing
```

has poor cohesion.

---

46. Coupling

Another concept is **coupling**.

Coupling asks:

> How strongly are different parts dependent on each other?

If every module directly depends on every other module:

```
A □ B □ C
□ □ □
D □ E □ F
```

the project becomes difficult to change.

A better architecture usually aims for clearer dependency boundaries.

---

47. A Practical Design Process

When starting a project:

Step 1

Start simple.

```
project/
□□□ main.py
```

Step 2

Identify responsibilities.

```
models
services
storage
```

Step 3

Split code when files become difficult to understand.

Step 4

Add tests.

```
tests/
```

Step 5

Add project configuration.

```
pyproject.toml
```

Step 6

Add documentation.

```
README.md
```

Step 7

Add more folders only when complexity requires them.

---

48. Example: Refactoring a Messy Project

Suppose you start with:

```
student_app/
□□□ main.py
```

`main.py` becomes 1,500 lines.

It contains:

```
Student class
database functions
validation
CLI
report generation
logging
```

Instead of keeping everything together:

```
main.py
```

refactor into:

```
student_app/
□□□ pyproject.toml
□□□ README.md
□□□ src/
□   □□□ student_app/
```

```
│   ├── models.py
│   ├── services.py
│   ├── repository.py
│   ├── reports.py
│   └── cli.py
└── tests/
```

Now each file has a clearer responsibility.

---

49. haas.dev Example

Imagine the backend of haas.dev has a resource system.

Instead of:

```
haas.py
```

containing everything, structure it as:

```
haas_resources/
├── pyproject.toml
├── README.md
│
└── src/
    └── haas_resources/
        ├── __init__.py
        ├── models.py
        ├── services.py
        ├── repositories.py
        ├── validators.py
        └── main.py
```

```
□
□□□ tests/
    □□□ test_models.py
    □□□ test_services.py
    □□□ test_validators.py
```

Possible responsibilities:

models.py
    □
Resource structure

validators.py
    □
Validate resource data

repositories.py
    □
Store/retrieve resources

services.py
    □
Business rules

main.py
    □
Application entry point

The structure communicates the architecture before you even open the files.

---

50. Common Mistakes

Mistake 1: One giant file

`main.py`

with thousands of lines.

Better

Split by responsibility when the project grows.

Mistake 2: Too many folders

Creating a huge architecture for a tiny project increases complexity.

Better

Start simple.

Mistake 3: Random file names

Names such as:

```
stuff.py
helpers2.py
final_utils.py
new_service.py
```

make the project harder to understand.

Better

Use names that communicate responsibility.

```
validators.py
repository.py
services.py
models.py
```

Mistake 4: Mixing tests with source code

Avoid:

```
src/
□□□ models.py
□□□ test_models.py
□□□ services.py
□□□ test_services.py
```

unless there is a deliberate reason.

A dedicated:

```
tests/
```

directory is usually clearer.

Mistake 5: Committing `.venv`

Do not commit the complete virtual environment.

Use:

`.venv/`

in `.gitignore`.

Mistake 6: Committing secrets

Avoid committing:

`.env`

when it contains:

`API_KEY`
`PASSWORD`
`DATABASE_URL`

Mistake 7: Circular imports

If modules depend on each other in circles, reconsider their responsibilities.

---

51. Best Practices

Keep the root clean

Prefer:

```
project/
├── pyproject.toml
├── README.md
├── src/
└── tests/
```

over dozens of unrelated files.

Use meaningful names

Prefer:

```
repositories.py
```

over:

```
stuff.py
```

Separate source and tests

```
src/
tests/
```

Keep configuration centralized

Use:

```
pyproject.toml
```

when appropriate.

Avoid premature abstraction

Create architecture because the project needs it.

Keep dependencies understandable

Make it easy to see which module depends on which.

Let structure evolve

A project's structure does not need to be perfect on day one.

---

52. A Recommended General Structure

For a medium-sized Python project, this is a useful starting point:

```
my_project/
│
├── pyproject.toml
├── README.md
├── LICENSE
├── .gitignore
│
├── src/
│   └── my_project/
│       ├── __init__.py
│       ├── main.py
│       ├── models.py
│       ├── services.py
│       └── repository.py
│
└── tests/
    ├── test_models.py
    ├── test_services.py
    └── test_repository.py
```

Add:

```
docs/
scripts/
config/
assets/
```

only when your project actually needs them.

---

53. Project Structure Checklist

Before calling your structure finished, ask:

Root

- Is `pyproject.toml` present when needed?
- Is there a README?
- Is `.gitignore` configured?
- Is a license needed?

Source

- Is application code organized by responsibility?
- Are module names meaningful?
- Are imports understandable?
- Are circular dependencies avoided?

Tests

- Are tests separated from source?
- Can important business logic be tested independently?

Configuration

- Are secrets outside source control?
- Are environment-specific settings handled properly?

Maintenance

- Can a new developer understand the project quickly?
- Can you find a feature without searching the entire codebase?

---

54. 5 Minute Challenge

Create this project:

```
book_manager/
```

Your goal is to organize it properly.

It should contain:

```
pyproject.toml
README.md
src/
tests/
```

Inside the package, create:

```
models.py
services.py
repository.py
main.py
```

Decide what each file should be responsible for.

Then answer:

1. Where should the `Book` class go?
2. Where should borrowing logic go?
3. Where should database/file storage logic go?
4. Where should the application start?
5. Where should automated tests live?

---

55. Mini Project

Build a Task Manager

Create:

```
task_manager/
□□□ pyproject.toml
□□□ README.md
□□□ .gitignore
□
□□□ src/
□   □□□ task_manager/
□       □□□ __init__.py
□       □□□ models.py
□       □□□ services.py
□       □□□ repository.py
□       □□□ main.py
□
□□□ tests/
    □□□ test_models.py
    □□□ test_services.py
```

models.py

Create:

```
class Task:
    def __init__(self, title):
        self.title = title
        self.completed = False
```

services.py

Implement:

```
def complete_task(task):
    task.completed = True
```

repository.py

Create a simple in-memory repository:

```
class TaskRepository:

    def __init__(self):
        self.tasks = []

    def add(self, task):
        self.tasks.append(task)

    def get_all(self):
        return self.tasks
```

main.py

Connect the components:

```
from .models import Task
from .repository import TaskRepository
from .services import complete_task


def main():
    repository = TaskRepository()

    task = Task("Learn Python project structure")

    repository.add(task)
    complete_task(task)

    print(repository.get_all()[0].completed)


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

The important lesson is not the task manager itself.

It is learning how different responsibilities can work together without putting everything into one file.

---

56. Interview Questions

Beginner

1. What is a Python project structure?
2. What is a project root?
3. What is a Python module?
4. What is a Python package?
5. What is the purpose of `__init__.py`?

6. What is the purpose of `README.md`?
7. Why is `.gitignore` useful?

Intermediate

8. What is the `src` layout?
9. Why separate `src` and `tests`?
10. What belongs in `pyproject.toml`?
11. What is the difference between a project and a package?
12. What is the purpose of `models.py`?
13. What is the purpose of a service layer?
14. What is the purpose of a repository?
15. What are circular imports?
16. Why should secrets not be committed?

Advanced

17. Why can the `src` layout prevent certain import mistakes?
18. What is cohesion?
19. What is coupling?
20. How should project structure evolve as an application grows?
21. How can project structure make dependencies easier to understand?
22. When should you split a module?
23. When can too much project structure become harmful?
24. How would you structure a reusable Python library?
25. How would you structure a larger web application?

---

57. Cheat Sheet

Small project

`project/`

├── main.py
└── utils.py

Medium project

```
project/
├── pyproject.toml
├── README.md
├── src/
│   └── project/
│       ├── __init__.py
│       ├── models.py
│       ├── services.py
│       └── repository.py
└── tests/
```

Larger project

```
project/
├── pyproject.toml
├── README.md
├── LICENSE
├── docs/
├── scripts/
├── src/
│   └── project/
│       ├── api/
│       ├── models/
│       ├── services/
│       ├── repositories/
│       ├── config/
│       └── main.py
└── tests/
    ├── unit/
    └── integration/
```

Important files

`pyproject.toml`
: Project configuration

`README.md`
: Project documentation

`LICENSE`
: Usage/distribution terms

`.gitignore`
: Files Git should ignore

`src/`
: Application/package source

`tests/`
: Automated tests

---

58. Key Takeaways

- A project structure organizes code and supporting files.
- Small projects should stay simple.
- Larger projects benefit from clear separation of responsibilities.
- `src/` is a common layout for Python package source code.
- `tests/` should generally be separated from production code.
- `pyproject.toml` contains modern project metadata, dependencies, build configuration, and supported tool configuration.
- `README.md` explains the project to humans.
- `.gitignore` prevents unwanted files from being tracked.
- `__init__.py` commonly marks a conventional Python package and can define package initialization behavior.

- Modules should contain related functionality.
- Avoid giant `utils.py` files containing unrelated code.
- Avoid both giant files and unnecessary overengineering.
- Good structure makes dependencies and responsibilities easier to understand.
- Project structure should evolve as complexity grows.

Final Mental Model

Do not memorize every possible folder.

Remember the purpose behind the structure:

```
                Python Project
                      │
      ┌───────────────┼──────────────┐
      │               │              │
  Configuration    Source         Tests
      │               │              │
pyproject.toml      src/          tests/
      │               │
      │       ┌───────┴──────┐
      │       │              │
      │    Modules      Packages
      │
      ├── Dependencies
      ├── Python version
      └── Tool settings
```

The core principle is:

> **A good Python project structure makes it obvious where code belongs, what each part is responsible for, and how the parts work together.**

Visit haas.dev for more resources and guides.

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

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