`pyproject.toml` in Python

Introduction

As Python projects grow, they need a clear place to store project configuration.

You may need to define:

- Project name and version
- Python version requirements
- Project dependencies
- Build system
- Package information
- Optional development dependencies
- Tool configuration
- Formatting and linting settings

Modern Python projects commonly use a file called:

`pyproject.toml`

It is one of the central configuration files in modern Python development.

---

1. What Is `pyproject.toml`?

`pyproject.toml` is a configuration file used to describe how a Python project is built, packaged, and configured.

A simple project might look like:

```
my_project/
□
□□□ pyproject.toml
□□□ README.md
□□□ LICENSE
□□□ src/
   □□□ my_project/
      □□□ __init__.py
```

The important file here is:

`pyproject.toml`

It uses the TOML format.

TOML stands for:

Tom's Obvious, Minimal Language

The syntax is designed to be easy for both humans and programs to read.

---

2. Why Does `pyproject.toml` Exist?

Older Python projects often spread configuration across multiple files.

For example:

```
setup.py
setup.cfg
requirements.txt
tox.ini
```

`.flake8`

Modern Python tooling allows much of this configuration to live in:

`pyproject.toml`

This gives a project a central configuration file.

Think of it as:

```
pyproject.toml
    
Project configuration
    
Build + package + dependencies + tools
```

It does not necessarily replace every configuration file, but it can centralize a large amount of project configuration.

---

3. `pyproject.toml` vs `requirements.txt`

These files are related, but they serve different purposes.

`requirements.txt`

Usually describes packages that an environment should install.

Example:

```
requests==2.32.3
pytest==8.3.3
```

It is commonly used for installing a particular environment's dependencies.

`pyproject.toml`

Describes the Python project itself.

For example:

```
[project]
name = "resource-manager"
version = "1.0.0"
dependencies = [
    "requests>=2.31"
]
```

Think:

`requirements.txt`
→
"What should this environment install?"

`pyproject.toml`
→
"What is this Python project and how is it configured?"

A project can use both.

---

4. Basic `pyproject.toml`

Here is a minimal example:

```
[project]
name = "resource-manager"
version = "1.0.0"
description = "A small resource management application"
requires-python = ">=3.11"
```

Let's understand each part.

---

5. The `[project]` Section

The `[project]` table contains information about the Python project.

```
[project]
```

You can think of it as:

```
project metadata
```

It can contain information such as:

```
name
version
description
readme
requires-python
license
authors
dependencies
optional-dependencies
```

---

6. Project Name

Example:

```
[project]
name = "resource-manager"
```

This identifies the project.

Use a valid package/project name.

For example:

```
name = "haas-resource-tools"
```

The distribution/project name and the Python import package name can be related without being identical.

For example:

```
Project name:
haas-resource-tools

Import package:
haas_resource_tools
```

---

7. Project Version

Example:

```
version = "1.0.0"
```

Version numbers communicate releases.

A common format is:

```
MAJOR.MINOR.PATCH
```

For example:

```
1.0.0
1.1.0
1.1.1
2.0.0
```

A project might move from:

```
1.0.0
```

to:

```
1.1.0
```

when new functionality is added.

---

8. Project Description

You can describe the project using:

```
description = "A resource management application for developers"
```

This helps tools and package repositories understand what your project does.

---

9. README

You can point to your README:

```
readme = "README.md"
```

Example:

```
[project]
name = "resource-manager"
version = "1.0.0"
description = "A resource management application"
readme = "README.md"
```

Now the project's README can provide the longer project description.

---

10. Specifying the Python Version

You can define the Python versions your project supports.

```
requires-python = ">=3.11"
```

This means:

```
Python 3.11 or newer
```

You can also define a range.

```
requires-python = ">=3.11,<3.14"
```

This means:

```
3.11 ≤ Python < 3.14
```

This is useful because your project may depend on Python features that do not exist in older versions.

---

11. Adding Dependencies

Dependencies are packages your project needs.

Example:

```
[project]
name = "resource-manager"
version = "1.0.0"
dependencies = [
    "requests>=2.31",
    "rich>=13.0"
]
```

Now the project declares that it needs:

```
requests
rich
```

with minimum versions.

---

12. Dependency Version Rules

You can specify different constraints.

Minimum version

`"requests>=2.31"`

Means:

`2.31 or newer`

Exact version

`"requests==2.32.3"`

Means:

`Only version 2.32.3`

Upper limit

`"requests<3"`

Means:

`Any version below 3`

Version range

`"requests>=2.31,<3"`

Means:

`2.31 or newer`
`but below 3`

---

13. Why Dependencies Belong in Project Configuration

Suppose your project imports:

```
import requests
```

Another developer clones the project.

Without dependency information, they may not know:

```
Which package?
Which version?
What needs to be installed?
```

Project metadata can declare this.

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

Now packaging tools can understand the requirement.

---

14. Optional Dependencies

Not every dependency is needed for every user.

For example:

```
Application
```

```
 □
 □□□ Core dependencies
 □
 □□□ Development dependencies
```

You can define optional dependencies.

Example:

```
[project.optional-dependencies]
dev = [
    "pytest",
    "ruff"
]
```

Now the project has a development group:

```
dev
```

This can contain tools needed while developing the project.

---

15. Multiple Optional Dependency Groups

You can create different groups.

```
[project.optional-dependencies]

dev = [
    "pytest",
    "ruff"
]
```

```
docs = [
    "mkdocs"
]

test = [
    "pytest",
    "coverage"
]
```

Conceptually:

```
project
│
├── core
│
├── dev
│   ├── pytest
│   └── ruff
│
├── test
│   ├── pytest
│   └── coverage
│
└── docs
    └── mkdocs
```

This makes project requirements more organized.

---

16. Authors

Project metadata can include authors.

Example:

```
authors = [
  { name = "Hafsa" }
]
```

You can also include an email address:

```
authors = [
  { name = "Hafsa", email = "example@example.com" }
]
```

Only include contact information that you actually want to publish.

---

17. License

A project can also describe its license.

For example:

```
license = { file = "LICENSE" }
```

The actual license text can remain in:

```
LICENSE
```

while `pyproject.toml` points to it.

---

18. Complete Basic Example

A simple project might have:

```
[project]
name = "resource-manager"
version = "1.0.0"
description = "A resource management application"
readme = "README.md"
requires-python = ">=3.11"

authors = [
    { name = "Hafsa" }
]

dependencies = [
    "requests>=2.31",
    "rich>=13.0"
]

[project.optional-dependencies]
dev = [
    "pytest>=8",
    "ruff>=0.6"
]
```

This tells Python tooling:

```
Project name
     
resource-manager

Version
     
1.0.0
```

Python requirement

3.11+

Runtime dependencies

requests
rich

Development dependencies

pytest
ruff

---

19. Build System

pyproject.toml can also tell tools how the project should be built.

Example:

```
[build-system]
requires = ["setuptools>=68"]
build-backend = "setuptools.build_meta"
```

This section answers:

> Which tools should be used to build this project?

The two important fields are:

`requires`

and:

`build-backend`

---

20. What Is a Build Backend?

A build backend is the component responsible for turning your source project into distributable Python packages.

For example:

```
Your source code
      
Build backend
      
Distribution package
```

The build backend could be provided by tools such as:

`setuptools`
`hatchling`
`flit`
`poetry-core`

The choice depends on the project's tooling.

---

21. Build Requirements

Example:

```
[build-system]
requires = [
    "setuptools>=68"
]
```

This means the build process needs that package.

Notice that this is different from:

```
[project]
dependencies = [...]
```

The distinction is important.

Project dependencies

Packages needed when the application runs.

Build dependencies

Packages needed to build the project.

---

22. Build Backend

Example:

```
build-backend = "setuptools.build_meta"
```

This tells the build frontend which backend to use.

The overall flow becomes:

pyproject.toml

Build frontend

Build backend

Wheel / Source distribution

---

23. Wheel and Source Distribution

Python projects can be distributed in formats such as:

.whl

and:

.tar.gz

A wheel is a built distribution.

A source distribution contains the source needed to build/install the project.

For example:

resource_manager-1.0.0-py3-none-any.whl

---

24. Building a Project

A common modern tool for building Python packages is:

```
python -m build
```

You may need to install the build package first:

```
python -m pip install build
```

Then:

```
python -m build
```

This can produce:

```
dist/
□□□ resource_manager-1.0.0-py3-none-any.whl
□□□ resource_manager-1.0.0.tar.gz
```

The exact filenames depend on your project metadata.

---

25. Why `pyproject.toml` Is More Than a Package File

One important idea is that `pyproject.toml` is not only about publishing packages.

Modern Python tools can use it for their configuration.

For example:

```
pyproject.toml
□
```

□□□ Project metadata
□□□ Dependencies
□□□ Build configuration
□□□ Formatter configuration
□□□ Linter configuration
□□□ Test configuration
□□□ Other tool settings

This makes it a central configuration point.

---

26. Tool Configuration

Many Python tools support configuration inside:

`pyproject.toml`

For example, a tool may have a section like:

```
[tool.some_tool]
option = true
```

The general structure is:

`[tool.tool_name]`

For example:

```
[tool.ruff]
line-length = 88
```

The exact supported options depend on the tool.

---

27. Why [tool] Exists

The [tool] namespace prevents unrelated tools from interfering with one another.

Conceptually:

```
[tool.ruff]
...

[tool.pytest]
...

[tool.black]
...
```

Each tool gets its own configuration area.

This is much cleaner than creating many separate configuration files when the tool supports pyproject.toml.

---

28. Example With Ruff

Suppose you use Ruff for linting.

You might configure:

```
[tool.ruff]
line-length = 88
```

Now Ruff can read its configuration from:

`pyproject.toml`

Your project does not need to keep every tool setting in a separate file.

---

29. Example Project Structure

A modern Python project might look like:

```
resource-manager/
│
├── pyproject.toml
├── README.md
├── LICENSE
├── .gitignore
│
├── src/
│   └── resource_manager/
│       ├── __init__.py
│       ├── models.py
│       ├── services.py
│       └── repository.py
│
└── tests/
    ├── test_models.py
    └── test_services.py
```

The central configuration is:

`pyproject.toml`

---

30. `pyproject.toml` and `src` Layout

A common package structure is:

```
project/
├── pyproject.toml
├── src/
│   └── project_name/
│       └── __init__.py
```

The `src` directory helps separate:

`project configuration`

from:

`actual Python package`

This can help catch packaging/import mistakes during development.

---

31. Installing a Local Project

A project with proper packaging metadata can be installed into its environment.

For example:

```
python -m pip install .
```

This means:

Current directory

Build/install project

For development, an editable installation is commonly used:

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

---

32. What Does -e Mean?

The `-e` option means editable installation.

Without editable installation:

```
pip install .
```

The installed package is built/installed from the project.

With:

```
pip install -e .
```

the environment points back to your working source tree.

This is useful during development because you can modify the source code without repeatedly reinstalling the project.

---

33. Optional Dependencies During Installation

If the project defines:

```
[project.optional-dependencies]
dev = [
    "pytest",
    "ruff"
]
```

you can install the development extras using:

```
python -m pip install -e ".[dev]"
```

Conceptually:

```
Install project
      +
Install dev dependencies
```

---

34. pyproject.toml and requirements.txt

These files can coexist.

For example:

```
project/
□
□□□ pyproject.toml
□□□ requirements.txt
□□□ requirements-dev.txt
```

They answer different questions.

File	Main Purpose
pyproject.toml	Project metadata, packaging, dependencies, tool configuration
requirements.txt	Installable environment dependency list
README.md	Project documentation
LICENSE	Legal license
.gitignore	Files Git should ignore

The exact setup depends on the project's workflow.

---

35. pyproject.toml vs setup.py

Older Python projects often used:

setup.py

Example:

```
from setuptools import setup

setup(
    name="resource-manager",
```

```
    version="1.0.0"
)
```

Modern packaging commonly puts project metadata in:

```
pyproject.toml
```

For example:

```
[project]
name = "resource-manager"
version = "1.0.0"
```

The important idea is not simply:

```
old = bad
new = good
```

Instead:

```
Python packaging evolved
        ↓
PEP 517 / PEP 518 / PEP 621
        ↓
Standardized project configuration
        ↓
pyproject.toml
```

---

36. `pyproject.toml` vs `setup.cfg`

Another older configuration approach is:

`setup.cfg`

Some projects still use it, especially legacy projects.

Modern projects can often centralize configuration in:

`pyproject.toml`

---

37. TOML Syntax Basics

Before working with `pyproject.toml`, understand basic TOML syntax.

A key-value pair:

```
name = "resource-manager"
```

A number:

```
version_number = 1
```

A boolean:

```
enabled = true
```

A list:

```
dependencies = [
    "requests",
    "rich"
]
```

A table:

```
[project]
```

---

38. Strings

Strings can be written like:

```
name = "resource-manager"
```

Single quotes are also possible:

```
name = 'resource-manager'
```

Double quotes are commonly used in Python project configuration examples.

---

39. Arrays

An array is a list:

```
dependencies = [
  "requests",
  "rich",
  "pydantic"
]
```

You can also write short arrays on one line:

```
dependencies = ["requests", "rich"]
```

---

40. Comments

TOML comments begin with:

`#`

Example:

```
# Project metadata
[project]

name = "resource-manager"
```

Comments are useful for explaining unusual configuration.

---

41. Tables

This:

`[project]`

creates a table named:

`project`

This:

`[build-system]`

creates another table:

`build-system`

Nested tool configuration can look like:

```
[tool.ruff]
line-length = 88
```

---

42. A More Complete Example

```
[build-system]
requires = ["setuptools>=68"]
build-backend = "setuptools.build_meta"

[project]
name = "resource-manager"
version = "1.0.0"
description = "A Python resource management application"
readme = "README.md"
requires-python = ">=3.11"

authors = [
   { name = "Hafsa" }
]

dependencies = [
   "requests>=2.31",
   "rich>=13.0"
]

[project.optional-dependencies]
dev = [
   "pytest>=8",
```

```
    "ruff>=0.6"
]

[tool.ruff]
line-length = 88
```

This one file now describes:

```
How to build
    □
What the project is
    □
What Python it needs
    □
What packages it needs
    □
What development tools it uses
    □
How a supported tool should behave
```

---

43. What `pyproject.toml` Does NOT Automatically Do

Creating this file does not magically:

- Create a virtual environment
- Install every dependency immediately
- Run your application
- Publish your package
- Replace Git
- Replace your source code
- Automatically lock every dependency version

For example:

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

does not mean the package has already been installed.

It declares the requirement.

A package manager or installer still has to process it.

---

44. `pyproject.toml` Is Not a Lock File

This is an important distinction.

Suppose you write:

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

This allows multiple versions.

For example:

```
2.31
2.32
2.33
...
```

depending on what satisfies the dependency constraints.

A lock file is designed to record a specific resolved dependency set.

Therefore:

```
pyproject.toml
      □
lock file
```

Some modern tools can generate lock files separately.

---

45. Dependency Declaration vs Dependency Resolution

These are two different jobs.

Declaration

You say:

```
I need requests >= 2.31
```

Resolution

A tool determines:

```
Which exact version should be installed?
What dependencies does that version require?
Are there conflicts?
```

Understanding this distinction becomes important as projects grow.

---

46. A Real World Project Flow

Imagine building a Python API.

You might start with:

```
api_project/
□□□ pyproject.toml
□□□ src/
   □□□ api_project/
```

Your `pyproject.toml` might declare:

```
[project]
name = "api-project"
version = "0.1.0"
requires-python = ">=3.11"

dependencies = [
   "fastapi",
   "uvicorn"
]
```

Development tools:

```
[project.optional-dependencies]
dev = [
   "pytest",
   "ruff"
]
```

Now the project has a clear description of its environment.

---

47. Example: haas.dev Resource Project

Imagine a Python service that manages resources for haas.dev.

Project structure:

```
haas-resource-service/
□
□□□ pyproject.toml
□□□ README.md
□
□□□ src/
□   □□□ haas_resources/
□       □□□ __init__.py
□       □□□ models.py
□       □□□ services.py
□       □□□ repository.py
□
□□□ tests/
    □□□ test_services.py
```

The configuration might contain:

```
[project]
name = "haas-resource-service"
version = "1.0.0"
description = "Resource management service for haas.dev"
requires-python = ">=3.11"

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

```
[project.optional-dependencies]
dev = [
    "pytest",
    "ruff"
]
```

Now anyone working on the project can understand its basic requirements from one central file.

---

48. Common Mistake: Confusing Package Name and Import Name

Suppose:

```
name = "my-resource-tools"
```

You might import:

```
import my_resource_tools
```

Notice:

```
Distribution/project:
my-resource-tools

Python import:
my_resource_tools
```

Hyphens and underscores can behave differently in these contexts.

Do not assume the two names must be identical.

---

49. Common Mistake: Putting Every Package at Exact Versions

You might see:

```
dependencies = [
    "requests==2.32.3",
    "rich==13.8.1",
    "pydantic==2.9.2"
]
```

Exact versions can be useful in some situations, but blindly pinning every dependency can make upgrades harder.

Choose dependency constraints intentionally.

---

50. Common Mistake: Treating `pyproject.toml` as `requirements.txt`

A beginner may think:

```
pyproject.toml = requirements.txt
```

They are not identical.

`pyproject.toml` can describe:

```
project metadata
build configuration
dependencies
optional dependencies
tool configuration
```

while `requirements.txt` is primarily an installation requirements file.

---

51. Common Mistake: Invalid TOML Syntax

For example:

```
name = resource-manager
```

This is incorrect because the string is not quoted.

Correct:

```
name = "resource-manager"
```

Another mistake:

```
dependencies = [
   "requests"
   "rich"
]
```

The items need commas.

Correct:

```
dependencies = [
   "requests",
   "rich"
]
```

---

52. Common Mistake: Putting Runtime Dependencies in the Build System

Do not confuse:

`[build-system]`

with:

`[project]`

For example:

```
[project]
dependencies = [
    "requests"
]
```

means the application/project needs `requests`.

Whereas:

```
[build-system]
requires = [
    "setuptools"
]
```

means the build process needs setuptools.

---

53. Common Mistake: Forgetting Python Version Compatibility

If your project requires Python 3.11 features but you write:

`requires-python = ">=3.8"`

you are making an inaccurate compatibility claim.

Your declared Python requirement should match what your project actually supports.

---

54. Best Practices

1. Keep project metadata in one place

Use:

`pyproject.toml`

when supported by your tooling.

2. Declare Python compatibility

Use:

`requires-python = ">=3.11"`

when appropriate.

3. Declare runtime dependencies

Keep them under:

`[project]`

4. Separate optional development tools

Use:

`[project.optional-dependencies]`

when appropriate.

5. Keep build configuration separate

Use:

`[build-system]`

for build requirements.

6. Configure supported tools consistently

Use:

`[tool.tool_name]`

when the tool supports it.

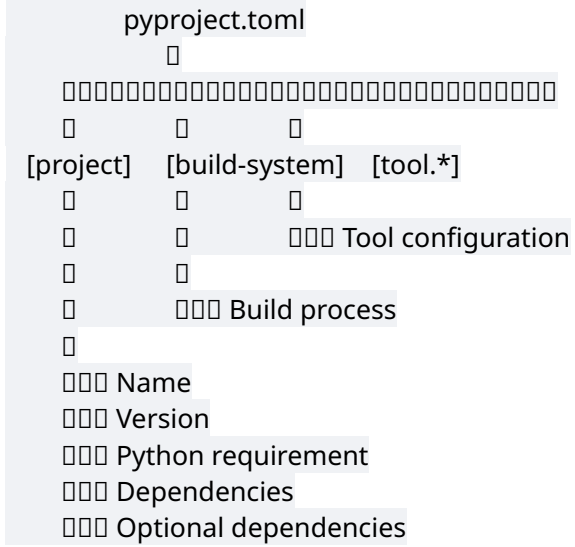
7. Do not blindly copy configuration

Understand what each field does.

---

55. `pyproject.toml` Mental Model

Remember this structure:



This is the most important mental model.

---

56. When Should You Use `pyproject.toml`?

Use it when:

- Building a modern Python project
- Creating a reusable package
- Publishing a Python package
- Defining project dependencies
- Defining Python compatibility
- Configuring supported development tools
- Defining a build system
- Creating a structured Python project

For a tiny one-file script, you may not need it.

For a serious Python project, it becomes much more useful.

---

57. `pyproject.toml` in a Team

Imagine three developers working on the same project.

Without centralized project metadata:

```
Developer A
"Which Python version?"

Developer B
"Which packages?"

Developer C
"Which build tool?"
```

With:

```
pyproject.toml
```

the project has a central place describing these requirements.

It becomes part of the project's source code and should normally be committed to Git.

---

58. `pyproject.toml` and Git

A typical repository should include:

`pyproject.toml`

but should generally not include:

`.venv/`

For example:

```
project/
├── pyproject.toml      ✅ commit
├── README.md           ✅ commit
├── src/                ✅ commit
├── tests/              ✅ commit
└── .venv/              ❌ do not commit
```

The environment can be recreated from the project's configuration and dependency workflow.

---

59. From Project to Package

A useful way to understand modern Python packaging is:

```
Write code
   ↓
Organize package
   ↓
Create pyproject.toml
   ↓
Declare metadata
   ↓
```

```
Declare dependencies
  □
Configure build system
  □
Build package
  □
Install or publish
```

`pyproject.toml` sits near the center of this workflow.

---

60. Mini Project: Create a Packaged Python Project

Create:

```
task_manager/
□
□□□ pyproject.toml
□□□ README.md
□
□□□ src/
   □□□ task_manager/
      □□□ __init__.py
      □□□ tasks.py
```

Step 1: Create `pyproject.toml`

```
[build-system]
requires = ["setuptools>=68"]
build-backend = "setuptools.build_meta"

[project]
name = "task-manager"
```

```
version = "0.1.0"
description = "A simple task management package"
readme = "README.md"
requires-python = ">=3.11"

dependencies = []

[project.optional-dependencies]
dev = [
    "pytest",
    "ruff"
]
```

Step 2: Create the package

```
# src/task_manager/tasks.py

def create_task(title):
    return {
        "title": title,
        "completed": False
    }
```

Step 3: Install it in editable mode

```
python -m pip install -e ".[dev]"
```

Step 4: Use the package

```
from task_manager.tasks import create_task

task = create_task("Learn pyproject.toml")

print(task)
```

Output:

```
{'title': 'Learn pyproject.toml', 'completed': False}
```

---

61. 5 Minute Challenge

Create a new Python project called:

```
student_manager
```

Your `pyproject.toml` should contain:

- Project name
- Version
- Description
- Python version requirement
- One runtime dependency
- One development dependency
- Build system

Then answer:

```
1. What is the project called?
2. Which Python versions does it support?
3. Which package does the application need?
4. Which package is only for development?
5. Which build backend is being used?
```

---

62. Exercises

Exercise 1

Write a `pyproject.toml` for:

`weather_app`

Requirements:

`Python 3.11+`
`requests`
`pytest for development`

Exercise 2

Identify the purpose of each section:

`[project]`

`[build-system]`

`[project.optional-dependencies]`

`[tool.ruff]`

Exercise 3

Explain the difference between:

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

```
]
```

and:

```
[build-system]
requires = [
    "setuptools>=68"
]
```

Exercise 4

Explain why this is useful:

```
requires-python = ">=3.11"
```

---

63. Mini Quiz

Question 1

What is `pyproject.toml` primarily used for?

A. Storing Python source code
B. Project configuration and packaging metadata
C. Storing database records
D. Running Python code

Answer: B

Question 2

Where are project runtime dependencies normally declared?

A. `[project]`
B. `[build-system]`
C. `[database]`
D. `[python]`

Answer: A

Question 3

What does this mean?

```
requires-python = ">=3.11"
```

A. Only Python 3.11.0
B. Python 3.11 or newer
C. Python older than 3.11
D. Python 2 only

Answer: B

Question 4

What is the purpose of `[build-system]`?

A. Configure the application's database
B. Describe how the project is built
C. Store passwords
D. Define Python variables

Answer: B

Question 5

What does this represent?

```
[tool.ruff]
```

A. Ruff's project configuration
B. A Python class
C. A dependency
D. A virtual environment

Answer: A

---

64. Interview Questions

Beginner

1. What is `pyproject.toml`?
2. What is TOML?
3. Why is `pyproject.toml` used in Python projects?
4. What is the `[project]` section?

5. How do you declare Python version requirements?
6. How do you declare project dependencies?
7. What are optional dependencies?

Intermediate

8. What is `[build-system]`?
9. What is a build backend?
10. What is the difference between build dependencies and runtime dependencies?
11. What is an editable installation?
12. How does `pyproject.toml` differ from `requirements.txt`?
13. How does it differ from `setup.py`?
14. Is `pyproject.toml` a lock file?
15. What are `[tool.*]` sections used for?

Advanced

16. What problem does PEP 517 solve?
17. What problem does PEP 518 solve?
18. What is PEP 621?
19. What is the difference between a build frontend and build backend?
20. Why might a project use a `src` layout?
21. Why can distribution names and import package names differ?
22. How are optional dependency groups useful?
23. Why should build dependencies be separated from runtime dependencies?
24. How does a project become installable from its `pyproject.toml`?
25. Why is dependency declaration different from dependency resolution?

---

65. Cheat Sheet

Basic project

```
[project]

name = "my-project"
version = "1.0.0"
description = "My Python project"
requires-python = ">=3.11"
```

README

```
readme = "README.md"
```

Dependencies

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

Optional dependencies

```
[project.optional-dependencies]

dev = [
    "pytest",
    "ruff"
]
```

Build system

```
[build-system]

requires = [
    "setuptools>=68"
]

build-backend = "setuptools.build_meta"
```

Tool configuration

```
[tool.ruff]

line-length = 88
```

Install project

```
python -m pip install .
```

Editable install

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

Install development extras

```
python -m pip install -e ".[dev]"
```

Build package

```
python -m pip install build
python -m build
```

---

66. Key Takeaways

- `pyproject.toml` is a modern central configuration file for Python projects.
- It uses the TOML format.
- `[project]` describes the project itself.
- Project dependencies can be declared in `[project]`.
- Optional dependencies can be grouped under `[project.optional-dependencies]`.
- `[build-system]` describes the tools required to build the project.
- `[tool.*]` sections can configure supported development tools.

- `pyproject.toml` can replace or reduce the need for older configuration approaches.
- It is not automatically a virtual environment.
- It is not the same thing as `requirements.txt`.
- It is not a dependency lock file.
- Runtime dependencies and build dependencies serve different purposes.
- A well-designed `pyproject.toml` makes a Python project's requirements and tooling easier to understand, reproduce, build, and distribute.

Final Mental Model

When you see:

`pyproject.toml`

think:

```
            Python Project
                 │
                 ▼
           pyproject.toml
                 │
   ┌─────────────┼─────────────┐
   ▼             ▼             ▼
Project       Build         Tools
Metadata      System        Configuration
   │             │             │
   ├─ name       ├─ backend    ├─ Ruff
   ├─ version    ├─ requirements ├─ pytest
   ├─ Python                   ├─ other tools
   ├─ dependencies
   └─ optional dependencies
```

The key idea is simple:

> **`pyproject.toml` tells Python tooling what your project is, what it needs, how it can be built, and how supported tools should configure themselves.**

Visit haas.dev for more resources and guides.

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

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