`requirements.txt` in Python

1. What Is `requirements.txt`?

`requirements.txt` is a text file that lists the Python packages a project depends on.

For example:

```
requests==2.32.3
python-dotenv==1.1.0
fastapi==0.115.6
```

It tells another developer, a server, or a deployment system:

> These are the Python dependencies needed by this project.

Instead of manually installing packages one by one, you can install them with:

```
python -m pip install -r requirements.txt
```

---

2. Why Do We Need `requirements.txt`?

Imagine you create a Python application using:

```
requests
python-dotenv
fastapi
uvicorn
```

Your computer already has all of them installed.

Another developer clones your project.

They only see:

`app.py`

They do not automatically know:

- which packages you installed
- which versions you used
- which dependencies the project needs

Without a dependency file, they may have to guess.

With:

`requirements.txt`

the setup becomes much easier.

```
Clone project
    
Create virtual environment
    
Install requirements.txt
    
Run project
```

---

3. What Does a Requirement Look Like?

The simplest form is:

```
requests
```

This means the project requires the `requests` package.

You can also specify a version:

```
requests==2.32.3
```

This means:

> Install exactly version `2.32.3`.

---

4. Version Operators

A requirements file can specify different version constraints.

Exact Version

```
requests==2.32.3
```

Exactly version `2.32.3`.

Minimum Version

```
requests>=2.31
```

Version `2.31` or newer.

Maximum Version

```
requests<3
```

Any version below `3`.

Minimum and Maximum

```
requests>=2.31,<3
```

At least `2.31`, but below `3`.

Not Equal

```
requests!=2.32.0
```

Do not install version `2.32.0`.

These constraints are useful when a project needs compatibility with a particular range of package versions.

---

5. Creating `requirements.txt` Manually

You can create a file named:

```
requirements.txt
```

and write:

```
requests==2.32.3
```

```
python-dotenv==1.1.0
```

Your project might look like:

```
my_project/
□□□ app.py
□□□ requirements.txt
□□□ .venv/
```

---

6. Installing From `requirements.txt`

Activate your virtual environment first.

Then run:

```
python -m pip install -r requirements.txt
```

The `-r` means:

```
read requirements from this file
```

`pip` reads each requirement and installs the necessary packages.

---

7. What Happens During Installation?

Suppose your file contains:

```
requests==2.32.3
python-dotenv==1.1.0
```

You run:

```
python -m pip install -r requirements.txt
```

Conceptually:

```
requirements.txt
      
pip reads requirements
      
Finds packages
      
Resolves dependencies
      
Downloads packages
      
Installs them into active environment
```

If those packages have their own dependencies, `pip` may install those too.

---

8. Generating `requirements.txt` With `pip freeze`

You can generate a requirements file from the current environment:

```
python -m pip freeze > requirements.txt
```

For example, your environment might contain:

```
requests==2.32.3
python-dotenv==1.1.0
urllib3==2.3.0
```

`certifi==2025.1.31`

The command saves that package information into:

`requirements.txt`

---

9. What Does `pip freeze` Actually Do?

`pip freeze` reports the installed packages and their versions in the current environment.

For example:

```
python -m pip freeze
```

might output:

```
certifi==2025.1.31
charset-normalizer==3.4.1
idna==3.10
requests==2.32.3
urllib3==2.3.0
```

When you use:

```
python -m pip freeze > requirements.txt
```

the `>` redirects that output into the file.

So:

`pip freeze`

means:

> Show the environment's installed packages.

while:

`pip freeze > requirements.txt`

means:

> Save that package list into `requirements.txt`.

---

10. Direct Dependencies vs Transitive Dependencies

This is an important concept.

Suppose you install:

`python -m pip install requests`

Your application directly depends on:

`requests`

But `requests` itself depends on other packages.

Conceptually:

```
Your App
  □
requests
  □□□ urllib3
  □□□ certifi
  □□□ charset-normalizer
  □□□ idna
```

So your environment contains:

```
requests
urllib3
certifi
charset-normalizer
idna
```

These are not all direct dependencies of your application.

Direct dependency

A package your project explicitly chooses to use.

Transitive dependency

A package required by another dependency.

---

11. Why `pip freeze` Can Include More Packages Than You Expect

Suppose you intentionally installed:

```
requests
```

You might expect:

```
requests==...
```

But:

```
python -m pip freeze
```

can include:

```
requests==...
urllib3==...
certifi==...
charset-normalizer==...
idna==...
```

That's because `pip freeze` describes what is installed in the environment rather than trying to determine which packages you personally intended as direct dependencies.

This distinction becomes important in larger projects.

---

12. A Simple Requirements File

For a small project, you might write:

```
requests==2.32.3
python-dotenv==1.1.0
```

This is easy to understand.

The file describes the dependencies your application needs.

---

13. Comments in `requirements.txt`

You can add comments using `#`.

Example:

```
# HTTP requests
requests==2.32.3

# Environment variables
python-dotenv==1.1.0
```

Comments help organize larger files.

They are ignored by `pip`.

---

14. Blank Lines

Blank lines are allowed.

For example:

```
# Web
fastapi==0.115.6
uvicorn==0.34.0
```

```
# Configuration
python-dotenv==1.1.0
```

They can make a requirements file easier to read.

---

15. Installing a Requirements File From Another Location

You can provide a path:

```
python -m pip install -r config/requirements.txt
```

For example:

```
project/
□□□ config/
□   □□□ requirements.txt
□□□ app/
    □□□ main.py
```

You can install it with:

```
python -m pip install -r config/requirements.txt
```

---

16. Multiple Requirements Files

Larger projects sometimes use multiple requirement files.

For example:

`requirements.txt`
`requirements-dev.txt`
`requirements-test.txt`

Each can serve a different purpose.

Production

`requirements.txt`

Could contain:

```
fastapi
uvicorn
python-dotenv
```

Development

`requirements-dev.txt`

Could contain:

```
-r requirements.txt
pytest
ruff
```

The line:

`-r requirements.txt`

means:

> Include the requirements from this other file.

Then install development dependencies with:

```
python -m pip install -r requirements-dev.txt
```

---

17. Development vs Production Dependencies

Not every package is needed when your application runs in production.

For example:

```
Application
□□□ fastapi
□□□ uvicorn
□□□ python-dotenv

Development
□□□ pytest
□□□ ruff
□□□ mypy
```

You may want production dependencies separated from tools used only during development.

This keeps the production environment focused on what the application actually needs.

---

18. Requirements Files for Testing

A project might have:

```
requirements.txt
```

`requirements-test.txt`

For example:

`requirements-test.txt`

could contain:

```
-r requirements.txt
pytest==8.3.4
```

Then:

```
python -m pip install -r requirements-test.txt
```

installs both the application dependencies and testing tools.

---

19. Requirements Files for Development

You can also have:

`requirements-dev.txt`

Example:

```
-r requirements.txt
pytest
ruff
mypy
```

Now the development environment contains the application's dependencies plus development tools.

---

20. The -r Option

The -r option tells pip to read requirements from a file.

Example:

```
python -m pip install -r requirements.txt
```

You can also use it inside a requirements file:

```
-r requirements.txt
pytest
```

This allows requirement files to reference other requirement files.

---

21. Editable Local Dependencies

A requirements file can reference a local project.

For example:

```
-e .
```

This means the current project can be installed in editable mode.

You might see:

```
-e .
pytest
```

```
ruff
```

This is more common in package development and advanced project setups.

---

22. Installing From Git

A requirements file can also reference a Git repository.

For example:

```
SomePackage @ git+https://github.com/example/SomePackage.git
```

This is useful in specialized cases where a dependency is not being installed from a normal package release.

For production systems, reproducibility should be considered carefully when using a moving Git branch.

A specific release or commit can provide a more predictable dependency source.

---

23. Why Committing `requirements.txt` Matters

A common Git project might contain:

```
my_project/
□□□ app/
□□□ requirements.txt
□□□ .gitignore
□□□ README.md
```

The developer commits:

`requirements.txt`

but normally does not commit:

`.venv/`

Why?

Because:

`requirements.txt`

describes the dependencies.

While:

`.venv/`

contains the actual local environment.

The environment can be recreated.

---

24. `requirements.txt` and `.venv`

These work together.

`requirements.txt`

    tells pip what to install

  `.venv`

```
    □
    □□□ installed packages
```

For example:

```
project/
□□□ .venv/
□□□ requirements.txt
□□□ app.py
```

The `.venv` directory is local.

The requirements file can be shared.

---

25. Recreating an Environment

Imagine you clone a project:

```
git clone <project>
cd project
```

You create an environment:

```
python -m venv .venv
```

Activate it.

Then:

```
python -m pip install -r requirements.txt
```

Now the dependencies are installed.

The workflow is:

```
Clone
 
Create .venv
 
Activate
 
Install requirements
 
Run application
```

---

26. Rebuilding a Broken Environment

Suppose your environment has become messy.

Instead of spending a long time trying to determine which package caused the problem, you can often recreate it.

```
Delete .venv
    
Create .venv again
    
Install requirements.txt
```

Commands:

```
python -m venv .venv
python -m pip install -r requirements.txt
```

This works well when your dependency configuration is reliable.

---

27. Updating `requirements.txt`

Suppose you currently have:

```
requests==2.31.0
```

You decide to upgrade:

```
python -m pip install --upgrade requests
```

Now your environment may contain:

```
requests==2.32.3
```

If your project tracks exact versions, the requirements file should be updated accordingly.

You can regenerate it:

```
python -m pip freeze > requirements.txt
```

However, remember that this can rewrite the file with all packages currently installed, including transitive dependencies.

For larger projects, dependency updates should be deliberate rather than blindly regenerating everything.

---

28. Should You Always Use `pip freeze`?

Not necessarily.

`pip freeze` is convenient, especially for simple projects.

But it reports the environment's installed packages rather than expressing only your application's direct dependency intentions.

For example, if your application directly needs:

```
requests
```

a manually maintained requirements file might contain:

```
requests==2.32.3
```

while `pip freeze` can also record packages that `requests` depends on.

Both approaches have legitimate uses.

---

29. `requirements.txt` vs `pyproject.toml`

Both can describe project dependencies, but they serve different roles.

`requirements.txt`

Commonly used for:

- application environments
- deployment
- simple projects
- installing a known set of dependencies

- existing Python workflows

Example:

```
requests==2.32.3
fastapi==0.115.6
```

pyproject.toml

A modern standard configuration file that can contain:

- project metadata
- dependencies
- build configuration
- tool configuration

Example:

```
[project]
dependencies = [
   "requests>=2.31",
   "fastapi>=0.115"
]
```

`pyproject.toml` is especially important when building Python packages.

---

30. `requirements.txt` Is Not a Lock File

This distinction matters.

A requirements file can contain exact versions:

`requests==2.32.3`

but the overall dependency-resolution behavior depends on the complete set of requirements and package metadata.

A fully locked environment can involve additional tooling that records the complete resolved dependency graph.

Therefore:

`requirements.txt`

should not automatically be treated as the same thing as a modern dependency lock file.

---

31. Reproducibility

One of the main goals of dependency management is reproducibility.

You want:

Developer A

requirements

Environment

Developer B

requirements

Environment

to produce compatible environments.

However, reproducibility depends on more than just package names.

It can also involve:

- Python version
- operating system
- platform-specific packages
- package versions
- dependency resolution
- system-level libraries

A requirements file helps, but it does not magically make every environment identical.

---

32. Python Version Matters Too

Suppose your project uses:

`Python 3.13`

and a package only supports certain Python versions.

Another developer uses:

`Python 3.9`

Even with the same:

`requirements.txt`

installation may behave differently.

So a reproducible project should document its supported Python version as well.

For example:

`Python 3.12+`

in the README or project configuration.

---

33. Platform Differences

Some packages behave differently across:

`Windows`
`macOS`
`Linux`

A requirements file may therefore need environment markers.

For example:

`some-package; sys_platform == "win32"`

This means the dependency applies only on Windows.

Another example:

`some-package; python_version >= "3.12"`

This makes the requirement conditional.

---

34. Dependency Constraints

Sometimes you want to control a version without directly declaring the package as an application dependency.

A constraints file can be used for this purpose:

`constraints.txt`

For example:

`urllib3==2.3.0`

Then:

```
python -m pip install -r requirements.txt -c constraints.txt
```

Conceptually:

`requirements.txt`
 What the project needs

`constraints.txt`
 Which versions are allowed

This is more advanced and is useful in larger dependency-management workflows.

---

35. Environment Markers

Requirements can be conditional.

For example:

```
colorama; platform_system == "Windows"
```

This means the dependency applies when the platform is Windows.

You can also use Python-version conditions:

```
package-name; python_version >= "3.12"
```

Environment markers allow one dependency file to represent different supported environments.

---

36. Hashes and Stronger Reproducibility

Requirements can also include hashes.

For example:

```
package==1.2.3 \
    --hash=sha256:...
```

Hashes can help verify that the downloaded distribution matches an expected artifact.

This is an advanced security and reproducibility technique.

You do not need hashes for basic Python projects, but they are useful in security-sensitive or highly controlled environments.

---

37. Common Mistake: Putting `.venv` in `requirements.txt`

Do not do this:

`.venv/`

inside `requirements.txt`.

`.venv/` belongs in:

`.gitignore`

The requirements file should describe packages and installable dependencies.

---

38. Common Mistake: Confusing `pip freeze` With Direct Dependencies

Suppose you install:

`fastapi`

and then run:

`python -m pip freeze > requirements.txt`

The file may contain many packages.

That does not mean your application directly uses every one of them.

Some may be dependencies of FastAPI or its dependency chain.

---

39. Common Mistake: Forgetting to Update the File

You install:

```
python -m pip install requests
```

Your program works.

But you forget to update the project's dependency configuration.

Another developer clones the repository and runs:

```
python -m pip install -r requirements.txt
```

`requests` is missing.

Their application fails.

The dependency file should stay synchronized with the project.

---

40. Common Mistake: Blindly Regenerating Requirements

Running:

```
python -m pip freeze > requirements.txt
```

can replace your carefully organized file with the entire current environment.

This may introduce:

- unused packages
- development tools
- transitive dependencies
- unrelated packages accidentally installed into the environment

Use `pip freeze` intentionally.

A clean virtual environment helps when you do need to generate a complete environment snapshot.

---

41. Best Practices

1. Keep dependencies explicit

Know what your application actually needs.

2. Use a virtual environment

`python -m venv .venv`

3. Keep `.venv` out of Git

`.venv/`

4. Commit dependency configuration

Usually:

`requirements.txt`

or an appropriate `pyproject.toml` configuration.

5. Be deliberate with versions

Do not randomly upgrade dependencies.

6. Keep development dependencies separate when useful

For example:

`requirements-dev.txt`

7. Document the Python version

Dependency compatibility also depends on Python itself.

8. Test after dependency changes

A package update can change application behavior.

9. Remove unused dependencies

Do not keep packages that the project no longer needs.

10. Rebuild environments periodically

A clean environment can reveal missing dependencies that were previously hidden by accidental global installations.

---

42. Real World Example: haas.dev Resource Service

Imagine a Python backend used to serve haas.dev resources.

The application directly uses:

`fastapi`

```
python-dotenv
requests
```

A simple dependency file might be:

```
fastapi==0.115.6
python-dotenv==1.1.0
requests==2.32.3
```

Project structure:

```
haas_resource_service/
□□□ .venv/
□□□ app/
□   □□□ main.py
□   □□□ services.py
□   □□□ config.py
□□□ requirements.txt
□□□ .gitignore
□□□ README.md
```

A new developer can run:

```
python -m venv .venv
```

Activate it and then:

```
python -m pip install -r requirements.txt
```

The dependency setup is now documented rather than existing only on the original developer's machine.

---

43. Complete Workflow

Step 1: Create the project

```
mkdir my_project
cd my_project
```

Step 2: Create the environment

```
python -m venv .venv
```

Step 3: Activate it

Windows:

```
.venv\Scripts\activate
```

macOS/Linux:

```
source .venv/bin/activate
```

Step 4: Install dependencies

```
python -m pip install requests python-dotenv
```

Step 5: Create the requirements file

```
python -m pip freeze > requirements.txt
```

Step 6: Add `.venv` to `.gitignore`

```
.venv/
```

Step 7: Commit the project

Commit:

```
app.py
requirements.txt
.gitignore
```

Do not normally commit:

```
.venv/
```

---

44. 5 Minute Challenge

Create:

```
inventory_app/
```

Inside it:

```
inventory_app/
□□□ .venv/
□□□ app.py
□□□ requirements.txt
□□□ .gitignore
```

Install:

```
python -m pip install requests python-dotenv
```

Create:

```
python -m pip freeze > requirements.txt
```

Now:

1. Open `requirements.txt`.
2. Identify the packages you directly installed.
3. Identify packages that were installed as dependencies.
4. Delete `.venv`.
5. Create it again.
6. Install everything using:

```
python -m pip install -r requirements.txt
```

7. Run your application.

Goal

Understand how a dependency file allows you to recreate an environment.

---

45. Mini Project: Reproducible Project Setup

Build a small Python application such as:

```
expense_tracker/
```

Use at least two third-party packages.

Your project must contain:

```
expense_tracker/
□□□ app/
□   □□□ main.py
□□□ requirements.txt
□□□ .gitignore
□□□ README.md
```

Requirements

Your README should document:

```
Python version
How to create the virtual environment
How to activate it
How to install requirements
How to run the application
```

The project should work after:

```
python -m venv .venv
python -m pip install -r requirements.txt
```

This is the real purpose of a dependency file:

> Another developer should be able to understand and reproduce the project's dependency setup without asking you what you installed.

---

46. Exercises

Exercise 1

Create a requirements file containing:

```
requests
python-dotenv
```

Exercise 2

Change the requirements to require an exact version of each package.

Exercise 3

Create a development requirements file:

`requirements-dev.txt`

Include:

```
-r requirements.txt
pytest
```

Exercise 4

Explain the difference between:

`requirements.txt`

and:

`.venv/`

Exercise 5

Run:

`python -m pip freeze`

and compare the output with your manually written requirements file.

Explain why they may not contain exactly the same packages.

Exercise 6

Create a requirement that only applies to Windows using an environment marker.

Exercise 7

Delete your virtual environment and recreate it using only:

`requirements.txt`

---

47. Mini Quiz

Question 1

What is the purpose of `requirements.txt`?

A. Store Python source code
B. Store project dependencies
C. Store passwords
D. Store virtual environments

Answer: B

Question 2

Which command installs dependencies from the file?

```
python -m pip install -r requirements.txt
```

Question 3

What does this mean?

```
requests==2.32.3
```

Answer: The project requires exactly version `2.32.3` of `requests`.

Question 4

What does this command do?

```
python -m pip freeze > requirements.txt
```

Answer: It writes the installed packages and versions from the current environment into `requirements.txt`.

Question 5

Should `.venv/` normally be placed inside `requirements.txt`?

Answer: No. `.venv/` belongs in `.gitignore`; requirements files describe installable dependencies.

Question 6

What is a transitive dependency?

Answer: A package required by another package in your project's dependency chain.

Question 7

Is `requirements.txt` exactly the same thing as a lock file?

Answer: No. A requirements file can specify dependencies and versions, but complete dependency locking may require additional tooling.

---

48. Interview Questions

Beginner

1. What is `requirements.txt`?
2. Why do Python projects use it?
3. How do you install packages from it?
4. What does `pip freeze` do?
5. What does `-r` mean?
6. How do you specify an exact package version?
7. Should `.venv` be committed to Git?
8. Why should dependency files be committed?

9. What is a direct dependency?
10. What is a transitive dependency?

Intermediate

11. Why might `pip freeze` contain more packages than your application directly uses?
12. What is the difference between `requirements.txt` and `pyproject.toml`?
13. Is `requirements.txt` a lock file?
14. What are development dependencies?
15. Why might a project have `requirements-dev.txt`?
16. What are environment markers?
17. What are constraints files?
18. Why does the Python version matter for dependency reproducibility?
19. Why can dependency updates break an application?
20. Why is blindly regenerating `requirements.txt` potentially problematic?
21. How can a requirements file reference another requirements file?
22. What is an editable requirement such as `-e .`?
23. How can a dependency be installed from Git?
24. Why might the same requirements file behave differently across operating systems?
25. What makes a Python project's dependency setup reproducible?

---

49. Quick Cheat Sheet

Install requirements

```
python -m pip install -r requirements.txt
```

Generate requirements

```
python -m pip freeze > requirements.txt
```

List installed packages

```
python -m pip list
```

Show package details

```
python -m pip show requests
```

Exact version

```
requests==2.32.3
```

Minimum version

```
requests>=2.31
```

Version range

```
requests>=2.31,<3
```

Exclude version

```
requests!=2.32.0
```

Include another requirements file

```
-r requirements.txt
```

Editable local project

```
-e .
```

Environment-specific dependency

```
some-package; sys_platform == "win32"
```

Typical project structure

```
project/
□□□ .venv/
```

```
├── app/
│   └── main.py
├── requirements.txt
├── .gitignore
└── README.md
```

.gitignore

```
.venv/
```

---

50. Key Takeaways

- `requirements.txt` documents a Python project's dependencies.
- It makes project setup easier for other developers and deployment environments.
- Dependencies can be specified with version constraints.
- `pip install -r requirements.txt` installs the listed requirements.
- `pip freeze` can create a snapshot of packages installed in an environment.
- `pip freeze` may include transitive dependencies, not only packages your application directly uses.
- `requirements.txt` should normally be committed to version control.
- `.venv/` should normally not be committed.
- Development and production dependencies can be separated when useful.
- `pyproject.toml` is an important modern alternative or complement for project dependency and packaging configuration.
- A requirements file helps reproducibility, but Python version, operating system, and platform-specific dependencies can also matter.
- Dependency management should be intentional: install what you need, track it, test changes, and remove what you no longer use.

The Dependency Mental Model

Before you finish a Python project, ask:

What does my application directly depend on?
- [ ]

Which versions are supported?
- [ ]

Are development dependencies separate?
- [ ]

Can another developer create .venv?
- [ ]

Can they run:
python -m pip install -r requirements.txt
- [ ]

Does the application work?

Visit haas.dev for more resources and guides.

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