[![logo](../../../assets/logo-letter.svg)](../../.. "uv") uv

[uv](https://github.com/astral-sh/uv "Go to repository")

Projects

Troubleshooting

Policies

Table of contents

* Editable mode
  * Conflicting dependencies

  1. [Introduction](../../..)
  2. [Concepts](../../)
  3. [Projects](../)

# Configuring projects

## Python version requirement

Projects may declare the Python versions supported by the project in the `project.requires-python` field of the `pyproject.toml`.

It is recommended to set a `requires-python` value:

pyproject.toml

```
[project]
name = "example"
version = "0.1.0"
requires-python = ">=3.12"
```

The Python version requirement determines the Python syntax that is allowed in the project and affects selection of dependency versions (they must support

the same Python version range).

## Entry points

[Entry points](https://packaging.python.org/en/latest/specifications/entry-points/#entry-points) are the official term for an installed package to advertise interfaces. These include:

  * Command line interfaces
  * Graphical user interfaces
  * Plugin entry points

Important

Using the entry point tables requires a build system to be defined.

### Command-line interfaces

Projects may define command line interfaces (CLIs) for the project in the `[project.scripts]` table of the `pyproject.toml`.

For example, to declare a command called `hello` that invokes the `hello` function in the `example` module:

pyproject.toml

```
[project.scripts]
hello = "example:hello"
```

Then, the command can be run from a console:

```
$ uv run hello
```

### Graphical user interfaces

Projects may define graphical user interfaces (GUIs) for the project in the `[project.gui-scripts]` table of the `pyproject.toml`.

Important

These are only different from command-line interfaces on Windows, where they are wrapped by a GUI executable so they can be started without a console. On other platforms, they behave the same.

For example, to declare a command called `hello` that invokes the `app` function in the `example` module:

pyproject.toml

```
    [project.gui-scripts]
    hello = "example:app"
```

### Plugin entry points

Projects may define entry points for plugin discovery in the [`[project.entry-points]`](https://packaging.python.org/en/latest/guides/creating-and-discovering-plugins/#using-package-metadata) table of the `pyproject.toml`.

For example, to register the `example-plugin-a` package as a plugin for `example`:

pyproject.toml

```
    [project.entry-points.'example.plugins']
    a = "example_plugin_a"
```

Then, in `example`, plugins would be loaded with:

example/__init__.py

```
    from importlib.metadata import entry_points

    for plugin in entry_points(group='example.plugins'):
        plugin.load()
```

Note

The `group` key can be an arbitrary value, it does not need to include the package name or "plugins". However, it is recommended to namespace the key by the package name to avoid collisions with other packages.

## Build systems

A build system determines how the project should be packaged and installed. Projects may declare and configure a build system in the `[build-system]` table of the `pyproject.toml`.

uv uses the presence of a build system to determine if a project contains a package that should be installed in the project virtual environment. If a build system is not defined, uv will not attempt to build or install the project itself, just its dependencies. If a build system is defined, uv will build and install the project into the project environment.

The `--build-backend` option can be provided to `uv init` to create a packaged project with an appropriate layout. The `--package` option can be provided to `uv init` to create a packaged project with the default build system.

Note

While uv will not build and install the current project without a build system definition, the presence of a `[build-system]` table is not required in other packages. For legacy reasons, if a build system is not defined, then `setuptools.build_meta:__legacy__` is used to build the package. Packages you depend on may not explicitly declare their build system but are still installable. Similarly, if you add a dependency on a local package or install it with `uv pip`, uv will always attempt to build and install it.

### Build system options

Build systems are used to power the following features:

* Including or excluding files from distributions
* Editable install behavior
* Dynamic project metadata
* Compilation of native code
* Vendoring shared libraries

To configure these features, refer to the documentation of your chosen build system.

## Project packaging

As discussed in build systems, a Python project must be built to be installed. This process is generally referred to as "packaging".

You probably need a package if you want to:

* Add commands to the project
* Distribute the project to others
* Use a `src` and `test` layout
* Write a library

You probably _do not_ need a package if you are:

* Writing scripts
* Building a simple application
* Using a flat layout

While uv usually uses the declaration of a build system to determine if a project should be packaged, uv also allows overriding this behavior with the [`tool.uv.package`](../../../reference/settings/#package) setting.

Setting `tool.uv.package = true` will force a project to be built and installed into the project environment. If no build system is defined, uv will use the setuptools legacy backend.

Setting `tool.uv.package = false` will force a project package _not_ to be

built and installed into the project environment. uv will ignore a declared

build system when interacting with the project.

## Project environment path

The `UV_PROJECT_ENVIRONMENT` environment variable can be used to configure the

project virtual environment path (`.venv` by default).

If a relative path is provided, it will be resolved relative to the workspace

root. If an absolute path is provided, it will be used as-is, i.e. a child

directory will not be created for the environment. If an environment is not

present at the provided path, uv will create it.

This option can be used to write to the system Python environment, though it

is not recommended. `uv sync` will remove extraneous packages from the

environment by default and, as such, may leave the system in a broken state.

To target the system environment, set `UV_PROJECT_ENVIRONMENT` to the prefix

of the Python installation. For example, on Debian-based systems, this is

usually `/usr/local`:

```
$ python -c "import sysconfig; print(sysconfig.get_config_var('prefix'))"
/usr/local
```

To target this environment, you'd export `UV_PROJECT_ENVIRONMENT=/usr/local`.

Important

If an absolute path is provided and the setting is used across multiple
projects, the environment will be overwritten by invocations in each project.
This setting is only recommended for use for a single project in CI or Docker
images.

Note

By default, uv does not read the `VIRTUAL_ENV` environment variable during
project operations. A warning will be displayed if `VIRTUAL_ENV` is set to a
different path than the project's environment. The `--active` flag can be used
to opt-in to respecting `VIRTUAL_ENV`. The `--no-active` flag can be used to
silence the warning.

## Limited resolution environments

If your project supports a more limited set of platforms or Python versions,
you can constrain the set of solved platforms via the `environments` setting,
which accepts a list of PEP 508 environment markers. For example, to constrain
the lockfile to macOS and Linux, and exclude Windows:

pyproject.toml

```
[tool.uv]
environments = [
    "sys_platform == 'darwin'",
    "sys_platform == 'linux'",
]
```

Or, to exclude alternative Python implementations:

pyproject.toml

```
[tool.uv]
environments = [
    "implementation_name == 'cpython'"
]
```

Entries in the `environments` setting must be disjoint (i.e., they must not overlap). For example, `sys_platform == 'darwin'` and `sys_platform == 'linux'` are disjoint, but `sys_platform == 'darwin'` and `python_version >= '3.9'` are not, since both could be true at the same time.

## Build isolation

By default, uv builds all packages in isolated virtual environments, as per [PEP 517](https://peps.python.org/pep-0517/). Some packages are incompatible with build isolation, be it intentionally (e.g., due to the use of heavy build dependencies, mostly commonly PyTorch) or unintentionally (e.g., due to the use of legacy packaging setups).

To disable build isolation for a specific dependency, add it to the `no-build-isolation-package` list in your `pyproject.toml`:

pyproject.toml

```
[project]
name = "project"
version = "0.1.0"
description = "..."
readme = "README.md"
requires-python = ">=3.12"
dependencies = ["cchardet"]

[tool.uv]
no-build-isolation-package = ["cchardet"]
```

Installing packages without build isolation requires that the package's build dependencies are installed in the project environment _prior_ to installing the package itself. This can be achieved by separating out the build dependencies and the packages that require them into distinct extras. For example:

pyproject.toml

```
[project]
name = "project"
version = "0.1.0"
description = "..."
readme = "README.md"
requires-python = ">=3.12"
dependencies = []

[project.optional-dependencies]
build = ["setuptools", "cython"]
compile = ["cchardet"]

[tool.uv]
no-build-isolation-package = ["cchardet"]
```

Given the above, a user would first sync the `build` dependencies:

```
$ uv sync --extra build
 + cython==3.0.11
 + foo==0.1.0 (from file:///Users/crmarsh/workspace/uv/foo)
 + setuptools==73.0.1
```

Followed by the `compile` dependencies:

```
$ uv sync --extra compile
 + cchardet==2.1.7
 - cython==3.0.11
 - setuptools==73.0.1
```

Note that `uv sync --extra compile` would, by default, uninstall the `cython` and `setuptools` packages. To instead retain the build dependencies, include both extras in the second `uv sync` invocation:

```
$ uv sync --extra build
$ uv sync --extra build --extra compile
```

Some packages, like `cchardet` above, only require build dependencies for the _installation_ phase of `uv sync`. Others, like `flash-attn`, require their build dependencies to be present even just to resolve the project's lockfile during the _resolution_ phase.

In such cases, the build dependencies must be installed prior to running any `uv lock` or `uv sync` commands, using the lower lower-level `uv pip` API. For example, given:

pyproject.toml

```toml
[project]
name = "project"
version = "0.1.0"
description = "..."
readme = "README.md"
requires-python = ">=3.12"
dependencies = ["flash-attn"]

[tool.uv]
no-build-isolation-package = ["flash-attn"]
```

You could run the following sequence of commands to sync `flash-attn`:

```
$ uv venv
$ uv pip install torch
$ uv sync
```

Alternatively, you can provide the `flash-attn` metadata upfront via the [`dependency-metadata`](../../../reference/settings/#dependency-metadata) setting, thereby forgoing the need to build the package during the dependency resolution phase. For example, to provide the `flash-attn` metadata upfront, include the following in your `pyproject.toml`:

pyproject.toml

```
[[tool.uv.dependency-metadata]]
name = "flash-attn"
version = "2.6.3"
requires-dist = ["torch", "einops"]
```

Tip

To determine the package metadata for a package like `flash-attn`, navigate to the appropriate Git repository, or look it up on

[PyPI](https://pypi.org/project/flash-attn) and download the package's source

distribution. The package requirements can typically be found in the

`setup.py` or `setup.cfg` file.

(If the package includes a built distribution, you can unzip it to find the

`METADATA` file; however, the presence of a built distribution would negate

the need to provide the metadata upfront, since it would already be available

to uv.)

Once included, you can again use the two-step `uv sync` process to install the

build dependencies. Given the following `pyproject.toml`:

pyproject.toml

```
[project]
name = "project"
version = "0.1.0"
description = "..."
readme = "README.md"
requires-python = ">=3.12"
dependencies = []

[project.optional-dependencies]
build = ["torch", "setuptools", "packaging"]
```

```
    compile = ["flash-attn"]


    [tool.uv]
    no-build-isolation-package = ["flash-attn"]


    [[tool.uv.dependency-metadata]]
    name = "flash-attn"
    version = "2.6.3"
    requires-dist = ["torch", "einops"]
```

You could run the following sequence of commands to sync `flash-attn`:

```
    $ uv sync --extra build
    $ uv sync --extra build --extra compile
```

Note

The `version` field in `tool.uv.dependency-metadata` is optional for registry-based dependencies (when omitted, uv will assume the metadata applies to all versions of the package), but _required_ for direct URL dependencies (like Git dependencies).

## Editable mode

By default, the project will be installed in editable mode, such that changes to the source code are immediately reflected in the environment. `uv sync` and `uv run` both accept a `--no-editable` flag, which instructs uv to install the project in non-editable mode. `--no-editable` is intended for deployment use-cases, such as building a Docker container, in which the project should be included in the deployed environment without a dependency on the originating source code.

## Conflicting dependencies

uv requires that all optional dependencies ("extras") declared by the project are compatible with each other and resolves all optional dependencies together when creating the lockfile.

If optional dependencies declared in one extra are not compatible with those in another extra, uv will fail to resolve the requirements of the project with an error.

To work around this, uv supports declaring conflicting extras. For example, consider two sets of optional dependencies that conflict with one another:

pyproject.toml

```
    [project.optional-dependencies]
```

```
extra1 = ["numpy==2.1.2"]
extra2 = ["numpy==2.0.0"]
```

If you run `uv lock` with the above dependencies, resolution will fail:

```
$ uv lock
  x No solution found when resolving dependencies:
    `-> Because myproject[extra2] depends on numpy==2.0.0 and myproject[extra1] depends on numpy==2.1.2, we can conclude that myproject[extra1] and
        myproject[extra2] are incompatible.
        And because your project requires myproject[extra1] and myproject[extra2], we can conclude that your projects's requirements are unsatisfiable.
```

But if you specify that `extra1` and `extra2` are conflicting, uv will resolve
them separately. Specify conflicts in the `tool.uv` section:

pyproject.toml

```
[tool.uv]
conflicts = [
    [
        { extra = "extra1" },
        { extra = "extra2" },
      ],
  ]
```

Now, running `uv lock` will succeed. Note though, that now you cannot install both `extra1` and `extra2` at the same time:

```
  $ uv sync --extra extra1 --extra extra2
  Resolved 3 packages in 14ms
  error: extra `extra1`, extra `extra2` are incompatible with the declared conflicts: {`myproject[extra1]`, `myproject[extra2]`}
```

This error occurs because installing both `extra1` and `extra2` would result in installing two different versions of a package into the same environment.

The above strategy for dealing with conflicting extras also works with dependency groups:

pyproject.toml

```
[dependency-groups]
group1 = ["numpy==2.1.2"]
group2 = ["numpy==2.0.0"]

[tool.uv]
conflicts = [
    [
      { group = "group1" },
      { group = "group2" },
    ],
]
```

The only difference with conflicting extras is that you need to use `group` instead of `extra`.

February 5, 2025

Made with [Material for MkDocs Insiders](https://squidfunk.github.io/mkdocs-material/)

[ ](https://github.com/astral-sh/uv "github.com") [ ](https://discord.com/invite/astral-sh "discord.com") [ ](https://pypi.org/project/uv/ "pypi.org") [ ](https://x.com/astral_sh

```
"x.com")
```