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

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

Projects

Configuration

The pip interface

Policies

Table of contents

# Locking environments

Locking is to take a dependency, e.g., `ruff`, and write an exact version to use to a file. When working with many dependencies, it is useful to lock the exact versions so the environment can be reproduced. Without locking, the versions of dependencies could change over time, when using a different tool, or across platforms.

## Locking requirements

uv allows dependencies to be locked in the `requirements.txt` format. It is recommended to use the standard `pyproject.toml` to define dependencies, but other dependency formats are supported as well. See the documentation on [declaring dependencies](../dependencies/) for more details on how to define dependencies.

To lock dependencies declared in a `pyproject.toml`:

```
$ uv pip compile pyproject.toml -o requirements.txt
```

Note by default the `uv pip compile` output is just displayed and `--output-file` / `-o` argument is needed to write to a file.

To lock dependencies declared in a `requirements.in`:

```
$ uv pip compile requirements.in -o requirements.txt
```

To lock dependencies declared in multiple files:

```
$ uv pip compile pyproject.toml requirements-dev.in -o requirements-dev.txt
```

uv also supports legacy `setup.py` and `setup.cfg` formats. To lock dependencies declared in a `setup.py`:

```
$ uv pip compile setup.py -o requirements.txt
```

To lock dependencies from stdin, use `-`:

```
$ echo "ruff" | uv pip compile -
```

To lock with optional dependencies enabled, e.g., the "foo" extra:

```
$ uv pip compile pyproject.toml --extra foo
```

To lock with all optional dependencies enabled:

```
$ uv pip compile pyproject.toml --all-extras
```

Note extras are not supported with the `requirements.in` format.

## Upgrading requirements

When using an output file, uv will consider the versions pinned in an existing output file. If a dependency is pinned it will not be upgraded on a subsequent compile run. For example:

```
$ echo "ruff==0.3.0" > requirements.txt
$ echo "ruff" | uv pip compile - -o requirements.txt
# This file was autogenerated by uv via the following command:
#    uv pip compile - -o requirements.txt
ruff==0.3.0
```

To upgrade a dependency, use the `--upgrade-package` flag:

```
$ uv pip compile - -o requirements.txt --upgrade-package ruff
```

To upgrade all dependencies, there is an `--upgrade` flag.

## Syncing an environment

Dependencies can be installed directly from their definition files or from compiled `requirements.txt` files with `uv pip install`. See the documentation on [installing packages from files](../packages/#installing-packages-from-files) for more details.

When installing with `uv pip install`, packages that are already installed will not be removed unless they conflict with the lockfile. This means that the environment can have dependencies that aren't declared in the lockfile, which isn't great for reproducibility. To ensure the environment exactly matches the lockfile, use `uv pip sync` instead.

To sync an environment with a `requirements.txt` file:

```
$ uv pip sync requirements.txt
```

To sync an environment with a `pyproject.toml` file:

```
$ uv pip sync pyproject.toml
```

## Adding constraints

Constraints files are `requirements.txt`-like files that only control the _version_ of a requirement that's installed. However, including a package in a constraints file will _not_ trigger the installation of that package. Constraints can be used to add bounds to dependencies that are not dependencies of the current project.

To define a constraint, define a bound for a package:

constraints.txt

```
pydantic<2.0
```

To use a constraints file:

```
$ uv pip compile requirements.in --constraint constraints.txt
```

Note that multiple constraints can be defined in each file and multiple files can be used.

## Overriding dependency versions

Overrides files are `requirements.txt`-like files that force a specific version of a requirement to be installed, regardless of the requirements declared by any constituent package, and regardless of whether this would be considered an invalid resolution.

While constraints are _additive_ , in that they're combined with the requirements of the constituent packages, overrides are _absolute_ , in that they completely replace the requirements of the constituent packages.

Overrides are most often used to remove upper bounds from a transitive dependency. For example, if `a` requires `c>=1.0,<2.0` and `b` requires `c>=2.0` and the current project requires `a` and `b` then the dependencies cannot be resolved.

To define an override, define the new requirement for the problematic package:

overrides.txt

```
c>=2.0
```

To use an overrides file:

```
$ uv pip compile requirements.in --override overrides.txt
```

Now, resolution can succeed. However, note that if `a` is _correct_ that it does not support `c>=2.0` then a runtime error will likely be encountered when using the packages.

Note that multiple overrides can be defined in each file and multiple files can be used.

September 6, 2024

"x.com")