Skip to content

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

uv

Compatibility with pip

Initializing search

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

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

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

Guides

* [Installing Python](../../guides/install-python/)
* [Running scripts](../../guides/scripts/)
* [Using tools](../../guides/tools/)
* [Working on projects](../../guides/projects/)
* [Publishing packages](../../guides/package/)
* [Integrations](../../guides/integration/)

Integrations

* [Docker](../../guides/integration/docker/)
* [Jupyter](../../guides/integration/jupyter/)
* [GitHub Actions](../../guides/integration/github/)
* [GitLab CI/CD](../../guides/integration/gitlab/)
* [Pre-commit](../../guides/integration/pre-commit/)
* [PyTorch](../../guides/integration/pytorch/)
* [FastAPI](../../guides/integration/fastapi/)
* [Alternative indexes](../../guides/integration/alternative-indexes/)
* [Dependency bots](../../guides/integration/dependency-bots/)
* [AWS Lambda](../../guides/integration/aws-lambda/)

* [Concepts](../../concepts/)

Concepts

* [Projects](../../concepts/projects/)

Projects

* [Structure and files](../../concepts/projects/layout/)
* [Creating projects](../../concepts/projects/init/)
* [Managing dependencies](../../concepts/projects/dependencies/)
* [Running commands](../../concepts/projects/run/)
* [Locking and syncing](../../concepts/projects/sync/)
* [Configuring projects](../../concepts/projects/config/)
* [Building distributions](../../concepts/projects/build/)
* [Using workspaces](../../concepts/projects/workspaces/)

* [Tools](../../concepts/tools/)
* [Python versions](../../concepts/python-versions/)
* [Resolution](../../concepts/resolution/)
* [Caching](../../concepts/cache/)

* [Configuration](../../configuration/)

Configuration

* [Configuration files](../../configuration/files/)
* [Environment variables](../../configuration/environment/)
* [Authentication](../../configuration/authentication/)
* [Package indexes](../../configuration/indexes/)
* [Installer](../../configuration/installer/)

* [The pip interface](../)

The pip interface

* [Using environments](../environments/)
* [Managing packages](../packages/)
* [Inspecting packages](../inspection/)
* [Declaring dependencies](../dependencies/)
* [Locking environments](../compile/)
* Compatibility with pip [Compatibility with pip](./) Table of contents
  * Configuration files and environment variables
  * Pre-release compatibility
  * Packages that exist on multiple indexes
  * PEP 517 build isolation
  * Transitive URL dependencies
  * Virtual environments by default
  * Resolution strategy
  * pip check
  * \--user and the user install scheme
  * \--only-binary enforcement
  * \--no-binary enforcement
  * manylinux_compatible enforcement
  * Bytecode compilation
  * Strictness and spec enforcement
  * pip command-line options and subcommands
  * Registry authentication
  * egg support
  * Build constraints
  * pip compile defaults
  * requires-python enforcement
  * Package priority

* [Reference](../../reference/)

Reference

* [Commands](../../reference/cli/)
* [Settings](../../reference/settings/)
* [Troubleshooting](../../reference/troubleshooting/)

Troubleshooting

* [Build failures](../../reference/troubleshooting/build-failures/)
* [Reproducible examples](../../reference/troubleshooting/reproducible-examples/)
* [Resolver](../../reference/resolver-internals/)
* [Benchmarks](../../reference/benchmarks/)
* [Policies](../../reference/policies/)

Policies

* [Versioning](../../reference/policies/versioning/)
* [Platform support](../../reference/policies/platforms/)
* [License](../../reference/policies/license/)

Table of contents

* PEP 517 build isolation
* Transitive URL dependencies
* Virtual environments by default
* Resolution strategy
* pip check
* \--user and the user install scheme
* \--only-binary enforcement
* \--no-binary enforcement
* manylinux_compatible enforcement
* Bytecode compilation
* Strictness and spec enforcement
* pip command-line options and subcommands
* Registry authentication
* egg support
* Build constraints
* pip compile defaults
* requires-python enforcement
* Package priority

1. [Introduction](../..)
2. [The pip interface](../)

Compatibility with `pip` and `pip-tools`

uv is designed as a drop-in replacement for common `pip` and `pip-tools`

workflows.

Informally, the intent is such that existing `pip` and `pip-tools` users can
switch to uv without making meaningful changes to their packaging workflows;
and, in most cases, swapping out `pip install` for `uv pip install` should
"just work".

However, uv is _not_ intended to be an _exact_ clone of `pip`, and the further
you stray from common `pip` workflows, the more likely you are to encounter
differences in behavior. In some cases, those differences may be known and
intentional; in others, they may be the result of implementation details; and
in others, they may be bugs.

This document outlines the known differences between uv and `pip`, along with
rationale, workarounds, and a statement of intent for compatibility in the
future.

Configuration files and environment variables

uv does not read configuration files or environment variables that are
specific to `pip`, like `pip.conf` or `PIP_INDEX_URL`.

Reading configuration files and environment variables intended for other tools
has a number of drawbacks:

1. It requires bug-for-bug compatibility with the target tool, since users end up relying on bugs in the format, the parser, etc.
2. If the target tool _changes_ the format in some way, uv is then locked-in to changing it in equivalent ways.

3. If that configuration is versioned in some way, uv would need to know _which version_ of the target tool the user is expecting to use.
4. It prevents uv from introducing any settings or configuration that don't exist in the target tool, since otherwise `pip.conf` (or similar) would no longer be usable with `pip`.
5. It can lead to user confusion, since uv would be reading settings that don't actually affect its behavior, and many users may _not_ expect uv to read configuration files intended for other tools.

Instead, uv supports its own environment variables, like `UV_INDEX_URL`. uv
also supports persistent configuration in a `uv.toml` file or a
`[tool.uv.pip]` section of `pyproject.toml`. For more information, see
[Configuration files](../../configuration/files/).

Pre-release compatibility

By default, uv will accept pre-release versions during dependency resolution
in two cases:

1. If the package is a direct dependency, and its version markers include a pre-release specifier (e.g., `flask>=2.0.0rc1`).
2. If _all_ published versions of a package are pre-releases.

If dependency resolution fails due to a transitive pre-release, uv will prompt
the user to re-run with `--prerelease allow`, to allow pre-releases for all
dependencies.

Alternatively, you can add the transitive dependency to your `requirements.in`
file with pre-release specifier (e.g., `flask>=2.0.0rc1`) to opt in to pre-

release support for that specific dependency.

In sum, uv needs to know upfront whether the resolver should accept pre-releases for a given package. `pip`, meanwhile, _may_ respect pre-release identifiers in transitive dependencies depending on the order in which the resolver encounters the relevant specifiers ([#1641](https://github.com/astral-sh/uv/issues/1641#issuecomment-1981402429)).

Pre-releases are [notoriously difficult](https://pubgrub-rs-guide.netlify.app/limitations/prerelease_versions) to model, and are a frequent source of bugs in packaging tools. Even `pip`, which is viewed as a reference implementation, has a number of open questions around pre-release handling ([#12469](https://github.com/pypa/pip/issues/12469), [#12470](https://github.com/pypa/pip/issues/12470), [#40505](https://discuss.python.org/t/handling-of-pre-releases-when-backtracking/40505/20), etc.). uv's pre-release handling is _intentionally_ limited and _intentionally_ requires user opt-in for pre-releases, to ensure correctness.

In the future, uv _may_ support pre-release identifiers in transitive dependencies. However, it's likely contingent on evolution in the Python packaging specifications. The existing PEPs [do not cover "dependency resolution"](https://discuss.python.org/t/handling-of-pre-releases-when-backtracking/40505/17) and are instead focused on behavior for a _single_ version specifier. As such, there are unresolved questions around the correct and intended behavior for pre-releases in the packaging ecosystem more

broadly.

Packages that exist on multiple indexes

In both uv and `pip`, users can specify multiple package indexes from which to search for the available versions of a given package. However, uv and `pip` differ in how they handle packages that exist on multiple indexes.

For example, imagine that a company publishes an internal version of `requests` on a private index (`--extra-index-url`), but also allows installing packages from PyPI by default. In this case, the private `requests` would conflict with the public [`requests`](https://pypi.org/project/requests/) on PyPI.

When uv searches for a package across multiple indexes, it will iterate over the indexes in order (preferring the `--extra-index-url` over the default index), and stop searching as soon as it finds a match. This means that if a package exists on multiple indexes, uv will limit its candidate versions to those present in the first index that contains the package.

`pip`, meanwhile, will combine the candidate versions from all indexes, and select the best version from the combined set, though it makes [no guarantees around the order](https://github.com/pypa/pip/issues/5045#issuecomment-369521345) in which it searches indexes, and expects that packages are unique up to name and version, even across indexes.

uv's behavior is such that if a package exists on an internal index, it should always be installed from the internal index, and never from PyPI. The intent is to prevent "dependency confusion" attacks, in which an attacker publishes a malicious package on PyPI with the same name as an internal package, thus causing the malicious package to be installed instead of the internal package. See, for example, [the `torchtriton` attack](https://pytorch.org/blog/compromised-nightly-dependency/) from December 2022.

As of v0.1.39, users can opt in to `pip`-style behavior for multiple indexes via the `--index-strategy` command-line option, or the `UV_INDEX_STRATEGY` environment variable, which supports the following values:

* `first-index` (default): Search for each package across all indexes, limiting the candidate versions to those present in the first index that contains the package, prioritizing the `--extra-index-url` indexes over the default index URL.
* `unsafe-first-match`: Search for each package across all indexes, but prefer the first index with a compatible version, even if newer versions are available on other indexes.
* `unsafe-best-match`: Search for each package across all indexes, and select the best version from the combined set of candidate versions.

While `unsafe-best-match` is the closest to `pip`'s behavior, it exposes users to the risk of "dependency confusion" attacks.

uv also supports pinning packages to dedicated indexes (see: [_Indexes_](../../configuration/indexes/#pinning-a-package-to-an-index)), such that a given package is _always_ installed from a specific index.

PEP 517 build isolation

uv uses [PEP 517](https://peps.python.org/pep-0517/) build isolation by default (akin to `pip install --use-pep517`), following `pypa/build` and in anticipation of `pip` defaulting to PEP 517 builds in the future ([pypa/pip#9175](https://github.com/pypa/pip/issues/9175)).

If a package fails to install due to a missing build-time dependency, try using a newer version of the package; if the problem persists, consider filing an issue with the package maintainer, requesting that they update the packaging setup to declare the correct PEP 517 build-time dependencies.

As an escape hatch, you can preinstall a package's build dependencies, then run `uv pip install` with `--no-build-isolation`, as in:

```
uv pip install wheel && uv pip install --no-build-isolation biopython==1.77
```

For a list of packages that are known to fail under PEP 517 build isolation, see [#2252](https://github.com/astral-sh/uv/issues/2252).

Transitive URL dependencies

While uv includes first-class support for URL dependencies (e.g., `ruff @

https://...`), it differs from pip in its handling of _transitive_ URL dependencies in two ways.

First, uv makes the assumption that non-URL dependencies do not introduce URL dependencies into the resolution. In other words, it assumes that dependencies fetched from a registry do not themselves depend on URLs. If a non-URL dependency _does_ introduce a URL dependency, uv will reject the URL dependency during resolution. (Note that PyPI does not allow published packages to depend on URL dependencies; other registries may be more permissive.)

Second, if a constraint (`--constraint`) or override (`--override`) is defined using a direct URL dependency, and the constrained package has a direct URL dependency of its own, uv _may_ reject that transitive direct URL dependency during resolution, if the URL isn't referenced elsewhere in the set of input requirements.

If uv rejects a transitive URL dependency, the best course of action is to provide the URL dependency as a direct dependency in the relevant `pyproject.toml` or `requirement.in` file, as the above constraints do not apply to direct dependencies.

Virtual environments by default

`uv pip install` and `uv pip sync` are designed to work with virtual environments by default.

Specifically, uv will always install packages into the currently active virtual environment, or search for a virtual environment named `.venv` in the current directory or any parent directory (even if it is not activated).

This differs from `pip`, which will install packages into a global environment if no virtual environment is active, and will not search for inactive virtual environments.

In uv, you can install into non-virtual environments by providing a path to a Python executable via the `--python /path/to/python` option, or via the `--system` flag, which installs into the first Python interpreter found on the `PATH`, like `pip`.

In other words, uv inverts the default, requiring explicit opt-in to installing into the system Python, which can lead to breakages and other complications, and should only be done in limited circumstances.

For more, see ["Using arbitrary Python environments"](../environments/#using-arbitrary-python-environments).

Resolution strategy

For a given set of dependency specifiers, it's often the case that there is no single "correct" set of packages to install. Instead, there are many valid sets of packages that satisfy the specifiers.

Neither `pip` nor uv make any guarantees about the _exact_ set of packages

that will be installed; only that the resolution will be consistent, deterministic, and compliant with the specifiers. As such, in some cases, `pip` and uv will yield different resolutions; however, both resolutions _should_ be equally valid.

For example, consider:

requirements.in

```
starlette
fastapi
```

At time of writing, the most recent `starlette` version is `0.37.2`, and the most recent `fastapi` version is `0.110.0`. However, `fastapi==0.110.0` also depends on `starlette`, and introduces an upper bound: `starlette>=0.36.3,<0.37.0`.

If a resolver prioritizes including the most recent version of `starlette`, it would need to use an older version of `fastapi` that excludes the upper bound on `starlette`. In practice, this requires falling back to `fastapi==0.1.17`:

requirements.txt

```
    # This file was autogenerated by uv via the following command:
    #    uv pip compile requirements.in
    annotated-types==0.6.0
        # via pydantic
    anyio==4.3.0
        # via starlette
    fastapi==0.1.17
    idna==3.6
        # via anyio
    pydantic==2.6.3
        # via fastapi
    pydantic-core==2.16.3
        # via pydantic
    sniffio==1.3.1
        # via anyio
    starlette==0.37.2
        # via fastapi
    typing-extensions==4.10.0
        # via
        #   pydantic
        #   pydantic-core
```

Alternatively, if a resolver prioritizes including the most recent version of `fastapi`, it would need to use an older version of `starlette` that satisfies the upper bound. In practice, this requires falling back to

`starlette==0.36.3`:

requirements.txt

```
# This file was autogenerated by uv via the following command:
#    uv pip compile requirements.in
annotated-types==0.6.0
    # via pydantic
anyio==4.3.0
    # via starlette
fastapi==0.110.0
idna==3.6
    # via anyio
pydantic==2.6.3
    # via fastapi
pydantic-core==2.16.3
    # via pydantic
sniffio==1.3.1
    # via anyio
starlette==0.36.3
    # via fastapi
typing-extensions==4.10.0
    # via
    #   fastapi
    #   pydantic
```

```
    # pydantic-core
```

When uv resolutions differ from `pip` in undesirable ways, it's often a sign that the specifiers are too loose, and that the user should consider tightening them. For example, in the case of `starlette` and `fastapi`, the user could require `fastapi>=0.110.0`.

`pip check`

At present, `uv pip check` will surface the following diagnostics:

* A package has no `METADATA` file, or the `METADATA` file can't be parsed.
* A package has a `Requires-Python` that doesn't match the Python version of the running interpreter.
* A package has a dependency on a package that isn't installed.
* A package has a dependency on a package that's installed, but at an incompatible version.
* Multiple versions of a package are installed in the virtual environment.

In some cases, `uv pip check` will surface diagnostics that `pip check` does not, and vice versa. For example, unlike `uv pip check`, `pip check` will _not_ warn when multiple versions of a package are installed in the current environment.

`--user` and the `user` install scheme

uv does not support the `--user` flag, which installs packages based on the

`user` install scheme. Instead, we recommend the use of virtual environments to isolate package installations.

Additionally, pip will fall back to the `user` install scheme if it detects that the user does not have write permissions to the target directory, as is the case on some systems when installing into the system Python. uv does not implement any such fallback.

For more, see [#2077](https://github.com/astral-sh/uv/issues/2077).

`--only-binary` enforcement

The `--only-binary` argument is used to restrict installation to pre-built binary distributions. When `--only-binary :all:` is provided, both pip and uv will refuse to build source distributions from PyPI and other registries.

However, when a dependency is provided as a direct URL (e.g., `uv pip install https://...`), pip does _not_ enforce `--only-binary`, and will build source distributions for all such packages.

uv, meanwhile, _does_ enforce `--only-binary` for direct URL dependencies, with one exception: given `uv pip install https://... --only-binary flask`, uv _will_ build the source distribution at the given URL if it cannot infer the package name ahead of time, since uv can't determine whether the package is "allowed" in such cases without building its metadata.

Both pip and uv allow editables requirements to be built and installed even

when `--only-binary` is provided. For example, `uv pip install -e . --only-binary :all:` is allowed.

`--no-binary` enforcement

The `--no-binary` argument is used to restrict installation to source distributions. When `--no-binary` is provided, uv will refuse to install pre-built binary distributions, but _will_ reuse any binary distributions that are already present in the local cache.

Additionally, and in contrast to pip, uv's resolver will still read metadata from pre-built binary distributions when `--no-binary` is provided.

`manylinux_compatible` enforcement

[PEP 600](https://peps.python.org/pep-0600/#package-installers) describes a mechanism through which Python distributors can opt out of `manylinux` compatibility by defining a `manylinux_compatible` function on the `_manylinux` standard library module.

uv respects `manylinux_compatible`, but only tests against the current glibc version, and applies the return value of `manylinux_compatible` globally.

In other words, if `manylinux_compatible` returns `True`, uv will treat the system as `manylinux`-compatible; if it returns `False`, uv will treat the system as `manylinux`-incompatible, without calling `manylinux_compatible` for every glibc version.

This approach is not a complete implementation of the spec, but is compatible with common blanket `manylinux_compatible` implementations like [`no-manylinux`](https://pypi.org/project/no-manylinux/):

```
    from __future__ import annotations
    manylinux1_compatible = False
    manylinux2010_compatible = False
    manylinux2014_compatible = False


    def manylinux_compatible(*_, **__):  # PEP 600
        return False
```

Bytecode compilation

Unlike `pip`, uv does not compile `.py` files to `.pyc` files during installation by default (i.e., uv does not create or populate `__pycache__` directories). To enable bytecode compilation during installs, pass the `--compile-bytecode` flag to `uv pip install` or `uv pip sync`, or set the `UV_COMPILE_BYTECODE` environment variable to `1`.

Skipping bytecode compilation can be undesirable in workflows; for example, we recommend enabling bytecode compilation in [Docker

builds](../../guides/integration/docker/) to improve startup times (at the cost of increased build times).

As bytecode compilation suppresses various warnings issued by the Python interpreter, in rare cases you may seen `SyntaxWarning` or `DeprecationWarning` messages when running Python code that was installed with uv that do not appear when using `pip`. These are valid warnings, but are typically hidden by the bytecode compilation process, and can either be ignored, fixed upstream, or similarly suppressed by enabling bytecode compilation in uv.

Strictness and spec enforcement

uv tends to be stricter than `pip`, and will often reject packages that `pip` would install. For example, uv rejects HTML indexes with invalid URL fragments (see: [PEP 503](https://peps.python.org/pep-0503/)), while `pip` will ignore such fragments.

In some cases, uv implements lenient behavior for popular packages that are known to have specific spec compliance issues.

If uv rejects a package that `pip` would install due to a spec violation, the best course of action is to first attempt to install a newer version of the package; and, if that fails, to report the issue to the package maintainer.

`pip` command-line options and subcommands

uv does not support the complete set of `pip`'s command-line options and subcommands, although it does support a large subset.

Missing options and subcommands are prioritized based on user demand and the complexity of the implementation, and tend to be tracked in individual issues. For example:

* [`--trusted-host`](https://github.com/astral-sh/uv/issues/1339)
* [`--user`](https://github.com/astral-sh/uv/issues/2077)

If you encounter a missing option or subcommand, please search the issue tracker to see if it has already been reported, and if not, consider opening a new issue. Feel free to upvote any existing issues to convey your interest.

Registry authentication

uv does not support `pip`'s `auto` or `import` options for `--keyring-provider`. At present, only the `subprocess` option is supported.

Unlike `pip`, uv does not enable keyring authentication by default.

Unlike `pip`, uv does not wait until a request returns a HTTP 401 before searching for authentication. uv attaches authentication to all requests for hosts with credentials available.

`egg` support

uv does not support features that are considered legacy or deprecated in `pip`. For example, uv does not support `.egg`-style distributions.

However, uv does have partial support for (1) `.egg-info`-style distributions (which are occasionally found in Docker images and Conda environments) and (2) legacy editable `.egg-link`-style distributions.

Specifically, uv does not support installing new `.egg-info`\- or `.egg-link`-style distributions, but will respect any such existing distributions during resolution, list them with `uv pip list` and `uv pip freeze`, and uninstall them with `uv pip uninstall`.

Build constraints

When constraints are provided via `--constraint` (or `UV_CONSTRAINT`), uv will _not_ apply the constraints when resolving build dependencies (i.e., to build a source distribution). Instead, build constraints should be provided via the dedicated `--build-constraint` (or `UV_BUILD_CONSTRAINT`) setting.

pip, meanwhile, applies constraints to build dependencies when specified via `PIP_CONSTRAINT`, but not when provided via `--constraint` on the command line.

For example, to ensure that `setuptools 60.0.0` is used to build any packages with a build dependency on `setuptools`, use `--build-constraint`, rather than `--constraint`.

`pip compile` defaults

There are a few small but notable differences in the default behaviors of `pip compile` and `pip-tools`.

By default, uv does not write the compiled requirements to an output file. Instead, uv requires that the user specify an output file explicitly with the `-o` or `--output-file` option.

By default, uv strips extras when outputting the compiled requirements. In other words, uv defaults to `--strip-extras`, while `pip-compile` defaults to `--no-strip-extras`. `pip-compile` is scheduled to change this default in the next major release (v8.0.0), at which point both tools will default to `--strip-extras`. To retain extras with uv, pass the `--no-strip-extras` flag to `uv pip compile`.

By default, uv does not write any index URLs to the output file, while `pip-compile` outputs any `--index-url` or `--extra-index-url` that does not match the default (PyPI). To include index URLs in the output file, pass the `--emit-index-url` flag to `uv pip compile`. Unlike `pip-compile`, uv will include all index URLs when `--emit-index-url` is passed, including the default index URL.

`requires-python` enforcement

When evaluating `requires-python` ranges for dependencies, uv only considers lower bounds and ignores upper bounds entirely. For example, `>=3.8, <4` is

treated as `>=3.8`. Respecting upper bounds on `requires-python` often leads to formally correct but practically incorrect resolutions, as, e.g., resolvers will backtrack to the first published version that omits the upper bound (see: [`Requires-Python` upper limits](https://discuss.python.org/t/requires-python-upper-limits/12663)).

When evaluating Python versions against `requires-python` specifiers, uv truncates the candidate version to the major, minor, and patch components, ignoring (e.g.) pre-release and post-release identifiers.

For example, a project that declares `requires-python: >=3.13` will accept Python 3.13.0b1. While 3.13.0b1 is not strictly greater than 3.13, it is greater than 3.13 when the pre-release identifier is omitted.

While this is not strictly compliant with [PEP 440](https://peps.python.org/pep-0440/), it _is_ consistent with [pip](https://github.com/pypa/pip/blob/24.1.1/src/pip/_internal/resolution/resolvelib/candidates.py#L540).

Package priority

There are usually many possible solutions given a set of requirements, and a resolver must choose between them. uv's resolver and pip's resolver have a different set of package priorities. While both resolvers use the user-provided order as one of their priorities, pip has additional [priorities](https://pip.pypa.io/en/stable/topics/more-dependency-resolution/#the-resolver-algorithm) that uv does not have. Hence, uv is more

likely to be affected by a change in user order than pip is.

For example, `uv pip install foo bar` prioritizes newer versions of `foo` over `bar` and could result in a different resolution than `uv pip install bar foo`. Similarly, this behavior applies to the ordering of requirements in input files for `uv pip compile`.

January 28, 2025

Back to top [Previous Locking environments](../compile/) [Next Index](../../reference/)