pip install

Usage

Unix/macOS Windows

```
python -m pip install [options] <requirement specifier> [package-index-options] ...
python -m pip install [options] -r <requirements file> [package-index-options] ...
python -m pip install [options] [-e] <vcs project url> ...
python -m pip install [options] [-e] <local project path> ...
python -m pip install [options] <archive url/path> ...
```

Description

Install packages from:

- PyPI (and other indexes) using requirement specifiers.
- VCS project urls.
- Local project directories.
- Local or remote source archives.

pip also supports installing from “requirements files”, which provide an easy way to specify a whole environment to be installed.

Overview

pip install has several stages:

1. Identify the base requirements. The user supplied arguments are processed here.
2. Resolve dependencies. What will be installed is determined here.
3. Build wheels. All the dependencies that can be are built into wheels.
4. Install the packages (and uninstall anything being upgraded/replaced).

Note that `pip install` prefers to leave the installed version as-is unless `--upgrade` is specified.

Skip to content

Argument Handling

When looking at the items to be installed, pip checks what type of item each is, in the following order:

1. Project or archive URL.
2. Local directory (which must contain a `pyproject.toml` or `setup.py`, otherwise pip will report an error).
3. Local file (a sdist or wheel format archive, following the naming conventions for those formats).
4. A version specifier.

Each item identified is added to the set of requirements to be satisfied by the install.

Working Out the Name and Version

For each candidate item, pip needs to know the project name and version. For wheels (identified by the `.whl` file extension) this can be obtained from the filename, as per the Wheel spec. For local directories, or explicitly specified sdist files, the `setup.py egg_info` command is used to determine the project metadata. For sdists located via an index, the filename is parsed for the name and project version (this is in theory slightly less reliable than using the `egg_info` command, but avoids downloading and processing unnecessary numbers of files).

Any URL may use the `#egg=name` syntax (see VCS Support) to explicitly state the project name.

Satisfying Requirements

Once pip has the set of requirements to satisfy, it chooses which version of each requirement to install using the simple rule that the latest version that satisfies the given constraints will be installed (but see here for an exception regarding pre-release versions). Where more than one source of the chosen version is available, it is assumed that any source is acceptable (as otherwise the versions would differ).

Obtaining information about what was installed

The install command has a `--report` option that will generate a JSON report of what pip has installed. In combination with the `--dry-run` and `--ignore-installed` it can be used to *resolve* a set of requirements without actually installing them.

The report can be written to a file, or to standard output (using `--report -` in combination with `--quiet`).

The format of the JSON report is described in Installation Report.

Skip to content

Installation Order

Note

This section is only about installation order of runtime dependencies, and does not apply to build dependencies (those are specified using the [build-system] table).

As of v6.1.0, pip installs dependencies before their dependents, i.e. in “topological order.” This is the only commitment pip currently makes related to order. While it may be coincidentally true that pip will install things in the order of the install arguments or in the order of the items in a requirements file, this is not a promise.

In the event of a dependency cycle (aka “circular dependency”), the current implementation (which might possibly change later) has it such that the first encountered member of the cycle is installed last.

For instance, if quux depends on foo which depends on bar which depends on baz, which depends on foo:

Unix/macOS Windows

```
$ python -m pip install quux
...
Installing collected packages baz, bar, foo, quux

$ python -m pip install bar
...
Installing collected packages foo, baz, bar
```

Prior to v6.1.0, pip made no commitments about install order.

The decision to install topologically is based on the principle that installations should proceed in a way that leaves the environment usable at each step. This has two main practical benefits:

1. Concurrent use of the environment during the install is more likely to work.
2. A failed install is less likely to leave a broken environment. Although pip would like to support failure rollbacks eventually, in the mean time, this is an improvement.

Skip to content

Although the new install order is not intended to replace (and does not replace) the use of `setup_requires` to declare build dependencies, it may help certain projects install from sdist (that might previously fail) that fit the following profile:

1. They have build dependencies that are also declared as install dependencies using `install_requires`.
2. `python setup.py egg_info` works without their build dependencies being installed.
3. For whatever reason, they don't or won't declare their build dependencies using `setup_requires`.

Requirements File Format

This section has been moved to Requirements File Format.

Requirement Specifiers

This section has been moved to Requirement Specifiers.

Per-requirement Overrides

This is now covered in Requirements File Format.

Pre-release Versions

Starting with v1.4, pip will only install stable versions as specified by pre-releases by default. If a version cannot be parsed as a compliant version then it is assumed to be a pre-release.

If a Requirement specifier includes a pre-release or development version (e.g. `>=0.0.dev0`) then pip will allow pre-release and development versions for that requirement. This does not include the != flag.

The `pip install` command also supports a --pre flag that enables installation of pre-releases and development releases.

VCS Support

This is now covered in VCS Support.

Finding Packages

pip searches for packages on PyPI using the HTTP simple interface, which is documented here and there.

pip offers a number of package index options for modifying how packages are found.

pip looks for packages in a number of places: on PyPI (or the index given as `--index-url`, if not

Skip to content

`-index`), in the local filesystem, and in any additional repositories specified via `--extra-index-url`. There is no priority in the locations that are searched. Rather they are all checked, and the "best" match for the requirements (in terms of version number - see the specification for details) is selected.

See the pip install Examples.

SSL Certificate Verification

This is now covered in HTTPS Certificates.

Caching

This is now covered in Caching.

Wheel Cache

This is now covered in Caching.

Hash checking mode

This is now covered in Secure installs.

Local Project Installs

This is now covered in Local project installs.

Editable installs

This is now covered in Local project installs.

Build System Interface

This is now covered in Build System Interface.

Options

-r, --requirement <file>

Install from the given requirements file. This option can be used multiple times.

(environment variable: `PIP_REQUIREMENT`)

-c, --constraint <file>

Constrain versions using the given constraints file. This option can be used multiple times.

(environment variable: `PIP_CONSTRAINT`)

--no-deps

Don’t install package dependencies.

` variable: `PIP_NO_DEPS`, `PIP_NO_DEPENDENCIES`)

Skip to content

`--pre`

Include pre-release and development versions. By default, pip only finds stable versions.

(environment variable: `PIP_PRE`)

`-e, --editable` `<path/url>`

Install a project in editable mode (i.e. setuptools “develop mode”) from a local project path or a VCS url.

(environment variable: `PIP_EDITABLE`)

`--dry-run`

Don’t actually install anything, just print what would be. Can be used in combination with --ignore-installed to ‘resolve’ the requirements.

(environment variable: `PIP_DRY_RUN`)

`-t, --target` `<dir>`

Install packages into <dir>. By default this will not replace existing files/folders in <dir>. Use --upgrade to replace existing packages in <dir> with new versions.

(environment variable: `PIP_TARGET`)

`--platform` `<platform>`

Only use wheels compatible with <platform>. Defaults to the platform of the running system. Use this option multiple times to specify multiple platforms supported by the target interpreter.

(environment variable: `PIP_PLATFORM`)

`--python-version` `<python_version>`

The Python interpreter version to use for wheel and “Requires-Python” compatibility checks. Defaults to a version derived from the running interpreter. The version can be specified using up to three dot-separated integers (e.g. “3” for 3.0.0, “3.7” for 3.7.0, or “3.7.3”). A major-minor version can also be given as a string without dots (e.g. “37” for 3.7.0).

(environment variable: `PIP_PYTHON_VERSION`)

`--implementation` `<implementation>`

Only use wheels compatible with Python implementation <implementation>, e.g. ‘pp’, ‘jy’, ‘cp’, or ‘ip’. If not specified, then the current interpreter implementation is used. Use ‘py’ to force

Skip to content on-agnostic wheels.

(environment variable: `PIP_IMPLEMENTATION`)

--abi `<abi>`

Only use wheels compatible with Python abi <abi>, e.g. 'pypy_41'. If not specified, then the current interpreter abi tag is used. Use this option multiple times to specify multiple abis supported by the target interpreter. Generally you will need to specify --implementation, --platform, and --python-version when using this option.

(environment variable: `PIP_ABI`)

--user

Install to the Python user install directory for your platform. Typically ~/.local/, or %APPDATA%Python on Windows. (See the Python documentation for site.USER_BASE for full details.)

(environment variable: `PIP_USER`)

--root `<dir>`

Install everything relative to this alternate root directory.

(environment variable: `PIP_ROOT`)

--prefix `<dir>`

Installation prefix where lib, bin and other top-level folders are placed. Note that the resulting installation may contain scripts and other resources which reference the Python interpreter of pip, and not that of `--prefix`. See also the `--python` option if the intention is to install packages into another (possibly pip-free) environment.

(environment variable: `PIP_PREFIX`)

--src `<dir>`

Directory to check out editable projects into. The default in a virtualenv is "<venv path>/src". The default for global installs is "<current dir>/src".

(environment variable: `PIP_SRC`, `PIP_SOURCE`, `PIP_SOURCE_DIR`, `PIP_SOURCE_DIRECTORY`)

-U, --upgrade

Upgrade all specified packages to the newest available version. The handling of dependencies depends on the upgrade-strategy used.

(environment variable: `PIP_UPGRADE`)

Skip to content

`--upgrade-strategy <upgrade_strategy>`

Determines how dependency upgrading should be handled [default: only-if-needed]. “eager” - dependencies are upgraded regardless of whether the currently installed version satisfies the requirements of the upgraded package(s). “only-if-needed” - are upgraded only when they do not satisfy the requirements of the upgraded package(s).

(environment variable: `PIP_UPGRADE_STRATEGY`)

`--force-reinstall`

Reinstall all packages even if they are already up-to-date.

(environment variable: `PIP_FORCE_REINSTALL`)

`-I, --ignore-installed`

Ignore the installed packages, overwriting them. This can break your system if the existing package is of a different version or was installed with a different package manager!

(environment variable: `PIP_IGNORE_INSTALLED`)

`--ignore-requires-python`

Ignore the Requires-Python information.

(environment variable: `PIP_IGNORE_REQUIRES_PYTHON`)

`--no-build-isolation`

Disable isolation when building a modern source distribution. Build dependencies specified by PEP 518 must be already installed if this option is used.

(environment variable: `PIP_NO_BUILD_ISOLATION`)

`--use-pep517`

Use PEP 517 for building source distributions (use --no-use-pep517 to force legacy behaviour).

(environment variable: `PIP_USE_PEP517`)

`--check-build-dependencies`

Check the build dependencies when PEP517 is used.

(environment variable: `PIP_CHECK_BUILD_DEPENDENCIES`)

`--break-system-packages`

Skip to content ıodify an EXTERNALLY-MANAGED Python installation

(environment variable: `PIP_BREAK_SYSTEM_PACKAGES`)

`-C, --config-settings` `<settings>`

Configuration settings to be passed to the PEP 517 build backend. Settings take the form KEY=VALUE. Use multiple --config-settings options to pass multiple keys to the backend.

(environment variable: `PIP_CONFIG_SETTINGS`)

`--global-option` `<options>`

Extra global options to be supplied to the setup.py call before the install or bdist_wheel command.

(environment variable: `PIP_GLOBAL_OPTION`)

`--compile`

Compile Python source files to bytecode

(environment variable: `PIP_COMPILE`)

`--no-compile`

Do not compile Python source files to bytecode

(environment variable: `PIP_NO_COMPILE`)

`--no-warn-script-location`

Do not warn when installing scripts outside PATH

(environment variable: `PIP_NO_WARN_SCRIPT_LOCATION`)

`--no-warn-conflicts`

Do not warn about broken dependencies

(environment variable: `PIP_NO_WARN_CONFLICTS`)

`--no-binary` `<format_control>`

Do not use binary packages. Can be supplied multiple times, and each time adds to the existing value. Accepts either ":all:" to disable all binary packages, ":none:" to empty the set (notice the colons), or one or more package names with commas between them (no colons). Note that some packages are tricky to compile and may fail to install when this option is used on them.

(environment variable: `PIP_NO_BINARY`)

Skip to content

`--only-binary <format_control>`

Do not use source packages. Can be supplied multiple times, and each time adds to the existing value. Accepts either “:all:” to disable all source packages, “:none:” to empty the set, or one or more package names with commas between them. Packages without binary distributions will fail to install when this option is used on them.

(environment variable: `PIP_ONLY_BINARY`)

`--prefer-binary`

Prefer binary packages over source packages, even if the source packages are newer.

(environment variable: `PIP_PREFER_BINARY`)

`--require-hashes`

Require a hash to check each requirement against, for repeatable installs. This option is implied when any package in a requirements file has a --hash option.

(environment variable: `PIP_REQUIRE_HASHES`)

`--progress-bar <progress_bar>`

Specify whether the progress bar should be used [on, off, raw] (default: on)

(environment variable: `PIP_PROGRESS_BAR`)

`--root-user-action <root_user_action>`

Action if pip is run as a root user [warn, ignore] (default: warn)

(environment variable: `PIP_ROOT_USER_ACTION`)

`--report <file>`

Generate a JSON file describing what pip did to install the provided requirements. Can be used in combination with --dry-run and --ignore-installed to ‘resolve’ the requirements. When - is used as file name it writes to stdout. When writing to stdout, please combine with the --quiet option to avoid mixing pip logging output with JSON output.

(environment variable: `PIP_REPORT`)

`--no-clean`

Don’t clean up build directories.

[illegible]t variable: `PIP_NO_CLEAN`)

Skip to content

`-i, --index-url <url>`

Base URL of the Python Package Index (default https://pypi.org/simple). This should point to a repository compliant with PEP 503 (the simple repository API) or a local directory laid out in the same format.

(environment variable: `PIP_INDEX_URL`, `PIP_PYPI_URL`)

`--extra-index-url <url>`

Extra URLs of package indexes to use in addition to --index-url. Should follow the same rules as --index-url.

(environment variable: `PIP_EXTRA_INDEX_URL`)

`--no-index`

Ignore package index (only looking at --find-links URLs instead).

(environment variable: `PIP_NO_INDEX`)

`-f, --find-links <url>`

If a URL or path to an html file, then parse for links to archives such as sdist (.tar.gz) or wheel (.whl) files. If a local path or file:// URL that's a directory, then look for archives in the directory listing. Links to VCS project URLs are not supported.

(environment variable: `PIP_FIND_LINKS`)

Skip to content

Examples

1. Install `SomePackage` and its dependencies from PyPI using Requirement Specifiers

   Unix/macOS Windows

   ```
   python -m pip install SomePackage            # latest version
   python -m pip install 'SomePackage==1.0.4'   # specific version
   python -m pip install 'SomePackage>=1.0.4'   # minimum version
   ```

2. Install a list of requirements specified in a file. See the Requirements files.

   Unix/macOS Windows

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

3. Upgrade an already installed `SomePackage` to the latest from PyPI.

   Unix/macOS Windows

   ```
   python -m pip install --upgrade SomePackage
   ```

4. Install a local project in “editable” mode. See the section on Editable Installs.

   Unix/macOS Windows

   ```
   python -m pip install -e .                # project in current directory
   python -m pip install -e path/to/project  # project in another directory
   ```

5. Install a project from VCS

   Unix/macOS Windows

   ```
   python -m pip install 'SomeProject@git+https://git.repo/some_pkg.git@1.3.1'
   ```

6. Install a project from VCS in “editable” mode. See the sections on VCS Support and Editable Installs.

   Unix/macOS Windows

Skip to content

```
python -m pip install -e 'git+https://git.repo/some_pkg.git#egg=SomePackage'          #
python -m pip install -e 'hg+https://hg.repo/some_pkg.git#egg=SomePackage'            #
python -m pip install -e 'svn+svn://svn.repo/some_pkg/trunk/#egg=SomePackage'         #
python -m pip install -e 'git+https://git.repo/some_pkg.git@feature#egg=SomePackage'  #
python -m pip install -e 'git+https://git.repo/some_repo.git#egg=subdir&subdirectory=su
```

7. Install a package with extras, i.e., optional dependencies (specification).

Unix/macOS Windows

```
python -m pip install 'SomePackage[PDF]'
python -m pip install 'SomePackage[PDF] @ git+https://git.repo/SomePackage@main#subdire
python -m pip install '.[PDF]'  # project in current directory
python -m pip install 'SomePackage[PDF]==3.0'
python -m pip install 'SomePackage[PDF,EPUB]'  # multiple extras
```

8. Install a particular source archive file.

Unix/macOS Windows

```
python -m pip install './downloads/SomePackage-1.0.4.tar.gz'
python -m pip install 'http://my.package.repo/SomePackage-1.0.4.zip'
```

9. Install a particular source archive file following direct references (specification).

Unix/macOS Windows

```
python -m pip install 'SomeProject@http://my.package.repo/SomeProject-1.2.3-py33-none-a
python -m pip install 'SomeProject @ http://my.package.repo/SomeProject-1.2.3-py33-none
python -m pip install 'SomeProject@http://my.package.repo/1.2.3.tar.gz'
```

10. Install from alternative package repositories.

    Install from a different index, and not PyPI

Unix/macOS Windows

```
python -m pip install --index-url http://my.package.repo/simple/ SomePackage
```

Install from a local flat directory containing archives (and don't scan indexes):

Unix/macOS Windows

Skip to content

```
          .p install --no-index --find-links=file:///local/dir/ SomePackage
python -m pip install --no-index --find-links=/local/dir/ SomePackage
python -m pip install --no-index --find-links=relative/dir/ SomePackage
```

Search an additional index during install, in addition to PyPI

> ⚠ Warning
>
> Using this option to search for packages which are not in the main repository (such as private packages) is unsafe, per a security vulnerability called dependency confusion: an attacker can claim the package on the public repository in a way that will ensure it gets chosen over the private package.

Unix/macOS **Windows**

```
python -m pip install --extra-index-url http://my.package.repo/simple SomePackage
```

11. Find pre-release and development versions, in addition to stable versions. By default, pip only finds stable versions.

Unix/macOS **Windows**

```
python -m pip install --pre SomePackage
```

12. Install packages from source.

Do not use any binary packages

Unix/macOS **Windows**

```
python -m pip install SomePackage1 SomePackage2 --no-binary :all:
```

Specify `SomePackage1` to be installed from source:

Unix/macOS **Windows**

```
python -m pip install SomePackage1 SomePackage2 --no-binary SomePackage1
```