Dependency Resolution

pip is capable of determining and installing the dependencies of packages. The process of determining which version of a dependency to install is known as dependency resolution. This behaviour can be disabled by passing `--no-deps`. See pip install.

How it works ¶

When a user does a `pip install` (e.g. `pip install tea`) pip needs to work out the package's dependencies (e.g. `spoon`, `hot-water`, `tea-leaves` etc.) and what the versions of each of those dependencies it should install.

At the start of a `pip install` run, pip does not have all the dependency information of the requested packages. It needs to work out the dependencies of the requested packages, the dependencies of those dependencies, and so on. Over the course of the dependency resolution process, pip will need to download distribution files of the packages which are used to get the dependencies of a package.

Backtracking

Changed in version 20.3 pip's dependency resolver is now capable of backtracking.

During dependency resolution, pip needs to make assumptions about the package versions it needs to install and, later, check these assumptions were not incorrect. When pip finds that an assumption it made earlier is incorrect, it has to backtrack, which means also discarding some of the work that has already been done, and going back to choose another path.

This can look like pip downloading multiple versions of the same package, since pip explicitly presents each download to the user. The backtracking of choices made during this step is not unexpected behaviour or a bug. It is part of how dependency resolution for Python packages works.

Example

The user requests `pip install tea`. The package `tea` declares a dependency on `hot-water`, `spoon`, `cup`, amongst others.

pip starts by picking the most recent version of `tea` and gets the list of dependencies of that version of `tea`. It will then repeat the process for those packages, picking the most recent version of `spoon` and then `cup`. Now, pip notices that the version of `cup` it has chosen is not compatible with the version of `spoon` it has chosen. Thus, pip will "go back" (backtrack) and try to use another version of `cup`. If it is successful, it will continue onto the next package (like `sugar`). Otherwise, it will continue to backtrack on `cup` until it finds a version of `cup` that is compatible with all the other packages.

This can look like:

```
$ pip install tea
Collecting tea
  Downloading tea-2.5.0-py2.py3-none-any.whl (340 kB)
     |████████████████████████████████| 340 kB 28.4 MB/s
Collecting spoon>=2.25.0
  Downloading spoon-2.27.0-py2.py3-none-any.whl (312 kB)
     |████████████████████████████████| 312 kB 19.2 MB/s
Collecting cup<2.0.0
  Downloading cup-1.0.0-py2.py3-none-any.whl (297 kB)
     |████████████████████████████████| 297 kB 29.3 MB/s
INFO: pip is looking at multiple versions of tea to determine
which version is compatible with other requirements.
This could take a while.
  Downloading cup-2.0.0-py2.py3-none-any.whl (295 kB)
     |████████████████████████████████| 295 kB 29.4 MB/s
  Downloading cup-1.0.0-py2.py3-none-any.whl (294 kB)
     |████████████████████████████████| 294 kB 24.4 MB/s
  Downloading cup-1.0.1-py2.py3-none-any.whl (294 kB)
     |████████████████████████████████| 294 kB 21.3 MB/s
  Downloading cup-2.0.1-py2.py3-none-any.whl (295 kB)
     |████████████████████████████████| 295 kB 26.2 MB/s
  Downloading cup-2.1.0-py2.py3-none-any.whl (293 kB)
     |████████████████████████████████| 293 kB 22.1 MB/s
  Downloading cup-2.1.1-py2.py3-none-any.whl (302 kB)
     |████████████████████████████████| 302 kB 29.3 MB/s
  Downloading cup-2.1.2-py2.py3-none-any.whl (370 kB)
     |████████████████████████████████| 370 kB 27.1 MB/s
  Downloading cup-2.1.3-py2.py3-none-any.whl (302 kB)
     |████████████████████████████████| 302 kB 30.4 MB/s
INFO: pip is looking at multiple versions of hot-water to determine
which version is compatible with other requirements.
This could take a while.
  Downloading cup-2.0.0-py2.py3-none-any.whl (370 kB)
     |████████████████████████████████| 370 kB 21.4 MB/s
  Downloading cup-2.0.0-py2.py3-none-any.whl (312 kB)
     |████████████████████████████████| 312 kB 21.3 MB/s
```

These multiple `Downloading cup (version)` lines show that pip is backtracking choices it is making during dependency resolution.

If pip starts backtracking during dependency resolution, it does not know how many choices it will reconsider, and how much computation would be needed.

For the user, this means it can take a long time to complete when pip starts backtracking. In the case where a package has a lot of versions, arriving at a good candidate can take a lot of time. The amount of time depends on the package size, the number of versions pip must try, and various other factors.

Backtracking reduces the risk that installing a new package will accidentally break an existing installed package, and so reduces the risk that your environment gets messed up. To do this, pip has to do more work, to find out which version of a package is a good candidate to install.

Possible ways to reduce backtracking

There is no one-size-fits-all answer to situations where pip is backtracking excessively during dependency resolution. There are ways to reduce the degree to which pip might backtrack though. Nearly all of these approaches require some amount of trial and error.

Allow pip to complete its backtracking

In most cases, pip will complete the backtracking process successfully. This could take a very long time to complete, so this may not be your preferred option.

However, it is a possible that pip will not be able to find a set of compatible versions. For this, pip will try every possible combination that it needs to and determine that there is no compatible set.

If you'd prefer not to wait, you can interrupt pip (Ctrl+C) and the strategies listed below.

Reduce the number of versions pip is trying to use

It is usually a good idea to add constraints the package(s) that pip is backtracking on (e.g. in the above example - `cup`).

You could try something like:

Linux MacOS **Windows**

```
C:\> py -m pip install tea "cup >= 3.13"
```

This will reduce the number of versions of `cup` it tries, and possibly reduce the time pip takes to install.

There is a possibility that the addition constraint is incorrect. When this happens, the reduced search space makes it easier for pip to more quickly determine what caused the conflict and present that to the user. It could also result in pip backtracking on a different package due to some other conflict.

Use constraint files or lockfiles

This option is a progression of the previous section. It requires users to know how to inspect:

- the packages they're trying to install
- the package release frequency and compatibility policies
- their release notes and changelogs from past versions

During deployment, you can create a lockfile stating the exact package and version number for each dependency of that package. You can create this with pip-tools.

This means the "work" is done once during development process, and thus will avoid performing dependency resolution during deployment.

Dealing with dependency conflicts

This section provides practical suggestions to pip users who encounter a `ResolutionImpossible` error, where pip cannot install their specified packages due to conflicting dependencies.

Understanding your error message

When you get a `ResolutionImpossible` error, you might see something like this:

Linux MacOS **Windows**

```
C:\> py -m pip install package_coffee==0.44.1 package_tea==4.3.0
[regular pip output]
ERROR: Cannot install package_coffee==0.44.1 and package_tea==4.3.0 because these package versions h

The conflict is caused by:
    package_coffee 0.44.1 depends on package_water<3.0.0,>=2.4.2
    package_tea 4.3.0 depends on package_water==2.3.1

To fix this you could try to:
1. loosen the range of package versions you've specified
2. remove package versions to allow pip attempt to solve the dependency conflict

ERROR: ResolutionImpossible: for help visit https://pip.pypa.io/en/latest/user_guide/#fixing-conflicting-dependencies
```

In this example, pip cannot install the packages you have requested, because they each depend on different versions of the same package (`package_water`)

- `package_coffee` version `0.44.1` depends on a version of `package_water` that is less than `3.0.0` but greater than or equal to `2.4.2`
- `package_tea` version `4.3.0` depends on version `2.3.1` of `package_water`

Sometimes these messages are straightforward to read, because they use commonly understood comparison operators to specify the required version (e.g. `<` or `>`).

However, Python packaging also supports some more complex ways for specifying package versions (e.g. `~=` or `*`):

Operator	Description	Example
`>`	Any version greater than the specified version.	`>3.1.1`: any version greater than `3.1.1`.
`<`	Any version less than the specified version.	`<3.1`: any version less than `3.1`.
`<=`	Any version less than or equal to the specified version.	`<=3.1`: any version less than or equal to `3.1`.
`!=`	Any version greater than or equal to the specified version.	`!=3.1`: any version greater than or equal to `3.1`.
`==`	Exactly the specified version.	`==3.1`: only version `3.1`.
`!=`	Any version not equal to the specified version.	`!=3.1`: any version other than `3.1`.
`~=`	Any compatible[1] version.	`~=3.1`: any version compatible[1] with `3.1`.
`*`	Can be used at the end of a version number to represent *all*.	`==3.1.*`: any version that starts with `3.1`.

[1] Compatible versions are higher versions that only differ in the final segment. `~=3.1.2` is equivalent to `>=3.1.2, ==3.1.*`. `~=3.1` is equivalent to `>=3.1, ==3.*`.

The detailed specification of supported comparison operators can be found in PEP 440.

Possible solutions

The solution to your error will depend on your individual use case. Here are some things to try:

Audit your top level requirements

As a first step, it is useful to audit your project and remove any unnecessary or out of date requirements (e.g. from your `setup.py` or `requirements.txt` files). Removing these can significantly reduce the complexity of your dependency tree, thereby reducing opportunities for conflicts to occur.

Loosen your top level requirements

Sometimes the packages that you have asked pip to install are incompatible because you have been too strict when you specified the package version.

In our first example both `package_coffee` and `package_tea` have been *pinned* to use specific versions (`package_coffee==0.44.1` `package_tea==4.3.0`).

To find a version of both `package_coffee` and `package_tea` that depend on the same version of `package_water`, you might consider:

- Loosening the range of packages that you are prepared to install (e.g. `pip install "package_coffee>0.44" "package_tea>4.0.0"`)
- Asking pip to install *any* version of `package_coffee` and `package_tea` by removing the version specifiers altogether (e.g. `pip install package_coffee package_tea`)

In the second case, pip will automatically find a version of both `package_coffee` and `package_tea` that depend on the same version of `package_water`, installing:

- `package_coffee 0.44.1`, which depends on `package_water 2.6.1`
- `package_tea 4.4.3` which *also* depends on `package_water 2.6.1`

If you want to prioritize one package over another, you can add version specifiers to *only* the more important package:

Linux MacOS **Windows**

```
C:\> py -m pip install package_coffee==0.44.1 package_tea
```

This will result in:

- `package_coffee 0.44.1`, which depends on `package_water 2.6.1`
- `package_tea 4.4.3` which *also* depends on `package_water 2.6.1`

Now that you have resolved the issue, you can repin the *compatible* package versions as required.

Loosen the requirements of your dependencies

Assuming that you cannot resolve the conflict by loosening the version of the package you require (as above), you can try to fix the issue on your *dependency* by:

- Requesting that the package maintainers loosen *their* dependencies
- Forking the package and loosening the dependencies yourself

Warning

If you choose to fork the package yourself, you are opting out of any support provided by the package maintainers. Proceed at your own risk!

All requirements are appropriate, but a solution does not exist

Sometimes it's simply impossible to find a combination of package versions that do not conflict. Welcome to dependency hell.

In this situation, you could consider:

- Using an alternative package, if that is acceptable for your project. See Awesome Python for similar packages.
- Refactoring your project to reduce the number of dependencies (for example, by breaking up a monolithic code base into smaller pieces).

Handling Resolution Too Deep Errors

Sometimes pip's dependency resolver may exceed its search depth and terminate with a `ResolutionTooDeep` error exception. This typically occurs when the dependency graph is extremely complex or when there are too many package versions to evaluate.

To address this error, consider the following strategies:

Specify Reasonable Lower Bounds

By setting a higher lower bound for your dependencies, you narrow the search space. This excludes older versions that might trigger excessive backtracking. For example:

Linux MacOS **Windows**

```
C:\> py -m pip install "package_coffee>=0.44.0" "package_tea>=4.0.0"
```

Use the `--upgrade` Flag

The `--upgrade` flag directs pip to ignore already installed versions and search for the latest versions that meet your requirements. This can help avoid unnecessary resolution paths:

Linux MacOS **Windows**

```
C:\> py -m pip install --upgrade package_coffee package_tea
```

Utilize Constraint Files

If you need to impose additional version restrictions on transitive dependencies (dependencies of dependencies), consider using a constraint file. A constraint file specifies version limits for packages that are indirectly required. For example:

```
# constraints.txt
indirect_dependency>=2.0.0
```

Then install your packages with:

Linux MacOS **Windows**

```
C:\> py -m pip install --constraint constraints.txt package_coffee package_tea
```

Use Upper Bounds Sparingly

Although upper bounds are generally discouraged because they can complicate dependency management, they may be necessary when certain versions are known to cause conflicts. Use them cautiously—for example:

Linux MacOS **Windows**

```
C:\> py -m pip install "package_coffee>=0.44.0,<1.0.0" "package_tea>=4.0.0"
```

Report ResolutionTooDeep Errors

If you encounter a `ResolutionTooDeep` error consider reporting it, to help the pip team have real world examples to test against, at the dedicated pip issue.

Getting help

If none of the suggestions above work for you, we recommend that you ask for help on:

- Python user Discourse
- Python user forums
- Python developers Slack channel
- Python IRC
- Stack Overflow

See "How do I ask a good question?" for tips on asking for help.

Unfortunately, **the pip team cannot provide support for individual dependency conflict errors**. Please only open a ticket on pip's issue tracker if you believe that your problem has exposed a bug in pip.