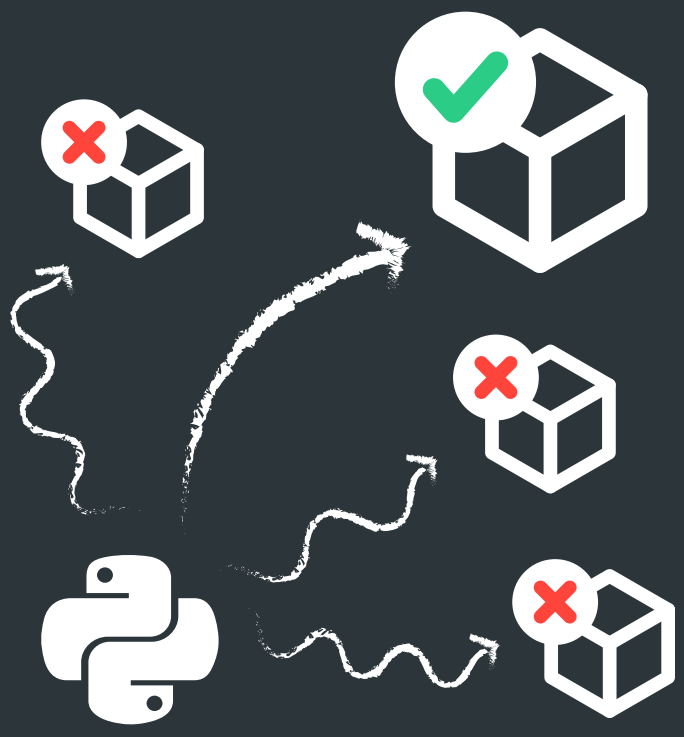
ActiveState

Python Package Management Guide for Enterprise Developers



Executive Summary

Package management continues to evolve, but traditional Python package managers are slow to catch up. Enterprise developers must deal with the consequences, including:

- Poor Environment Reproducibility - slightly different configurations across environments result in “works on my machine” issues and time wasted reproducing bugs, delaying time to market.
- Supply Chain Security - installing unsigned binaries with package managers is convenient, but risky. On the other hand, building packages from source for multiple operating systems is painful, especially if they require linked C libraries.
- Choosing the Right Packages/ Versions - how can you be sure you are always choosing the correct, approved open source components and versions required by your organization?
- Fixing Vulnerabilities - investigating vulnerabilities, patching/updating components and rebuilding environments are time and resource intensive, leaving less time for coding.

The ActiveState Platform addresses these issues, helping Python development teams in enterprises to:

- Create consistent, reproducible Python environments that can be deployed on a given system with a single command.
- Automatically build Python environments (including C libraries) from source code, resolving dependencies and packaging the result for all popular operating systems.
- Always know which components/versions are approved for use.
- Identify vulnerable components, fix them, and automatically rebuild secure environments quickly and easily.

If you wrestle with any of these issues, adopting the ActiveState Platform will allow you to spend more time coding and less time managing packages and environments. All of which means you’re more likely to complete your sprint deliverables on time.

The State of Python Package Management

As a Python programmer, you know that package management has been a work in progress for decades. Defined narrowly, package management is the ability to install, configure, upgrade and uninstall a package and its dependencies. In practice however, package management is more broadly concerned with managing the development environment created by installing multiple packages against a specific version of a programming language on a specific version of an Operating System (OS).

Modern package management is concerned with solving the dependency and environment management issues that arise from this combination of components, languages and OS's that original package managers were never designed to deal with, including:

- **Multiple Environments** - how can I work with multiple projects on my local system if each requires a different version of the language and/or different versions of packages?
- **Dependency Conflicts** - if one package requires version X of dependency Z, but another package requires version Y, how can the conflict be resolved?
- **Reproducibility** - how can I ensure that my project can be consistently deployed and run on other systems?

Package Management Solutions

While pip has long been the standard for installing and managing Python packages, it doesn't address key issues around environment and dependency management, such as creating and managing virtual environments or dependency resolution. Numerous solutions have been introduced to try and bridge the gap, including:

- **venv** - built into the Python standard library since v3.3, venv provides support for creating and managing virtual environments. You'll still need to use pip to install and manage packages, though.
- **virtualenv** - a more full-featured version of venv that ships as a third party package rather than in the Python core, but you'll need pip to manage the packages in your virtual environment.
- **pyenv** - adds the capability to manage installed versions of Python, alongside pyenv-virtualenv to create the actual virtual environments. Still need pip though.
- **pipenv** - effectively combines pip+venv into a single tool that lets you create virtual environments with whatever Python version you want, and then install and manage packages therein. It even builds a dependency graph for your project, flagging any issues and generating a Pipfile.lock that specifies every dependency and version in the project. You'll need to manually denote sub-dependencies, however, to ensure deterministic builds across platforms.
- **poetry** - combines environment control with dependency resolution, letting you manage your project, virtual environment and packages with a single tool. It also flags any dependency conflicts that arise. Unfortunately, it's still quite slow at resolving dependencies.

There are other package managers available, as well, such as conda for Anaconda Python, or apt and yum for Linux distributions. All of these alternative ecosystem package management tools have their pros and cons, but all are capable of managing packages and environments, as well as resolving dependencies. However, neither apt nor yum natively supports virtual environments, so you'll have to rely on one of the solutions listed above. And like poetry, conda's dependency resolution can be very slow.

Given the plethora of package management choices, seasoned Python developers often gravitate to their favourite sets of tools to help manage their environments and dependencies. However, when it comes to issues like dependency conflicts, fixing vulnerabilities, or troubleshooting "works on my machine" issues, today's package managers leave developers to manually implement their own workarounds.

Why Modern Python Package Management is Needed

Developers have worked around the shortcomings of Python package management tools for decades. They've also found creative ways to better create and manage each of their development and CI/CD environments, as well. However, as organizations have adopted agile software development processes, the pressure to deliver code faster has increased, making creative workarounds for common package and environment management shortcomings less and less viable.

ActiveState is no stranger to this pressure. We handcrafted our ActivePython distribution, which contains the latest version of Python and hundreds of popular packages, for decades. But depending on how drastically each version of Python, key packages, compilers, and patches changed between releases, the process could take weeks to months. To speed things up, we built the ActiveState Platform, which automates everything from dependency resolution to compiling linked C libraries to packaging the environment for Windows, Linux and macOS. The process now takes days, most of which is manual verification.

We've made the ActiveState Platform free for use so that Pythonistas can use it in combination with the ActiveState Platform's command line interface (CLI), the State Tool, to manage packages, environments and dependencies in a standard, reproducible way across Windows, Linux and macOS.

This section discusses a mix of traditional and evolving use cases that are either not addressed, or else poorly addressed by traditional package management solutions. The ActiveState Platform has been specifically designed to address these gaps.

Dependency Resolution & Conflicts

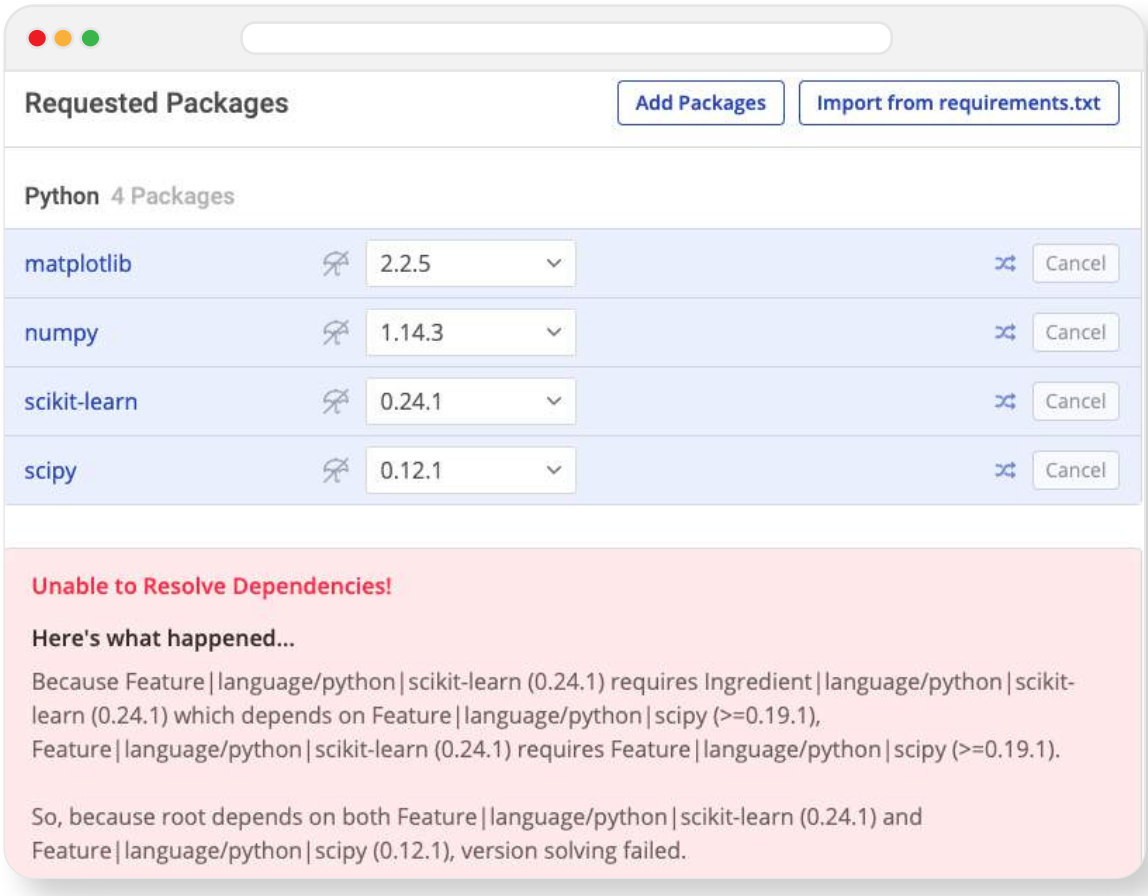
While some package management solutions will resolve dependencies, and even flag conflicts, most are incapable of resolving a dependency conflict. This results in developers manually testing various versions of packages to see if they can resolve the conflict themselves. In some cases, the solution is fairly straightforward, but in other cases developers waste time and resources in dependency hell.

The ActiveState Platform includes a dependency solver that not only resolves dependencies but also flags conflicts at multiple levels, including:

- Top level package dependencies
- Linked C/Fortran library dependencies
- Transitive dependencies (ie,. dependencies of dependencies)
- OS-level dependencies
- Shared libraries across multiple languages (ie., openSSL)

Most package managers only resolve dependencies at the language level (ie., package dependencies), which can lead to problems when your project is deployed on a different operating system, for example.

The ActiveState Platform is also unique in suggesting ways to solve a dependency conflict if it is unable to resolve the issue automatically. For example:



Simply following the instructions and changing the version of scipy to be >=0.19.1 (ie., by clicking on the dropdown and selecting a more recent version) would solve this conflict, eliminating dependency hell.

Supply Chain Security

Most developers prefer to install Python packages as binaries offered by the community, even though they haven't been signed. If you're careful to avoid typo-squatted packages, the risk of being compromised is typically acceptable in most organizations. After all, the alternative would be building every package and dependency from source for every operating system required, which can be a huge time sink over the life of a project. But what if there was a way to automate builds from source code?

Most pure Python packages are trivially easy to build from source. The problem arises when a package has a dependency on a linked C library. In this case, you'll need to source, create and maintain a build environment for your operating system, featuring:

- C compiler
- Fortran compiler (for certain data science packages)
- Build scripts
- Installer/packager

But you may also have to manually:

- Download Ingredients - data files containing the all the metadata (version, requirements, build/install commands, etc) for each component to be built
- Correct all metadata errors for each component to be built
- Patch any known vulnerabilities
- Resolve all conflicts between components and their dependencies, as well as any OS-level dependencies

Finally, you can now compile and resolve the inevitable issues.

By contrast, the ActiveState Platform provides a cloud-based build farm that will automatically build Python packages (as well as their dependencies) from source code in parallel, including any linked C libraries, ready to install on Windows and Linux. As a result, there's no need to maintain a local build environment, or even a need for language or operating system expertise. While not every package can be automatically built at any point in time (ie., newer versions may introduce a new build method), the ActiveState Platform often provides the simplest way for developers to build their environment from source.

Environment Reproducibility

Ensuring that your project can be deployed in a consistent, reproducible manner is a key goal for any software development team.

Python's requirements.txt and pipfile.lock files specify the exact versions of dependencies in your project. These files go a long way to ensuring consistent deployability, given a specific version of the programming language.

Of course, you need to remember to update these files before deploying, but there are other issues, as well:

- Requirements.txt and pipfile.lock files can get out of sync as different developers on a team update their development environment for their own purposes.
- Even when all development team members are using identical Docker images or Virtual Machines (VMs) for their development environments, you still need to ensure that they have been built with the latest Python environment, and that all development environments are up to date.
- When developing on one OS but deploying on a different OS, you may be missing OS-level dependencies.
- DevOps is often left to resolve multiple conflicting code check-ins, leading to CI/CD environments that differ from development environments

All of these issues are likely to lead to environment inconsistencies. The result can be bugs found in the CI/CD process that developers need to spend time reproducing by rebuilding the CI/CD environment where the bug was found.

The ActiveState Platform takes a different approach to ensuring environment consistency and reproducibility by:

- Automatically building your Python environment for Windows and Linux.
- Providing a central "source of truth" for the Python runtime environment that all developers (and DevOps) can pull to build their local environments, ensuring everyone is using the same environment, no matter their OS.
- Updates to the environment are made centrally, and can be updated locally with a single command.

Centrally managing and deploying your Python environments means that development and DevOps teams remain in sync, eliminating "works on my machine" issues.

Choosing the Right Packages

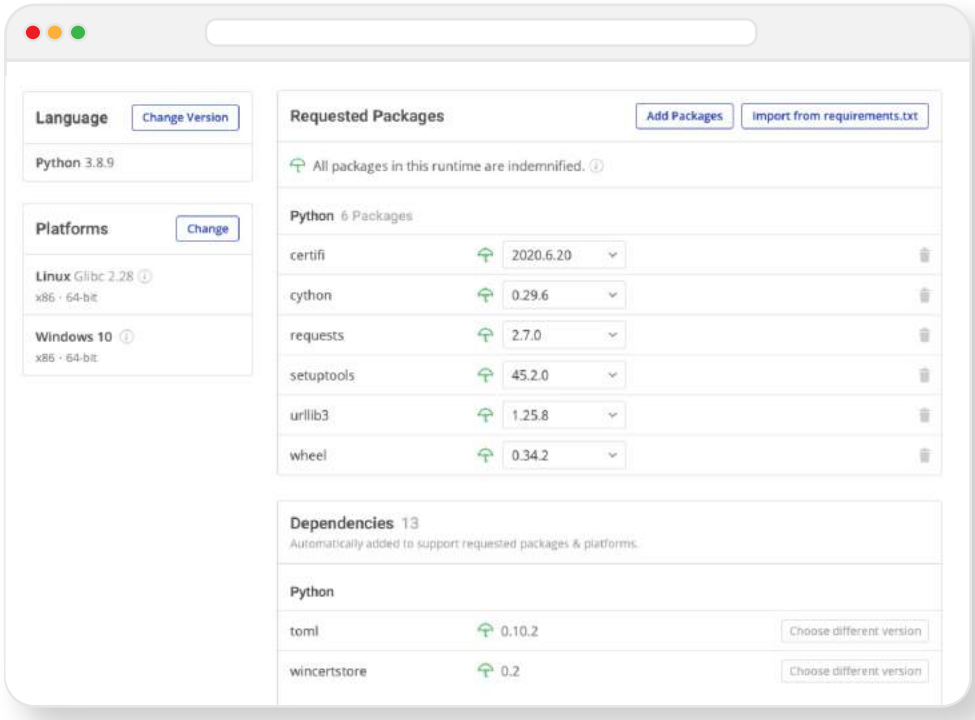
While not traditionally considered a package management issue, one of the most common problems we hear from customers is, "How can I ensure my developers are using only the approved set of components/component versions?"

Currently, most enterprises either:

- Use a local "walled garden" repository of packages, which can severely limit developers that want to experiment with new packages and/or new versions since they are simply not available. Typically, the approval process for adding new components to the walled garden can be quite lengthy and/or complex.
- Manually maintain a list of approved packages/versions, which can quickly get out of date.
- Use a third-party tool, which can help restrict use of packages that feature unapproved licenses and/or those that have a vulnerability. However, developers can still gain access to unapproved packages that meet these criteria.

The ActiveState Platform offers two approaches, depending on the enterprise's needs:

- **Unrestricted:** developers gain access to all packages in our catalog (which is updated from PyPI, GitHub, and other sources on a regular basis), but are provided guidance as to which packages/versions are appropriate for use through our indemnification offering. Indemnified packages are well maintained and feature licenses appropriate for creating commercial offerings.



- **Restricted:** developers are provided access to a walled garden in the form of a hosted repository that features only those Python packages that have been approved for use.

Finding and Fixing Vulnerabilities

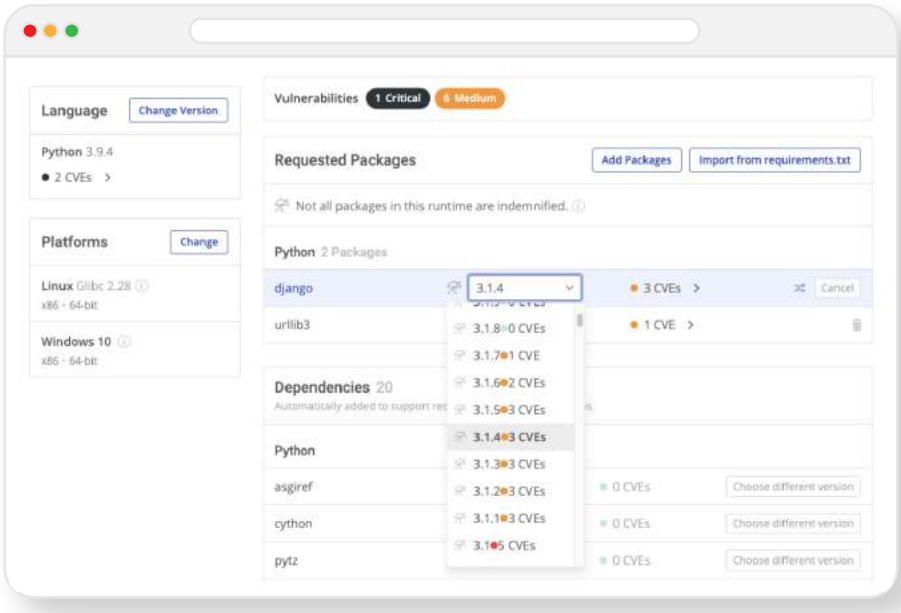
Again, although fixing package vulnerabilities are not typically considered a key requirement of package management, securing Python environments is a common use case that currently requires an inordinate amount of time and resources to:

- Discover the Common Vulnerabilities and Exposures (CVEs)
- Investigate the impact
- Patch/upgrade/downgrade the affected package
- Rebuild the environment
- Retest

While there are a number of solutions that can notify developers when a vulnerability is discovered, and even suggest a solution, all the other steps remain manual tasks. It's no wonder that the Mean Time To Remediation (MTTR) for these kinds of issues is weeks instead of days or hours.

The ActiveState Platform helps automate a number of the time consuming tasks associated with finding and fixing vulnerabilities, including:

- Automated status updates
- A list of non-vulnerable package versions that you can simply point and click to upgrade or downgrade to
- Visibly showing the cascading effect on all other dependencies when you upgrade/downgrade a package
- Automatically rebuilding and testing the environment for you



As a result, developers can spend less time finding and fixing vulnerabilities, and enterprises can dramatically decrease MTTR.

How ActiveState Can Help

The ActiveState Platform takes a holistic approach to package management, providing developers with a single, unified, cloud-based toolchain that works for both Python and Perl on Windows and Linux.

By adopting the ActiveState Platform, enterprise developers can benefit from many of the same advantages, including:

- Automated building of packages from source, including link C libraries without the need for a local build environment.
- Automated resolution of dependencies (or suggestions on how to manually resolve conflicts), ensuring that your environment always contains a set of known good dependencies that work together.
- Central management of a single source of truth for your environment that can be deployed with a single command to all development and CI/CD environments, ensuring consistent reproducibility.
- Automated installation of virtual Python environments on Windows or Linux without requiring prior setup.
- The ability to find, fix and automatically rebuild vulnerable environments, thereby enhancing security and dramatically reducing time and effort involved in resolving CVEs.
- Visually seeing which versions of which packages are approved for use, thereby taking the guesswork out of development.

Those that prefer to work from the command line can leverage the ActiveState Platform's CLI, the State Tool, which acts as a universal package manager for Python, and provides access to most of the features offered by the Platform.

Conclusions

ActiveState provides a unified cross-platform toolchain for modern Python package management. It can replace the complex and hard-to-maintain in-house solutions built from multiple package managers, environment management tools and other solutions. By adopting the ActiveState Platform, developers can:

- Increase the security of Python environments
- Improve the transparency of your open source supply chain
- Dramatically reduce package and environment management overhead
- Eliminate dependency hell
- Reduce "works on my machine" issues

Ultimately, developers that are willing to adopt the ActiveState Platform will spend less time wrestling with tooling and more time focused on doing what they do best: coding.

To try the ActiveState Platform for yourself, sign up for a free account at **https://platform.activestate.com**