**Package management system**

A package manager or package management system is a collection of software tools that automate the process of installing, upgrading, configuring, and removing computer programs for a [computer](https://en.wikipedia.org/wiki/Computer)'s [operating system](https://en.wikipedia.org/wiki/Operating_system) in a consistent manner.

A package manager deals with packages, distributions of software and data in [archive files](https://en.wikipedia.org/wiki/Archive_file). Packages contain [metadata](https://en.wikipedia.org/wiki/Metadata), such as the software's name, description of its purpose, version number, vendor, [checksum](https://en.wikipedia.org/wiki/Checksum), and a list of [dependencies](https://en.wikipedia.org/wiki/Coupling_(computer_programming)) necessary for the software to run properly. Upon installation, metadata is stored in a local package database. Package managers typically maintain a database of software dependencies and version information to prevent software mismatches and missing prerequisites. They work closely with [software repositories](https://en.wikipedia.org/wiki/Software_repository), [binary repository managers](https://en.wikipedia.org/wiki/Binary_repository_manager), and [app stores](https://en.wikipedia.org/wiki/App_store).

Package managers are designed to eliminate the need for manual installs and updates. This can be particularly useful for large enterprises whose operating systems are based on [Linux](https://en.wikipedia.org/wiki/Linux) and other [Unix-like](https://en.wikipedia.org/wiki/Unix-like) systems, typically consisting of hundreds or even tens of thousands of distinct software packages.

**What is PIP?**

Pip is a package management system used to install and manage software packages written in python. Many packages can be found in the default source for packages and their dependencies----Python Package Index (PyPI).

pip is a recursive acronym for Pip Installs Packages.

One major advantage of pip is the ease of its command-line interface, which makes installing Python software packages as easy as issuing one command:

Most importantly **pip** has a feature to manage full lists of packages and corresponding version numbers, possible through a "requirements" file. This permits the efficient re-creation of an entire group of packages in a separate environment (e.g. another computer) or virtual environment. This can be achieved with a properly formatted requirements file and the following command

### Why use Pip over easy-install?

All packages are downloaded before installation.

Partially-completed installation doesn’t occur as a result.

Care is taken to present useful output on the console.

The reasons for actions are kept track of.

For instance, if a package is being installed, pip keeps track of why that

package was required.

Error messages should be useful.

The code is relatively concise and cohesive, making it easier to use

programmatically.

Packages don’t have to be installed as egg archives, they can be installed flat.

Native support for other version control systems (Git, Mercurial and Bazaar)

Uninstallation of packages.

Simple to define fixed sets of requirements and reliably reproduce a set of

packages.

**SOFTWARE PACKAGE**

A software package is an [archive file](https://en.wikipedia.org/wiki/Archive_file) containing a computer program as well as necessary metadata for its deployment. The computer program can be in [source code](https://en.wikipedia.org/wiki/Source_code) that has to be compiled and built first.[[3]](https://en.wikipedia.org/wiki/Package_manager#cite_note-3) Package metadata include package description, package version, and dependencies (other packages that need to be installed beforehand).

Package managers are charged with the task of finding, installing, maintaining or uninstalling software packages upon the user's command. Typical functions of a package management system include:

* Working with [file archivers](https://en.wikipedia.org/wiki/File_archiver) to extract package archives
* Ensuring the integrity and authenticity of the package by verifying their [digital certificates](https://en.wikipedia.org/wiki/Digital_certificate) and [checksums](https://en.wikipedia.org/wiki/Checksum)
* Looking up, downloading, installing or updating existing software from a [software repository](https://en.wikipedia.org/wiki/Software_repository) or [app store](https://en.wikipedia.org/wiki/App_store)
* Grouping packages by function to reduce user confusion
* Managing dependencies to ensure a package is installed with all packages it requires, thus avoiding "[dependency hell](https://en.wikipedia.org/wiki/Dependency_hell)"

**#Requirements Files:**

Requirements files are the files containing a list of items to be installed using pip install. Each line of the R-file indicates something to install. Logically, a Requirements file is just a list of pip install arguments placed in a file. In practice, there are 4 common uses of Requirements files:

1: Requirements files are used to hold the result from pip freeze for the purpose of achieving repeatable installations. In this case, your requirement file contains a pinned version of everything that was installed when pip freeze was run.

2: Requirements files are used to force pip to properly resolve dependencies.

3: Requirements files are used to force pip to install an alternate version of a sub-dependency.

4: Requirements files are used to override a dependency with a local patch that lives in version control.

**#Constraints Files**

Constraints files are requirements files that only control which version of a requirement is installed, not whether it is installed or not.

There is one key difference: Including a package in a constraints file does not trigger installation of the package.

Constraints files are used for exactly the same reason as requirements files when you don't know exactly what things you want to install.

One way to ensure that the patched version is used consistently is to manually audit the dependencies of Everything you install, and if "hello world" is present, write a requirements file to use when installing that thing.

Constraints files offer a better way: write a single constraints file for your organisation and use that everywhere. If the thing being installed requires "hello world" to be installed, your fixed version specified in your constraints file will be used.

[Source Distributions vs Wheels](https://packaging.python.org/tutorials/installing-packages/#id18)

[pip](https://packaging.python.org/key_projects/#pip) can install from either [Source Distributions (sdist)](https://packaging.python.org/glossary/#term-source-distribution-or-sdist) or [Wheels](https://packaging.python.org/glossary/#term-wheel), but if both are present on PyPI, pip will prefer a compatible [wheel](https://packaging.python.org/glossary/#term-wheel).

[Wheels](https://packaging.python.org/glossary/#term-wheel) are a pre-built [distribution](https://packaging.python.org/glossary/#term-distribution-package) format that provides faster installation compared to [Source Distributions (sdist)](https://packaging.python.org/glossary/#term-source-distribution-or-sdist), especially when a project contains compiled extensions.

If [pip](https://packaging.python.org/key_projects/#pip) does not find a wheel to install, it will locally build a wheel and cache it for future installs, instead of rebuilding the source distribution in the future.

Build Wheel archives for your requirements and dependencies.

Wheel is a built-package format, and offers the advantage of not recompiling your software during every install.

## Benefits of packaging

While it’s possible to unpack [tarballs](https://the-hitchhikers-guide-to-packaging.readthedocs.io/en/latest/glossary.html" \l "term-tarball) and manually put them into your Python installation directories (see [Explicitly Including a Package Location](https://the-hitchhikers-guide-to-packaging.readthedocs.io/en/latest/introduction.html#explicitly-including-a-package-location)), using a package management system gives you some significant benefits. Here are some of the obvious ones:

* **Dependency management**

Often, the package you want to install requires that others be there. A package management system can automatically resolve dependencies and make your installation pain free and quick. This is one of the basic facilities offered by **distutils**. However, other extensions to **distutils** do a better job of installing dependencies. (see [Distribute](https://the-hitchhikers-guide-to-packaging.readthedocs.io/en/latest/installation.html#distribute-info))

* **Accounting**

Package managers can maintain lists of things installed and other metadata like the version installed etc. which makes is easy for the user to know what are the things his system has. (see [Pip Installs Python (Pip)](https://the-hitchhikers-guide-to-packaging.readthedocs.io/en/latest/installation.html#pip-info))

* **Uninstall**

Package managers can give you push button ways of removing a package from your environment. (see [Pip Installs Python (Pip)](https://the-hitchhikers-guide-to-packaging.readthedocs.io/en/latest/installation.html#pip-info))

* **Search**

Find packages by searching a [package index](https://the-hitchhikers-guide-to-packaging.readthedocs.io/en/latest/glossary.html#term-package-index) for specific terminology. (see [Pip Installs Python (Pip)](https://the-hitchhikers-guide-to-packaging.readthedocs.io/en/latest/installation.html#pip-info))

**TWINE**

twine is [a utility](https://pypi.org/project/twine/) for [publishing](https://packaging.python.org/tutorials/distributing-packages/) packages on [PyPI](https://pypi.org/).

The goal of twine is to improve PyPI interaction by improving security and testability.

The biggest reason to use twine is that it securely authenticates you to [PyPI](https://pypi.org/) over HTTPS using a verified connection, while python setup.py upload [only recently stopped using HTTP](https://bugs.python.org/issue12226) in Python 2.7.9+ and Python 3.2+. This means anytime you use python setup.py upload with an older Python version, you expose your username and password to being easily sniffed. Twine uses only verified TLS to upload to PyPI, protecting your credentials from theft.

Secondly, it allows you to precreate your distribution files. python setup.py upload only allows you to upload something that you’ve created in the same command invocation. This means that you cannot test the exact file you’re going to upload to PyPI to ensure that it works before uploading it.

Finally, twine allows you to pre-sign your files and pass the .asc files into the command line invocation (twine upload myproject-1.0.1.tar.gz myproject-1.0.1.tar.gz.asc). This enables you to be assured that you’re typing your gpgpassphrase into gpg itself and not anything else, since you will be the one directly executing gpg --detach-sign -a <filename>.

## Features

* Verified HTTPS connections
* Uploading doesn’t require executing setup.py
* Uploading files that have already been created, allowing testing of distributions before release
* Supports uploading any packaging format (including [wheels](https://packaging.python.org/glossary/#term-wheel))