pip is a Python software product developed by the Python Packaging Authority that can install Python packages. The packages are listed on PyPI servers. pip downloads the packages from these servers or optionally from other sources. It can be used directly by users or may be combined with an installer tool. Functionality of pip is exposed through a command line interface and the system is built on a layered architecture. pip is released in a single package and there are no product versions with different features. The source is hosted on GitHub and an automatic build and test suite is used.

Introduction to pip

pip is the preferred Python program by the Python community for handling any packaging related actions in any environment. The environment can be either an operating system or a virtual environment.

pip uses a Command Line Interface which exposes several commands. These commands will be explained later in this chapter. Each of the commands of pip focuses on a specific part of handling dependencies. There are command for (un)installing, listing, searching, and updating packages on the system.

The architecture of pip is interesting since it is a package dependency manager which is the default across all operating systems which Python supports. Not only is the software well-written and stable; its community support and availability also impact the future of the project. Maybe even more importantly, pip lies at the heart of many great Python projects, since a lot of developers consider it the best way to handle a projects dependencies. This combined shows a well structured case why the pip software is interesting to take a closer look at.

The Python Packaging Authority (PyPA) is the working group that officially maintains pip. The working group currently consists of 8 members who are the owners of the pip repository on Github. The PyPA states they pursue the architecture to be fast, reliable and "reasonably secure". They recognise that they can only pursue reasonable security, as they still have to provide backward compatibility. This requirement prevents PyPA to fix known insecure features [1]. PyPA provides development of pip throughout its life cycle, deploys new versions of pip and tests the system. Other developers can also contribute since pip is an open source project.

In this chapter the detailed working of pip will be explored and the context in which it operates will be shown. Also the architecture of pip and its module structure will be described. Next the development process will be made clear. Finally, the future of pip will be outlined.

Functionality of pip

As mentioned before, functionality of pip is exposed through a command line interface (CLI). Users (e.g. developers) and installers communicate with pip through the CLI, giving the software different commands. pip has eight different commands at this moment, the most important ones being `pip install` and `pip wheel`. In this sections these will be briefly explained however the developers of pip also provide a thorough Reference Guide.

The most used command is `pip install <package>`. This simple command triggers pip to install the named package. When the command doesn't contain any option or a version number of the package, the latest version of the package is simply installed. (How the installation process works is discussed in the section about pip's interactions.) Using pip, it is possible to install packages on the global system or in virtual environments. Complementary to `pip install`, pip has the command `pip uninstall <package>`. As the name suggest this takes care of uninstalling a package.

The command `pip freeze` is used to get an overview of all packages that are installed on the system or virtual environments. It outputs the packages in the requirements format, so it can be easily used by the command `pip install -r <requirements-file>` Similar is the command `pip list`, which also provides a list of packages, but in a reader friendly form.

It is possible for users to search through PI servers to get more information about packages (this is also possible using a web browser). The command `pip search <query>` searches for PyPI packages whose name or summary contains `<query>`. In the next section more is explained about PI servers and PyPI.

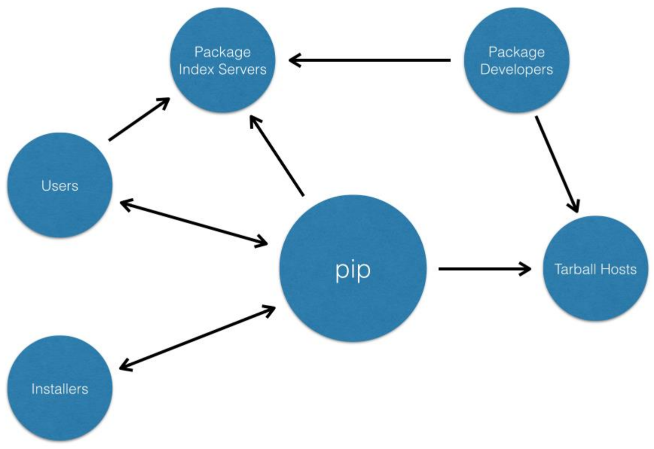
To get more information about an installed package, a user can use `pip show <package>`. This will then show information like the requirements of that package and the location of installed files.

pip also works with *Wheel archives*. Wheel is a built-package format and offers the advantage of not having to redownload and recompile your software during every install. For example, by using the command `pip wheel`, it is possible to collect all installed packages in a virtual environment. Having this archive, all these packages can simple be installed in a new virtual environment, without the need to search for and download all the packages again.

pip and its interactions

Before we can zoom in to the architecture of pip, we have to take a look at the context in which pip works. Therefore, we now will discuss the context of pip or in other words; the systems and parties pip interacts with. After presenting the complete situation in which pip occurs, we will focus on the entities that play a role and will describe the entities and their respective roles, after which we will discuss some interactions in detail.

Context view diagram



Global overview of the context of pip

In the diagram above, you can see all the entities that (in)directly interact with pip and how all interactions are situated. Most of the entities in this diagram are straightforward and do not need much further explanation. Shortly,

- *Users* are the actual users (e.g. developers) that use pip to install packages. Users interact with pip through a command line interface.
- *Installers* are a considered part of the installation of a software artifact, which also interact with pip through a command line interface.
- The *Package developers* are the group of developers which create and want to distribute Python Software.
- The *Operating System* is important to pip because it provides services to pip and libraries to programs.

We will discuss *Tarball Hosts* and *PI Servers* in more detail because these are less known, and some precise distinctions need to be made clear to fully understand the most important interactions.

Tarball Hosts and PI Servers

Tarball hosts are servers which host the files needed to install requested packages. Most often this is a tarball with the package source. As follows from its definition, pip heavily depends on tarball hosts, since they provide all the files needed to install packages.

Next to tarball hosts there are the *Package Index servers*, these hold a repository of different packages. pip uses the metadata stored on these servers to get all the information needed to install a requested package. *Package Index servers* can be seen as a passive system since it is only queried but never initiates an exchange of information.

The distinction between PI servers and tarball hosts is that PI servers acts as an index of all packages whereas a tarball host only actually stores the packages. It must be noted that a PI server can also act as a tarball host, this is the case by default for PyPI itself. Concluding, pip collects all needed information about packages through PI servers and uses this information (metadata) to retrieve the packages through a tarball host. In the next section a more detailed view on this interaction will be given.

Interactions

Having introduced the systems that are important for pip and naming what their roles are, most interactions from the context diagram follow straightforward.

An interesting interaction is the one between users and the PI servers. Namely, users are able to learn more about all the packages that are available on the servers. In this way, any user can gain information about which particular package they want to install (through pip). Users can browse the indexes using their browsers for example or use pip itself to search for certain packages. Also, besides finding information about packages, package developers can upload distribution to the servers. Uploading happens by the package developers by calling a specific Python script which handles the addition/mutation of a package in the index.

Interactions for installing packages

Above are some examples of single interactions between entities. However, the most important action, namely the installation of packages using pip, happens in a chain of interactions.

Stepwise, we find that the installation (from a users perspective) of packages (and corresponding interactions) is as follows:

1. The user uses the `pip install <package>` command to ask pip to install a certain package.
2. pip searches through one (or more) PI server(s) to retrieve information about the package. In this way, pip gains information about the tarball host(s) that provide the requested package and for example other packages the requested package relies upon.
3. pip connects to the correct tarball host to download all the needed data to complete the installation.
4. pip uses the downloaded data to complete the compilation and installation.

Installations using installers happen the same way. From this example, you can conclude that users and installers handle the server side operations of the pip ecosystem. They simple request certain packages to be installed and the data of this comes directly form those two aforementioned hosts.

Package Developers

The Package Developers have an important role in the pip ecosystem, after all they provide all the packages that pip enables you to install. The Package Developer adds their package to the Package Index servers and decide what to use as a tarball host. They either use the Package Index server if this is possible, or they can decide to use another server as the tarball host. In either case they upload the necessary files to the appropriate servers, and user can begin downloading their software package via pip.

The architecture of pip

In this section the architecture of pip will be introduced. First the modules of pip will be outlined. These modules can then used the define the architecture of pip. Besides these modules common processing and libraries can be identified within the architecture that pip depends on. Finally how pip tracks what packages are installed will be discussed.

Modules

Based on the source code of pip the following architecture has been conceived. pip is architecturally divided in three layers containing a total of six modules. The modules will be discussed first, subsequently the layers will be explained.

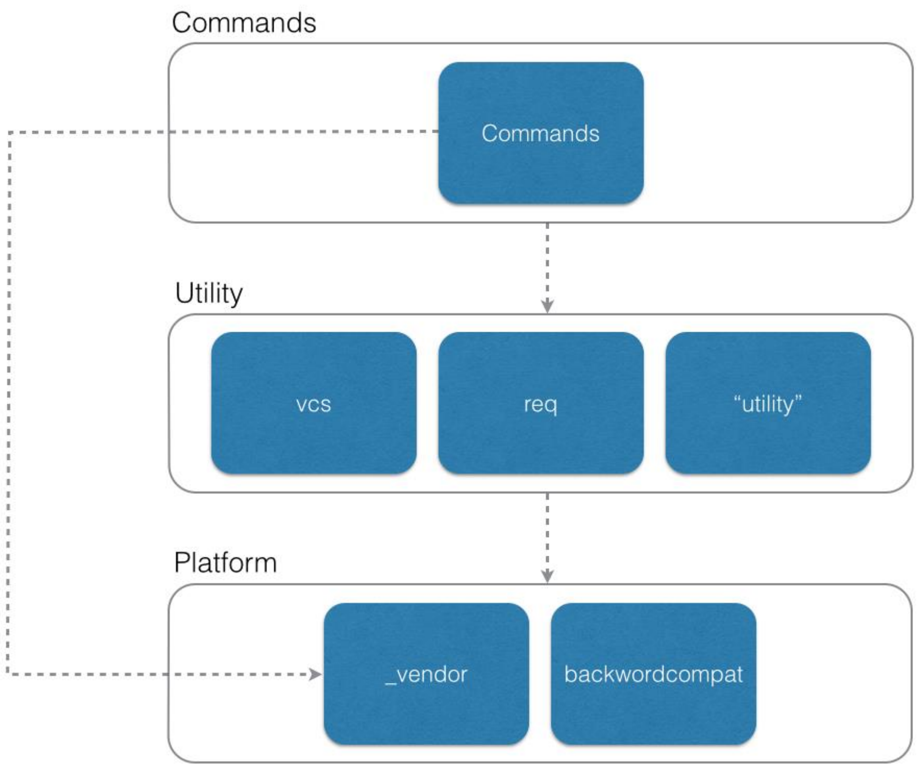
The following modules are present in pip:

- **_vendor** contains all vendored libraries needed to build pip. These libraries are in this module in order to prevent end users from needing to manually install packages if they accidentally remove something that pip depends on. In other words, when a user deletes a dependency by mistake, they can still use pip.

- **backwardcompat** handles everything that is needed to let pip work on different Python versions and platform distributions.
- **req** contains all files that handles requirements that are needed for some commands. As an example, the module contains the file *req-install*, that contains code which deals with the requirements of the `install` command.
- **vcs** stands for version control support. This module is used for downloading packages from tarball hosts. It is not used for the version control of packages.
- The **utility** module is the collection of files with code of several utility classes. In the pip organisation, these files are present in the root of the code organisation. Examples of files in this module are `exceptions.py` and `log.py`.
- **commands** contains the code of all possible pip commands, like `install`, `freeze` and `search`.

Layering Structure

There are several dependency relations between the modules. Given these dependencies, the six modules can be divided in three layers. A graphical representation of the Module Structure of pip is provided below. This representation clearly shows the different dependencies among the modules.



Module Structure diagram

The lowest layer contains the modules *_vendor* and *backwardcompat*. These modules contains the code which ensures that pip can be build and is compatible with the system it runs on. This layer is therefore the base of the architecture. It should be noted that the two modules in this layer do not share any dependencies with each other.

The second layer can be seen as the base for the functionality of pip itself. It contains the modules *req*, *vcs* and the utility unit. These three modules do share dependencies with each other, and also, there are interlayer dependencies with the first layer.

The top layer of the system only contains the commands module, which implements the commands of pip. Of all modules this is the least abstract one, and is therefore put in a higher layer. This layer has interlayer dependencies with both the req module as the utility unit. Besides these dependencies, there is also an explicit intermodule dependency with the *_vendor* module. Because this module is not in the layer below the commands module, pip has an architecture with a non strict layering structure.

Common processing

Pip has a small codebase, this limits the needs for isolated and common processing. This need is further limited by the nature of pip.

There are two ways of how pip exposes common processing. The first way is to expose functions that can be called that will handle common processing. The second way is to use one-time common processing.

When initialising the system executes the common processing and the results are afterwards saved in global variables. Python restricts name space on a module level, but pip only has one module so these variables are accessible system wide. The reason for the adoption of this practice is that the result of these calculations are always the same. This provides an easy way to share common information and limit calculation. But the one-time calculation imposes restrictions on the system, due to invariants that have to be maintained during execution. Examples of these variables contain what operating system is used, or the username of the user which executes the command. During execution of pip you cannot change what operating system you are running or the user name that executed the pip command.

Configuration

There are several ways to configure the way pip operates. Because of there being several ways, precedence rules are in place. Higher in the list means more precedence.

1. Values set in the CLI
2. Environment variables
3. Environment configuration file
4. User configuration file

Environment information and environment specific file locations are combined into another code unit. It gives information about what operating system is used, other operating system related information, and locations of files, like where to install packages or the location of the configuration file. This information is passed by global variables to other parts of pip.

State

Of course, since pip handles the installation of packages and its dependencies it needs to keep track of its internal state. For this, pip uses an internal registry. In this registry pip saves which packages are installed, where they are placed and which versions are available. It also keeps track of packages in virtual environments. To see the current state of the system, one can query pip using `pip list` which will show a complete overview of the installed packages. Using `pip show <package>` one can see specific information about a package. To do this automated (such as generating a requirements file) the option `pip freeze` is present.

The development process of pip

pip bases the entire development process upon git. Specifically they use GitHub as their git host, which provides them all sorts of feature that benefit their development process. In this section we will describe the development process pip uses.

GitHub

The use of GitHub enables the official maintainers of pip to track issues. This is used by users to report bugs, but it is mostly used to enable official maintainers and occasional contributors to track what needs to be done to fix any problems and to implement new features.

Aside from tracking issues, GitHub is also used for the release of pip. GitHub provides ways of tagging groups of commits as a release. PyPA uses tagging to create releases, but they also use GitHub to host the files that are needed to install pip.

Even pip's documentation is hosted on GitHub (in a git repository). To do this a service called ReadTheDocs is used to generate a documentation website from markdown files. As an alternative, support for Sphinx is provided, which is a third party service that is able to generate documentation from markdown in a variety of formats.

Branching strategy

By default pip uses a branch called *develop*. This is the branch on which all development is done. If a new feature is implemented a separate branch is created for the feature, this feature branch will eventually be merged to the *develop* branch via a pull request. The section on 'Contributing' will expand upon this. At some point it is decided to release a new version of pip, at that moment a new release candidate branch is create based on the *develop* branch. This release candidate is reviewed by the community until it is deemed ready to be released. The changes are then merged to the *master* branch of pip, and the release system of GitHub is then used to make the new release available.

Building & testing

Before changes are merged to master pip is automatically built and tested using Travis CI. Travis is a third party service that allows you to automatically build the system in different environments and run tests on it. For pip this works as follows: Whenever changes are submitted to the *develop* branch via a pull request, Travis automatically builds the resulting system and runs it against the test suite defined by the developers of pip. Every time changes are pushed this process is repeated, so at every point in time it is know whether or not merging a pull request will break the system. Although Travis is convenient for reviewing a pull request, the test can also run locally. In fact, it is asked to make sure that your changes make it through the tests before you submit a pull request.

The test suite consists of an elaborate set of unit tests, which cover most parts of pip, and also a large set of integration tests, which cover most of the functions of pip. The tests cover functionality like install and uninstalling packages, searching PyPI or installing from different kind of sources.

The developers use an external package Flake8 to run the tests on the source code. Flake8 can be used to run tests locally, but to fully automate things Tox is used. Tox allows automated testing using Flake8 for different Python version. Travis CI in turn uses Tox to run automatically when pull requests are made.

Pip also uses Jenkins CI to test the system for specific python versions on windows and centos. This is done regularly, but not automatically.

Contributing

Contribution from the community is an important part of the development process of pip. Anyone can fork the repository, create a new feature branch and submit a pull request to have their changes merged into the final version of pip. Although most development is done by a small group of developers, there are also a lot of contributions done by other people. Often these are just small changes to fix bugs they encounter, but sometimes these are more significant contributions.

Since 2008 close to 50% of the pull requests have actually been merged into the *develop* branch. A rather low number, but it turns out that pull-request are often done against the wrong branch, and are therefore closed and reopend on another branch. Another issue has to do with pull request by developers who are not official maintainers, it happens often that these pull requests don't match the requirements of the official maintainers or the changes do not pass the automatic testing after which the developer gives up on the pull request. This all leads to the fact that most merged pull request are done by official maintainers. Those pull request are usually merged within a couple of days, whereas pull request from external parties usually take longer. That seems perfectly reasonable since some things like coding style don't match that of the official maintainers these things need to be adjusted before any changes can be merged.

In conclusion, the development process allows for everybody to contribute but in practice a significant amount of the code base is done by the official maintainers. It seems though that this openness of the development process attracts people that eventually become official maintainers.

The future of pip

Just as many parts of Python, pip is also modified and extended by community inputs; *PEP*'s. PEP is an abbreviation for Python Enhancement Proposal and is intended to propose new features, getting input from the Python community, and documenting specific design decisions. Since pip is a project backed by PyPA it will come as no surprise a large number of PEPs has been created for pip itself.

For Python, pip can be seen as a special project. It is not part of the official CPython release software, although the makers of CPython encourage the use of pip as a Python package manager. pip is a project which is of enormous importance for the Python development and distribution since it is the official program to handle these issues.

However, at this point pip is at a special place:

1. it is not part of the official CPython project
2. almost anyone depends on its availability (which depends on the user platform)
3. older pip versions hinder updates to a better PyPI (the index)

In PEP-453 a number of authors therefore propose a radical change to the way pip is seen by the community. The proposal includes that future CPython versions should come with the pip installation files bundled (though not installed); moreover, functions in Python 3.4 should allow the bootstrapping of pip on the system.

This proposed system has a number of benefits:

1. First of all this allows for easier installation on a number of platform (specifically Windows, OSX and slower moving Linux/Unix distributions)
2. Secondly it will be easier for new users to start using Python since the installation may install pip itself as well
3. It allows the evolution of the packaging system which is used in Python in the future.

Scripts can call `python -m ensurepip` to allow the installation of pip (global or user-specific). The packaging of pip in the Python environment also ensures that pip can be installed without network access (and let pip install using local sources).

This proposal marks an important step in the development of Python and the acceptance of pip by its users. As far as the authors are concerned, pip is the first language-specific package management system to be embedded into the very language. Compared to other languages, such as Java, these package management tools still need to be installed independently (Maven or Ant).

In short: the addition of pip to the language has a number of large advantages for users, while it has no serious downsides. e.g. Users and downstream maintainers can still install pip manually, pip and CPython keep their own release schemas; moreover, their development cycles won't influence each other. Also this makes it easier for first-time Python users to start working with dependencies straight-away.

Challenges

In this chapter pip was discussed in various aspects. However there are also a few of problems with pip, these challenges need to be taken on by the developers in the following years to ensure the continued usage of pip.

The main issues of pip are: 1. pip uses a non-standard registry; 1. there is platform-specific static metadata in PyPI; 1. PyPI does not have a useful categorization system.

The first issue is that pip has an own internal registry. Although this does not pose a problem directly, it might be problematic if on the same machine one user uses `pip` and another user uses `easy_install`. In this case both systems will install dependencies without knowing about each other, and these dependencies may therefore conflict. Python is currently working on a better installation procedure: it aims to replace Distutils by Distutils2 to mitigate this issue.

Secondly, pip has platform-specific metadata in the central index. For most packages this is not directly an issue, but for some it might be. Assume that some package was developed which communicates over hardware sockets. In Windows this package would also need an extra package `win32comm`. However this is not required on Linux and Unix-like operating systems. Once the metadata for PyPI is generated, the metadata is static and uploaded to the servers. Hence, in case the metadata is generated under Linux, the dependency for Windows would be missing, resulting in an incorrectly installed dependency. And when the metadata is generated under Windows, any Linux machine would try to install a package `win32comm`, which will not work either. Hereunder it is clear why having static metadata is not always useful for the index, and why there should be a possibility to make this dynamic. Once again, this is addressed in the same `Distutils2` package that is currently under development.

Finally, PyPI does not have a categorization system, which means the user (or developer) should know the exact name of the package. This name might be obtained from a `pip search <query>` command, from browsing the PyPI index, or from any other source. However, this is not yet as user friendly as pip (and Python) aim to be. Therefore the projects have declared that such a functionality should exist in PyPI, since it gives users a better handle of what can be expected from a certain package.

The flaws described above do not make pip less attractive. It is its simplicity for the users that makes pip a very attractive product, and therefore pip will be the first language-specific package management system to be embedded into a language, Python.

Concluding: In the near future, the simple architecture discussed in this chapter isn't going to change. But, the way development is organised for pip is one that ensures people keep working on the product. For Python itself it is an aim to replace `Distutils` by `Disutils2`, is order to solve some of the flaws discussed above. This means that pip needs to evolve as well. E.g. PyPA names that pip schould gain support for using `distutils2/packaging` as an alternative to `setuptools/distribute`. So, this will probably the main focus in upcoming development. Even better for the users, because this will make package handling even easier!