Survey of software architecture

Name of system: Python Packaging

Reviewer: Trevor Bekolay

Date: October 3, 2011

Author of software: The chapter on Python Packaging discusses a number of different pieces of software. Below are those packages and their authors. Note that the so-called "Fellowship of the Packaging" is a group of several Python developers led by Tarek Ziadé. The "Python Packaging Authority" is a group of several Python developers including Ian Bicking, Carl Meyer, Brian Rosner, Jannis Leidel, and others.

`distutils` Greg Ward, Anthony Baxter

`distutils2` "Fellowship of the Packaging"

`setuptools/easy_install` Phillip J. Eby, Ian Bicking, and Jim Fulton

`distribute` "Fellowship of the Packaging"

`pip` "Python Packaging Authority"

`virtualenv` "Python Packaging Authority"

Pypi PEP's contributed by A.M Kuchling, Sean Reifschneider, and Richard Jones[1], implemented by Richard Jones

Author of book chapter: Tarek Ziadé

Five star rating of book chapter: Perhaps it makes sense that this chapter comes off as somewhat disorganized, given that it is discussing the disorganization involved with Python Packaging at the time the chapter was written. It is, in a sense, a case study of what not to do, and since it discussed a number of different packages, it was difficult to learn anything important about any one of those packages. It gave me more insight into the Python project and language itself, rather than the architecture of a piece of software. Overall: **

Purpose of system: The Python Packaging tools are used to manage Python packages installed on a user's system. Python adopts a Linux-like approach to installing packages, in which they are tracked and managed, such that a Python package, such as `NumPy`, is installed to the appropriate place on the user's system, and any package that uses `NumPy` uses that installed version. That means that updates to `NumPy` are immediately available to all packages that use `NumPy`, but it also means that a package must keep track of the packages it depends on, and whether updates to those packages cause unwanted issues. The set of tools described in this chapter are what enable Python programmers to install packages to the appropriate place and deal with dependency issues. Below is a short description of the purpose of each system discussed.

`distutils` A module used by developers to enable their package to be easily installed on a user's system.

`distutils2` An improved version of `distutils`. Fixes a number of issues discussed in the chapter. Aims to replace all of the tools in this list, except `pip` and `virtualenv`.

`setuptools/easy_install` A module used by users to download, build, and install packages easily, through a common interface.

`distribute` An improved version of `setuptools`. Fixes a number of issues discussed in the chapter, but will eventually be obsoleted by `distutils2`.

[1] PEP stands for "Python Enhancement Proposal," and is the primary method through which Python developers suggest new Python features.

`pip` An improved version of `easy_install`. Fixes a number of issues discussed in the chapter.

`virtualenv` A module used by developers to create isolated Python environments. Allows users in restricted environments to make local package repositories, for example.

Pypi The central Internet server that accepts Python packages from developers and distributes them to users.

Basic metrics

Lines of code: These totals determined using the CLOC tool (`http://cloc.sourceforge.net/`) on each program's source code.

`distutils` 22 KLOC

`distutils2` 36 KLOC

`setuptools/easy_install` 14 KLOC

`distribute` 16 KLOC

`pip` 11.5 KLOC

`virtualenv` 4 KLOC

Pypi 14.5 KLOC

Project start-up: Each project's startup is listed below, in chronological order (which is illustrative).

`distutils` First released along with Python 1.5.2 in April, 1999. Included in Python 1.6 in September, 2000.

Pypi Operational since February, 2003. First code committed in November, 2002.

`setuptools/easy_install` Documentation was first provided in July, 2005. Version numbers indicate that the first release was likely shortly before that date.

`virtualenv` First committed September, 2007.

`pip` First committed with the name `pip` (previously `pyinstall`) in October, 2008.

`distribute` First committed in July, 2009. A fork of `setuptools`.

`distutils2` First committed in February, 2010, though development was originally a series of edits to `distutils`, which had to be reverted to ensure backwards compatibility.

Number of major releases:

`distutils` New release with each version of Python since 1.6 (approximately 12 major releases).

Pypi Doesn't have major versions, code rolls out regularly.

`setuptools/easy_install` Four (0.3 - 0.6).

`virtualenv` Nine (0.8 - 1.7).

`pip` Ten (0.1 - 0.8, 1.0 - 1.1).

`distribute` One (0.6).

`distutils2` None (1.0 still in alpha).

Number of developers: There is a lot of overlap in the developers for all of these tools. Below, they are listed separately; in aggregate, there are perhaps twenty to forty active developers working on various pieces of the Python Packaging system.

`distutils` Four primary developers.

Pypi Thirteen contributors to the SVN repository.

`setuptools/easy_install` Approximately five developers.

`virtualenv` Seven primary developers, approximately twenty contributors.

`pip` Seven primary developers, approximately thirty contributors.

`distribute` Two or three primary developers, approximately ten contributors.

`distutils2` Four or five primary developers, approximately twenty contributors (the "Fellowship of the Packaging" mailing list has 89 members, though it is unlikely that all members contribute code).

Size of user community or number of installations: `distutils` is included in the standard Python distribution, meaning that the number of installations is in the millions (though many of those installations, such as the one included by default on Mac OS X, may not ever be used). Even those tools not included in the standard library see much use; `setuptools`, for example, has been downloaded over a million times. Python is increasing in popularity, and with many applications starting to use it, these packaging tools will affect millions, though only tens or hundreds of thousands of people may use them directly.

Major stakeholders: Almost all Python developers are stakeholders for this set of tools. Even developers who do not plan to distribute their packages widely will likely need to install dependencies using these tools.

Use of concurrency: These tools are not resource-intensive, and thus do not require the use of concurrency.

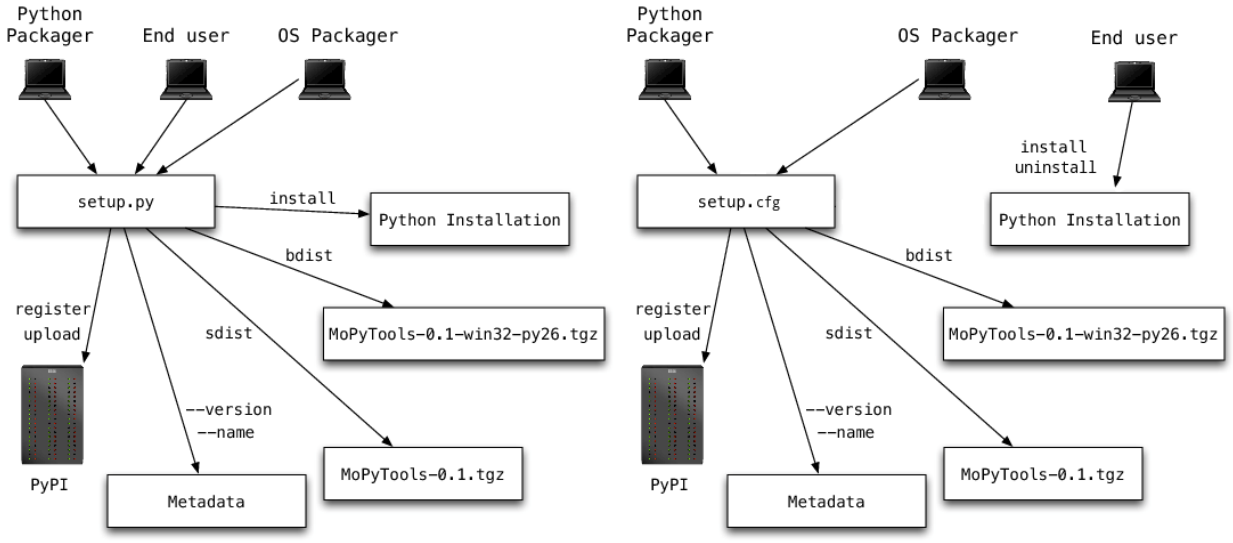
Implementation language: All of these tools are implemented in Python.

Supporting software: All of the tools rely on elements of the standard Python library. `distribute` depends on `distutils2`. Everything else is independent.

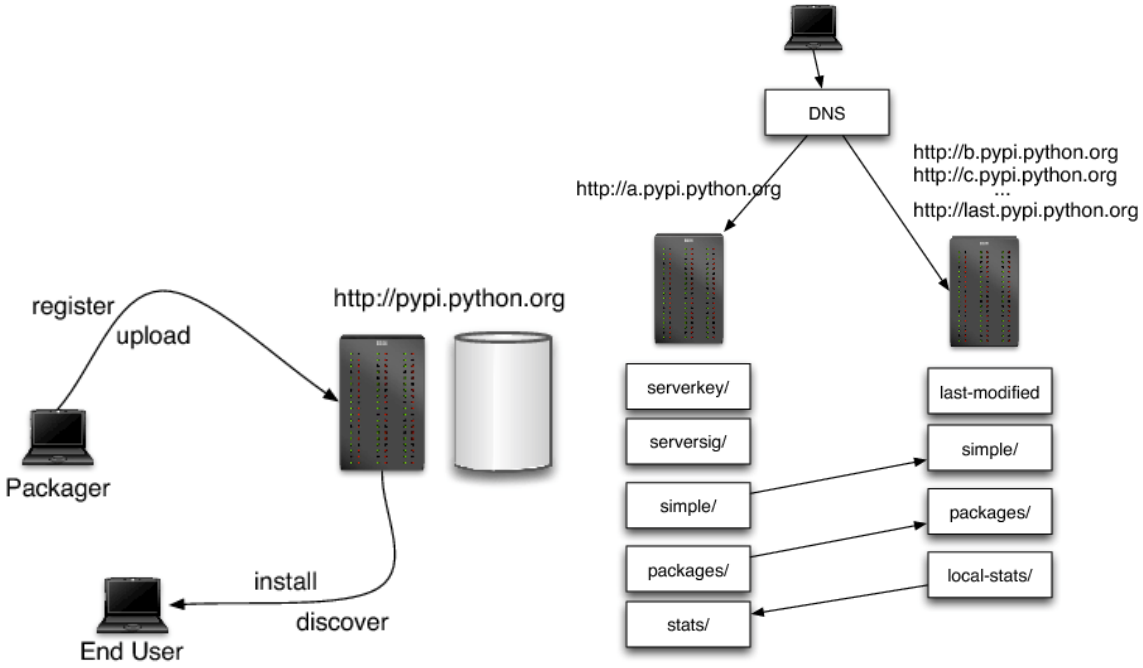
High level architecture

Diagram of software architecture The original design of Python packaging relied on `distutils` for almost all operations. Every Python package would include a `setup.py` file that would include `distutils`, allowing for the functionality described in the left figure below.

As the packaging effort continues, the intention is to change this picture slightly, separating out the developer and user interactions, among other things, resulting in an architecture more like the right figure below.

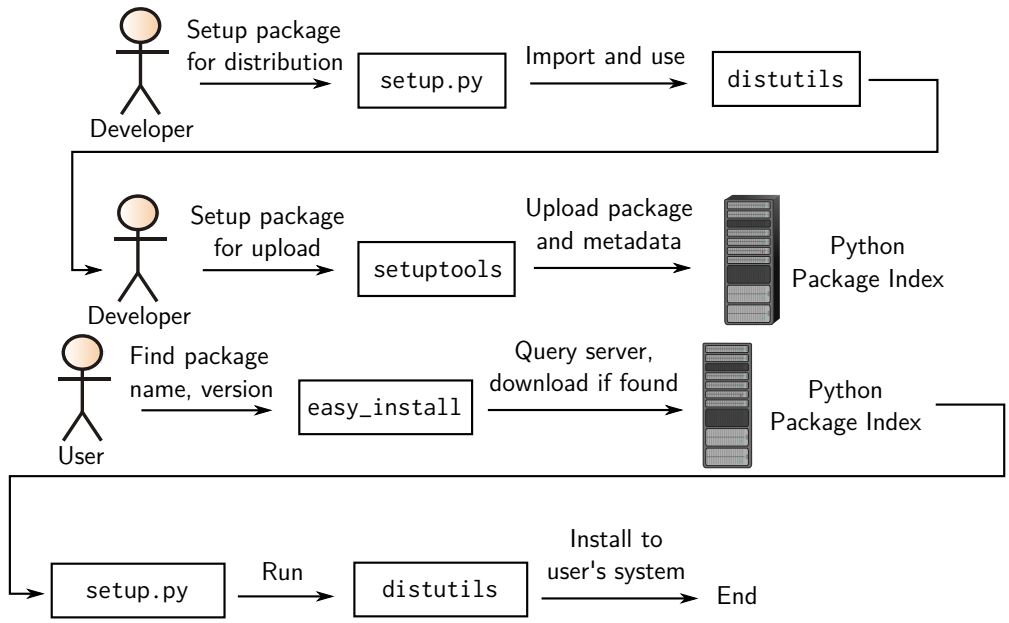


PyPI's architecture is interesting in its own right. Its interactions can be seen in the left figure below, and its mirroring system is summarized in the right figure.

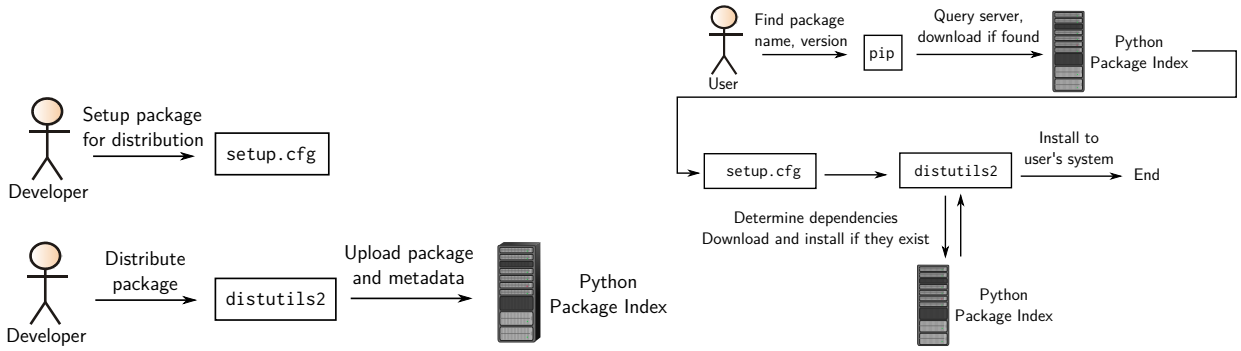


High level scenarios: Due to the transitional nature of the architecture described in this chapter, it is useful to examine two sets of scenarios: one describing the current state of Python Packaging, and one describing the desired future state of Python Packaging.

The current way can be summarized with these two use cases, one for the package developer and one for the package user.



The new way can be summarized with these two use cases, one for the package developer and one for the package user.



Data structures or algorithms: While not an algorithm or data structure per se, I found the description of version numbers quite interesting, and sensible. Python developers can be somewhat whimsical, using clever names for versions. One of the accepted proposals described in the chapter is a reasonable format for version numbers. The format is: `N.N[.N]+[a|b|c|rcN[.N]+][.postN][.devN]` where

- N is an integer; MAJOR.MINOR.OTHERS*,
- a, b, c and rc are alpha, beta and release candidate markers,
- dev followed by a number is a dev marker, and
- post followed by a number is a post-release marker.

Architectural style: The software packages described in the chapter are all relatively small sets of Python scripts, usually consisting of less than 2 MB of code, and generally using the Object-Oriented style that is common for Python programs. However, the overarching style is perhaps better described as being a client-server architecture, with PyPI acting as the server and both developers and users using these tools as clients to access and interact with the PyPI server.

Major evolutionary changes: This chapter describes an evolutionary change that is still taking place. The original way to do things was to use `distutils` and `setuptools` to interact with PyPI. The proposed change is to move towards `distutils2` and `pip` to interact with PyPI. This change is still occurring. `distutils2` is part of the Python 3.3 alpha release, but the eventual goal is to use `distutils2` in Python 2.4 and above.

Performance bottlenecks: PyPI presents a large bottleneck for Python packaging, though mainly in terms of stability rather than raw performance. Having a single server present a single point of failure; if it goes down, then all Python users can be inconvenienced. The chapter describes a mirroring scheme to alleviate these concerns. More recent discussions have also indicated PyPI may be distributed over a content delivery network in the near future.

Real time: No real time concerns, except possibly the latency and bandwidth of the PyPI server (or servers).

Notation for architecture: N/A

Methodology: All of these projects are open source, so individual developers will use whatever methodology suits them best. However, many of the development pages indicate that significant work was done at "Code Sprints" during various Python conferences. The idea of a Code Sprint fits with the Extreme Programming methodology.

Appendix: Kruchten's context attributes

Size M (over 100KLOC in total)

Criticality Med

Age of system L (`distutils` has been around since 1999)

Rate of change Med (while it took a while for change to happen, it's rapid now)

Business model Open source

Stable architecture Hi (at the moment)

Team distribution Hi (online, except at Python conferences)

Governance Med (PEP's give some structure)