

Distutils: Packaging, Metadata and Pushups

Author: Jeff Rush <jeff@taupro.com>
Copyright: 2009 Tau Productions Inc.
License: Creative Commons Attribution-ShareAlike 3.0
Date: March 25, 2009
Series: Python Eggs and Buildout Deployment - PyCon 2009 Chicago

An introduction to the *distutils* module that provides a standard way of **building, distributing** and **installing** one or a group of Python modules, across platforms. *distutils* has shipped as part of the Python standard library since version 1.6.



What is the Problem?

- a standard mechanism for
 - building , packaging (distributing) , installing
- one or more modules of
 - Python source , C/C++ code , data files
- be cross-platform
- fit into existing packaging technologies
- an extensible tool
 - distribution file formats
 - special processing commands

We want a mechanism standardized within the Python community for building, packaging, distributing, and installing one or more modules that may consist of Python source, compiled C source and bundled data files.

A solution should do all this in a manner that works across operating system platforms, plays nicely with existing packaging technologies, and provides for an easily extensible set of distribution file formats and special processing commands.

Roadmap of Topics

- What is the Problem?
- Facts About *distutils*
- What is a Distribution?
- How are Distributions Used?
- Examples of Invocation
- Applying *distutils* to Your Software
- About Package Servers
- Further Reading

Facts About *distutils*

- Developer's Day session (IPC7 Nov 1998)
- shipped with Python in version 1.6
- *setuptools* and *buildout* make use of it

Missing Features:

- can upload but not download packages
- no *automatic* dependency satisfaction
- no way to list **installed** distributions
- no official way to uninstall a module
- no dev mode; have to re-install each time to test
- no help/documentation bundling yet
- executable scripts are only copied and are platform-specific
- distributions are always unpacked for use

Scattered work on a solution to meet these requirements had been underway for quite a few years but in 1998 started to come together under Greg Ward at IPC7. This led to a version of *distutils* shipping with Python 1.6.

There is an old version linked to from the *distutils-sig* page but it is out-of-date. The official version is that which ships with Python.

The *setuptools* module that is the basis for *eggs* and the *buildout* deployment tool both make heavy use of *distutils* and leverage its concepts.

What is a Distribution?

A Distribution

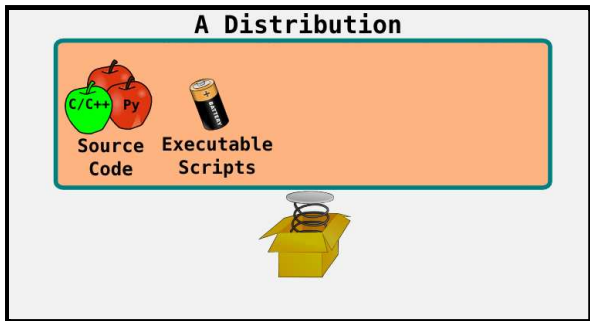


A "distribution" is a single downloadable resource, a single-file importable distribution format.

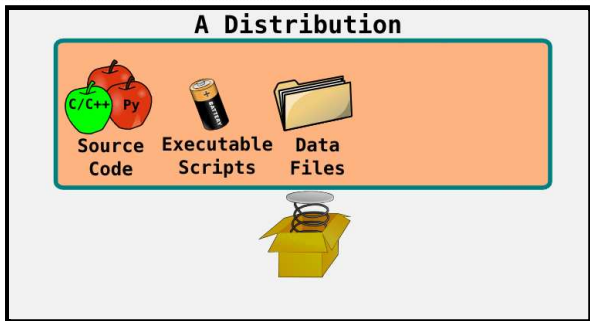
A Distribution



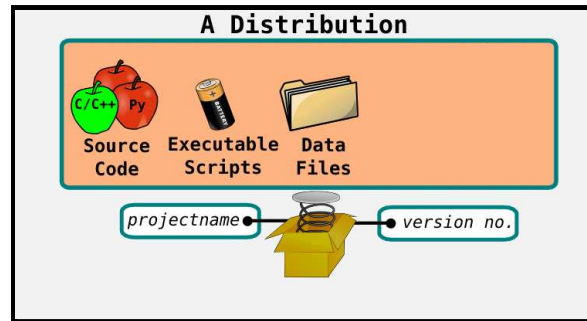
made up of one or more modules, each of which may be a single .py file, a directory of modules organized into a Python package, or a C/C++ extension. These modules may be independent or unrelated sibling packages - no relationship is assumed.



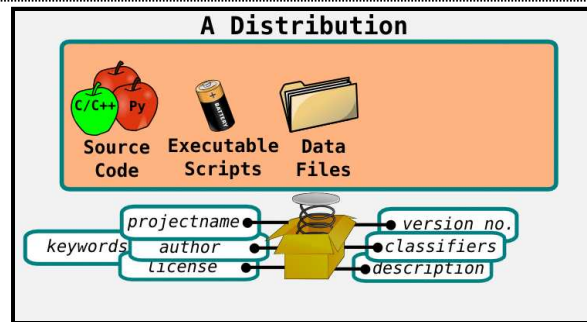
It may contain executable Python scripts which get installed as additional system commands, depending upon the operating system.



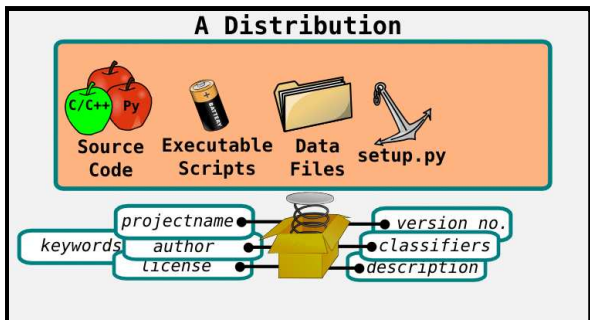
And it may contain data files, such as icons or reference material. All of which are intended to be installed or uninstalled together.



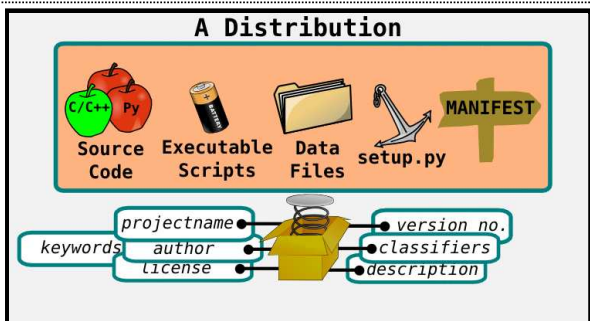
A distribution has certain metadata, such of which is optional. Every distribution is required to have a name (supposed to be) unique within the Python community, and a version identifier.



It may also have information to classify the distribution, keywords to help locate it, and contact information for the author or maintainer.

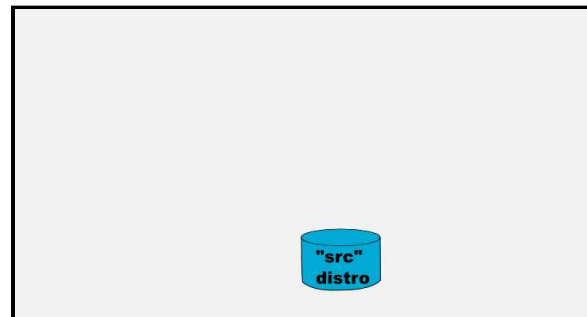


The place all these are specified and controlled is within a special Python module named `setup.py`. This module is executed as a command with differing arguments to actually perform package generation and installation operations.

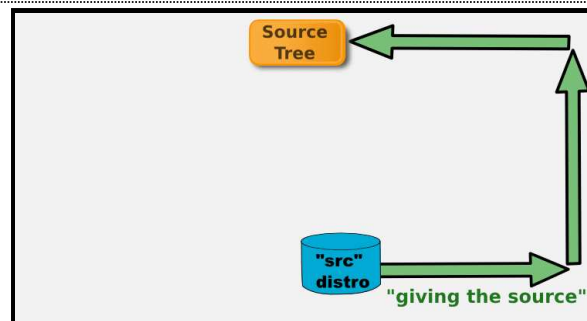


We've talked mostly about items from the perspective of what gets installed to *use* the package. There is one other piece of metadata needed, that specifies all the files needed by a developer to work on the distribution. For example, if our distribution included C/C++ source code, the installation would only bundle binaries compiled from that source and omit that C/C++ source. However a developer needs that source to do work. This leads to the distinction between "binary" distributions for the end-user and "source" distributions for other developers.

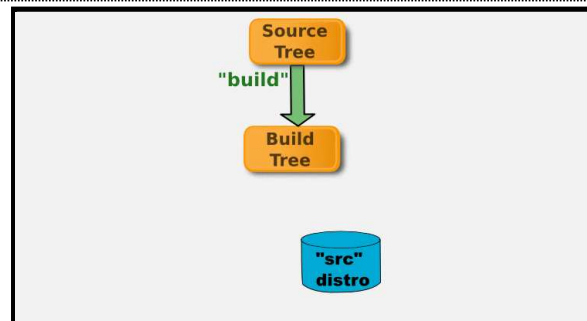
How are Distributions Used?



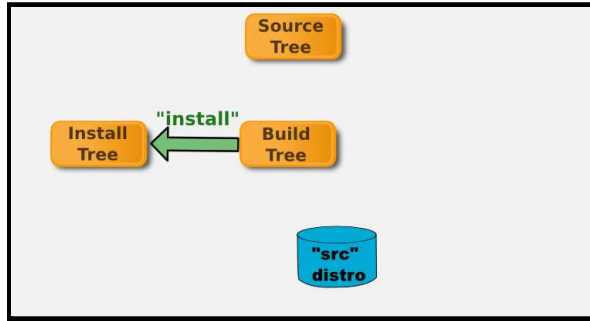
Say someone gives you a source distribution, often as a compressed archive file. We unpack this into a temporary directory:



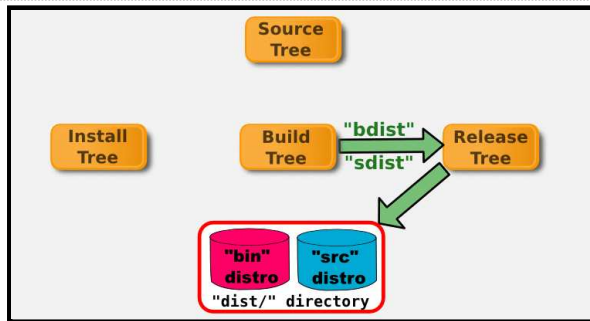
After reading the docs we decide to try it out by building it,



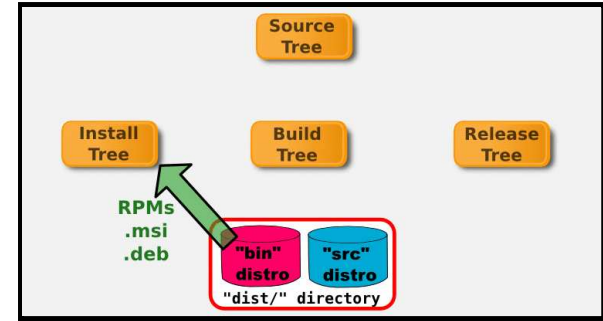
and then installing it into our Python *site-packages* directory. After testing it we decide to repackage it into a file format different than we got it for sharing with some friends.



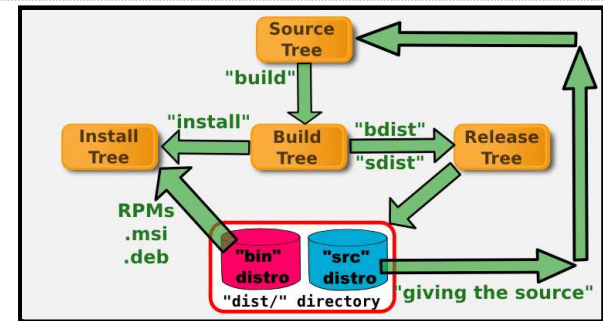
On the other hand, we may want to generate a platform-specific form of distribution, such as RPMs or .deb files.



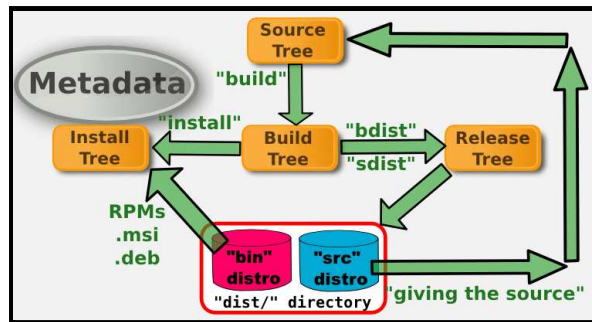
And give them to a non-developer to install on her system.



Note: Binary distributions are intended for installation into environments that lack development tools such as a C/C++ compiler, and usually come in the form of existing package technologies, such as RPMs, .debs, .msi, .dpg.



All together now the diagram looks like this.



Metadata about its characteristics, its ownership and dependencies is retained past the build process into the installation itself, so that other packages that depend upon it can know of its presence and version. This metadata is also what is transmitted up to index servers like the Cheeseshop.

Examples of Invocation

- uses a custom *setup.py* in project root


```
from distutils.core import setup; setup()
```
- is command-line driven


```
python setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
```
- `python setup.py --help-commands`
- `python setup.py build --help`
- `export DISTUTILS_DEBUG=yes`

distutils makes its appearance in a project via a customized Python source file named *setup.py* placed in the project root directory.

A minimal such file looks like this. To interactively follow along with me, you may want to create such a file if you don't have one handy from an existing project.

Invocation of *distutils* is command-line driven as follows. You change into the directory of the project and pass *setup.py* explicitly to the Python interpreter.

Some of the key global options, their meaning pretty much self-explanatory, are `--verbose`, `--quiet`, `--dry-run`, and `--help`.

distutils accepts one or more commands per invocation, each with a variable number of arguments. The boundaries are found by matching arguments against commands in the mapping.

To obtain a list of defined commands, use the `--help-commands` option. This list can vary since *distutils* supports extension through addition of new commands.

Each command has its own help which can be accessed like this.

To get internal processing details while *distutils* is working, define the `DISTUTILS_DEBUG=yes` environment and invoke *setup.py*. Any error will now give you a traceback telling you more about the who and why.

Invocation: Installing from Source

```
$ cd /tmp
$ wget http://www.crummy.com/BeautifulSoup-3.0.5.tgz
$ tar xvzf BeautifulSoup-3.0.5.tgz
$ cd BeautifulSoup-3.0.5
$ less README

$ sudo /wherever/python setup.py install

$ /sandbox/bin/python setup.py install
```

The most common operation is probably installing someone else's distribution from the source. The distribution is installed into the *site-packages* directory of the specific instance of Python invoked. To install into a *virtualenv* sandbox, give the explicit path to the sandbox's interpreter.

Invocation: Building a Binary Distribution

```
$ cd /tmp
$ wget http://www.crummy.com/BeautifulSoup-3.0.5.tgz
$ tar xvzf BeautifulSoup-3.0.5.tgz
$ cd BeautifulSoup-3.0.5
$ python setup.py bdist_rpm
$ cp dist/BeautifulSoup-3.0.5-1.noarch.rpm /releases

$ python setup.py bdist --formats=tar,zip,rpm
```

The next most common operation may be building platform-specific packages for distribution. A single binary distribution may be given as the command "bdist_<format>" or one or more formats can be specified with the "--formats=" option.

Invocation: Building a Binary (formats)

```
$ python setup.py bdist --help-formats
```

- bdist (gztar, bztar, ztar, tar, zip)
- bdist_msi (Microsoft Installer binary)
- bdist_rpm (both source and binary RPMs)
- planned for Python 2.6
 - bdist_deb
 - bdist_egg
- 3rd-party add-ons
 - bdist_mpkg (for Mac OSX)
 - bdist_pkgtool (Solaris pkgtool)
 - bdist_sdux (HP-UX swinstall)

Invocation: Building a Source Distribution

```
$ python setup.py sdist --help-formats
$ cd /tmp
$ svn co http://www.example.com/foo/trunk foo
$ cd foo
$ python setup.py sdist --formats=zip
$ cp dist/foo-1.0.0.tar /releases
```

The *sdist* command extracts metadata and writes it to a file by the name of PKG-INFO in the top directory of the generated zipfile or tarball. This file is a single set of RFC822 headers parseable by the `rfc822.py` module. <http://www.python.org/dev/peps/pep-0241/>

This PKG-INFO file is what gets POST'd to the index server.

Metadata - Basic Identification

```
from distutils.core import setup
from setuptools import setup

from mymodule import __version__ as VERSION

setup (name='foo',
       version=VERSION,
       description='XYZZY Magic Generator',
       keywords='apple orange farming well',
       long_description=open('README.txt').read(),
       )
```

The *setup.py* file is just Python source, that invokes a single function `setup()`, with a variety of keyword arguments. This allows you to bring to bear on the configuration problem the full power of Python, without having to learn another configuration-specific language.

The *setup.py* file should not be marked executable as it is generally invoked with explicit reference to a Python interpreter. This is because the binary formats generated and the directory locations written to are specific to that instance of interpreter.

In general it is best to obtain the version of your distribution from within your source code, to avoid duplication of information.

The *long_description* keyword is a document, usually written in reStructuredText, that gets displayed on the project page in the Cheeseshop. Since *setup.py* is basically an executable configuration file, it can construct the fields at runtime.

Possible Layouts for Your Project Directory

For a simple distribution consisting of a single source file, placing it in the project root directory is common.

Or if the source is a single package that works fine too,

foo-1.0/ setup.py setup.cfg README MANIFEST.in build/ dist/ mymodule.py	foo-1.0/ setup.py setup.cfg README MANIFEST.in build/ dist/ mymodule/ __init__.py	foo-1.0/ setup.py setup.cfg README MANIFEST.in build/ dist/ src/ module1/ __init__.py module2/ __init__.py
----------------------------------------------------------------------------------------------	-----------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------

although some prefer to put it under a common directory like *src/*.

Metadata - Distribution Ownership

```
setup(...
    author='Sherlock Holmes',
    author_email='sholmes@baker.org',
    maintainer='John Watson',
    maintainer_email='jwatson@baker.org',
    url='http://baker.org/casehistory/',
    download_url='http://baker.org/downloads/holmes.tgz',
)
```

Metadata - Classification of Distribution

```
setup(...
    license='MIT',
    classifiers=[
        'Development Status :: 4 - Beta',
        'Intended Audience :: Developers',
        'License :: OSI Approved :: MIT License',
    ],
)
```

- `python setup.py register --list-classifiers`

The `classifiers` keyword is a list of strings, representing official tags used by the Cheeseshop. The official list at any time can be retrieved from the Cheeseshop with the `--list-classifiers` option to the `register` command.

Specifying Your Python Sources

```
from distutils.core import setup
from setuptools import setup, find_packages
setup(...
    # specify each module (not filename) separately
    py_modules=['foo', 'pkg.bar'],

    # or refer to entire packages
    packages=['foo', 'foo.command'], # package dirnames
    packages=find_packages(exclude=['ez_setup']),

    package_dir = { # map package names to directories
        ': 'src/lib',
        'pkg': 'other' }
)
```

- NO automatic recursion (until *setuptools*)

The names of "py_modules" are relative to the `setup.py` file itself. To locate modules within non-package subdirectories, use the `package_dir` keyword, which is a mapping of module names to directory paths. These paths are written in the Unix convention, i.e. slash-separated. A module name of empty string is the root package of all Python packages.

All packages must be explicitly listed; *distutils* will **not** recursively scan your source tree or package hierarchy looking for any directory with an `__init__.py` file. The *setuptools* module provides a `find_packages()` function however that does.

Setuptools is smart about recognizing packages, so invoke `find_packages()` instead of manually itemizing them. This can save you from accidentally forgetting to add one to the list when you add a new module to your source tree.

Specifying Your C/C++ Sources (and Build Flags)

```
from distutils.core import setup, Extension
setup(...
```

```
    ext_modules=[
        Extension('pkg.foo', ['foo.c'],
```

```
            include_dirs=['/usr/include/X11'])
        define_macros=[('DEBUG', '1')],
        undef_macros=['HAVE_BAR'],
        libraries=['X11', 'Xt'],
        library_dirs=['/usr/X11R6/lib'],
        # and many other options!
```

```
    ],
```

```
)
```

- rough support for SWIG

distutils has rough support for SWIG, processing .i files into C/C++ code.

Specifying Your Executable Scripts

```
from distutils.core import setup
setup(...
```

```
    # filenames of Python scripts
    scripts=[
        'scripts/xml_parse',
        'scripts/xml_dump',
```

```
)
```

- copies them into OS-specific directory
- under Unix, applies: `chmod +x yourscript`
- rewrites: `#!/usr/bin/env python`
- NOT versioned, can collide
- *setuptools* has a more sophisticated approach

Scripts are files containing Python source code, intended to be started from the command line. The "scripts=" keyword specifies a list of paths to the scripts.

These scripts get installed in the system command area and, because they are not renamed by version, may conflict if other distributions use the same name. This makes it difficult to install multiple versions of a distribution.

distutils takes care of marking the scripts executable for POSIX.

If the first line of the script starts with `#!` and contains the word "python", *distutils* will adjust the first line to refer to the current interpreter location. This value can be overridden with the `--executable` option to the *setup.py* invocation..

Specifying Package-Relative Datafiles

```
mypkg/
  __init__.py
  data/
    tables.dat
```

```
setup(...
```

```
    package_data={'mypkg': ['data/*.dat']},
```

```
    include_package_data=True, # setuptools only
    exclude_package_data={'': ['README.txt'] },
    zip_safe=False,
```

```
)
```

- paths are relative to `package_dir` entries, NOT `setup.py`
- glob filename patterns ok
- will create parent dirs as needed
- runtime access to data files in archives

The `package_data` keyword is a mapping from package name to a list of pathnames (relative to the package) of files to copy into the package.

`zip_safe` (*setuptools* only)

A boolean (True or False) flag specifying whether the project can be safely installed and run from a zip file. If this argument is not supplied, the `bdist_egg` command will have to analyze all of your project's contents for possible problems each time it builds an egg.

`package_data` (*setuptools* only)

The `package_data` keyword came from *distutils* but is less needed with *setuptools*. It is only needed to add, for example, files

generated by `setup.py` or the build process that are not otherwise under source control.

`include_package_data` (*setuptools* only)

Tells *setuptools* to install any data files it finds in your packages. The data files must be under CVS or Subversion control, or else they must be specified via the distutils' `MANIFEST.in` file. They can also be tracked by another revision control system, using an appropriate plugin.

Specifying Non-Package Datafiles

```
setup(...
```

```
    data_files=[ # destdir, list-of-files
                 ('bitmaps', ['bm/b1.gif', 'bm/b2.gif']),
                 ('config', ['cfg/data.cfg']),
                 ('/etc/init.d', ['init-script'])
    ]
)
```

- source relative to *setup.py* file
- destination relative to install prefix
- source paths NOT retained, unlike *package_data*=
- can relocate but **not** rename files
- feature removed in *setuptools*

The *data_files* keyword is for placement of datafiles unrelated to any specific Python package.

You can specify any destination directory for a file, but no mechanism is provided to rename it.

Expressing Inter-Distribution Dependencies

```
setup(...
```

```
    provides=['foomatic', 'washer.log (2.1)'],
```

```
    obsoletes=['foomatic (1.0, 1.1, 1.2)'],
```

```
    requires=['magicmod', 'puff.cmd (>=1.3)'],
```

```
    # setuptools only
```

```
    install_requires=[ 'magicmod', 'puff.cmd>=1.3' ],
```

```
    setup_requires=[ 'plughcmd' ],
```

```
    tests_require=[ 'deflea>=1.0' ],
```

```
    extras_require={ 'pdf': ['adobe'], 'ps': ['ghostscript'] }
```

```
)
```

setuptools enriches the approach taken to for declaring dependencies, providing a finer-grained set of keywords for different kinds of relations. These new keywords replace the *requires* keyword from *distutils*.

install_requires

A list of other distributions that need to be installed when this one is.

setup_requires

A list of distributions that need to be present in order for the setup script to run. Needed for *distutils* extensions. Distributions listed here are NOT installed onto the system but stashed locally to run the *setup.py* command.

tests_require

A list of distributions necessary for running your test suites. Distributions listed here are NOT installed onto the system but

stashed locally to run the *setup.py* command.

extras_require

A mapping of a feature name (optional features of your project) to a list of distributions that must be installed to support those features.

setuptools provides a powerful syntax for expressing requirements, either a list of strings or one long string broken across lines.

Any scripts in your project will be installed with wrappers that verify the availability of the specified dependencies at runtime, and ensure that the correct versions are added to `sys.path` (e.g. if multiple versions have been installed).

Supplying a MANIFEST for Source Distributions

- MANIFEST file (created if absent)
 - for extra files into source distro only
 - template (macros) in `MANIFEST.in`
- auto-included
 - referenced Python, scripts, C/C++ source
 - recognized test scripts (`test/test*.py`)
 - `README.txt`, `setup.py`, `setup.cfg`
- omitted by default
 - C/C++ header files (flaw)
 - the `build/` tree
 - version-control metadirs
- **setuptools:**
 - automatically adds files under version control
 - `MANIFEST.in` only for non-versioned files
 - different approach, mostly forget about it now

Just as with *distutils*, *setuptools* creates a MANIFEST file to specify a list of files to include in a source distribution.

However, *setuptools* tries to make the life of the developer easier by being smart and including all relevant files in your source tree. Relevancy is determined by whether the file is under CVS/Subversion control.

But when using *setuptools* be sure to ignore any part of the *distutils* documentation that deals with MANIFEST or how it's generated from `MANIFEST.in`; *setuptools* shields you from these issues and doesn't work the same way in any case. Unlike *distutils*, *setuptools* regenerates the source distribution manifest file every time you build a source distribution, and it builds it inside the project's `.egg-info` directory (`.egg-info/SOURCES.txt`), out of the way of your main project directory. You therefore need not worry about whether it is up-to-date or not.

Introduction to Package Servers

- public distribution repositories
 - a machine-readable registry format
 - with dependency information
- examples
 - CPAN (Perl) and PEAR (PHP)
 - Package Index (PyPI) or Cheeseshop
- what does it serve?
 - index server - *records pointing to websites*
 - upload server - *holds distributions*
 - link server - *scraped HTML server*

To make the existence of distributions visible to others in an automated form, suitable for dependency resolution, there are package servers.

For Perl and PHP, respectively, there are the CPAN and PEAR package servers. The one for Python is named the Cheeseshop or sometimes the Package Index (PyPI).

A package server may serve one or more kinds of information:

An index server holds records containing metadata about many different distributions, including a URL where the actual distribution files may be found. Index servers are cross-project in nature.

An upload server receives distribution files, both source and binary, along with metadata, from developers and makes them available for download from a centralized place.

A link server holds HTML about a specific or related set of projects, which has links sprinkled on it that point to actual distribution files. Link servers are usually browseable project sites, and are used in connection with buildout.

Package Servers: About The Cheeseshop

- PyPI is an index server and upload server
- *distutils* pushes up, never pulls down
- *setuptools* does both
- need an account to write to it
 - visit site and register
 - or do a pushup, get queried for ID
- obtaining its source (or use *basketweaver*)

The PyPI server is both an index server and an upload server. It is developer discretion whether to keep actual distributions on it or just rely upon the URL in the metadata to point to a project website.

distutils only knows how to push up to PyPI, both metadata with the "register" command and distributions (source and binary) with the "upload" command. Later we'll see how *setuptools* and *buildout* add the capability to pull from PyPI.

User accounts on PyPI can be obtained by visiting the site in a browser, or by pushing up metadata for a project and being prompted by *setup.py*.

The source to PyPI is available for running your own, such as behind a corporate firewall. The *basketweaver* package in the Cheeseshop looks like another good way to run your own package index server.

Package Servers: Posting Your Metadata

```
$ svn co http://www.demo.com/foo/trunk foo
$ cd foo
$ python setup.py register
  1. existing login
  2. register anew
  3. reset/email password
```

- entries keyed by (*projectname*, *version*)
- minimum metadata required

projectname, *version*, *URL*, *contact info*

Here we see use of the *register* command for pushing metadata for a distribution up to a package server, by default PyPI.

Sending data to a package server requires a username and password. *distutils* will prompt for an existing one, permission to create a new one or reset your password and have PyPI email a new, random one to you.

Within PyPI, entries are uniquely identified by the (*projectname*, *version*).

A package server such as PyPI, besides checking the correctness of metadata, enforces a minimum set of fields.

Package Servers: Stashing Credentials

- *register* asks to save auth info
- but NOT in a *distutils* config file
- instead in `$HOME/.pypirc`:

```
[server-login]
username:USERNAME
password:PASSWORD
repository:SOMEURL
```

- must use *setup.cfg* for register command

```
[register]
repository=http://www.democorp.com/
```

Upon completion, the *register* command asks if you want to save the username and password entered into a local configuration file.

This file is NOT one of the *distutils* configuration files but one specific to PyPI.

Or you can place the information in the file yourself.

Oddly, the "repository" field is only used with the *upload* command which we cover next, not the *register* command. To convince *register* to use a repository other than PyPI, add something like this to one of the *distutils* configuration files.

Package Servers: Posting Distributions

```
$ svn co http://www.demo.com/foo/trunk foo
$ cd foo
```

```
$ python setup.py sdist upload
$ python setup.py sdist --formats=gz tar,zip upload
$ python setup.py sdist upload --sign
$ python setup.py bdist --formats=rpm upload
```

- metadata sent as well
- new releases hide previous releases

The "upload" command is used to upload actual distribution files. Since there are many potential distribution flavors and formats, the choice of what to upload is given earlier in the command-line.

Uploads can be signed with a GnuPG key by adding the `--sign` option. There is also an `--identity` option that supplies a user ID to pass to the GnuPG tools.

To upload a binary distribution instead or in addition to a source distribution, use the `bdist` command.

Uploads can also be performed manually by visiting the PyPI website.

Submitting a distribution file automatically submits the metadata.

A new release of a package hides all previous releases, wrt listings and searches. You can manually override this by visiting the PyPI website, until the next submission of metadata.

Further Reading

- the Distutils-SIG and Mailing List
<http://www.python.org/sigs/distutils-sig/>
<http://mail.python.org/pipermail/distutils-sig/>
- "Distributing Python Modules" (guide for developers)
<http://docs.python.org/dist/dist.html>
- "Installing Python Modules" (guide for sysadmins)
<http://docs.python.org/inst/inst.html>
- Distutils Cookbook - Collection of Recipes
<http://wiki.python.org/moin/DistutilsCookbook>
- Community Wiki for Distutils (links to useful info)
<http://wiki.python.org/moin/DistUtils>
- Source to Python Package Index
<http://sourceforge.net/projects/pypi/>
- "Cleaning Up PyBlosxom Using Cheesecake"
<http://pycheesecake.org/wiki/CleaningUpPyBlosxom>