The Hitchhiker's Guide to Packaging" Documentation

Release 1.0

Tarek Ziadé

Jun 27, 2017

Contents

Warning: This document is currently under development. Thank you for your interest.

CHAPTER 1

Table of Contents

Quick Start

Here's how to create a new project, which we'll call **TowelStuff**.

1. Lay out your project

The smallest python project is two files. A *setup.py* file which describes the metadata about your project, and a file containing Python code to implement the functionality of your project.

In this example project we are going to add a little more to the project to provide the typical minimal layout of a project. We'll create a full Python package, a directory with an *__init__.py* file, called *towelstuff/*. This anticipates future growth as our project's source code is likely to grow beyond a single module file.

We'll also create a *README.txt* file describing an overview of your project, and a *LICENSE.txt* file containing the license of your project.

That's enough to start. There are a number of other types of files a project will often have, see the *Arranging your file and directory structure* for an example of a more fully fleshed out project.

Your project will now look like:

```
TowelStuff/
    LICENSE.txt
    README.txt
    setup.py
    towelstuff/
        __init__.py
```

2. Describe your project

The *setup.py* file is at the heart of a Python project. It describes all of the metadata about your project. There a quite a few fields you can add to a project to give it a rich set of metadata describing the project. However, there are only three

required fields: *name*, *version*, and *packages*. The *name* field must be unique if you wish to publish your package on the Python Package Index (PyPI). The *version* field keeps track of different releases of the project. The *packages* field describes where you've put the Python source code within your project.

Our initial *setup.py* will also include information about the license and will re-use the *README.txt* file for the *long_description* field. This will look like:

```
from distutils.core import setup

setup(
    name='TowelStuff',
    version='0.1dev',
    packages=['towelstuff',],
    license='Creative Commons Attribution-Noncommercial-Share Alike license',
    long_description=open('README.txt').read(),
)
```

3. Create your first release

In the *version* field, we specified *0.1dev*. This indicates that we are *developing* towards the *0.1* version. Right now there isn't any code in the project. After you've written enough code to make your first release worthwhile, you will change the version field to just *0.1*, dropping the *dev* marker.

To create a release, your source code needs to be packaged into a single archive file. This can be done with the *sdist* command:

```
$ python setup.py sdist
```

This will create a *dist* sub-directory in your project, and will wrap-up all of your project's source code files into a distribution file, a compressed archive file in the form of:

```
TowelStuff-0.1.tar.gz
```

The compressed archive format defaults to *.tar.gz* files on POSIX systems, and *.zip* files on Windows systems.

By default, Distutils does **not** include all files in your project's directory. Only the following files will be included by default:

- all Python source files implied by the py_modules and packages options
- all C source files mentioned in the ext_modules or libraries options
- scripts identified by the scripts option
- anything that looks like a test script: test/test*.py
- Top level files named: README.txt, README, setup.py, or setup.cfg

If you want to include additional files, then there are a couple options for including those files:

- Use a package which extends Distutils with more functionality. Setuptools and Distribute allow you to include all files checked into your version control system.
- Write a top-level MANIFEST.in file. This is a template file which specifies which files (and file patterns) should be included. (TO-DO: link to a MANIFEST.in document)

When you run the **sdist** command, a file named MANIFEST will be generated which contains a complete list of all files included. You can view this file to double check that you've setup your project to include all files correctly. Now your project is ready for a release. Before releasing, it's a good idea to double check to make sure that you have:

- The correct version number.

  While it's handy to append a *dev* marker to the version number during development, so that you can distinguish between code under development and a released version, you **never** want to publish a release with *dev* in the version name.

- All desired project files are included.

  Go over the MANIFEST file, or open the archive file generated by running the **sdist** command.

4. Register your package with the Python Package Index (PyPI)

The distribution file generated by running **sdist** can be published anywhere. There is a central index of all publically available Python projects maintained on the python.org web site named the *The Python Package Index (PyPI)*. This is where you will want to release your distribution if you intend to make your project public.

You will first have to visit that site, where you can register for an account. Project's are published on PyPI in the format of:

```
http://pypi.python.org/pypi/<projectname>
```

Your project will have to choose a name which is not already taken on PyPI. You can then claim your new project's name by registering the package by running the command:

```
$ python setup.py register
```

5. Upload your release, then grab your towel and save the Universe!

Now that you are happy that you can create a valid source distribution, it's time to upload the finished product to PyPI. We'll also create a *bdist_wininst* distribution file of our project, which will create a Windows installable file. There are a few different file formats that Python distributions can be created for. Each format must be specified when you run the upload command, or they won't be uploaded (even if you've previously built them previously). You should always upload a source distribution file. The other formats are optional, and will depend upon the needs of your anticipated user base:

```
$ python setup.py sdist bdist_wininst upload
```

At this point you should announce your package to the community!

Finally, in your *setup.py* you can make plans for your next release, by changing the *version* field to indicate which version you want to work towards next (e.g. *0.2dev*).

This *Quick Start* is a good brief introduction, but does not cover a lot of obscure use-cases. For more details, please see *Introduction to Packaging*, to gain a better understanding of the *Current State of Packaging*.

Introduction to Packaging

Abstract

This document describes the current state of packaging in Python using Distribution Utilities ("Distutils") and its extensions from the end-user's point-of-view, describing how to extend the capabilities of a standard Python installation by building packages and installing third-party packages, modules and extensions.

Python is known for it's "batteries included" philosophy and has a rich *standard library*. However, being a popular language, the number of third party packages is much larger than the number of *standard library* packages. So it eventually becomes necessary to discover how packages are *used*, *found* and *created* in Python.

It can be tedious to manually install extra packages that one needs or requires. Therefore, Python has a packaging system that allows people to distribute their programs and libraries in a standard format that makes it easy to install and use them. In addition to distributing a package, Python also provides a central service for contributing packages. This service is known as *The Python Package Index (PyPI)*. Information about *The Python Package Index (PyPI)* will be provided throughout this documentation. This allows a *developer* to distribute a package to the greater community with little effort.

This documentation aims to explain how to *install packages* and *create packages* for the end-user, while still providing references to advanced topics.

The Packaging Ecosystem

A Package

A *package* is simply a directory with an `__init__.py` file inside it. For example:

```
$ mkdir mypackage
$ cd mypackage
$ touch __init__.py
$ echo "# A Package" > __init__.py
$ cd ..
```

This creates a package that can be imported using the `import`. Example:

```
>>> import mypackage
>>> mypackage.__file__
'mypackage/__init__.py'
```

Discovering a Python Package

Using *packages* in the current working directory only works for small projects in most cases. Using the working directory as a package location usually becomes a problem when distributing packages for larger systems. Therefore, `distutils` was created to **install** packages into the `PYTHONPATH` with little difficulty. The `PYTHONPATH`, also `sys.path` in code, is a list of locations to look for Python packages. Example:

```
>>> import sys
>>> sys.path
['',
 '/usr/local/lib/python2.6',
 '/usr/local/lib/python2.6/site-packages',
 ...]
>>> import mypackage
>>> mypackage.__file__
'mypackage/__init__.py'
```

The first value, the null or empty string, in `sys.path` is the current working directory, which is what allows the packages in the current working directory to be found.

Note: Your `PYTHONPATH` values will likely be different from those displayed.

Explicitly Including a Package Location

The convention way of manually installing packages is to put them in the `.../site-packages/` directory. But one may want to install Python modules into some arbitrary directory. For example, your site may have a convention of keeping all software related to the web server application under `/www`. Add-on Python modules might then belong in `/www/pythonx.y/`, and in order to import them, this directory must be added to `sys.path`. There are several different ways to add the directory.

Note: TODO Better define where the `.../site-packages/` directory is located.

The most convenient way is to add a path configuration file to a directory that's already in Python's path, which could be the `.../site-packages/` directory. Path configuration files have an extension of `.pth`, and each line must contain a single path that will be appended to `sys.path`. (Because the new paths are appended to `sys.path`, modules in the added directories will not override standard modules. This means you can't use this mechanism for installing fixed versions of standard modules.)

Paths can be absolute or relative, in which case they're relative to the directory containing the `.pth` file. See the documentation of the `site` module for more information.

In addition there are two environment variables that can modify `sys.path`. `PYTHONHOME` sets an alternate value for the prefix of the Python installation. For example, if `PYTHONHOME` is set to `/www/python/lib/python2.6/`, the search path will be set to `['', '/www/python/lib/python2.6/', ...]`.

The `PYTHONPATH` variable can be set to a list of paths that will be added to the beginning of `sys.path`. For example, if `PYTHONPATH` is set to `/www/python:/opt/py`, the search path will begin with `['', '/www/python', '/opt/py', ...]`.

Note: Directories must exist in order to be added to `sys.path`. The `site` module removes paths that don't exist.

Finally, `sys.path` is just a regular Python list, so any Python application can modify it by adding or removing entries.

Note: The zc.buildout package modifies the sys.path in order to include all packages relative to a buildout. The `zc.buildout` package is often used to build large projects that have external build requirements.

Python file layout

A Python installation has a `site-packages` directory inside the module directory. This directory is where user installed packages are dropped. A `.pth` file in this directory is maintained which contains paths to the directories where the extra packages are installed.

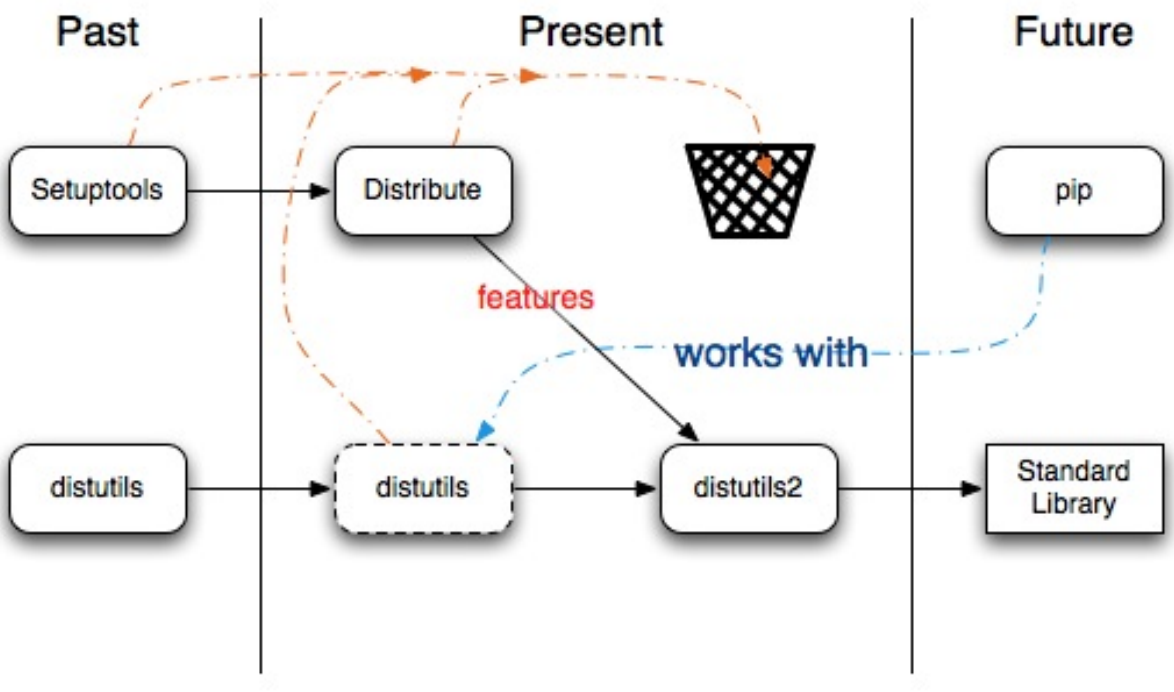
Note: For details on the `.pth` file, please refer to modifying Python's search path. In short, when a new package is installed using `distutils` or one of its extenders, the contents of the package are dropped into the `site-packages` directory and then the name of the new package directory is added to a `.pth` file. This allows Python upon the next startup to see the new package.

Benefits of packaging

While it's possible to unpack *tarballs* and manually put them into your Python installation directories (see *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*)
- **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)*)
- **Uninstall** Package managers can give you push button ways of removing a package from your environment. (see *Pip Installs Python (Pip)*)
- **Search** Find packages by searching a *package index* for specific terminology. (see *Pip Installs Python (Pip)*)

Current State of Packaging



The `distutils` modules is part of the *standard library* and will be until Python 3.3. The `distutils` module will be discontinued in Python 3.3. The `distutils2` (note the number two) will be backwards compatible for Python 2.4 onward; and will be part of the *standard library* in Python 3.3.

The `distutils` module provides the basics for packaging Python. Unfortunately, the `distutils` module is riddled with problems, which is why a small group of python developers are working on `distutils2`. However, until `distutils2` is complete it is recommended that the *Developer* either use pure `distutils` or the **'Distribute package <distribute_info_>'_** for packaging Python software.

In the mean time, if a package requires the `setuptools` package, it is our recommendation that you install the *Distribute* package, which provides a more up to date version of `setuptools` than does the original Setuptools

package.

In the *future* `distutils2` will replace `setuptools` and `distutils`, which will also remove the need for *Distribute*. And as stated before `distutils` will be removed from the *standard library*. For more information, please refer to the *Future of Packaging*.

Warning: Please use the *Distribute package* rather than the Setuptools package because there are problems in this package that can and will not be fixed.

Creating a Micro-Ecosystem with `virtualenv`

Here we have a small digression to briefly discuss *Virtual Environments*, which will be covered later in this guide. In most situations, the `site-packages` directory is part of the system Python installation and not writable by unprivileged users. Also, it's useful to have a solid reliable installation of the language which we can use. Experimental packages shouldn't be mixed with the stable ones if we want to keep this quality. In order to achieve this, most Python developers use the *virtualenv* package which allows people to create a *virtual* installation of Python. This replicates the `site-packages` directory in an user writable area. The `site-packages` directory located in the *Virtual Environments* is in addition to the global one. While orthogonal to the whole package installation process, it's an extremely useful and natural way to work and so the whole thing will be mentioned again. The installation and usage of `virtualenv` is covered in *Virtual Environments* document.

Installing the Package Tools

In the *current state of packaging* in Python, one needs a set of tools to easily manipulate the packaging ecosystem. There are two tools in particular that are extremely handy in the current ecosystem. There is a third tool, *Virtual Environments*, that will be discussed later in this documentation that will assist in isolating a packaging ecosystem from the global one. The combination of these tools will help to find, install and uninstall packages.

Distribute

Distribute is a collection of enhancements to the Python standard library module: `distutils` (for Python 2.3.5 and up on most platforms; 64-bit platforms require a minimum of Python 2.4) that allows you to more easily build and distribute Python packages, especially ones that have dependencies on other packages.

Distribute was created because the Setuptools package is no longer maintained. Third-party packages will likely require `setuptools`, which is provided by the Distribute package. Therefore, anytime time a packages depends on the Setuptools package, Distribute will step in to say it already provides the `setuptools` module.

See also:

Distribute documentation.

Installation Instructions

Distribute can be installed using the distribute_setup.py script. It can also be installed using easy_install, pip, the source tarball, or the egg distribution. distribute_setup.py is the simplest and preferred way to install Distribute on all systems.

Download distribute_setup.py and execute it, using the Python interpreter of your choice.

From the *nix shell you can do:

```
$ wget http://python-distribute.org/distribute_setup.py
$ python distribute_setup.py
```

Note: For those on Mac OS X, you can use `curl -O <url>` instead of `wget`.

See also:

The development version of Distribute can be found at: http://bitbucket.org/tarek/distribute/

Pip Installs Python (Pip)

Pip is an installer for Python packages written by Ian Bicking. It can install packages, list installed packages, upgrade packages and uninstall packages.

The `pip` application is a replacement for easy_install. It uses mostly the same techniques for finding packages, so packages that were made easy_installable should be pip-installable as well.

See also:

Pip Documentation and Pip PyPI description

Note: ??? Pip requirements

Installing Pip

The Pip installer can be installed using the source tarball or using `easy_install`. The source tarball is the recommended method of installation.

The latest version of the source tarball can be obtained from PyPI:

```
$ wget http://pypi.python.org/packages/source/p/pip/pip-0.7.2.tar.gz
$ tar xzf pip-0.7.2.tar.gz
$ cd pip-0.7.2
$ python setup.py install
```

Or the `easy_install` application can be used:

```
$ easy_install pip
```

The `pip` application is now installed.

Note: `pip` is complementary with *Virtual Environments*, and it is encouraged that you use *Virtual Environments* to isolate your installation.

Installing a package

Let's install the Markdown package:

```
$ pip install Markdown
```

Markdown is now installed; you can import and use it:

```
$ python -c "import markdown; print markdown.markdown('**Excellent**')"
```

Listing installed packages

To list installed packages and versions, use the `freeze` command:

```
$ pip freeze
Markdown==2.0.3
wsgiref==0.1.2
```

Note: The `wsgiref` package is a part of the Python standard library. Currently it is the only standard library package that includes package metadata, so it is the only standard library package whose presence pip reports.

Installing specific versions

You can also give pip a version specifier for a package using one or more of ==, >=, >, <, <=:

```
$ pip install 'Markdown<2.0'
```

This will find your current installation of Markdown 2.0.3, automatically uninstall it, and install Markdown 1.7 (the latest version in the 1.x series) in its place. You can even combine version specifiers with a comma:

```
$ pip install 'Markdown>2.0,<2.0.3'
```

Upgrading

If you want to upgrade a package to its most recent available version, use the `-U` or `--upgrade` flag:

```
$ pip install -U Markdown
```

Uninstalling

Now let's uninstall Markdown:

```
$ pip uninstall Markdown
```

After showing you which files/directories will be removed and requesting confirmation, pip will uninstall everything installed by the Markdown package.

Note: Pip inside a *Virtual Environment* will only uninstall packages installed within that virtual environment. For instance, if you try to `pip uninstall wsgiref` it will refuse, because the reference is within the global Python's standard library.

Using Packages

Finding Packages

Note: TODO I'm mentioning *The Python Package Index (PyPI)* everywhere. Should probably move this up in the documentation instead of explaining it over and over.

How does one find a package. Well, the simple answer is to check *The Python Package Index (PyPI)* first. The other options are:

- To do a simple web search with google.com, yahoo, etc.
- Ask around the python community using *IRC*

We will cover the *The Python Package Index (PyPI)* later in the documentation.

Installing from other sources

When using the *Pip Installs Python (Pip)* application, how does it know what to install when you run `pip install Markdown`? By default, it checks the *The Python Package Index (PyPI)* for a package of that name. In this case, it found one; but what if you want to install a package that hasn't been uploaded to PyPI?

You have several options:

- *Installing from a tarball*
- *Installing from a Version Control System (VCS)*
- *Add URLs to search for links*

Installing from a tarball

You can install directly from a tarball or zip file, as long as there is a working `setup.py` file in the root directory of the unzipped contents:

```
$ pip install path/to/mypackage.tgz
```

You can also install from a tarball/zip file over the network:

```
$ pip install http://dist.repoze.org/PIL-1.1.6.tar.gz
```

Installing from a Version Control System (VCS)

Using the `--editable` or `-e` option, pip has the capability to install directly from a version control repository (it currently supports Subversion, Mercurial, Git, and Bazaar):

```
$ pip install -e svn+http://svn.colorstudy.com/INITools/trunk#egg=initools-dev
```

This option shells out to the command-line client for each respective VCS, so you must have the VCS installed on your system. The repo URL must begin with `svn+` (or `hg+`, `git+`, or `bzr+`) and end with `#egg=packagename`; otherwise, pip supports the same URL formats and wire protocols supported by the VCS itself.

Pip will checkout the source repo into a `src/` directory inside the virtualenv (i.e. `pip_test_env/src/initools-dev`), and then run `python setup.py develop` in that source repo. This "links" the code directly

from the repo into the virtualenv's `site-packages` directory (by adding the repo directory into `easy-install.pth`), so changes you make in the source checkout are effective immediately.

If you already have a local VCS checkout you want to keep using, you can just use `pip install -e path/to/repo` to install it "editable" in the same way.

Add URLs to search for links

You can use the `-f` or `--find-links` option to add another URL pip should search for links to the package. If you dump some package tarballs in a webserver directory and turn on automatic indexing, you can point pip at that index page and install any of those packages, assuming you named the files in the pattern `packagename-version.ext`.

For example, if you upload a tarball `MyApp-1.0.tgz` to a `my-packages` directory on your webserver, and make sure indexing is on for that directory, you can run:

```
$ pip install MyApp -f http://www.example.com/my-packages/
```

Running your own package index

If you want more of the features provided by PyPI (including the ability to upload packages with `python setup.py sdist upload`), you can run software such as chishop, which implements the PyPI API, on your own server. Then you can use pip's `-i` (or `--index-url`) or `--extra-index-url` options to point it at your index.

For instance, if you set up your own index at http://www.example.com/chishop/, you might run:

```
$ pip install MyPrivateApp -i http://www.example.com/chishop/simple/
```

If you use `-i` pip won't check PyPI, only the index you provide. If you are installing multiple packages at once, some from your index and some from PyPI, you may want to use `--extra-index-url` instead, so pip will check both indexes.

Creating a Package

Basics: Creating and Distributing Distributions

If you have some useful Python *modules* that you think others might benefit from, but aren't sure how to go about packaging them up and distributing them, then this short document is for you. By the end of it, you'll be a contributor to the *The Python Package Index (PyPI)*.

For a more detailed look at packaging a larger project, see this example.

Let's begin.

Background

Suppose you've written a couple modules to help you keep track of your towel (`location.py` and `utils.py`), and you'd like to share them. First thing to do is come up with a CamelCase project name for them. Let's go with "TowelStuff" since it seems appropriate and also it has not yet been used on the *The Python Package Index (PyPI)*.

Arranging your file and directory structure

"TowelStuff" will be the name of our project as well as the name of our *distribution*. We should also come up with a *package* name within which our modules will reside (to avoid naming conflicts with other modules). For this example, there's only one package, so let's reuse the project name and go with "towelstuff". Make the layout of your project directory (described below) look like this:

```
TowelStuff/
    bin/
    CHANGES.txt
    docs/
    LICENSE.txt
    MANIFEST.in
    README.txt
    setup.py
    towelstuff/
        __init__.py
        location.py
        utils.py
        test/
            __init__.py
            test_location.py
            test_utils.py
```

Here's what you should do for each of those listed above:

- Put into `bin` any scripts you've written that use your `towelstuff` package and which you think would be useful for your users. If you don't have any, then remove the `bin` directory.
- For now, the `CHANGES.txt` file should only contain:

  ```
  v<version>, <date> -- Initial release.
  ```

  since this is your very first version (version number will be described below) and there are no changes to report.
- The `docs` dir should contain any design docs, implementation notes, a FAQ, or any other docs you've written. For now, stick to plain text files ending in ".txt". This author (JohnMG) likes to use Pandoc's Markdown, but many Pythoneers use *reStructuredText*.
- The `LICENSE.txt` file is often just a copy/paste of your license of choice. We recommend going with a commonly-used license, such as the GPL, BSD, or MIT.
- The `MANIFEST.in` file should contain this:

  ```
  include *.txt
  recursive-include docs *.txt
  ```

- The `README.txt` file should be written in *reST* so that the PyPI can use it to generate your project's PyPI page. Here's a 10-second intro to reST that you might use to start with:

  ```
  ===========
  Towel Stuff
  ===========

  Towel Stuff provides such and such and so and so. You might find
  it most useful for tasks involving <x> and also <y>. Typical usage
  often looks like this::

      #!/usr/bin/env python
  ```

```
    from towelstuff import location
    from towelstuff import utils

    if utils.has_towel():
        print "Your towel is located:", location.where_is_my_towel()

(Note the double-colon and 4-space indent formatting above.)

Paragraphs are separated by blank lines. *Italics*, **bold**,
and ``monospace`` look like this.


A Section
=========

Lists look like this:

* First

* Second. Can be multiple lines
  but must be indented properly.

A Sub-Section
-------------

Numbered lists look like you'd expect:

1. hi there

2. must be going

Urls are http://like.this and links can be
written `like this <http://www.example.com/foo/bar>`_.
```

You might also consider adding a "Contributors" section and/or a "Thanks also to" section to list the names of people who've helped.

By the way, to see how the above `README.txt` looks rendered in html, see the TowelStuff project at the PyPI.

- `setup.py` – Create this file and make it look like this:

```
from distutils.core import setup

setup(
    name='TowelStuff',
    version='0.1.0',
    author='J. Random Hacker',
    author_email='jrh@example.com',
    packages=['towelstuff', 'towelstuff.test'],
    scripts=['bin/stowe-towels.py','bin/wash-towels.py'],
    url='http://pypi.python.org/pypi/TowelStuff/',
    license='LICENSE.txt',
    description='Useful towel-related stuff.',
    long_description=open('README.txt').read(),
    install_requires=[
        "Django >= 1.1.1",
        "caldav == 0.1.4",
    ],
```

```
)
```

but, of course, replace the towel stuff with your own project and package names. For more details about picking version numbers, see versioning, but '0.1.0' will work just fine for a first release (this is using the common "major.minor.micro" numbering convention).

Use the install_requires argument to automatically install dependencies when your package will be installed and include information about dependencies (so that package management tools like Pip can use the information). It takes a string or list of strings containing requirement specifiers.

The syntax consists of a project's PyPI name, optionally followed by a comma-separated list of version specifiers. Modern packaging tools implement version specifiers syntax described in PEP 345 and resolve version comparison in compliance with PEP 386.

If you have no scripts to distribute (and thus no `bin` dir), you can remove the above line which begins with "scripts".

- Inside the `towelstuff` directory, `__init__.py` can be empty. Likewise, inside `towelstuff/test`, *that* `__init__.py` can be empty as well. If you have no tests written yet, you can leave the two other module files in `towelstuff/test` empty for now too. When writing your tests, use the standard `unittest` module.

For our example, TowelStuff does not depend upon any other distributions (it only depends upon what's already in the Python standard library). To specify dependencies upon other distributions, see the more detailed example.

Creating your distribution file

Create your distribution file like so:

```
$ cd path/to/TowelStuff
$ python setup.py sdist
```

Running that last command will create a `MANIFEST` file in your project directory, and also a `dist` and `build` directory. Inside that `dist` directory is the distribution that you'll be uploading to the PyPI. In our case, the distribution file will be named `TowelStuff-0.1.0.tgz`. Feel free to poke around in the `dist` directory to look at your distribution.

Uploading your distribution file

Before uploading you first need to create an account at http://pypi.python.org/pypi . Once that's complete, register your distribution at the PyPI like so:

```
$ cd path/to/TowelStuff
$ python setup.py register
```

Use your existing login (choice #1). It will prompt you to save the login info for future use (to which I agree). Then upload:

```
$ python setup.py sdist upload
```

This builds the distribution one last time and then uploads it.

Thanks for your contribution!

Updating your distribution

Down the road, after you've made updates to your distribution and wish to make a new release:

1. increment the version number in your `setup.py` file,
2. update your `CHANGES.txt` file,
3. if necessary, update the "Contributors" and "Thanks also to" sections of your `README.txt` file.
4. run `python setup.py sdist upload` again.

Entry points

Entry points are a Setuptools/Distribute feature that's really handy in one specific case: register something under a specific key in package A that package B can query for.

Distribute itself uses it. If you're packaging your project up properly, you've probably used the `console_scripts` entry point:

```
setup(name='zest.releaser',
      ...
      entry_points={
          'console_scripts':
              ['release = zest.releaser.release:main',
               'prerelease = zest.releaser.prerelease:main',
               ]}
      )
```

`console_scripts` is an entry point that Setuptools looks up. It looks up all entry points registered under the name console_scripts and uses that information to generate scripts. In the above example that'd be a bin/release script that runs the main() method in zest/releaser/release.py.

You can use that for your own extension mechanism. For `zest.releaser` I needed some extension mechanism. I wanted to be able to do extra things on prerelease/release/postrelease time.

- Downloading an external javascript library into a package that cannot be stored in (zope's) svn repository directly due to licensing issues. Before packaging and releasing it, that is. Automatically so you don't forget it.
- Uploading a version.cfg to `scp://somewhere/kgs/ourmainproduct-version.cfg` after making a release to use it as a so-called "known good set" (KGS).
- Possibly modifying values (like a commit message) inside zest.releaser itself while doing a release. (I do get modification requests from time to time "hey, can you make x and y configurable"). So now every zest.releaser step (prerelease, release, postrelease) is splitted in two: a calculation phase and a "doing" phase. The results of the first phase are stored in a dict that gets used in the second phase. And you can register an entry point that gets passed that dict so you can modify it. See the entry point documentation of zest.releaser for details.

An entry point for zest.releaser is configured like this in your setup.py:

```
entry_points={
    'console_scripts':
        ['myscript = my.package.scripts:main'],
    'zest.releaser.prereleaser.middle':
        ['dosomething = my.package.some:some_entrypoint, ]
}
```

Replace prereleaser and middle in zest.releaser.prereleaser.middle with prerelease/release/postrelease and before/middle/after where needed. (For this specific zest.releaser example).

Now, how to use this in your program? The best way is to show a quick example from zest.releaser where we query and use one of our entry points:

```
import pkg_resources

...
def run_entry_point(data):
    # Note: data is zest.releaser specific: we want to pass
    # something to the plugin group = 'zest.releaser.prerelease.middle'

    for entrypoint in pkg_resources.iter_entry_points(group=group):
        # Grab the function that is the actual plugin.
        plugin = entrypoint.load() # Call the plugin
        plugin(data)
```

So: pretty easy and simple way to allow other packages to register something that you want to know. Extra plugins, extra render methods, extra functionality you want to register in your web application, etcetera.

Packaging for a Particular Operating System (OS)

General Packaging Guidelines for Unix

General Packaging Guidelines for Windows

Contribute Your Package to the World

The Python Package Index (PyPI)

The Python Package Index (PyPI), formerly known as the Cheeseshop, is to Python what *CPAN* is to *Perl*: a central repository of projects and *distributions*.

Note: XXX put here the Monty Python Cheeseshop extract

PyPI is located at http://pypi.python.org and contains more than 9000 projects registered by developers from the community.

Tools like *Pip* or zc.buildout are using PyPI as the default location to find *distributions* to install. When `pip install Foo` is called, it will browse PyPI to find the latest available version of the `Foo` project using *The Simple Index Protocol*. If it finds it, it will download it and install it automatically.

This automatic installation ala *Pip Installs Python (Pip)* will work only if the distribution is using the Distutils-based structure and contains a `setup.py` file.

This means that any serious Python project should use Distutils (see the basics for doing this) and should at the minimum be registered at PyPI. Uploading releases there is also a good practice.

Note: ??? Why should a project at a minimum register at PyPI? good practice, yes; at a minimum, no

---

Registering projects

Registering a project at PyPI is done by using *Distutils*' `register` command. This command performs a simple HTTP post using basic authentication with the login name and password stored in the `.pypirc` file located in your

home directory. This login has to be registered at PyPI, so you should go there and create an account before running `register` for the first time.

Another option is to call `register` once on any Distutils-based project. It will register the new account for you and all you'll have to do is to reply to the confirmation e-mail PyPI will send you:

```
$ python setup.py register
running register
warning: register: missing required meta-data: version
We need to know who you are, so please choose either:
1. use your existing login,
2. register as a new user,
3. have the server generate a new password for you (and email it to you), or
4. quit
Your selection [default 1]:
...
```

Once this is done, `register` will ask you if you want to save your login information in the `.pypirc` file. By default, this will store the login name **and** the password:

```
[distutils]
index-servers =
    pypi

[pypi]
username:tarek
password:sigourney_as_an_avatar_is_scary
```

For security reasons, starting at Python 2.6, you can remove the password from the file if you want `register` to prompt you to type it everytime.

Note: A recent GSoC project called Keyring was created in 2009 in order to use any available system keyring like KWallet or KeyChain to store the PyPI password. The project exists and may be used in Distutils by the `register` and `upload` commands.

See: http://pypi.python.org/pypi/keyring

Once your account is ready, registering it at PyPI will create a new web page there, using the metadata fields of your project.

- **name** will be used as the unique identifier in PyPI. For example, if your project's name is `Foo`, its page will be located at *http://pypi.python.org/pypi/foo*. The rule at PyPI is *first in, first served*, meaning that once you have registered the *Foo* project, your login is the owner of the PyPI *foo* identifier and no one else (unless you authorize them) will be able to register a project under that name.
- **long_description** will be used to fill that page. An Html to *reStructuredText* rendered will be used on the field before it is displayed.

A good practice is to use reST, and make sure your `long_description` field doesn't contain any reStructuredText syntax error. See rest_example for a quick introduction on how to write a reST compatible field.

To perform a check, install `docutils` by using Pip (`pip install docutils`) and run:

```
$ python setup.py --long-description | rst2html.py > /dev/null
```

Under Windows, make sure the sys.prefix + 'Scripts/' path is in the `PATH` environment variable and run:

```
$ python.exe setup.py --long-description | rst2html.py > dummy.html
```

If your text contains any reST error or warning, they will be displayed.

Starting at Python 2.7, you can use the `check` command instead of calling the `rst2html.py` script, as long as `docutils` is installed:

```
$ python setup.py check
```

The check command will check that all fields are compliant before you register the project at PyPI.

Uploading distributions

PyPI also allows developers to upload their project's distributions. This can be done manually via a web form, but also through Distutils by using the `upload` command.

This command will upload freshly created archives via HTTP to PyPI. The usual way to perform this is to call `upload` right after `register` and the commands used to create archives. For instance, to upload a source distribution and update the project's page, one may do:

```
$ python setup.py register sdist upload
```

Note that you can upload several archives in one step if wanted:

```
$ python setup.py register sdist bdist upload
```

A good practice when uploading distributions at PyPI is to always upload the source distribution, unless your project is not open source of course. Binary distribution are optional especially if your project doesn't contain any extension to be compiled. This will help automatic installers like Pip to get and install your project on any platform. Uploading only a binary distribution will restrict automatic installation to the platform and Python version it was compiled with.

The Simple Index protocol

Besides its web pages, PyPI provides a tree structure at http://pypi.python.org/simple called the `Simple Index`. This structure allows installers like Pip to look for distribution archives.

For example, calling:

```
$ pip install distribute
```

will look at the `http://pypi.python.org/simple/distribute` page, which is a list of URLs for the Distribute project. These URLs include:

- the Distribute archives uploaded at PyPI
- a link to the project's home page
- extra links contained in the project's description fields

The Package Index follows these rules (taken from Setuptools' documentation):

1. Its pages are in HTML
2. Individual project version pages' URLs must be of the form `base/projectname/version`, where `base` is the package index's base URL. The base URL for PyPI is : `http://pypi.python.org/simple`.
3. Omitting the `/version` part of a project page's URL (but keeping the trailing `/`) should result in a page that is either:

(a) The single active version of that project, as though the version had been explicitly included, OR

(b) A page with links to all of the active version pages for that project.

Depending on the project's configuration, PyPI will display a) or b).

4. Individual project version pages should contain direct links to downloadable distributions where possible. The project's "long_description" field may contain URLs that will be displayed.
5. Where available, MD5 information should be added to download URLs by appending a fragment identifier of the form `#md5=...`, where `...` is the 32-character hex MD5 digest.
6. Individual project version pages should identify any "homepage" or "download" URLs using `rel="homepage"` and `rel="download"` attributes on the HTML elements linking to those URLs. Use of these attributes will cause EasyInstall to always follow the provided links, unless it can be determined by inspection that they are downloadable distributions. If the links are not to downloadable distributions, they are retrieved, and if they are HTML, they are scanned for download links. They are *not* scanned for additional "homepage" or "download" links, as these are only processed for pages that are part of a package index site.
7. The root URL of the index, if retrieved with a trailing `/`, must result in a page containing links to *all* projects' active version pages.

   (Note: This requirement is a workaround for the absence of case-insensitive `safe_name()` matching of project names in URL paths. If project names are matched in this fashion (e.g. via the PyPI server, mod_rewrite, or a similar mechanism), then it is not necessary to include this all-packages listing page.)
8. If a package index is accessed via a `file://` URL, then EasyInstall will automatically use `index.html` files, if present, when trying to read a directory with a trailing `/` on the URL.

The XML-RPC interface

Note: XXX pointer to PyPI developer doc + demo with yolk

Limitations of the system

Within the last few years, PyPI has become a first class citizen in the Python development ecosystem. For instance Plone developers that are using `zc.buildout` to install Plone-based applications are making hundreds of calls to PyPI each time they build their projects. That's because many web applications are using hundreds of small Python distributions nowadays.

This has a bad side effect: if PyPI is down, it becomes impossible to build some applications. PyPI acts as a `Single Point of Failure` (SPOF).

The PyPI server is quite robust and its uptime is probably around 99.99%, although people hit the SPOF problem once or twice per year. The community has started to develop strategies to avoid it:

- `zc.buildout` comes with a cache that will keep a copy of downloaded archives on the local system. This avoids getting packages at PyPI when they were already downloaded once.
- the collective.eggproxy project provides a cache server that acts like a proxy between developers and PyPI, and caches PyPI files like `zc.buildout` does.
- the z3c.pypimirror project is a mirroring script that syncs a local copy of the simple index so tools can use it instead of calling PyPI.

In the last year, PEP 381 was added to describe a mirroring protocol and set up an official "ring" of PyPI mirrors. This system should be ready by the end of the year 2010, and will provide a DNS entry under `mirrors.pypi.org`

listing all mirrors IPs. This will allow tools like *Pip Installs Python (Pip)* to list all mirrors and pick the closest one, or fall back in case a server is down.

Private and secondary PyPI servers

Note: XXX explain here that other servers can have a pypi feature (like plone.org, or private)

Note: XXX explain that pip can point to any pypi-like server

Note: XXX conclude on the multiple-index merge idea

Virtual Environments

Getting Started

Creating a virtualenv

We'll be using virtualenv so our installation experiments are contained and don't modify your system Python environment. If you aren't already familiar with virtualenv, you may want to read up on it first.

Create a virtualenv:

```
$ virtualenv --no-site-packages pip_test_env
```

We use the `--no-site-packages` flag to prevent this virtualenv from "seeing" your global Python "site-packages" directory, so that our experiments aren't confused by any Python packages you happen to already have installed globally.

Recent versions of virtualenv (1.4+) automatically install pip for you inside the virtualenv (there will already be a `pip` script in `pip_test_env/bin/`). If you are using a pre-1.4 virtualenv, run `pip_test_env/bin/easy_install pip` to install pip in the virtual environment.

Note: If you are using Windows, executable scripts in the virtualenv will be located at `pip_test_env\Scripts\` rather than `pip_test_env/bin/`. Just replace all occurrences of the latter with the former.

Let's "activate" the virtualenv to put `pip_test_env/bin` on our default PATH, so we can just type `pip` instead of `pip_test_env/bin/pip`:

```
$ . pip_test_env/bin/activate
```

Note: The leading dot is important. Without it, a subshell is spawned and only that subshell gets the virtualenv activated. The leading dot tells the shell to run the *activate* script in the current shell.

On Windows, this is:

```
$ pip_test_env\Scripts\activate.bat
```

In either case, your shell prompt should now begin with `(pip_test_env)` to remind you that a virtualenv is activated. When you are done working with this virtualenv type `deactivate` to remove its bin directory from your PATH.

Specifications

Naming Specification

Versioning Specification

How to version your Python projects

The version of a package is defined in the `setup.py` file. A `setup.py` looks something like this:

```
from distutils.core import setup

setup(name='MyProject', version='1.0', author='Tarek',
      author_email='tarek@ziade.org',
      url='http://example.com',)
```

And this gives you the power to register and upload your project to *The Python Package Index (PyPI)*, as simply as:

```
$ python setup.py register sdist upload
```

Your project is then added in PyPI, among over 9000 other projects and people can start using it, by downloading the archive you have created with the `sdist` command.

With `pip` they can even install it with a simple command, ala `apt-get`:

```
$ pip install MyProject
```

Why you need version numbers?

As soon as you start publishing your project to the world, you need to version it. For example, the first version of your software can be `1.0`.

Everytime you are releasing a new version with new features and bugfixes, raising the version number will let your end users know that it is a newer version. This is usally done by incrementing the version, so your next version could be `1.1`.

Once this new version is made available to the world on PyPI, people will be able to install it, for example using `pip`:

```
$ pip install --upgrade MyProject
```

The `upgrade` option here means that Pip will look for all published versions of `MyProject` on PyPI and see which one is the latest, then upgrade your system to that version if you are not up-to-date.

Now imagine that you have introduced a small bug in `1.1` that makes your project unusable on Windows. You are working on it, but you know it will take you some time to resolve it.

The best strategy here is to tell you Windows user to stick with `1.0` until you have fixed the issue. They can downgrade to `1.0` because it is still available on PyPI:

```
$ pip install MyProject==1.0
```

This is possible because `pip` is able to sort the various versions of your project, as long as it follows a standard version scheme.

Standard versioning schemes

The two most commone schemes used to version a software are **date-based** schemes and **sequence-based** schemes.

Date scheme

Some projects use dates for the version numbers. That was the case for Wine before it started using a sequence-based scheme. A date scheme is usually using a `YYYY-MM-DD` form so versions can be sorted alphanumerically:

- 2009-08-10
- 2005-02-03
- etc.

This versioning scheme has a few limitations:

- you need to add hours to the scheme if you do more than one release per day.
- if you have maintenance branches over older releases, the overall order will not work anymore: a maintenance release of an older version will appear to be newer than a recent release.

Although, date schemes are commonly used as extra markers in versions schemes.

Sequence-based scheme

The most common scheme is the sequence-based scheme, where each version is a sequence of numerical values usually separated by dots (.):

- 1.0
- 1.1
- 1.0.1
- 1.3.4

This scheme removes the limitations we have seen with date-based schemes since you can release maintenance versions and keep the proper order: versions are ordered by comparing alphanumerically each segment of the version.

The most frequent sequence-based scheme is:

```
MAJOR.MINOR[.MICRO]
```

where `MAJOR` designates a major revision number for the software, like `2` or `3` for Python. Usually, raising a major revision number means that you are adding a lot of features, breaking backward-compatibility or drastically changing the APIs or ABIs.

`MINOR` usually groups moderate changes to the software like bug fixes or minor improvements. Most of the time, end users can upgrade with no risks their software to a new minor release. In case an API changes, the end users will be notified with deprecation warnings. In other words, API and ABI stability is usually a promise between two minor releases.

Some softwares use a third level: `MICRO`. This level is used when the release cycle of minor release is quite long. In that case, micro releases are dedicated to bug fixes.

Choosing between a major-based scheme and a micro-based one is really a matter of taste. The most important thing is to document how your version scheme works and what your end users should expect when you release a new version. So be sure to define your release cycle properly before you start releasing to the wild !

Development releases

When working for the next version of your application, you might need to release a development version in order to share it with other developers. Many projects provide *nightly builds*, which are daily snapshots of the code repository people can install to try out a cutting edge version.

Like regular versions, development versions have to be numbered so they can be sorted and compared with any other version. The simplest way to perform this numbering is to use the current repository version number when creating the release. The next version is suffixed by a *dev* number. Of course this implies that your code lives in a repository, like Subversion or Mercurial.

Examples:

- 1.2.dev1234 : a development version of the future 1.2 release. 1234 is the current repository version.
- 1.3a2.dev12 : a development version of the future 1.3 alpha 2 release. 12 is the current repository version.

Package managers that will install those versions will need to sort them correctly: 1.2.dev1234 < 1.2 < 1.3a2.dev12 < 1.3a2 < 1.3

Here's an example of a `setup.py` file in a project that lives in a Mercurial repository, that will automatically generate the right development version when creating a source distribution.:

```
from distutils.command.sdist import sdist
import os

class sdist_hg(sdist):

    user_options = sdist.user_options + [
             ('dev', None, "Add a dev marker")
             ]

    def initialize_options(self):
        sdist.initialize_options(self)
        self.dev = 0

    def run(self):
        if self.dev:
            suffix = '.dev%d' % self.get_tip_revision()
            self.distribution.metadata.version += suffix
        sdist.run(self)

    def get_tip_revision(self, path=os.getcwd()):
        from mercurial.hg import repository
        from mercurial.ui import ui
        from mercurial import node
        repo = repository(ui(), path)
        tip = repo.changelog.tip()
        return repo.changelog.rev(tip)

setup(name='MyProject',
      version='1.0',
```

```
    packages=['package'],
    cmdclass={'sdist': sdist_hg})
```

This development marker will be added when the `dev` option is used.

Pre-releases

Major versions of a software are often preceded by pre-releases. These previews are comparable to development versions and are released for testing purposes. Publishing them will let your end-users try them out and send you valuable feedback before the *final* version is published. It also helps avoiding releases of *brown-bag* versions. A brown-bag version is a version that is badly broken and that cannot be used by some of all of the end users.

Usually, pre-releases are organized in three phases:

- **alpha** releases: early pre-releases. A lot of changes can occur between alphas and the final release, like feature additions or refactorings. But they are minor changes and the software should stay pretty unchanged by the time the first beta is reached.
- **beta** releases: at this stage, no new features are added and developers are tracking remaning bugs.
- **release candidate** releases: a release candidate is an ultimate release before the final release. Unless something bad happens, nothing is changed.

Of course, the number of pre-releases varies a lot depending on the size of your project and the numbers of end-users. Python itself has several alpha, beta and release candidate releases when a minor version is being released, because it impacts a lot of people and organizations.

Small projects often don't have any pre-releases because they can publish a new final version anytime. Some small projects do have pre-releases even if they could skip it because it is benefic for increasing feedback, and has a positive psychological effect on end-users. For instance, having a series of pre-releases for your "1.0" version will increase the chances to publish a rock-solid version. And the "1.0" milestone can mean a lot to your end users: round numbers like this make them feel that your software has reached an important step in its maturity.

Post-releases

Some projects use post-releases when they need to publish a version that simply doesn't fit in the next series or can't be a new release in the current series. Let's take an example: "1.9" is the last release of your "1.x" series, and you have started a backward incompatible "2.x" series. You want all your users to drop any "1.x" release as soon as possible. You have released debug versions in the "1.x" series for quite a time and stated that "1.9" was the last one. Although, a user finds a really bad bug in "1.9" and asks you to fix it because he can't switch to the new series. In that case it is useful to publish a post release.

This use case is quite rare, and it's likely that you will never have to do post releases.

Defining your release cycle

A sofware usually follows this cycle:

- internal development versions might be released, for testing purposes. These releases can be private to a circle of developers, but they still need to be versioned. It's a good practice to do them because, unlike grabbing the current development version out of the code repository, it will help testing your release process: if there's a bug in your setup.py file, or in any part of your release/deploy/install process, they can be caught in early stage that way.

- one or several pre-release are made for getting end-users feedback and avoiding releasing a *brown bag* release, as we saw in the previous section. Having just release candidates, or also apha and betas is up to you. There's no rule about it, but it is considered overkill to have alphas and betas for small projects (small here would be less that 5k SLOC. or less than 50 end-users)
- a final version is released to the world, usually starting at `1.0` or `0.1`. Micro versions are considered overkill, unless the release cycle of your minor version lasts for months, and you want to keep the same major number for years.
- bugfixes and new features are added and new versions are released
- when a new major version is started, like "2.0", bug fixes releases may continue in the "1.x" series, depending on the cost of moving to "2.0" for your end-users.

Using a PEP 386 compatible scheme

PEP 386 defines a standard version scheme that should be used by Python applications that are publishing releases at PyPI. It ensures interoperability among all major package managers and installers. In other words, it will make sure that all versions of a project are proprely recognized and sorted as long as a PEP 386-compatible scheme is used.

XXX put a link, more info

XXX say here that ppl should look at how big projects are doing (python, zope, twisted)

Advanced Topics

Future of Packaging

Follow @*packagingpig* <*http://twitter.com/packagingpig*> on twitter.

Inshort...

- `setup.py` gone!
- `distutils` gone!
- `distribute` gone!
- `pip` and `virtualenv` here to stay!
- eggs ... gone!

See also:

http://bitbucket.org/tarek/distutils2/src/tip/docs/design/wiki.rst

Glossary

Buildout A python-based build system for creating, assembling and deploying applications from multiple parts, any of which may be non-python based. It lets you create a buildout configuration and reproduce the same software later.

Buildout is commonly used to install and manage Python distributions. It differentiates itself from <pip> in a few ways: Buildout takes a configuration file as input, where as pip is run from the command-line. Buildout can

run any arbitrary recipe during installation and so can manage non-python parts, such as config files, databases, etc. Buildout by default installs packages in a multi-version manner, so that each distribution is contained in a separate directory, and Buildout be configured so that other Buildout installations can re-use a multi-version archive of installed distributions. In contrast, Pip installs distributions into a single location, such that it's only possible to have a single version of each distribution installed at a time. However, it's possible to use different recipes with Buildout to create single location installations in the same fashion as pip.

CPAN CPAN is the Comprehensive Perl Archive Network, a large collection of *Perl* software and documentation. You can begin exploring from either http://www.cpan.org/, http://www.perl.com/CPAN/ or any of the mirrors listed at http://www.cpan.org/SITES.html. *[CPAN]*

Developer The person developing the package.

Distribution A Python distribution is a versioned compressed archive file that contains Python packages, modules, and other resource files. The distribution file is what an end-user will download from the internet and install.

A distribution is often also called a package. This is the term commonly used in other fields of computing. For example, Mac OS X and Debian call these files package files. However, in Python, the term package refers to an importable directory. In order to distinguish between these two concepts, the compressed archive file containing code is called a distribution.

However, it is not uncommon in Python to refer to a distribution using the term package. While the two meanings of the term package is not always 100% unambigous, the context of the term package is usually sufficient to distinguish the meaning of the word. For example, the python installation tool pip is an acronym for "pip installs packages", while technically the tool installs distributions, the name package is used as it's meaning is more widely understood. Even the site where distributions are distributed at is called the Python Package Index (and not the Python Distribution Index).

Distutils A standard and basic package that comes with the Python standard library. It is used for creating distributions (where it is imported in the `setup.py` file for that distribution).

Extension Module A module written in the low-level language of the Python implementation: C/C++ for Python, Java for Jython. Typically contained in a single dynamically loadable pre-compiled file, e.g. a shared object (.so) file for Python extensions on Unix, a DLL (given the .pyd extension) for Python extensions on Windows, or a Java class file for Jython extensions.

IRC Internet Relay Chat (IRC) is a form of real-time Internet text messaging (chat) or synchronous conferencing. It is mainly designed for group communication in discussion forums, called channels, but also allows one-to-one communication via private message as well as chat and data transfers via Direct Client-to-Client. *[WikipediaIRC]*

Known Good Set (KGS) A set of distributions at specified versions which are compatible with each other. Typically a test suite will be run which passes all tests before a specific set of packages is declared a known good set. This term is commonly used by frameworks and toolkits which are compromised of multiple individual distributions.

Module A python source code file (ex. `mymodule.py`), most often found in a package (`mypackage/mymodule.py`). You can import a module: `import mymodule`, or `import mypackage.mymodule` if it lives in `mypackage`.

Package A directory containing an `__init__.py` file (ex. `mypackage/__init__.py`), and also usually containing modules (possibly along with other packages). You can import a package: `import mypackage`

A package should not be confused with a compressed archive file used to install Python code.

Package Index A repository of distributions with a web interface to automate distribution discovery and consumption.

See also:

The *The Python Package Index (PyPI)* is the default packaging index for the Python community. It is open to all Python developers to consume and distribute their distributions.

Packager The person packaging the package for a particular operating system (e.g. Debian).

Perl Perl is a high-level programming language with an eclectic heritage written by Larry Wall and a cast of thousands. It derives from the ubiquitous C programming language and to a lesser extent from sed, awk, the Unix shell, and at least a dozen other tools and languages. Perl's process, file, and text manipulation facilities make it particularly well-suited for tasks involving quick prototyping, system utilities, software tools, system management tasks, database access, graphical programming, networking, and world wide web programming. These strengths make it especially popular with system administrators and CGI script authors, but mathematicians, geneticists, journalists, and even managers also use Perl. *[PERL]*

Pip <pip> is a command-line tool for downloading and installing Python distributions.

Project A library, framework, script, plugin, application, or collection of data or other resources, or some combination thereof.

Python projects must have unique names, which are registered on PyPI. Each project will then contain one or more releases, and each release may comprise one or more distributions.

Note that there is a strong convention to name a project after the name of the package that is imported to run that project. However, this doesn't have to hold true. It's possible to install a distribution from the project 'spam' and have it provide a package importable only as 'eggs'.

Pure Python Module A module written in Python and contained in a single .py file (and possibly associated .pyc and/or .pyo files). Sometimes referred to as a "pure module."

Release A snapshot of a project at a particular point in time, denoted by a version identifier.

Making a release may entail the publishing of multiple distributions. For example, if version 1.0 of a project was released, it could be available in both a source distribution format and a Windows installer distribution format.

reStructuredText A plain text format used in many Python projects for documentation. The reStructuredText format is used in this document. For more information, please see the reStructuredText Documentation.

Standard Library Python's standard library is very extensive, offering a wide range of facilities as indicated by the long table of contents listed below. The library contains built-in modules (written in C) that provide access to system functionality such as file I/O that would otherwise be inaccessible to Python programmers, as well as modules written in Python that provide standardized solutions for many problems that occur in everyday programming. Some of these modules are explicitly designed to encourage and enhance the portability of Python programs by abstracting away platform-specifics into platform-neutral APIs. *[StandardLibrary]*

See also:

Python Standard Library Documentation

System Package A package provided by in a format native to the operating system. e.g. rpm or dpkg file.

Tarball An archive format for collected a group of files together as one. The format's extension is usually `.tar`, which represents its meaning, Tape ARchive. The format is often used in conjunction with a compression format such as gzip or bzip.

Working Set A collection of distributions available for importing. These are the distributions that are on the *sys.path* variable. At most one version a distribution is possible in a working set.

Working sets include all distributions available for importing, not just the sub-set of distributions which have actually been imported.

Citations

Credits

Here's the list of contributors: writers, editors, reviewers, and people who allowed some of their own content to be included in the guide.

– Tarek

Contributors

- Jean-Paul Calderone
- Martijn Faassen
- Carl Meyer
- John M. Gabriele
- Reinout van Rees
- Michael Mulich
- C. Titus Brown
- Kevin Teague

Referenced Materials

- Installing Python Modules
- Distributing Python Modules

License

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Bibliography

[CPAN] What is CPAN?

[PERL] What is Perl?

[StandardLibrary] Python Standard Library Documentation

[WikipediaIRC] Internet Relay Chat (Wikipedia)

Index