1. An Introduction to Distutils

This document covers using the Distutils to distribute your Python modules, concentrating on the role of developer/distributor: if you're looking for information on installing Python modules, you should refer to the Installing Python Modules (Legacy version) chapter.

1.1. Concepts & Terminology

Using the Distutils is quite simple, both for module developers and for users/administrators installing third-party modules. As a developer, your responsibilities (apart from writing solid, well-documented and well-tested code, of course!) are:

- write a setup script (`setup.py` by convention)
- (optional) write a setup configuration file
- create a source distribution
- (optional) create one or more built (binary) distributions

Each of these tasks is covered in this document.

Not all module developers have access to a multitude of platforms, so it's not always feasible to expect them to create a multitude of built distributions. It is hoped that a class of intermediaries, called *packagers*, will arise to address this need. Packagers will take source distributions released by module developers, build them on one or more platforms, and release the resulting built distributions. Thus, users on the most popular platforms will be able to install most popular Python module distributions in the most natural way for their platform, without having to run a single setup script or compile a line of code.

1.2. A Simple Example

The setup script is usually quite simple, although since it's written in Python, there are no arbitrary limits to what you can do with it, though you should be careful about putting arbitrarily expensive operations in your setup script. Unlike, say, Autoconf-style configure scripts, the setup script may be run multiple times in the course of building and installing your module distribution.

If all you want to do is distribute a module called `foo`, contained in a file `foo.py`, then your setup script can be as simple as this:

```
from distutils.core import setup
setup(name='foo',
      version='1.0',
```

```
      py_modules=['foo'],
      )
```

Some observations:

- most information that you supply to the Distutils is supplied as keyword arguments to the `setup()` function
- those keyword arguments fall into two categories: package metadata (name, version number) and information about what's in the package (a list of pure Python modules, in this case)
- modules are specified by module name, not filename (the same will hold true for packages and extensions)
- it's recommended that you supply a little more metadata, in particular your name, email address and a URL for the project (see section Writing the Setup Script for an example)

To create a source distribution for this module, you would create a setup script, `setup.py`, containing the above code, and run this command from a terminal:

```
python setup.py sdist
```

For Windows, open a command prompt window (Start ▸ Accessories) and change the command to:

```
setup.py sdist
```

sdist will create an archive file (e.g., tarball on Unix, ZIP file on Windows) containing your setup script `setup.py`, and your module `foo.py`. The archive file will be named `foo-1.0.tar.gz` (or `.zip`), and will unpack into a directory `foo-1.0`.

If an end-user wishes to install your `foo` module, all she has to do is download `foo-1.0.tar.gz` (or `.zip`), unpack it, and—from the `foo-1.0` directory—run

```
python setup.py install
```

which will ultimately copy `foo.py` to the appropriate directory for third-party modules in their Python installation.

This simple example demonstrates some fundamental concepts of the Distutils. First, both developers and installers have the same basic user interface, i.e. the setup script. The difference is which Distutils *commands* they use: the **sdist** command is almost exclusively for module developers, while **install** is more often for installers (although most developers will want to install their own code occasionally).

If you want to make things really easy for your users, you can create one or more built distributions for them. For instance, if you are running on a Windows machine,

and want to make things easy for other Windows users, you can create an executable installer (the most appropriate type of built distribution for this platform) with the **bdist_wininst** command. For example:

```
python setup.py bdist_wininst
```

will create an executable installer, `foo-1.0.win32.exe`, in the current directory.

Other useful built distribution formats are RPM, implemented by the **bdist_rpm** command, Solaris **pkgtool** (**bdist_pkgtool**), and HP-UX **swinstall** (**bdist_sdux**). For example, the following command will create an RPM file called `foo-1.0.noarch.rpm`:

```
python setup.py bdist_rpm
```

(The **bdist_rpm** command uses the **rpm** executable, therefore this has to be run on an RPM-based system such as Red Hat Linux, SuSE Linux, or Mandrake Linux.)

You can find out what distribution formats are available at any time by running

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

1.3. General Python terminology

If you're reading this document, you probably have a good idea of what modules, extensions, and so forth are. Nevertheless, just to be sure that everyone is operating from a common starting point, we offer the following glossary of common Python terms:

module
: the basic unit of code reusability in Python: a block of code imported by some other code. Three types of modules concern us here: pure Python modules, extension modules, and packages.

pure Python module
: a module written in Python and contained in a single `.py` file (and possibly associated `.pyc` files). Sometimes referred to as a "pure module."

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. (Note that currently, the Distutils only handles C/C++ extensions for Python.)

package
: a module that contains other modules; typically contained in a directory in the filesystem and distinguished from other directories by the presence of a file `__init__.py`.

root package
: the root of the hierarchy of packages. (This isn't really a package, since it doesn't have an `__init__.py` file. But we have to call it something.) The vast majority of the standard library is in the root package, as are many small, standalone third-party modules that don't belong to a larger module collection. Unlike regular packages, modules in the root package can be found in many directories: in fact, every directory listed in `sys.path` contributes modules to the root package.

1.4. Distutils-specific terminology

The following terms apply more specifically to the domain of distributing Python modules using the Distutils:

module distribution
: a collection of Python modules distributed together as a single downloadable resource and meant to be installed *en masse*. Examples of some well-known module distributions are NumPy, SciPy, Pillow, or mxBase. (This would be called a *package*, except that term is already taken in the Python context: a single module distribution may contain zero, one, or many Python packages.)

pure module distribution
: a module distribution that contains only pure Python modules and packages. Sometimes referred to as a "pure distribution."

non-pure module distribution
: a module distribution that contains at least one extension module. Sometimes referred to as a "non-pure distribution."

distribution root
: the top-level directory of your source tree (or source distribution); the directory where `setup.py` exists. Generally `setup.py` will be run from this directory.