CHAPTER 18

Packaging Your Programs

Once your program is ready for release, you will probably want to package it properly before distributing it. If it consists of a single `.py` file, this might not be much of an issue. If you're dealing with nonprogrammer users, however, even placing a simple Python library in the right place or fiddling with the `PYTHONPATH` may be more than they want to deal with. Users normally want to simply double-click an installation program, follow some installation wizard, and then have your program ready to run.

Lately, Python programmers have also become used to a similar convenience, although with a slightly more low-level interface. The Distutils toolkit for distributing Python packages makes it easy to write install scripts in Python. You can use these scripts to build archive files for distribution, which the programmer (user) can then use for compiling and installing your libraries.

In this chapter, I focus on Distutils, because it is an essential tool in every Python programmer's toolkit. And Distutils actually goes beyond the script-based installation of Python libraries. Using Distutils, you can build simple Windows installers and, with the extension `py2exe`, you can also build stand-alone Windows executable programs. And if you want a self-installing archive for your binaries, I provide a few pointers for achieving that as well.

Distutils Basics

Distutils is documented thoroughly in the two documents "Distributing Python Modules" and "Installing Python Modules," both available from the Python Library Reference (`http://python.org/doc/lib/module-distutils.html`). You can use Distutils to do all manner of useful things by writing a script as simple as the one shown in Listing 18-1.

Listing 18-1. *Simple Distutils Setup Script (setup.py)*

```
from distutils.core import setup

setup(name='Hello',
      version='1.0',
      description='A simple example',
      author='Magnus Lie Hetland',
      py_modules=['hello'])
```

You don't really *have* to supply all of this information in the `setup` function (you don't actually need to supply any arguments at all), and you certainly can supply more (such as `author_email` or `url`). The names should be self-explanatory.

■ **Tip** The `setuptools` project (`http://peak.telecommunity.com/DevCenter/setuptools`) is based on Distutils, but includes several enhancements. For example, `setuptools` lets you create so-called "Python eggs," which are portable, single-file bundles designed for distributing Python packages. It also provides quite a bit of automatic interaction with the Python Package Index (`http://pypi.python.org`), a centralized index of Python packages.

Save the script in Listing 18-1 as `setup.py` (this is a universal convention for Distutils setup scripts), and make sure that you have a simple module called `hello.py` in the same directory.

■ **Caution** The `setup` script will create new files and subdirectories in the current directory when you run it, so you should probably experiment with it in a fresh directory to avoid having old files being overwritten.

Now let's see how you can put this simple script to use. Execute it as follows:

```
python setup.py
```

You should get some output like the following:

```
usage: setup.py [global_opts] cmd1 [cmd1_opts] [cmd2 [cmd2_opts] ...]
   or: setup.py --help [cmd1 cmd2 ...]
   or: setup.py --help-commands
   or: setup.py cmd --help

error: no commands supplied
```

As you can see, you can get more information using the `--help` or `--help-commands` switches. Try issuing the `build` command, just to see Distutils in action:

```
python setup.py build
```

You should now see output like the following:

```
running build
running build_py
creating build
creating build/lib
copying hello.py -> build/lib
```

Distutils has created a subdirectory called `build`, with yet another subdirectory named `lib`, and placed a copy of `hello.py` in `build/lib`. The `build` subdirectory is a sort of working area where Distutils assembles a package (and compiles extension libraries, for example). You don't really need to run the `build` command when installing, because it will be run automatically, if needed, when you run the `install` command.

■ **Note** In this example, the `install` command will copy the `hello.py` module to some system-specific directory in your `PYTHONPATH`. This should not pose a risk, but if you don't want to clutter your system, you might want to remove it afterward. Make a note of the specific location where it is placed, as output by `setup.py`. You could also use the `-n` switch to do a dry run. At the time of writing, there is no standard `uninstall` command (although you can find custom uninstallation implementations online), so you'll need to uninstall the module by hand.

Speaking of which . . . let's try to install the module:

```
python setup.py install
```

Now you should see something like the following:

```
running install
running build
running build_py
running install_lib
copying build/lib/hello.py -> /path/to/python/lib/python2.5/site-packages
byte-compiling /path/to/python/lib/python2.5/site-packages/hello.py to hello.pyc
```

■ **Note** If you're running a version of Python that you didn't install yourself, and don't have the proper privileges, you may not be allowed to install the module as shown, because you don't have write permissions to the correct directory.

This is the standard mechanism used to install Python modules, packages, and extensions. All you need to do is provide the little setup script.

The sample script uses only the Distutils directive `py_modules`. If you want to install entire packages, you can use the directive `packages` in an equivalent manner (just list the package names). You can set many other options (some of which are covered in the section "Compiling Extensions," later in this chapter). You can also create configuration files for Distutils to set various properties (see the section "Distutils Configuration Files" in "Installing Python Modules," `http://python.org/doc/inst/config-syntax.html`).

The various ways of providing options (command-line switches, keyword arguments to `setup`, and Distutils configuration files) let you specify such things as *what* to install and *where* to install it. And these options can be used for more than one thing. The following section shows you how to wrap the modules you specified for installation as an archive file, ready for distribution.

Wrapping Things Up

Once you've written a `setup.py` script that will let the user install your modules, you can use it yourself to build an archive file, a Windows installer, or an RPM package.

Building an Archive File

You do this with the `sdist` (for "source distribution") command:

```
python setup.py sdist
```

If you run this, you will probably get quite a bit of output, including some warnings. The warnings I get include a complaint about a missing `author_email` option, a missing `MANIFEST.in` file, and a missing `README` file. You can safely ignore all of these (although feel free to add an `author_email` option to your `setup.py` script, similar to the `author` option, a `README` or `README.txt` text file, and an empty file called `MANIFEST.in` in the current directory).

After the warnings you should see output like the following:

```
writing manifest file 'MANIFEST'
creating Hello-1.0
making hard links in Hello-1.0...
hard linking hello.py -> Hello-1.0
hard linking setup.py -> Hello-1.0
tar -cf dist/Hello-1.0.tar Hello-1.0
gzip -f9 dist/Hello-1.0.tar
removing 'Hello-1.0' (and everything under it)
```

As you can see, when you create a source distribution, a file called `MANIFEST` is created. This file contains a list of all your files. The `MANIFEST.in` file is a template for the manifest, and it is used when figuring out what to install. You can include lines like the following to specify files that you want to have included, if Distutils hasn't figured it out by itself, using your `setup.py` script (and default includes, such as `README`):

```
include somedirectory/somefile.txt

include somedirectory/*
```

■ **Note** If you've run the `sdist` command before, and you have a file called `MANIFEST` already, you will see the word `reading` instead of `writing` at the beginning. If you've restructured your package and want to repackage it, deleting the `MANIFEST` file can be a good idea, in order to start afresh.

Now, in addition to the `build` subdirectory, you should have one called `dist`. Inside it, you will find a `gzip`'ed `tar` archive called `Hello-1.0.tar.gz`. This can now be distributed to others, and they can unpack it and install it using the included `setup.py` script. If you don't want a `.tar.gz` file, plenty of other distribution formats are available, and you can set them all through the command-line switch `--formats`. (As the plural name indicates, you can supply more than one format, separated by commas, to create more archive files in one go.) The format names available in Python 2.5 (accessible through the `--help-formats` switch to the `sdist` command) are `bztar` (for `bzip2`'ed `tar` files), `gztar` (the default, for `gzip`'ed `tar` files), `tar` (for uncompressed `tar` files), `zip` (for ZIP files), and `ztar` (for compressed `tar` files, using the UNIX command `compress`).

Creating a Windows Installer or an RPM Package

Using the command `bdist`, you can create simple Windows installers and Linux RPM files. (You normally use this to create *binary* distributions, where extensions have been compiled for a particular architecture. See the following section for information about compiling extensions.) The formats available for `bdist` (in addition to the ones available for `sdist`) are `rpm` (for RPM packages) and `wininst` (for Windows executable installer).

One interesting twist is that you can, in fact, build Windows installers for your package in non-Windows systems, provided that you don't have any extensions you need to compile. If you have access to both, say, a Linux machine and a Windows box, you could try running the following on a Linux machine:

```
python setup.py bdist --formats=wininst
```

Then (after ignoring a few warnings about compiler settings) copy the file `dist/Hello-1.0.win32.exe` to your Windows machine and run it. You should be presented with a rudimentary installer wizard. (You can cancel the process before actually installing the module.)

USING A *REAL* INSTALLER

The installer you get with the `wininst` format in Distutils is very basic. As with normal Distutils installation, it will not let you uninstall your packages, for example. This may be acceptable in some situations, but sometimes you may want a more professional look, especially if you're creating an executable using `py2exe` (as described in this chapter). In this case, you might want to consider using some standard installer such as Inno Setup (`http://jrsoftware.org/isinfo.php`), which works very well with executables created with `py2exe`. This type of installer will install your program in a more normal Windows fashion and give you functionality such as the ability to uninstall the program.

A more Python-centric (but, at present, unmaintained) option is the McMillan installer (a web search should give you an updated download location), which can also work as an alternative to `py2exe` when building executable programs. Other options include InstallShield (`http://installshield.com`), Wise installer (`http://wise.com`), Installer VISE (`http://www.mindvision.com`), Nullsoft Scriptable Install System (`http://nsis.sf.net`), Youseful Windows Installer (`http://youseful.com`), and Ghost Installer (`http://ethalone.com`). A web search will probably turn up several other solutions.

For more information about Windows installer technology, see Phil Wilson's *The Definitive Guide to Windows Installer* (Apress, 2004).

Compiling Extensions

In Chapter 17, you saw how to write extensions for Python. You may agree that compiling these extensions could be a bit cumbersome at times. Luckily, you can use Distutils for this as well. You may want to refer back to Chapter 17 for the source code to the program `palindrome` (in Listing 17-6). Assuming that you have the source file `palindrome2.c` in the current (empty) directory, the following `setup.py` script could be used to compile (and install) it:

```
from distutils.core import setup, Extension

setup(name='palindrome',
      version='1.0',
      ext_modules = [
          Extension('palindrome', ['palindrome2.c'])
      ])
```

If you run the `install` command with this `setup.py` script, the `palindrome` extension module should be compiled automatically before it is installed. As you can see, instead of specifying a list of module names, you give the `ext_modules` argument a list of `Extension` instances. The constructor takes a name and a list of related files; this is where you would specify header (`.h`) files, for example.

If you would rather just compile the extension in place (resulting in a file called `palindrome.so` in the current directory for most UNIX systems), you can use the following command:

```
python setup.py build_ext --inplace
```

Now we get to a real juicy bit. If you have SWIG installed (see Chapter 17), you can have Distutils use it directly!

Take a look at the source for the original `palindrome.c` (without all the wrapping code) in Listing 17-3. It's certainly much simpler than the wrapped-up version. Being able to compile it directly as a Python extension, having Distutils use SWIG for you, can be very convenient. It's all very simple, really—you just add the name of the interface (`.i`) file (see Listing 17-5) to the list of files in the `Extension` instance:

```
from distutils.core import setup, Extension

setup(name='palindrome',
      version='1.0',
      ext_modules = [
          Extension('palindrome', ['palindrome.c',
                                   'palindrome.i'])
      ])
```

If you run this script using the same command as before (`build_ext`, possibly with the `--inplace` switch), you should end up with a `palindrome.so` file again, but this time without needing to write all the wrapper code yourself.

Creating Executable Programs with py2exe

The `py2exe` extension to Distutils (available from `http://www.py2exe.org`) allows you to build executable Windows programs (`.exe` files), which can be useful if you don't want to burden your users with having to install a Python interpreter separately.

■ **Tip** After creating your executable program, you may want to use an installer, such as Inno Setup (`http://jrsoftware.org/isinfo.php`), to distribute the executable program and the accompanying files created by `py2exe`. See the "Using a Real Installer" sidebar.

The `py2exe` package can be used to create executables with GUIs (such as `wx`, as described in Chapter 12). Let's use a very simple example here (it uses the `raw_input` trick first discussed in the section "What About Double-Clicking?" in Chapter 1):

```
print 'Hello, world!'
raw_input('Press <enter>')
```

Again, starting in an empty directory containing only this file, called `hello.py`, create a `setup.py` file like this:

```
from distutils.core import setup
import py2exe

setup(console=['hello.py'])
```

You can run this script like this:

```
python setup.py py2exe
```

This will create a console application (called `hello.exe`) along with a couple of other files in the `dist` subdirectory. You can either run it from the command line or double-click it.

For more information about how `py2exe` works, and how you can use it in more advanced ways, visit the `py2exe` web site (`http://www.py2exe.org`).

■ **Tip** If you're using Mac OS, you might want to check out Bob Ippolito's `py2app` (`http://undefined.org/python/py2app.html`).

LETTING THE WORLD KNOW

You have a choice of many places to announce your new software, such as Freshmeat (`http://freshmeat.net`). There is, however, a standard, centralized index of Python packages called, fittingly, the Python Package Index, or simply PyPI. Visit the PyPI web site (`http://pypi.python.org`) to look for new packages or new versions of old packages, or to publish your own packages.

In addition to the packages themselves, you can register a lot of useful metadata (possibly with the aid of Distutils or its relation `setuptools`), such as author, license, platform, categories, and descriptive keywords. The `register` command in Distutils will do most of the work for you.

A Quick Summary

Finally, you now know how to create shiny, professional-looking software with fancy GUI installers—or how to automate the generation of those precious `.tar.gz` files. Here is a summary of the specific concepts covered:

Distutils: The Distutils toolkit lets you write installer scripts, conventionally called `setup.py`. With these scripts, you can install modules, packages, and extensions. You can also build distributable archives and simple Windows installers.

Distutils commands: You can run your `setup.py` script with several commands, such as `build`, `build_ext`, `install`, `sdist`, and `bdist`.

Installers: Many installer generators are available. Using an installer to install your Python program makes the process easier for your users.

Compiling extensions: You can use Distutils to have your C extensions compiled automatically, with Distutils automatically locating your Python installation and figuring out which compiler to use. You can even have it run SWIG automatically.

Executable binaries: The `py2exe` extension to Distutils can be used to create executable binaries from your Python programs. Along with a couple of extra files (which can be

conveniently installed with an installer), these `.exe` files can be run without installing a Python interpreter separately.

New Functions in This Chapter

Function	Description
`distutils.core.setup(...)`	Configures Distutils with keyword arguments in your `setup.py` script

What Now?

That's it for the technical stuff—sort of. In the next chapter, you get some programming methodology and philosophy, and then come the projects. Enjoy!