Distributing Python Modules

Greg Ward

September 28, 2000

E-mail: gward@python.net

Abstract

This document describes the Python Distribution Utilities ("Distutils") from the module developer's point-of-view, describing how to use the Distutils to make Python modules and extensions easily available to a wider audience with very little overhead for build/release/install mechanics.

Contents

1 Introduction

In the past, Python module developers have not had much infrastructure support for distributing modules, nor have Python users had much support for installing and maintaining third-party modules. With the introduction of the Python Distribution Utilities (Distutils for short) in Python 1.6, this situation should start to improve.

This document only covers using the Distutils to distribute your Python modules. Using the Distutils does not tie you to Python 1.6, though: the Distutils work just fine with Python 1.5.2, and it is reasonable (and expected to become commonplace) to expect users of Python 1.5.2 to download and install the Distutils separately before they can install your modules. Python 1.6 (or later) users, of course, won't have to add anything to their Python installation in order to use the Distutils to install third-party modules.

This document concentrates on the role of developer/distributor: if you're looking for information on installing Python modules, you should refer to the *Installing Python Modules* manual.

2 Concepts & Terminology

Using the Distutils is quite simple, both for module developers and for users/administrators installing third-party modules. As a developer, your responsibilites (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.

2.1 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. 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 little 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 meta-data (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 meta-data, in particular your name, email address and a URL for the project

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

```
python setup.py sdist
```

which 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 cre-

ate 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.

****not implemented yet**** (Another way to create executable installers for Windows is with the `bdist_wise` command, which uses Wise—the commercial installer-generator used to create Python's own installer—to create the installer. Wise-based installers are more appropriate for large, industrial-strength applications that need the full capabilities of a "real" installer. `bdist_wininst` creates a self-extracting zip file with a minimal user interface, which is enough for small- to medium-sized module collections. You'll need to have version XXX of Wise installed on your system for the `bdist_wise` command to work; it's available from http://foo/bar/baz.)

Currently (Distutils 0.9.1), the are only other useful built distribution format is RPM, implemented by the `bdist_rpm` command. For example, the following command will create an RPM file called 'Foo-1.0.noarch.rpm':

```
python setup.py bdist_rpm
```

(This uses the `rpm` command, so 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
```

2.2 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' and/or '.pyo' files). Sometimes referred to as a "pure module."

extension module a module written in the low-level language of the Python implemention: C/C++ for CPython, Java for JPython. Typically contained in a single dynamically loadable pre-compiled file, e.g. a shared object ('.so') file for CPython extensions on Unix, a DLL (given the '.pyd' extension) for CPython extensions on Windows, or a Java class file for JPython extensions. (Note that currently, the Distutils only handles C/C++ extensions for CPython.)

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` can contribute modules to the root package.

2.3 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 Numeric Python, PyXML, PIL (the Python Imaging Library), or mxDateTime. (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 and is run from

3 Writing the Setup Script

The setup script is the centre of all activity in building, distributing, and installing modules using the Distutils. The main purpose of the setup script is to describe your module distribution to the Distutils, so that the various commands that operate on your modules do the right thing. As we saw in section 2.1 above, the setup script consists mainly of a call to `setup()`, and most information supplied to the Distutils by the module developer is supplied as keyword arguments to `setup()`.

Here's a slightly more involved example, which we'll follow for the next couple of sections: the Distutils' own setup script. (Keep in mind that although the Distutils are included with Python 1.6 and later, they also have an independent existence so that Python 1.5.2 users can use them to install other module distributions. The Distutils' own setup script, shown here, is used to install the package into Python 1.5.2.)

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

from distutils.core import setup

setup (name = "Distutils",
       version = "1.0",
       description = "Python Distribution Utilities",
       author = "Greg Ward",
       author_email = "gward@python.net",
       url = "http://www.python.org/sigs/distutils-sig/",

       packages = ['distutils', 'distutils.command'],
      )
```

There are only two differences between this and the trivial one-file distribution presented in section 2.1: more meta-data, and the specification of pure Python modules by package, rather than by module. This is important since the Distutils consist of a couple of dozen modules split into (so far) two packages; an explicit list of every module would be tedious to generate and difficult to maintain.

Note that any pathnames (files or directories) supplied in the setup script should be written using the Unix convention, i.e. slash-separated. The Distutils will take care of converting this platform-neutral representation into whatever is appropriate on your current platform before actually using the pathname. This makes your setup script portable across operating systems, which of course is one of the major goals of the Distutils. In this spirit, all pathnames in this document are slash-separated (Mac OS programmers should keep in mind that the *absence* of a leading slash indicates a relative path, the opposite of the Mac OS convention with colons).

3.1 Listing whole packages

The packages option tells the Distutils to process (build, distribute, install, etc.) all pure Python modules found in each package mentioned in the packages list. In order to do this, of course, there has to be a correspondence between package names and directories in the filesystem. The default correspondence is the most obvious one, i.e. package `distutils` is found in the directory 'distutils' relative to the distribution root. Thus, when you say `packages = ['foo']` in your setup script, you are promising that the Distutils will find a file 'foo/__init__.py' (which might be spelled differently on your system, but you get the idea) relative to the directory where your setup script lives. (If you break this promise, the Distutils will issue a warning but process the broken package anyways.)

If you use a different convention to lay out your source directory, that's no problem: you just have to supply the package_dir option to tell the Distutils about your convention. For example, say you keep all Python source under 'lib', so that modules in the "root package" (i.e., not in any package at all) are right in 'lib', modules in the `foo` package are in 'lib/foo', and so forth. Then you would put

```
package_dir = {'': 'lib'}
```

in your setup script. (The keys to this dictionary are package names, and an empty package name stands for the root package. The values are directory names relative to your distribution root.) In this case, when you say `packages = ['foo']`, you are promising that the file 'lib/foo/__init__.py' exists.

Another possible convention is to put the `foo` package right in 'lib', the `foo.bar` package in 'lib/bar', etc. This would be written in the setup script as

```
package_dir = {'foo': 'lib'}
```

A *package*: *dir* entry in the package_dir dictionary implicitly applies to all packages below *package*, so the `foo.bar` case is automatically handled here. In this example, having `packages = ['foo', 'foo.bar']` tells the Distutils to look for 'lib/__init__.py' and 'lib/bar/__init__.py'. (Keep in mind that although package_dir applies recursively, you must explicitly list all packages in packages: the Distutils will *not* recursively scan your source tree looking for any directory with an '__init__.py' file.)

3.2 Listing individual modules

For a small module distribution, you might prefer to list all modules rather than listing packages—especially the case of a single module that goes in the "root package" (i.e., no package at all). This simplest case was shown in section 2.1; here is a slightly more involved example:

```
py_modules = ['mod1', 'pkg.mod2']
```

This describes two modules, one of them in the "root" package, the other in the `pkg` package. Again, the default package/directory layout implies that these two modules can be found in 'mod1.py' and 'pkg/mod2.py', and that 'pkg/__init__.py' exists as well. And again, you can override the package/directory correspondence using the package_dir option.

3.3 Describing extension modules

Just as writing Python extension modules is a bit more complicated than writing pure Python modules, describing them to the Distutils is a bit more complicated. Unlike pure modules, it's not enough just to list modules or packages and expect the Distutils to go out and find the right files; you have to specify the extension name, source file(s), and any compile/link requirements (include directories, libraries to link with, etc.).

All of this is done through another keyword argument to `setup()`, the extensions option. extensions is just a list of `Extension` instances, each of which describes a single extension module. Suppose your distribution includes a single extension, called `foo` and implemented by 'foo.c'. If no additional instructions to the compiler/linker are needed, describing this extension is quite simple:

```
Extension("foo", ["foo.c"])
```

The `Extension` class can be imported from `distutils.core`, along with `setup()`. Thus, the setup script for a module distribution that contains only this one extension and nothing else might be:

```
from distutils.core import setup, Extension
setup(name = "foo", version = "1.0",
      extensions = [Extension("foo", ["foo.c"])])
```

The `Extension` class (actually, the underlying extension-building machinery implemented by the `built_ext` command) supports a great deal of flexibility in describing Python extensions, which is explained in the following sections.

Extension names and packages

The first argument to the `Extension` constructor is always the name of the extension, including any package names. For example,

```
Extension("foo", ["src/foo1.c", "src/foo2.c"])
```

describes an extension that lives in the root package, while

```
Extension("pkg.foo", ["src/foo1.c", "src/foo2.c"])
```

describes the same extension in the `pkg` package. The source files and resulting object code are identical in both cases; the only difference is where in the filesystem (and therefore where in Python's namespace hierarchy) the resulting extension lives.

If you have a number of extensions all in the same package (or all under the same base package), use the ext_package keyword argument to `setup()`. For example,

```
setup(...
      ext_package = "pkg",
      extensions = [Extension("foo", ["foo.c"]),
                    Extension("subpkg.bar", ["bar.c"])]
     )
```

will compile 'foo.c' to the extension `pkg.foo`, and 'bar.c' to `pkg.subpkg.bar`.

Extension source files

The second argument to the `Extension` constructor is a list of source files. Since the Distutils currently only support C/C++ extensions, these are normally C/C++ source files. (Be sure to use appropriate extensions to distinguish C++ source files: '.cc' and '.cpp' seem to be recognized by both Unix and Windows compilers.)

However, you can also include SWIG interface ('.i') files in the list; the `build_ext` command knows how to deal with SWIG extensions: it will run SWIG on the interface file and compile the resulting C/C++ file into your extension.

****SWIG support is rough around the edges and largely untested; especially SWIG support of C++ extensions! Explain in more detail here when the interface firms up.****

On some platforms, you can include non-source files that are processed by the compiler and included in your extension. Currently, this just means Windows resource files for Visual C++. ****get more detail on this feature from Thomas Heller!****

Preprocessor options

Three optional arguments to `Extension` will help if you need to specify include directories to search or preprocessor macros to define/undefine: `include_dirs`, `define_macros`, and `undef_macros`.

For example, if your extension requires header files in the 'include' directory under your distribution root, use the `include_dirs` option:

```
Extension("foo", ["foo.c"], include_dirs=["include"])
```

You can specify absolute directories there; if you know that your extension will only be built on Unix systems with X11R6 installed to '/usr', you can get away with

```
Extension("foo", ["foo.c"], include_dirs=["/usr/include/X11"])
```

You should avoid this sort of non-portable usage if you plan to distribute your code: it's probably better to write your code to include (e.g.) `<X11/Xlib.h>`.

If you need to include header files from some other Python extension, you can take advantage of the fact that the Distutils install extension header files in a consistent way. For example, the Numerical Python header files are installed (on a standard Unix installation) to '/usr/local/include/python1.5/Numerical'. (The exact location will differ according to your platform and Python installation.) Since the Python include directory—'/usr/local/include/python1.5' in this case—is always included in the search path when building Python extensions, the best approach is to include (e.g.) `<Numerical/arrayobject.h>`. If you insist on putting the 'Numerical' include directory right into your header search path, though, you can find that directory using the Distutils `sysconfig` module:

```
from distutils.sysconfig import get_python_inc
incdir = os.path.join(get_python_inc(plat_specific=1), "Numerical")
setup(...,
      Extension(..., include_dirs=[incdir]))
```

Even though this is quite portable—it will work on any Python installation, regardless of platform—it's probably easier to just write your C code in the sensible way.

You can define and undefine pre-processor macros with the `define_macros` and `undef_macros` options. `define_macros` takes a list of `(name, value)` tuples, where `name` is the name of the macro to define (a string) and `value` is its value: either a string or `None`. (Defining a macro `FOO` to `None` is the equivalent of a bare `#define FOO` in your C source: with most compilers, this sets `FOO` to the string `1`.) `undef_macros` is just a list of macros to undefine.

For example:

```
Extension(...,
          define_macros=[('NDEBUG', '1')],
                         ('HAVE_STRFTIME', None),
          undef_macros=['HAVE_FOO', 'HAVE_BAR'])
```

is the equivalent of having this at the top of every C source file:

```
#define NDEBUG 1
#define HAVE_STRFTIME
#undef HAVE_FOO
#undef HAVE_BAR
```

Library options

You can also specify the libraries to link against when building your extension, and the directories to search for those libraries. The `libraries` option is a list of libraries to link against, `library_dirs` is a list of directories to search for libraries at link-time, and `runtime_library_dirs` is a list of directories to search for shared (dynamically loaded) libraries at run-time.

For example, if you need to link against libraries known to be in the standard library search path on target systems

```
Extension(...,
          libraries=["gdbm", "readline"])
```

If you need to link with libraries in a non-standard location, you'll have to include the location in `library_dirs`:

```
Extension(...,
          library_dirs=["/usr/X11R6/lib"],
          libraries=["X11", "Xt"])
```

(Again, this sort of non-portable construct should be avoided if you intend to distribute your code.)

****still undocumented: extra_objects, extra_compile_args, extra_link_args, export_symbols—none of which are frequently needed, some of which might be completely unnecessary!****

4 Writing the Setup Configuration File

Often, it's not possible to write down everything needed to build a distribution *a priori*. You need to get some information from the user, or from the user's system, in order to proceed. For example, you might include an optional extension module that provides an interface to a particular C library. If that library is installed on the user's system, then you can build your optional extension—but you need to know where to find the header and library file. If it's not installed, you need to know this so you can omit your optional extension.

The preferred way to do this, of course, would be for you to tell the Distutils which optional features (C libraries, system calls, external utilities, etc.) you're looking for, and it would inspect the user's system and try to find them. This functionality may appear in a future version of the Distutils, but it isn't there now. So, for the time being, we rely on the user building and installing your software to provide the necessary information. The vehicle for doing so is the setup configuration file, 'setup.cfg'.

****need more here!****

5 Creating a Source Distribution

As shown in section 2.1, you use the `sdist` command to create a source distribution. In the simplest case,

```
python setup.py sdist
```

(assuming you haven't specified any `sdist` options in the setup script or config file), `sdist` creates the archive of the default format for the current platform. The default formats are:

Platform	Default archive format for source distributions
Unix	gzipped tar file ('.tar.gz')
Windows	zip file

You can specify as many formats as you like using the **--formats** option, for example:

```
python setup.py sdist --formats=gztar,zip
```

to create a gzipped tarball and a zip file. The available formats are:

Format	Description	Notes
`zip`	zip file ('.zip')	(1)
`gztar`	gzipped tar file ('.tar.gz')	(2)
`ztar`	compressed tar file ('.tar.Z')	
`tar`	tar file ('.tar')	

Notes:

(1) default on Windows

(2) default on Unix

5.1 The manifest and manifest template

Without any additional information, the `sdist` command puts a minimal set of files into the source distribution:

- all Python source files implied by the py_modules and packages options
- all C source files mentioned in the ext_modules or libraries options (****getting C library sources currently broken – no get_source_files() method in build_clib.py!****)
- anything that looks like a test script: 'test/test*.py' (currently, the Distutils don't do anything with test scripts except include them in source distributions, but in the future there will be a standard for testing Python module distributions)
- 'README.txt' (or 'README') and 'setup.py'

Sometimes this is enough, but usually you will want to specify additional files to distribute. The typical way to do this is to write a *manifest template*, called 'MANIFEST.in' by default. The `sdist` command processes this template and

generates a manifest file, ‘MANIFEST’. (If you prefer, you can skip the manifest template and generate the manifest yourself: it just lists one file per line.)

The manifest template has one command per line, where each command specifies a set of files to include or exclude from the source distribution. For an example, again we turn to the Distutils’ own manifest template:

```
include *.txt
recursive-include examples *.txt *.py
prune examples/sample?/build
```

The meanings should be fairly clear: include all files in the distribution root matching `*.txt`, all files anywhere under the ‘examples’ directory matching `*.txt` or `*.py`, and exclude all directories matching `examples/sample?/build`. There are several other commands available in the manifest template mini-language; see section 9.4.

The order of commands in the manifest template very much matters: initially, we have the list of default files as described above, and each command in the template adds to or removes from that list of files. When we have fully processed the manifest template, we have our complete list of files. This list is written to the manifest for future reference, and then used to build the source distribution archive(s).

Following the Distutils’ own manifest template, let’s trace how the `sdist` command will build the list of files to include in the Distutils source distribution:

1. include all Python source files in the ‘distutils’ and ‘distutils/command’ subdirectories (because packages corresponding to those two directories were mentioned in the packages option in the setup script)
2. include ‘test/test*.py’ (always included)
3. include ‘README.txt’ and ‘setup.py’ (always included)
4. include ‘*.txt’ in the distribution root (this will find ‘README.txt’ a second time, but such redundancies are weeded out later)
5. in the sub-tree under ‘examples’, include anything matching ‘*.txt’
6. in the sub-tree under ‘examples’, include anything matching ‘*.py’
7. remove all files in the sub-trees starting at directories matching ‘examples/sample?/build’—this may exclude files included by the previous two steps, so it’s important that the `prune` command in the manifest template comes after the two `recursive-include` commands

Just like in the setup script, file and directory names in the manifest template should always be slash-separated; the Distutils will take care of converting them to the standard representation on your platform. That way, the manifest template is portable across operating systems.

5.2 Manifest-related options

The normal course of operations for the `sdist` command is as follows:

- if the manifest file, ‘MANIFEST’ doesn’t exist, read ‘MANIFEST.in’ and create the manifest
- if either ‘MANIFEST.in’ or the setup script (‘setup.py’) are more recent than ‘MANIFEST’, recreate ‘MANIFEST’ by reading ‘MANIFEST.in’
- use the list of files now in ‘MANIFEST’ (either just generated or read in) to create the source distribution archive(s)

There are a couple of options that modify this behaviour.

First, you might want to force the manifest to be regenerated—for example, if you have added or removed files or directories that match an existing pattern in the manifest template, you should regenerate the manifest:

```
python setup.py sdist --force-manifest
```

Or, you might just want to (re)generate the manifest, but not create a source distribution:

```
python setup.py sdist --manifest-only
```

(**--manifest-only** implies **--force-manifest**.)

If you don't want to use the default file set, you can supply the **--no-defaults** option. If you use **--no-defaults** and don't supply a manifest template (or it's empty, or nothing matches the patterns in it), then your source distribution will be empty.

6 Creating Built Distributions

A "built distribution" is what you're probably used to thinking of either as a "binary package" or an "installer" (depending on your background). It's not necessarily binary, though, because it might contain only Python source code and/or byte-code; and we don't call it a package, because that word is already spoken for in Python. (And "installer" is a term specific to the Windows world. ****do Mac people use it?****)

A built distribution is how you make life as easy as possible for installers of your module distribution: for users of RPM-based Linux systems, it's a binary RPM; for Windows users, it's an executable installer; for Debian-based Linux users, it's a Debian package; and so forth. Obviously, no one person will be able to create built distributions for every platform under the sun, so the Distutils is designed to enable module developers to concentrate on their specialty—writing code and creating source distributions—while an intermediary species of *packager* springs up to turn source distributions into built distributions for as many platforms as there are packagers.

Of course, the module developer could be his own packager; or the packager could be a volunteer "out there" somewhere who has access to a platform which the original developer does not; or it could be software periodically grabbing new source distributions and turning them into built distributions for as many platforms as the software has access to. Regardless of the nature of the beast, a packager uses the setup script and the `bdist` command family to generate built distributions.

As a simple example, if I run the following command in the Distutils source tree:

```
python setup.py bdist
```

then the Distutils builds my module distribution (the Distutils itself in this case), does a "fake" installation (also in the 'build' directory), and creates the default type of built distribution for my platform. Currently, the default format for built distributions is a "dumb" archive—tarball on Unix, ZIP file on Windows. (These are called "dumb" built distributions, because they must be unpacked in a specific location to work.)

Thus, the above command on a Unix system creates 'Distutils-0.9.1.*plat*.tar.gz'; unpacking this tarball from the root of the filesystemq installs the Distutils just as though you had downloaded the source distribution and run `python setup.py install`. (Assuming that the target system has their Python installation laid out the same as you do—another reason these are called "dumb" distributions.) Obviously, for pure Python distributions, this isn't a huge win—but for non-pure distributions, which include extensions that would need to be compiled, it can mean the difference between someone being able to use your extensions or not.

****filenames are inaccurate here!****

The `bdist` command has a **--format** option, similar to the `sdist` command, which you can use to select the types of built distribution to generate: for example,

```
python setup.py bdist --format=zip
```

would, when run on a Unix system, create 'Distutils-0.8.*plat*.zip'—again, this archive would be unpacked from the root directory to install the Distutils.

The available formats for built distributions are:

Format	Description	Notes
`zip`	zip file ('.zip')	
`gztar`	gzipped tar file ('.tar.gz')	(1)
`ztar`	compressed tar file ('.tar.Z')	
`tar`	tar file ('.tar')	
`rpm`	RPM	
`srpm`	source RPM	****to do!****
`wininst`	self-extracting ZIP file for Windows	(2)

Notes:

(1) default on Unix

(2) default on Windows ****to-do!****

You don't have to use the `bdist` command with the **--formats** option; you can also use the command that directly implements the format you're interested in. Some of these `bdist` "sub-commands" actually generate several similar formats; for instance, the `bdist_dumb` command generates all the "dumb" archive formats (`tar`, `ztar`, `gztar`, and `zip`), and `bdist_rpm` generates both binary and source RPMs. The `bdist` sub-commands, and the formats generated by each, are:

Command	Formats
`bdist_dumb`	tar, ztar, gztar, zip
`bdist_rpm`	rpm, srpm
`bdist_wininst`	wininst

7 Examples

7.1 Pure Python distribution (by module)

7.2 Pure Python distribution (by package)

7.3 Single extension module

7.4 Multiple extension modules

7.5 Putting it all together

8 Extending the Distutils

8.1 Extending existing commands

8.2 Writing new commands

9 Reference

9.1 Building modules: the `build` command family

`build`

`build_py`

`build_ext`

`build_clib`

9.2 Installing modules: the `install` command family

The install command ensures that the build commands have been run and then runs the subcommands `install_lib`, `install_data` and `install_scripts`.

`install_lib`

`install_data`

This command installs all data files provided with the distribution.

`install_scripts`

This command installs all (Python) scripts in the distribution.

9.3 Cleaning up: the `clean` command

9.4 Creating a source distribution: the `sdist` command

****fragment moved down from above: needs context!**** The manifest template commands are:

Command	Description
`include` *pat1 pat2 ...*	include all files matching any of the listed patterns
`exclude` *pat1 pat2 ...*	exclude all files matching any of the listed patterns
`recursive-include` *dir pat1 pat2 ...*	include all files under *dir* matching any of the listed patterns
`recursive-exclude` *dir pat1 pat2 ...*	exclude all files under *dir* matching any of the listed patterns
`global-include` *pat1 pat2 ...*	include all files anywhere in the source tree matching any of the listed patterns
`global-exclude` *pat1 pat2 ...*	exclude all files anywhere in the source tree matching any of the listed patterns
`prune` *dir*	exclude all files under *dir*
`graft` *dir*	include all files under *dir*

The patterns here are Unix-style “glob” patterns: `*` matches any sequence of regular filename characters, `?` matches any single regular filename character, and `[`*range*`]` matches any of the characters in *range* (e.g., `a-z`, `a-zA-Z`, `a-f0-9_.`). The definition of “regular filename character” is platform-specific: on Unix it is anything except slash; on Windows anything except backslash or colon; on Mac OS anything except colon. ****Windows and Mac OS support not there yet****

9.5 Creating a “built” distribution: the `bdist` command family

`blib`

`blib_dumb`

`blib_rpm`

`blib_wise`