Distributing Python Modules

Greg Ward

October 14, 2002

Email: 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 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.

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.[1] 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:

[1] But 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 you need to insert potentially expensive processing steps into the Distutils chain, see section **??** on extending the Distutils.

```
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 (see section 3 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:

```
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 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.

Currently (Distutils 0.9.2), the 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 implementation: C/C++ for Python, Java for JPython. 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 JPython 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` 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 (MacOS programmers should keep in mind that the *absence* of a leading slash indicates a relative path, the opposite of the MacOS convention with colons).

This, of course, only applies to pathnames given to Distutils functions. If you, for example, use standard python functions such as glob.glob or os.listdir to specify files, you should be careful to write portable code instead of hardcoding path separators:

```
    glob.glob(os.path.join('mydir', 'subdir', '*.html'))
    os.listdir(os.path.join('mydir', 'subdir'))
```

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

The Extension class (actually, the underlying extension-building machinery implemented by the build_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",
      ext_modules=[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 message text ('.mc') files and resource definition ('.rc') files for Visual C++. These will be compiled to binary resource ('.res') files and linked into the executable.

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.)

****Should mention clib libraries here or somewhere else!****

Other options

There are still some other options which can be used to handle special cases.

The extra_objects option is a list of object files to be passed to the linker. These files must not have extensions, as the default extension for the compiler is used.

extra_compile_args and extra_link_args can be used to specify additional command line options for the compiler resp. the linker command line.

export_symbols is only useful on windows, it can contain a list of symbols (functions or variables) to be exported. This option is not needed when building compiled extensions: the `initmodule` function will automatically be added to the exported symbols list by Distutils.

3.4 Listing scripts

So far we have been dealing with pure and non-pure Python modules, which are usually not run by themselves but imported by scripts.

Scripts are files containing Python source code, indended to be started from the command line. Distutils doesn't provide much functionality for the scripts: the only support Distutils gives is to adjust the first line of the script if it starts with `#!` and contains the word "python" to refer to the current interpreter location.

The scripts option simply is a list of files to be handled in this way.

3.5 Listing additional files

The data_files option can be used to specify additional files needed by the module distribution: configuration files, data files, anything which does not fit in the previous categories.

data_files specifies a sequence of (*directory*, *files*) pairs in the following way:

```
setup(...
      data_files=[('bitmaps', ['bm/b1.gif', 'bm/b2.gif']),
                  ('config', ['cfg/data.cfg'])])
```

Note that you can specify the directory names where the data files will be installed, but you cannot rename the data files themselves.

Each (*directory*, *files*) pair in the sequence specifies the installation directory and the files to install there. If *directory* is a relative path, it is interpreted relative to the installation prefix (Python's `sys.prefix` for pure-Python packages, `sys.exec_prefix` for packages that contain extension modules). Each file name in *files* is interpreted relative to the 'setup.py' script at the top of the package source distribution. No directory information from *files* is used to determine the final location of the installed file; only the name of the file is used.

You can specify the data_files options as a simple sequence of files without specifying a target directory, but this is not recommended, and the `install` command will print a warning in this case. To install data files directly in the target directory, an empty string should be given as the directory.

4 Writing the Setup Configuration File

Often, it's not possible to write down everything needed to build a distribution *a priori*: you may need to get some information from the user, or from the user's system, in order to proceed. As long as that information is fairly simple—a list of directories to search for C header files or libraries, for example—then providing a configuration file, 'setup.cfg', for users to edit is a cheap and easy way to solicit it. Configuration files also let you provide default values for any command option, which the installer can then override either on the command-line or by editing the config file.

(If you have more advanced needs, such as determining which extensions to build based on what capabilities are present on the target system, then you need the Distutils "auto-configuration" facility. This started to appear in Distutils 0.9 but, as of this writing, isn't mature or stable enough yet for real-world use.)

The setup configuration file is a useful middle-ground between the setup script—which, ideally, would be opaque to installers[2]—and the command-line to the setup script, which is outside of your control and entirely up to the installer. In fact, 'setup.cfg' (and any other Distutils configuration files present on the target system) are processed after the contents of the setup script, but before the command-line. This has several useful consequences:

- installers can override some of what you put in 'setup.py' by editing 'setup.cfg'
- you can provide non-standard defaults for options that are not easily set in 'setup.py'
- installers can override anything in 'setup.cfg' using the command-line options to 'setup.py'

The basic syntax of the configuration file is simple:

```
[command]
option=value
...
```

where *command* is one of the Distutils commands (e.g. `build_py`, `install`), and *option* is one of the options that command supports. Any number of options can be supplied for each command, and any number of command sections can be included in the file. Blank lines are ignored, as are comments (from a '`#`' character to end-of-line). Long option values can be split across multiple lines simply by indenting the continuation lines.

You can find out the list of options supported by a particular command with the universal **--help** option, e.g.

```
> python setup.py --help build_ext
[...]
Options for 'build_ext' command:
  --build-lib (-b)     directory for compiled extension modules
  --build-temp (-t)    directory for temporary files (build by-products)
  --inplace (-i)       ignore build-lib and put compiled extensions into the
                       source directory alongside your pure Python modules
  --include-dirs (-I)  list of directories to search for header files
  --define (-D)        C preprocessor macros to define
  --undef (-U)         C preprocessor macros to undefine
[...]
```

Or consult section 7 of this document (the command reference).

Note that an option spelled **--foo-bar** on the command-line is spelled foo_bar in configuration files.

For example, say you want your extensions to be built "in-place"—that is, you have an extension `pkg.ext`, and you want the compiled extension file ('ext.so' on UNIX, say) to be put in the same source directory as your pure Python modules `pkg.mod1` and `pkg.mod2`. You can always use the **--inplace** option on the command-line to ensure this:

[2]This ideal probably won't be achieved until auto-configuration is fully supported by the Distutils.

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

But this requires that you always specify the `build_ext` command explicitly, and remember to provide **--inplace**. An easier way is to "set and forget" this option, by encoding it in 'setup.cfg', the configuration file for this distribution:

```
[build_ext]
inplace=1
```

This will affect all builds of this module distribution, whether or not you explcitly specify `build_ext`. If you include 'setup.cfg' in your source distribution, it will also affect end-user builds—which is probably a bad idea for this option, since always building extensions in-place would break installation of the module distribution. In certain peculiar cases, though, modules are built right in their installation directory, so this is conceivably a useful ability. (Distributing extensions that expect to be built in their installation directory is almost always a bad idea, though.)

Another example: certain commands take a lot of options that don't change from run-to-run; for example, `bdist_rpm` needs to know everything required to generate a "spec" file for creating an RPM distribution. Some of this information comes from the setup script, and some is automatically generated by the Distutils (such as the list of files installed). But some of it has to be supplied as options to `bdist_rpm`, which would be very tedious to do on the command-line for every run. Hence, here is a snippet from the Distutils' own 'setup.cfg':

```
[bdist_rpm]
release = 1
packager = Greg Ward <gward@python.net>
doc_files = CHANGES.txt
            README.txt
            USAGE.txt
            doc/
            examples/
```

Note that the doc_files option is simply a whitespace-separated string split across multiple lines for readability.

See Also:

Installing Python Modules
(../inst/config-syntax.html)
More information on the configuration files is available in the manual for system administrators.

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 format is gzip'ed tar file ('.tar.gz') on UNIX, and ZIP file on Windows. ****no MacOS support here****

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),(3)
`gztar`	gzip'ed tar file ('.tar.gz')	(2),(4)
`bztar`	bzip2'ed tar file ('.tar.gz')	(4)
`ztar`	compressed tar file ('.tar.Z')	(4)
`tar`	tar file ('.tar')	(4)

Notes:

(1) default on Windows

(2) default on UNIX

(3) requires either external **zip** utility or `zipfile` module (not part of the standard Python library)

(4) requires external utilities: **tar** and possibly one of **gzip**, **bzip2**, or **compress**

5.1 Specifying the files to distribute

If you don't supply an explicit list of files (or instructions on how to generate one), the `sdist` command puts a minimal default set 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'), 'setup.py' (or whatever you called your setup script), and 'setup.cfg'

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 manifest template is just a list of instructions for how to generate your manifest file, 'MANIFEST', which is the exact list of files to include in your source distribution. The `sdist` command processes this template and generates a manifest based on its instructions and what it finds in the filesystem.

If you prefer to roll your own manifest file, the format is simple: one filename per line, regular files (or symlinks to them) only. If you do supply your own 'MANIFEST', you must specify everything: the default set of files described above does not apply in this case.

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`. All of this is done *after* the standard include set, so you can exclude files from the standard set with explicit instructions in the manifest template. (Or, you can use the **--no-defaults** option to disable the standard set entirely.) There are several other commands available in the manifest template mini-language; see section 7.2.

The order of commands in the manifest template 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. Once we have fully processed the manifest template, we remove files that should not be included in the source distribution:

- all files in the Distutils "build" tree (default 'build/')
- all files in directories named 'RCS' or 'CVS'

Now we have our complete list of files, which is written to the manifest for future reference, and then used to build the source distribution archive(s).

You can disable the default set of included files with the **--no-defaults** option, and you can disable the standard exclude set with **--no-prune**.

Following the Distutils' own manifest template, let's trace how the `sdist` command builds 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—see section 3)
2. include 'README.txt', 'setup.py', and 'setup.cfg' (standard files)
3. include 'test/test*.py' (standard files)
4. include '*.txt' in the distribution root (this will find 'README.txt' a second time, but such redundancies are weeded out later)
5. include anything matching '*.txt' or '*.py' in the sub-tree under 'examples',
6. exclude 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 `recursive-include` command
7. exclude the entire 'build' tree, and any 'RCS' or 'CVS' directories

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 neither 'MANIFEST' nor 'MANIFEST.in' exist, create a manifest with just the default file set[3]
- 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)

[3]In versions of the Distutils up to and including 0.9.2 (Python 2.0b1), this feature was broken; use the **-f** (**--force-manifest**) option to work around the bug.

There are a couple of options that modify this behaviour. First, use the **--no-defaults** and **--no-prune** to disable the standard "include" and "exclude" sets.[4]

Second, 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**. **-o** is a shortcut for **--manifest-only**, and **-f** for **--force-manifest**.

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 are 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. The default format for built distributions is a "dumb" tar file on UNIX, and an simple executable installer on Windows. (That tar file is considered "dumb" because it has to 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 right place installs the Distutils just as though you had downloaded the source distribution and run `python setup.py install`. (The "right place" is either the root of the filesystem or Python's *prefix* directory, depending on the options given to the `bdist_dumb` command; the default is to make dumb distributions relative to *prefix*.)

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. And creating "smart" built distributions, such as an RPM package or an executable installer for Windows, is a big win for users even if your distribution doesn't include any extensions.

[4]Note that if you have no manifest template, no manifest, and use the **--no-defaults**, you will get an empty manifest. Another bug in Distutils 0.9.2 and earlier causes an uncaught exception in this case. The workaround is: Don't Do That.

The `bdist` command has a **--formats** 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
`gztar`	gzipped tar file ('.tar.gz')	(1),(3)
`ztar`	compressed tar file ('.tar.Z')	(3)
`tar`	tar file ('.tar')	(3)
`zip`	zip file ('.zip')	(4)
`rpm`	RPM	(5)
`srpm`	source RPM	(5) ****to do!****
`wininst`	self-extracting ZIP file for Windows	(2),(4)

Notes:

(1) default on UNIX

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

(3) requires external utilities: **tar** and possibly one of **gzip**, **bzip2**, or **compress**

(4) requires either external **zip** utility or `zipfile` module (not part of the standard Python library)

(5) requires external **rpm** utility, version 3.0.4 or better (use `rpm --version` to find out which version you have)

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

The following sections give details on the individual `bdist_*` commands.

6.1 Creating dumb built distributions

****Need to document absolute vs. prefix-relative packages here, but first I have to implement it!****

6.2 Creating RPM packages

The RPM format is used by many of popular Linux distributions, including Red Hat, SuSE, and Mandrake. If one of these (or any of the other RPM-based Linux distributions) is your usual environment, creating RPM packages for other users of that same distribution is trivial. Depending on the complexity of your module distribution and differences between Linux distributions, you may also be able to create RPMs that work on different RPM-based distributions.

The usual way to create an RPM of your module distribution is to run the `bdist_rpm` command:

```
python setup.py bdist_rpm
```

or the `bdist` command with the **--format** option:

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

The former allows you to specify RPM-specific options; the latter allows you to easily specify multiple formats in one run. If you need to do both, you can explicitly specify multiple `bdist_*` commands and their options:

```
python setup.py bdist_rpm --packager="John Doe <jdoe@python.net>" \
                bdist_wininst --target_version="2.0"
```

Creating RPM packages is driven by a '.spec' file, much as using the Distutils is driven by the setup script. To make your life easier, the `bdist_rpm` command normally creates a '.spec' file based on the information you supply in the setup script, on the command line, and in any Distutils configuration files. Various options and sections in the '.spec' file are derived from options in the setup script as follows:

RPM '.spec' file option or section	Distutils setup script option
Name	name
Summary (in preamble)	description
Version	version
Vendor	author and author_email, or maintainer and maintainer_email
Copyright	licence
Url	url
%description (section)	long_description

Additionally, there many options in '.spec' files that don't have corresponding options in the setup script. Most of these are handled through options to the `bdist_rpm` command as follows:

RPM '.spec' file option or section	`bdist_rpm` option	default value
Release	release	"1"
Group	group	"Development/Libraries"
Vendor	vendor	(see above)
Packager	packager	(none)
Provides	provides	(none)
Requires	requires	(none)
Conflicts	conflicts	(none)
Obsoletes	obsoletes	(none)
Distribution	distribution_name	(none)
BuildRequires	build_requires	(none)
Icon	icon	(none)

Obviously, supplying even a few of these options on the command-line would be tedious and error-prone, so it's usually best to put them in the setup configuration file, 'setup.cfg'—see section 4. If you distribute or package many Python module distributions, you might want to put options that apply to all of them in your personal Distutils configuration file ('~/.pydistutils.cfg').

There are three steps to building a binary RPM package, all of which are handled automatically by the Distutils:

1. create a '.spec' file, which describes the package (analogous to the Distutils setup script; in fact, much of the information in the setup script winds up in the '.spec' file)

2. create the source RPM

3. create the “binary” RPM (which may or may not contain binary code, depending on whether your module distribution contains Python extensions)

Normally, RPM bundles the last two steps together; when you use the Distutils, all three steps are typically bundled together.

If you wish, you can separate these three steps. You can use the **--spec-only** option to make `bdist_rpm` just create the ‘.spec’ file and exit; in this case, the ‘.spec’ file will be written to the “distribution directory”—normally ‘dist/’, but customizable with the **--dist-dir** option. (Normally, the ‘.spec’ file winds up deep in the “build tree,” in a temporary directory created by `bdist_rpm`.)

****this isn’t implemented yet—is it needed?!**** You can also specify a custom ‘.spec’ file with the **--spec-file** option; used in conjunction with **--spec-only**, this gives you an opportunity to customize the ‘.spec’ file manually:

```
> python setup.py bdist_rpm --spec-only
# ...edit dist/FooBar-1.0.spec
> python setup.py bdist_rpm --spec-file=dist/FooBar-1.0.spec
```

(Although a better way to do this is probably to override the standard `bdist_rpm` command with one that writes whatever else you want to the ‘.spec’ file; see section **??** for information on extending the Distutils.)

6.3 Creating Windows installers

Executable Windows installers are the natural format for binary distributions on Windows. They display a nice graphical user interface, display some information of the module distribution to be installed, taken from the meta-data in the setup script, let the user select a few (currently maybe too few) options, and start or cancel the installation.

Since the meta-data is taken from the setup script, creating Windows installers is usually as easy as running:

```
python setup.py bdist_wininst
```

or the `bdist` command with the **--format** option:

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

If you have a pure module distribution (only containing pure Python modules and packages), the resulting installer will be version independent and have a name like ‘Foo-1.0.win32.exe’. These installers can even be created on UNIX or MacOS platforms.

If you have a non-pure distribution, the extensions can only be created on a Windows platform, and will be Python version dependent. The installer filename will reflect this and now has the form ‘Foo-1.0.win32-py2.0.exe’. You have to create a separate installer for every Python version you want to support.

The installer will try to compile pure modules into bytecode after installation on the target system in normal and optimizing mode. If you don’t want this to happen for some reason, you can run the bdist_wininst command with the **--no-target-compile** and/or the **--no-target-optimize** option.

7 Reference

7.1 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_data`

This command installs all data files provided with the distribution.

`install_scripts`

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

7.2 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 MacOS anything except colon.

****Windows and MacOS support not there yet****

8 `distutils.sysconfig` — System configuration information

The `distutils.sysconfig` module provides access to Python's low-level configuration information. The specific configuration variables available depend heavily on the platform and configuration. The specific variables depend on the build process for the specific version of Python being run; the variables are those found in the 'Makefile' and configuration header that are installed with Python on UNIX systems. The configuration header is called 'pyconfig.h' for Python versions starting with 2.2, and 'config.h' for earlier versions of Python.

Some additional functions are provided which perform some useful manipulations for other parts of the `distutils` package.

`PREFIX`
: The result of `os.path.normpath(sys.prefex)`.

`EXEC_PREFIX`
: The result of `os.path.normpath(sys.exec_prefex)`.

`get_config_var`(*name*)
: Return the value of a single variable. This is equivalent to `get_config_vars().get(`*name*`)`.

`get_config_vars`(...)
: Return a set of variable definitions. If there are no arguments, this returns a dictionary mapping names of

configuration variables to values. If arguments are provided, they should be strings, and the return value will be a sequence giving the associated values. If a given name does not have a corresponding value, `None` will be included for that variable.

get_config_h_filename()
: Return the full path name of the configuration header. For UNIX, this will be the header generated by the **configure** script; for other platforms the header will have been supplied directly by the Python source distribution. The file is a platform-specific text file.

get_makefile_filename()
: Return the full path name of the 'Makefile' used to build Python. For UNIX, this will be a file generated by the **configure** script; the meaning for other platforms will vary. The file is a platform-specific text file, if it exists. This function is only useful on POSIX platforms.

get_python_inc([*plat_specific*[, *prefix*]])
: Return the directory for either the general or platform-dependent C include files. If *plat_specific* is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If *prefix* is given, it is used as either the prefix instead of `PREFIX`, or as the exec-prefix instead of `EXEC_PREFIX` if *plat_specific* is true.

get_python_lib([*plat_specific*[, *standard_lib*[, *prefix*]]])
: Return the directory for either the general or platform-dependent library installation. If *plat_specific* is true, the platform-dependent include directory is returned; if false or omitted, the platform-independent directory is returned. If *prefix* is given, it is used as either the prefix instead of `PREFIX`, or as the exec-prefix instead of `EXEC_PREFIX` if *plat_specific* is true. If *standard_lib* is true, the directory for the standard library is returned rather than the directory for the installation of third-party extensions.

The following function is only intended for use within the `distutils` package.

customize_compiler(*compiler*)
: Do any platform-specific customization of a `distutils.ccompiler.CCompiler` instance.

  This function is only needed on UNIX at this time, but should be called consistently to support forward-compatibility. It inserts the information that varies across UNIX flavors and is stored in Python's 'Makefile'. This information includes the selected compiler, compiler and linker options, and the extension used by the linker for shared objects.

This function is even more special-purpose, and should only be used from Python's own build procedures.

set_python_build()
: Inform the `distutils.sysconfig` module that it is being used as part of the build process for Python. This changes a lot of relative locations for files, allowing them to be located in the build area rather than in an installed Python.