Distributing Python Modules

Release 2.5.3

Greg Ward
Anthony Baxter

19th December, 2008

Python Software Foundation
Email: distutils-sig@python.org

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

CHAPTER

ONE

An Introduction to Distutils

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

1.1 Concepts & Terminology

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

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

Each of these tasks is covered in this document.

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

1.2 A Simple Example

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

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

```
from distutils.core import setup
setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Some observations:

- most information that you supply to the Distutils is supplied as keyword arguments to the `setup()` function
- those keyword arguments fall into two categories: package metadata (name, version number) and information about what's in the package (a list of pure Python modules, in this case)
- modules are specified by module name, not filename (the same will hold true for packages and extensions)
- it's recommended that you supply a little more metadata, in particular your name, email address and a URL for the project (see section 2 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.

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

```
python setup.py bdist_rpm
```

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

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

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

1.3 General Python terminology

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

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

pure Python module a module written in Python and contained in a single '.py' file (and possibly associated '.pyc' 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 Jython. Typically contained in a single dynamically loadable pre-compiled file, e.g. a shared object ('.so') file for Python extensions on UNIX, a DLL (given the '.pyd' extension) for Python extensions on Windows, or a Java class file for Jython extensions. (Note that currently, the Distutils only handles C/C++ extensions for Python.)

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

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

1.4 Distutils-specific terminology

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

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

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

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

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

CHAPTER

TWO

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 1.2 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 1.2: more metadata, 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. For more information on the additional meta-data, see section 2.8.

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 9 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.)

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'))
```

2.1 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 still 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 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.)

2.2 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 1.2; 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.

2.3 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 ext_modules option. ext_modules 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.

2.3.1 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.

2.3.2 Extension source files

The second argument to the Extension constructor is a list of source files. Since the Distutils currently only support C, C++, and Objective-C extensions, these are normally C/C++/Objective-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!****

This warning notwithstanding, options to SWIG can be currently passed like this:

```
setup(...
      ext_modules=[Extension('_foo', ['foo.i'],
                             swig_opts=['-modern', '-I../include'])],
      py_modules=['foo'],
     )
```

Or on the commandline like this:

```
> python setup.py build_ext --swig-opts="-modern -I../include"
```

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.

2.3.3 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 C code like

```
#include <X11/Xlib.h>
```

If you need to include header files from some other Python extension, you can take advantage of the fact that header files are installed in a consistent way by the Distutils `install_header` command. 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 write C code like

```
#include <Numerical/arrayobject.h>
```

If you must put the 'Numerical' include directory right into your header search path, though, you can find that directory using the Distutils `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
```

2.3.4 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!****

2.3.5 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 respective compiler and linker command lines.

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: Distutils will automatically add `initmodule` to the list of exported symbols.

2.4 Relationships between Distributions and Packages

A distribution may relate to packages in three specific ways:

1. It can require packages or modules.
2. It can provide packages or modules.
3. It can obsolete packages or modules.

These relationships can be specified using keyword arguments to the `distutils.core.setup()` function.

Dependencies on other Python modules and packages can be specified by supplying the *requires* keyword argument to `setup()`. The value must be a list of strings. Each string specifies a package that is required, and optionally what versions are sufficient.

To specify that any version of a module or package is required, the string should consist entirely of the module or package name. Examples include `'mymodule'` and `'xml.parsers.expat'`.

If specific versions are required, a sequence of qualifiers can be supplied in parentheses. Each qualifier may consist of a comparison operator and a version number. The accepted comparison operators are:

```
<    >    ==
<=   >=   !=
```

These can be combined by using multiple qualifiers separated by commas (and optional whitespace). In this case, all of the qualifiers must be matched; a logical AND is used to combine the evaluations.

Let's look at a bunch of examples:

Requires Expression	Explanation
`==1.0`	Only version `1.0` is compatible
`>1.0, !=1.5.1, <2.0`	Any version after `1.0` and before `2.0` is compatible, except `1.5.1`

Now that we can specify dependencies, we also need to be able to specify what we provide that other distributions can require. This is done using the *provides* keyword argument to `setup()`. The value for this keyword is a list of strings, each of which names a Python module or package, and optionally identifies the version. If the version is not specified, it is assumed to match that of the distribution.

Some examples:

Provides Expression	Explanation
`mypkg`	Provide `mypkg`, using the distribution version
`mypkg (1.1)`	Provide `mypkg` version 1.1, regardless of the distribution version

A package can declare that it obsoletes other packages using the *obsoletes* keyword argument. The value for this is similar to that of the *requires* keyword: a list of strings giving module or package specifiers. Each specifier consists of a module or package name optionally followed by one or more version qualifiers. Version qualifiers are given in parentheses after the module or package name.

The versions identified by the qualifiers are those that are obsoleted by the distribution being described. If no qualifiers are given, all versions of the named module or package are understood to be obsoleted.

2.5 Installing 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, intended to be started from the command line. Scripts don't require Distutils to do anything very complicated. The only clever feature is that if the first line of the script starts with `#!` and contains the word "python", the Distutils will adjust the first line to refer to the current interpreter location. By default, it is replaced with the current interpreter location. The **--executable** (or **-e**) option will allow the interpreter path to be explicitly overridden.

The scripts option simply is a list of files to be handled in this way. From the PyXML setup script:

```
setup(...
      scripts=['scripts/xmlproc_parse', 'scripts/xmlproc_val']
      )
```

2.6 Installing Package Data

Often, additional files need to be installed into a package. These files are often data that's closely related to the package's implementation, or text files containing documentation that might be of interest to programmers using the package. These files are called *package data*.

Package data can be added to packages using the `package_data` keyword argument to the `setup()` function. The value must be a mapping from package name to a list of relative path names that should be copied into the package. The paths are interpreted as relative to the directory containing the package (information from the `package_dir` mapping is used if appropriate); that is, the files are expected to be part of the package in the source directories. They may contain glob patterns as well.

The path names may contain directory portions; any necessary directories will be created in the installation.

For example, if a package should contain a subdirectory with several data files, the files can be arranged like this in the source tree:

```
setup.py
src/
    mypkg/
        __init__.py
        module.py
        data/
            tables.dat
            spoons.dat
            forks.dat
```

The corresponding call to `setup()` might be:

```
setup(...,
      packages=['mypkg'],
      package_dir={'mypkg': 'src/mypkg'},
      package_data={'mypkg': ['data/*.dat']},
      )
```

New in version 2.4.

2.7 Installing Additional Files

The data_files option can be used to specify additional files needed by the module distribution: configuration files, message catalogs, data files, anything which doesn't 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']),
                  ('/etc/init.d', ['init-script'])]
     )
```

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.

2.8 Additional meta-data

The setup script may include additional meta-data beyond the name and version. This information includes:

Meta-Data	Description	Value	Notes
`name`	name of the package	short string	(1)
`version`	version of this release	short string	(1)(2)
`author`	package author's name	short string	(3)
`author_email`	email address of the package author	email address	(3)
`maintainer`	package maintainer's name	short string	(3)
`maintainer_email`	email address of the package maintainer	email address	(3)
`url`	home page for the package	URL	(1)
`description`	short, summary description of the package	short string	
`long_description`	longer description of the package	long string	
`download_url`	location where the package may be downloaded	URL	(4)
`classifiers`	a list of classifiers	list of strings	(4)
`platforms`	a list of platforms	list of strings	

Notes:

(1) These fields are required.

(2) It is recommended that versions take the form *major.minor*[.*patch*[.*sub*]].

(3) Either the author or the maintainer must be identified.

(4) These fields should not be used if your package is to be compatible with Python versions prior to 2.2.3 or 2.3. The list is available from the PyPI website.

'short string' A single line of text, not more than 200 characters.

'long string' Multiple lines of plain text in reStructuredText format (see http://docutils.sf.net/).

'list of strings' See below.

None of the string values may be Unicode.

Encoding the version information is an art in itself. Python packages generally adhere to the version format *major.minor*[.*patch*][*sub*]. The major number is 0 for initial, experimental releases of software. It is incremented for releases that represent major milestones in a package. The minor number is incremented when important new features are added to the package. The patch number increments when bug-fix releases are made. Additional trailing version information is sometimes used to indicate sub-releases. These are "a1,a2,...,aN" (for alpha releases, where functionality and API may change), "b1,b2,...,bN" (for beta releases, which only fix bugs) and "pr1,pr2,...,prN" (for final pre-release release testing). Some examples:

0.1.0 the first, experimental release of a package

1.0.1a2 the second alpha release of the first patch version of 1.0

classifiers are specified in a python list:

```
setup(...
      classifiers=[
          'Development Status :: 4 - Beta',
          'Environment :: Console',
          'Environment :: Web Environment',
          'Intended Audience :: End Users/Desktop',
          'Intended Audience :: Developers',
          'Intended Audience :: System Administrators',
          'License :: OSI Approved :: Python Software Foundation License',
          'Operating System :: MacOS :: MacOS X',
          'Operating System :: Microsoft :: Windows',
          'Operating System :: POSIX',
          'Programming Language :: Python',
          'Topic :: Communications :: Email',
          'Topic :: Office/Business',
          'Topic :: Software Development :: Bug Tracking',
          ],
      )
```

If you wish to include classifiers in your 'setup.py' file and also wish to remain backwards-compatible with Python releases prior to 2.2.3, then you can include the following code fragment in your 'setup.py' before the `setup()` call.

```
# patch distutils if it can't cope with the "classifiers" or
# "download_url" keywords
from sys import version
if version < '2.2.3':
    from distutils.dist import DistributionMetadata
    DistributionMetadata.classifiers = None
    DistributionMetadata.download_url = None
```

2.9 Debugging the setup script

Sometimes things go wrong, and the setup script doesn't do what the developer wants.

Distutils catches any exceptions when running the setup script, and print a simple error message before the script is terminated. The motivation for this behaviour is to not confuse administrators who don't know much about Python and are trying to install a package. If they get a big long traceback from deep inside the guts of Distutils, they may think the package or the Python installation is broken because they don't read all the way down to the bottom and see that it's a permission problem.

On the other hand, this doesn't help the developer to find the cause of the failure. For this purpose, the DISTUTILS_-DEBUG environment variable can be set to anything except an empty string, and distutils will now print detailed information what it is doing, and prints the full traceback in case an exception occurs.

CHAPTER

THREE

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.

The setup configuration file is a useful middle-ground between the setup script—which, ideally, would be opaque to installers[1]—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, which run from a '`#`' character until the end of the 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.

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

```
> 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
  --swig-opts          list of SWIG command line options
[...]
```

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:

```
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 explicitly 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.

CHAPTER

FOUR

Creating a Source Distribution

As shown in section 1.2, 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 a gzip'ed tar file ('.tar.gz') on UNIX, and ZIP file on Windows.

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.bz2')	(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 (part of the standard Python library since Python 1.6)

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

4.1 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'!****)
- scripts identified by the scripts option
- 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 10.2.

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', 'CVS', '.svn', '.hg', '.git', '.bzr', or '_darcs'

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 2)
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', 'CVS', '.svn', '.hg', '.git', '.bzr', or '_darcs' 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.

4.2 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
- 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, use the **--no-defaults** and **--no-prune** to disable the standard "include" and "exclude" sets.

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**.

CHAPTER

FIVE

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 world of mainstream desktop systems.)

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 called *packagers* 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 who they are, 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 a 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-1.0.*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 any simpler than just running `python setup.py install`—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 far more convenient for users even if your distribution doesn't include any extensions.

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-1.0.*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)
pkgtool	Solaris **pkgtool**	
sdux	HP-UX **swinstall**	
rpm	RPM	(5)
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 (part of the standard Python library since Python 1.6)

(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.

5.1 Creating dumb built distributions

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

5.2 Creating RPM packages

The RPM format is used by many 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@example.org>" \
                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 are 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 3. 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`.)

5.3 Creating Windows Installers

Executable installers are the natural format for binary distributions on Windows. They display a nice graphical user interface, display some information about the module distribution to be installed taken from the metadata in the setup script, let the user select a few options, and start or cancel the installation.

Since the metadata 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 **--formats** 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 Mac OS 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.

By default the installer will display the cool "Python Powered" logo when it is run, but you can also supply your own bitmap which must be a Windows '.bmp' file with the **--bitmap** option.

The installer will also display a large title on the desktop background window when it is run, which is constructed from the name of your distribution and the version number. This can be changed to another text by using the **--title** option.

The installer file will be written to the "distribution directory" — normally 'dist/', but customizable with the **--dist-dir** option.

5.3.1 The Postinstallation script

Starting with Python 2.3, a postinstallation script can be specified which the **--install-script** option. The basename of the script must be specified, and the script filename must also be listed in the scripts argument to the setup function.

This script will be run at installation time on the target system after all the files have been copied, with `argv[1]` set to **-install**, and again at uninstallation time before the files are removed with `argv[1]` set to **-remove**.

The installation script runs embedded in the windows installer, every output (`sys.stdout`, `sys.stderr`) is redirected into a buffer and will be displayed in the GUI after the script has finished.

Some functions especially useful in this context are available as additional built-in functions in the installation script.

`directory_created`(*path*)
`file_created`(*path*)

These functions should be called when a directory or file is created by the postinstall script at installation time. It will register *path* with the uninstaller, so that it will be removed when the distribution is uninstalled. To be safe, directories are only removed if they are empty.

`get_special_folder_path`(*csidl_string*)

This function can be used to retrieve special folder locations on Windows like the Start Menu or the Desktop. It returns the full path to the folder. *csidl_string* must be one of the following strings:

```
"CSIDL_APPDATA"

"CSIDL_COMMON_STARTMENU"
"CSIDL_STARTMENU"

"CSIDL_COMMON_DESKTOPDIRECTORY"
"CSIDL_DESKTOPDIRECTORY"

"CSIDL_COMMON_STARTUP"
"CSIDL_STARTUP"

"CSIDL_COMMON_PROGRAMS"
"CSIDL_PROGRAMS"

"CSIDL_FONTS"
```

If the folder cannot be retrieved, `OSError` is raised.

Which folders are available depends on the exact Windows version, and probably also the configuration. For details refer to Microsoft's documentation of the `SHGetSpecialFolderPath()` function.

`create_shortcut`(*target, description, filename*[*, arguments*[*, workdir*[*, iconpath*[*, iconindex*]]]])

This function creates a shortcut. *target* is the path to the program to be started by the shortcut. *description* is the description of the shortcut. *filename* is the title of the shortcut that the user will see. *arguments* specifies the command line arguments, if any. *workdir* is the working directory for the program. *iconpath* is the file containing the icon for the shortcut, and *iconindex* is the index of the icon in the file *iconpath*. Again, for details consult the Microsoft documentation for the `IShellLink` interface.

CHAPTER

SIX

Registering with the Package Index

The Python Package Index (PyPI) holds meta-data describing distributions packaged with distutils. The distutils command `register` is used to submit your distribution's meta-data to the index. It is invoked as follows:

```
python setup.py register
```

Distutils will respond with the following prompt:

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

Note: if your username and password are saved locally, you will not see this menu.

If you have not registered with PyPI, then you will need to do so now. You should choose option 2, and enter your details as required. Soon after submitting your details, you will receive an email which will be used to confirm your registration.

Once you are registered, you may choose option 1 from the menu. You will be prompted for your PyPI username and password, and `register` will then submit your meta-data to the index.

You may submit any number of versions of your distribution to the index. If you alter the meta-data for a particular version, you may submit it again and the index will be updated.

PyPI holds a record for each (name, version) combination submitted. The first user to submit information for a given name is designated the Owner of that name. They may submit changes through the `register` command or through the web interface. They may also designate other users as Owners or Maintainers. Maintainers may edit the package information, but not designate other Owners or Maintainers.

By default PyPI will list all versions of a given package. To hide certain versions, the Hidden property should be set to yes. This must be edited through the web interface.

6.1 The .pypirc file

The format of the '.pypirc' file is formated as follows:

```
[server-login]
repository: <repository-url>
username: <username>
password: <password>
```

repository can be ommitted and defaults to `http://www.python.org/pypi`.

CHAPTER

SEVEN

Uploading Packages to the Package Index

New in version 2.5.

The Python Package Index (PyPI) not only stores the package info, but also the package data if the author of the package wishes to. The distutils command `upload` pushes the distribution files to PyPI.

The command is invoked immediately after building one or more distribution files. For example, the command

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

will cause the source distribution and the Windows installer to be uploaded to PyPI. Note that these will be uploaded even if they are built using an earlier invocation of 'setup.py', but that only distributions named on the command line for the invocation including the `upload` command are uploaded.

The `upload` command uses the username, password, and repository URL from the '$HOME/.pypirc' file (see section 6.1 for more on this file).

You can use the **--sign** option to tell `upload` to sign each uploaded file using GPG (GNU Privacy Guard). The **gpg** program must be available for execution on the system PATH. You can also specify which key to use for signing using the **--identity=*name*** option.

Other `upload` options include **--repository=*url*** (which lets you override the repository setting from '$HOME/.pypirc'), and **--show-response** (which displays the full response text from the PyPI server for help in debugging upload problems).

CHAPTER

EIGHT

Examples

This chapter provides a number of basic examples to help get started with distutils. Additional information about using distutils can be found in the Distutils Cookbook.

See Also:

Distutils Cookbook
    Collection of recipes showing how to achieve more control over distutils.

8.1 Pure Python distribution (by module)

If you're just distributing a couple of modules, especially if they don't live in a particular package, you can specify them individually using the py_modules option in the setup script.

In the simplest case, you'll have two files to worry about: a setup script and the single module you're distributing, 'foo.py' in this example:

```
<root>/
        setup.py
        foo.py
```

(In all diagrams in this section, `<root>` will refer to the distribution root directory.) A minimal setup script to describe this situation would be:

```
from distutils.core import setup
setup(name='foo',
      version='1.0',
      py_modules=['foo'],
      )
```

Note that the name of the distribution is specified independently with the name option, and there's no rule that says it has to be the same as the name of the sole module in the distribution (although that's probably a good convention to follow). However, the distribution name is used to generate filenames, so you should stick to letters, digits, underscores, and hyphens.

Since py_modules is a list, you can of course specify multiple modules, eg. if you're distributing modules `foo` and `bar`, your setup might look like this:

```
<root>/
        setup.py
        foo.py
        bar.py
```

and the setup script might be

```
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      py_modules=['foo', 'bar'],
      )
```

You can put module source files into another directory, but if you have enough modules to do that, it's probably easier to specify modules by package rather than listing them individually.

8.2 Pure Python distribution (by package)

If you have more than a couple of modules to distribute, especially if they are in multiple packages, it's probably easier to specify whole packages rather than individual modules. This works even if your modules are not in a package; you can just tell the Distutils to process modules from the root package, and that works the same as any other package (except that you don't have to have an '__init__.py' file).

The setup script from the last example could also be written as

```
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      packages=[''],
      )
```

(The empty string stands for the root package.)

If those two files are moved into a subdirectory, but remain in the root package, e.g.:

```
<root>/
        setup.py
        src/      foo.py
                  bar.py
```

then you would still specify the root package, but you have to tell the Distutils where source files in the root package live:

```
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'': 'src'},
      packages=[''],
      )
```

More typically, though, you will want to distribute multiple modules in the same package (or in sub-packages). For example, if the `foo` and `bar` modules belong in package `foobar`, one way to layout your source tree is

```
<root>/
        setup.py
        foobar/
                 __init__.py
                 foo.py
                 bar.py
```

This is in fact the default layout expected by the Distutils, and the one that requires the least work to describe in your setup script:

```
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      packages=['foobar'],
      )
```

If you want to put modules in directories not named for their package, then you need to use the package_dir option again. For example, if the 'src' directory holds modules in the `foobar` package:

```
<root>/
        setup.py
        src/
                 __init__.py
                 foo.py
                 bar.py
```

an appropriate setup script would be

```
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'foobar': 'src'},
      packages=['foobar'],
      )
```

Or, you might put modules from your main package right in the distribution root:

```
<root>/
        setup.py
        __init__.py
        foo.py
        bar.py
```

in which case your setup script would be

```
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      package_dir={'foobar': ''},
      packages=['foobar'],
      )
```

(The empty string also stands for the current directory.)

If you have sub-packages, they must be explicitly listed in packages, but any entries in package_dir automatically extend to sub-packages. (In other words, the Distutils does *not* scan your source tree, trying to figure out which directories correspond to Python packages by looking for '__init__.py' files.) Thus, if the default layout grows a sub-package:

```
<root>/
        setup.py
        foobar/
                 __init__.py
                 foo.py
                 bar.py
                 subfoo/
                           __init__.py
                           blah.py
```

then the corresponding setup script would be

```
from distutils.core import setup
setup(name='foobar',
      version='1.0',
      packages=['foobar', 'foobar.subfoo'],
      )
```

(Again, the empty string in package_dir stands for the current directory.)

8.3 Single extension module

Extension modules are specified using the ext_modules option. package_dir has no effect on where extension source files are found; it only affects the source for pure Python modules. The simplest case, a single extension module in a single C source file, is:

```
<root>/
        setup.py
        foo.c
```

If the foo extension belongs in the root package, the setup script for this could be

```
from distutils.core import setup
from distutils.extension import Extension
setup(name='foobar',
      version='1.0',
      ext_modules=[Extension('foo', ['foo.c'])],
      )
```

If the extension actually belongs in a package, say foopkg, then

With exactly the same source tree layout, this extension can be put in the foopkg package simply by changing the name of the extension:

```
from distutils.core import setup
from distutils.extension import Extension
setup(name='foobar',
      version='1.0',
      ext_modules=[Extension('foopkg.foo', ['foo.c'])],
      )
```

CHAPTER

NINE

Extending Distutils

Distutils can be extended in various ways. Most extensions take the form of new commands or replacements for existing commands. New commands may be written to support new types of platform-specific packaging, for example, while replacements for existing commands may be made to modify details of how the command operates on a package.

Most extensions of the distutils are made within 'setup.py' scripts that want to modify existing commands; many simply add a few file extensions that should be copied into packages in addition to '.py' files as a convenience.

Most distutils command implementations are subclasses of the `Command` class from `distutils.cmd`. New commands may directly inherit from `Command`, while replacements often derive from `Command` indirectly, directly subclassing the command they are replacing. Commands are required to derive from `Command`.

9.1 Integrating new commands

There are different ways to integrate new command implementations into distutils. The most difficult is to lobby for the inclusion of the new features in distutils itself, and wait for (and require) a version of Python that provides that support. This is really hard for many reasons.

The most common, and possibly the most reasonable for most needs, is to include the new implementations with your 'setup.py' script, and cause the `distutils.core.setup()` function use them:

```
from distutils.command.build_py import build_py as _build_py
from distutils.core import setup

class build_py(_build_py):
    """Specialized Python source builder."""

    # implement whatever needs to be different...

setup(cmdclass={'build_py': build_py},
      ...)
```

This approach is most valuable if the new implementations must be used to use a particular package, as everyone interested in the package will need to have the new command implementation.

Beginning with Python 2.4, a third option is available, intended to allow new commands to be added which can support existing 'setup.py' scripts without requiring modifications to the Python installation. This is expected to allow third-party extensions to provide support for additional packaging systems, but the commands can be used for anything distutils commands can be used for. A new configuration option, command_packages (command-line option **--command-packages**), can be used to specify additional packages to be searched for modules implementing commands. Like all distutils options, this can be specified on the command line or in a configuration file. This option can only be set in the `[global]` section of a configuration file, or before any commands on the command line. If set in a configuration file, it can be overridden from the command line; setting it to an empty string on the command line causes the default to be used. This should never be set in a configuration file provided with a package.

This new option can be used to add any number of packages to the list of packages searched for command implementations; multiple package names should be separated by commas. When not specified, the search is only performed in the `distutils.command` package. When 'setup.py' is run with the option **--command-packages distcmds,buildcmds**, however, the packages `distutils.command`, `distcmds`, and `buildcmds` will be searched in that order. New commands are expected to be implemented in modules of the same name as the command by classes sharing the same name. Given the example command line option above, the command `bdist_openpkg` could be implemented by the class `distcmds.bdist_openpkg.bdist_openpkg` or `buildcmds.bdist_openpkg.bdist_openpkg`.

9.2 Adding new distribution types

Commands that create distributions (files in the 'dist/' directory) need to add (*command*, *filename*) pairs to `self.distribution.dist_files` so that `upload` can upload it to PyPI. The *filename* in the pair contains no path information, only the name of the file itself. In dry-run mode, pairs should still be added to represent what would have been created.

CHAPTER

TEN

Command Reference

10.1 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`.

10.1.1 `install_data`

This command installs all data files provided with the distribution.

10.1.2 `install_scripts`

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

10.2 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 Mac OS 9 anything except colon.

****Windows support not there yet****

CHAPTER

ELEVEN

API Reference

11.1 distutils.core — Core Distutils functionality

The distutils.core module is the only module that needs to be installed to use the Distutils. It provides the setup() (which is called from the setup script). Indirectly provides the distutils.dist.Distribution and distutils.cmd.Command class.

setup(*arguments*)

The basic do-everything function that does most everything you could ever ask for from a Distutils method. See XXXXX

The setup function takes a large number of arguments. These are laid out in the following table.

argument name	value	type
name	The name of the package	a string
version	The version number of the package	See distutils.version
description	A single line describing the package	a string
long_description	Longer description of the package	a string
author	The name of the package author	a string
author_email	The email address of the package author	a string
maintainer	The name of the current maintainer, if different from the author	a string
maintainer_email	The email address of the current maintainer, if different from the author	
url	A URL for the package (homepage)	a URL
download_url	A URL to download the package	a URL
packages	A list of Python packages that distutils will manipulate	a list of strings
py_modules	A list of Python modules that distutils will manipulate	a list of strings
scripts	A list of standalone script files to be built and installed	a list of strings
ext_modules	A list of Python extensions to be built	A list of instances of distutils.core.Extension
classifiers	A list of categories for the package	The list of available categorizations is at http://cheeseshop.python.org/pypi?:action=list_classifiers.
distclass	the Distribution class to use	A subclass of distutils.core.Distribution
script_name	The name of the setup.py script - defaults to sys.argv[0]	a string
script_args	Arguments to supply to the setup script	a list of strings
options	default options for the setup script	a string
license	The license for the package	
keywords	Descriptive meta-data. See PEP 314	
platforms		
cmdclass	A mapping of command names to Command subclasses	a dictionary

run_setup(*script_name*[, *script_args*=None, *stop_after*='run'])

Run a setup script in a somewhat controlled environment, and return the distutils.dist.Distribution instance that drives things. This is useful if you need to find out the distribution meta-data (passed as keyword args from *script* to setup()), or the contents of the config files or command-line.

script_name is a file that will be run with `execfile()` `sys.argv[0]` will be replaced with *script* for the duration of the call. *script_args* is a list of strings; if supplied, `sys.argv[1:]` will be replaced by *script_args* for the duration of the call.

stop_after tells `setup()` when to stop processing; possible values:

value	description
	Stop after the `Distribution` instance has been created and populated with the keyword arguments to `setup()`
	Stop after config files have been parsed (and their data stored in the `Distribution` instance)
	Stop after the command-line (`sys.argv[1:]` or *script_args*) have been parsed (and the data stored in the `Distribution` instance.)
	Stop after all commands have been run (the same as if `setup()` had been called in the usual way). This is the default value.

In addition, the `distutils.core` module exposed a number of classes that live elsewhere.

- `Extension` from `distutils.extension`
- `Command` from `distutils.cmd`
- `Distribution` from `distutils.dist`

A short description of each of these follows, but see the relevant module for the full reference.

class `Extension`

The Extension class describes a single C or C++extension module in a setup script. It accepts the following keyword arguments in its constructor

argument name	value	type
name	the full name of the extension, including any packages — ie. *not* a filename or pathname, but Python dotted name	string
sources	list of source filenames, relative to the distribution root (where the setup script lives), in UNIX form (slash-separated) for portability. Source files may be C, C++, SWIG (.i), platform-specific resource files, or whatever else is recognized by the `build_ext` command as source for a Python extension.	string
include_dirs	list of directories to search for C/C++ header files (in UNIX form for portability)	string
define_macros	list of macros to define; each macro is defined using a 2-tuple, where 'value' is either the string to define it to or `None` to define it without a particular value (equivalent of `#define FOO` in source or **-DFOO** on UNIX C compiler command line)	(string,string) tuple or (name,`None`)
undef_macros	list of macros to undefine explicitly	string
library_dirs	list of directories to search for C/C++ libraries at link time	string
libraries	list of library names (not filenames or paths) to link against	string
runtime_library_dirs	list of directories to search for C/C++ libraries at run time (for shared extensions, this is when the extension is loaded)	string
extra_objects	list of extra files to link with (eg. object files not implied by 'sources', static library that must be explicitly specified, binary resource files, etc.)	string
extra_compile_args	any extra platform- and compiler-specific information to use when compiling the source files in 'sources'. For platforms and compilers where a command line makes sense, this is typically a list of command-line arguments, but for other platforms it could be anything.	string
extra_link_args	any extra platform- and compiler-specific information to use when linking object files together to create the extension (or to create a new static Python interpreter). Similar interpretation as for 'extra_compile_args'.	string
export_symbols	list of symbols to be exported from a shared extension. Not used on all platforms, and not generally necessary for Python extensions, which typically export exactly one symbol: `init` + extension_name.	string
depends	list of files that the extension depends on	string
language	extension language (i.e. `'c'`, `'c++'`, `'objc'`). Will be detected from the source extensions if not provided.	string

class `Distribution`

A `Distribution` describes how to build, install and package up a Python software package.

See the `setup()` function for a list of keyword arguments accepted by the Distribution constructor. `setup()` creates a Distribution instance.

class `Command`

A `Command` class (or rather, an instance of one of its subclasses) implement a single distutils command.

11.2 `distutils.ccompiler` — CCompiler base class

This module provides the abstract base class for the `CCompiler` classes. A `CCompiler` instance can be used for all the compile and link steps needed to build a single project. Methods are provided to set options for the compiler — macro definitions, include directories, link path, libraries and the like.

This module provides the following functions.

gen_lib_options (*compiler, library_dirs, runtime_library_dirs, libraries*)
Generate linker options for searching library directories and linking with specific libraries. *libraries* and *library_dirs* are, respectively, lists of library names (not filenames!) and search directories. Returns a list of command-line options suitable for use with some compiler (depending on the two format strings passed in).

gen_preprocess_options (*macros, include_dirs*)
Generate C pre-processor options (**-D**, **-U**, **-I**) as used by at least two types of compilers: the typical UNIX compiler and Visual C++. *macros* is the usual thing, a list of 1- or 2-tuples, where `(`*name*`,)` means undefine (**-U**) macro *name*, and `(`*name*`, `*value*`)` means define (**-D**) macro *name* to *value*. *include_dirs* is just a list of directory names to be added to the header file search path (**-I**). Returns a list of command-line options suitable for either UNIX compilers or Visual C++.

get_default_compiler (*osname, platform*)
Determine the default compiler to use for the given platform.

osname should be one of the standard Python OS names (i.e. the ones returned by `os.name`) and *platform* the common value returned by `sys.platform` for the platform in question.

The default values are `os.name` and `sys.platform` in case the parameters are not given.

new_compiler (*plat=None, compiler=None, verbose=0, dry_run=0, force=0*)
Factory function to generate an instance of some CCompiler subclass for the supplied platform/compiler combination. *plat* defaults to `os.name` (eg. `'posix'`, `'nt'`), and *compiler* defaults to the default compiler for that platform. Currently only `'posix'` and `'nt'` are supported, and the default compilers are "traditional UNIX interface" (`UnixCCompiler` class) and Visual C++(`MSVCCompiler` class). Note that it's perfectly possible to ask for a UNIX compiler object under Windows, and a Microsoft compiler object under UNIX—if you supply a value for *compiler*, *plat* is ignored.

show_compilers ()
Print list of available compilers (used by the **--help-compiler** options to `build`, `build_ext`, `build_clib`).

class **CCompiler** ([*verbose=0, dry_run=0, force=0*])
The abstract base class `CCompiler` defines the interface that must be implemented by real compiler classes. The class also has some utility methods used by several compiler classes.

The basic idea behind a compiler abstraction class is that each instance can be used for all the compile/link steps in building a single project. Thus, attributes common to all of those compile and link steps — include directories, macros to define, libraries to link against, etc. — are attributes of the compiler instance. To allow for variability in how individual files are treated, most of those attributes may be varied on a per-compilation or per-link basis.

The constructor for each subclass creates an instance of the Compiler object. Flags are *verbose* (show verbose output), *dry_run* (don't actually execute the steps) and *force* (rebuild everything, regardless of dependencies). All of these flags default to `0` (off). Note that you probably don't want to instantiate `CCompiler` or one of its subclasses directly - use the `distutils.CCompiler.new_compiler()` factory function instead.

The following methods allow you to manually alter compiler options for the instance of the Compiler class.

add_include_dir (*dir*)
Add *dir* to the list of directories that will be searched for header files. The compiler is instructed to search directories in the order in which they are supplied by successive calls to `add_include_dir()`.

set_include_dirs (*dirs*)
Set the list of directories that will be searched to *dirs* (a list of strings). Overrides any preceding calls to `add_include_dir()`; subsequent calls to `add_include_dir()` add to the list passed to `set_include_dirs()`. This does not affect any list of standard include directories that the compiler may search by default.

add_library (*libname*)
Add *libname* to the list of libraries that will be included in all links driven by this compiler object. Note that *libname* should *not* be the name of a file containing a library, but the name of the library itself:

the actual filename will be inferred by the linker, the compiler, or the compiler class (depending on the platform).

The linker will be instructed to link against libraries in the order they were supplied to `add_library()` and/or `set_libraries()`. It is perfectly valid to duplicate library names; the linker will be instructed to link against libraries as many times as they are mentioned.

`set_libraries`(*libnames*)
Set the list of libraries to be included in all links driven by this compiler object to *libnames* (a list of strings). This does not affect any standard system libraries that the linker may include by default.

`add_library_dir`(*dir*)
Add *dir* to the list of directories that will be searched for libraries specified to `add_library()` and `set_libraries()`. The linker will be instructed to search for libraries in the order they are supplied to `add_library_dir()` and/or `set_library_dirs()`.

`set_library_dirs`(*dirs*)
Set the list of library search directories to *dirs* (a list of strings). This does not affect any standard library search path that the linker may search by default.

`add_runtime_library_dir`(*dir*)
Add *dir* to the list of directories that will be searched for shared libraries at runtime.

`set_runtime_library_dirs`(*dirs*)
Set the list of directories to search for shared libraries at runtime to *dirs* (a list of strings). This does not affect any standard search path that the runtime linker may search by default.

`define_macro`(*name*[, *value*=`None`])
Define a preprocessor macro for all compilations driven by this compiler object. The optional parameter *value* should be a string; if it is not supplied, then the macro will be defined without an explicit value and the exact outcome depends on the compiler used (XXX true? does ANSI say anything about this?)

`undefine_macro`(*name*)
Undefine a preprocessor macro for all compilations driven by this compiler object. If the same macro is defined by `define_macro()` and undefined by `undefine_macro()` the last call takes precedence (including multiple redefinitions or undefinitions). If the macro is redefined/undefined on a per-compilation basis (ie. in the call to `compile()`), then that takes precedence.

`add_link_object`(*object*)
Add *object* to the list of object files (or analogues, such as explicitly named library files or the output of "resource compilers") to be included in every link driven by this compiler object.

`set_link_objects`(*objects*)
Set the list of object files (or analogues) to be included in every link to *objects*. This does not affect any standard object files that the linker may include by default (such as system libraries).

The following methods implement methods for autodetection of compiler options, providing some functionality similar to GNU **autoconf**.

`detect_language`(*sources*)
Detect the language of a given file, or list of files. Uses the instance attributes `language_map` (a dictionary), and `language_order` (a list) to do the job.

`find_library_file`(*dirs, lib*[, *debug*=`0`])
Search the specified list of directories for a static or shared library file *lib* and return the full path to that file. If *debug* is true, look for a debugging version (if that makes sense on the current platform). Return `None` if *lib* wasn't found in any of the specified directories.

`has_function`(*funcname* [, *includes*=`None`, *include_dirs*=`None`, *libraries*=`None`, *library_dirs*=`None`])
Return a boolean indicating whether *funcname* is supported on the current platform. The optional arguments can be used to augment the compilation environment by providing additional include files and paths and libraries and paths.

`library_dir_option`(*dir*)
Return the compiler option to add *dir* to the list of directories searched for libraries.

library_option(*lib*)
Return the compiler option to add *dir* to the list of libraries linked into the shared library or executable.

runtime_library_dir_option(*dir*)
Return the compiler option to add *dir* to the list of directories searched for runtime libraries.

set_executables(***args*)
Define the executables (and options for them) that will be run to perform the various stages of compilation. The exact set of executables that may be specified here depends on the compiler class (via the 'executables' class attribute), but most will have:

attribute	description
compiler	the C/C++ compiler
linker_so	linker used to create shared objects and libraries
linker_exe	linker used to create binary executables
archiver	static library creator

On platforms with a command-line (UNIX, DOS/Windows), each of these is a string that will be split into executable name and (optional) list of arguments. (Splitting the string is done similarly to how UNIX shells operate: words are delimited by spaces, but quotes and backslashes can override this. See `distutils.util.split_quoted()`.)

The following methods invoke stages in the build process.

compile(*sources*[, *output_dir*=`None`, *macros*=`None`, *include_dirs*=`None`, *debug*=`0`, *extra_preargs*=`None`, *extra_postargs*=`None`, *depends*=`None`])
Compile one or more source files. Generates object files (e.g. transforms a '.c' file to a '.o' file.)

sources must be a list of filenames, most likely C/C++files, but in reality anything that can be handled by a particular compiler and compiler class (eg. `MSVCCompiler` can handle resource files in *sources*). Return a list of object filenames, one per source filename in *sources*. Depending on the implementation, not all source files will necessarily be compiled, but all corresponding object filenames will be returned.

If *output_dir* is given, object files will be put under it, while retaining their original path component. That is, 'foo/bar.c' normally compiles to 'foo/bar.o' (for a UNIX implementation); if *output_dir* is *build*, then it would compile to 'build/foo/bar.o'.

macros, if given, must be a list of macro definitions. A macro definition is either a (*name*, *value*) 2-tuple or a (*name*,) 1-tuple. The former defines a macro; if the value is `None`, the macro is defined without an explicit value. The 1-tuple case undefines a macro. Later definitions/redefinitions/undefinitions take precedence.

include_dirs, if given, must be a list of strings, the directories to add to the default include file search path for this compilation only.

debug is a boolean; if true, the compiler will be instructed to output debug symbols in (or alongside) the object file(s).

extra_preargs and *extra_postargs* are implementation-dependent. On platforms that have the notion of a command-line (e.g. UNIX, DOS/Windows), they are most likely lists of strings: extra command-line arguments to prepend/append to the compiler command line. On other platforms, consult the implementation class documentation. In any event, they are intended as an escape hatch for those occasions when the abstract compiler framework doesn't cut the mustard.

depends, if given, is a list of filenames that all targets depend on. If a source file is older than any file in depends, then the source file will be recompiled. This supports dependency tracking, but only at a coarse granularity.

Raises `CompileError` on failure.

create_static_lib(*objects, output_libname*[, *output_dir*=`None`, *debug*=`0`, *target_lang*=`None`])
Link a bunch of stuff together to create a static library file. The "bunch of stuff" consists of the list of object files supplied as *objects*, the extra object files supplied to `add_link_object()` and/or `set_link_objects()`, the libraries supplied to `add_library()` and/or `set_libraries()`, and the libraries supplied as *libraries* (if any).

output_libname should be a library name, not a filename; the filename will be inferred from the library name. *output_dir* is the directory where the library file will be put. XXX defaults to what?

debug is a boolean; if true, debugging information will be included in the library (note that on most platforms, it is the compile step where this matters: the *debug* flag is included here just for consistency).

target_lang is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.

Raises `LibError` on failure.

`link`(*target_desc, objects, output_filename*[, *output_dir*=`None`, *libraries*=`None`, *library_dirs*=`None`, *runtime_library_dirs*=`None`, *export_symbols*=`None`, *debug*=`0`, *extra_preargs*=`None`, *extra_postargs*=`None`, *build_temp*=`None`, *target_lang*=`None`])

Link a bunch of stuff together to create an executable or shared library file.

The "bunch of stuff" consists of the list of object files supplied as *objects*. *output_filename* should be a filename. If *output_dir* is supplied, *output_filename* is relative to it (i.e. *output_filename* can provide directory components if needed).

libraries is a list of libraries to link against. These are library names, not filenames, since they're translated into filenames in a platform-specific way (eg. *foo* becomes 'libfoo.a' on UNIX and 'foo.lib' on DOS/Windows). However, they can include a directory component, which means the linker will look in that specific directory rather than searching all the normal locations.

library_dirs, if supplied, should be a list of directories to search for libraries that were specified as bare library names (ie. no directory component). These are on top of the system default and those supplied to `add_library_dir()` and/or `set_library_dirs()`. *runtime_library_dirs* is a list of directories that will be embedded into the shared library and used to search for other shared libraries that *it* depends on at run-time. (This may only be relevant on UNIX.)

export_symbols is a list of symbols that the shared library will export. (This appears to be relevant only on Windows.)

debug is as for `compile()` and `create_static_lib()`, with the slight distinction that it actually matters on most platforms (as opposed to `create_static_lib()`, which includes a *debug* flag mostly for form's sake).

extra_preargs and *extra_postargs* are as for `compile()` (except of course that they supply command-line arguments for the particular linker being used).

target_lang is the target language for which the given objects are being compiled. This allows specific linkage time treatment of certain languages.

Raises `LinkError` on failure.

`link_executable`(*objects, output_progname*[, *output_dir*=`None`, *libraries*=`None`, *library_dirs*=`None`, *runtime_library_dirs*=`None`, *debug*=`0`, *extra_preargs*=`None`, *extra_postargs*=`None`, *target_lang*=`None`])

Link an executable. *output_progname* is the name of the file executable, while *objects* are a list of object filenames to link in. Other arguments are as for the `link` method.

`link_shared_lib`(*objects, output_libname*[, *output_dir*=`None`, *libraries*=`None`, *library_dirs*=`None`, *runtime_library_dirs*=`None`, *export_symbols*=`None`, *debug*=`0`, *extra_preargs*=`None`, *extra_postargs*=`None`, *build_temp*=`None`, *target_lang*=`None`])

Link a shared library. *output_libname* is the name of the output library, while *objects* is a list of object filenames to link in. Other arguments are as for the `link` method.

`link_shared_object`(*objects, output_filename*[, *output_dir*=`None`, *libraries*=`None`, *library_dirs*=`None`, *runtime_library_dirs*=`None`, *export_symbols*=`None`, *debug*=`0`, *extra_preargs*=`None`, *extra_postargs*=`None`, *build_temp*=`None`, *target_lang*=`None`])

Link a shared object. *output_filename* is the name of the shared object that will be created, while *objects* is a list of object filenames to link in. Other arguments are as for the `link` method.

`preprocess`(*source*[, *output_file*=`None`, *macros*=`None`, *include_dirs*=`None`, *extra_preargs*=`None`, *extra_postargs*=`None`])

Preprocess a single C/C++ source file, named in *source*. Output will be written to file named *output_file*, or *stdout* if *output_file* not supplied. *macros* is a list of macro definitions as for `compile()`, which will augment the macros set with `define_macro()` and `undefine_macro()`. *include_dirs* is a list of directory names that will be added to the default list, in the same way as `add_include_dir()`.

Raises `PreprocessError` on failure.

The following utility methods are defined by the `CCompiler` class, for use by the various concrete subclasses.

`executable_filename` (*basename*[, *strip_dir=0, output_dir=''*])
Returns the filename of the executable for the given *basename*. Typically for non-Windows platforms this is the same as the basename, while Windows will get a '.exe' added.

`library_filename` (*libname*[, *lib_type='static', strip_dir=0, output_dir=''*])
Returns the filename for the given library name on the current platform. On UNIX a library with *lib_type* of `'static'` will typically be of the form 'liblibname.a', while a *lib_type* of `'dynamic'` will be of the form 'liblibname.so'.

`object_filenames` (*source_filenames*[, *strip_dir=0, output_dir=''*])
Returns the name of the object files for the given source files. *source_filenames* should be a list of filenames.

`shared_object_filename` (*basename*[, *strip_dir=0, output_dir=''*])
Returns the name of a shared object file for the given file name *basename*.

`execute` (*func, args*[, *msg=None, level=1*])
Invokes `distutils.util.execute()` This method invokes a Python function *func* with the given arguments *args*, after logging and taking into account the *dry_run* flag. XXX see also.

`spawn` (*cmd*)
Invokes `distutils.util.spawn()`. This invokes an external process to run the given command. XXX see also.

`mkpath` (*name*[, *mode=511*])
Invokes `distutils.dir_util.mkpath()`. This creates a directory and any missing ancestor directories. XXX see also.

`move_file` (*src, dst*)
Invokes `distutils.file_util.move_file()`. Renames *src* to *dst*. XXX see also.

`announce` (*msg*[, *level=1*])
Write a message using `distutils.log.debug()`. XXX see also.

`warn` (*msg*)
Write a warning message *msg* to standard error.

`debug_print` (*msg*)
If the *debug* flag is set on this `CCompiler` instance, print *msg* to standard output, otherwise do nothing.

11.3 `distutils.unixccompiler` — Unix C Compiler

This module provides the `UnixCCompiler` class, a subclass of `CCompiler` that handles the typical UNIX-style command-line C compiler:

- macros defined with **-D*name*[=value]**
- macros undefined with **-U*name***
- include search directories specified with **-I*dir***
- libraries specified with **-l*lib***
- library search directories specified with **-L*dir***
- compile handled by **cc** (or similar) executable with **-c** option: compiles '.c' to '.o'
- link static library handled by **ar** command (possibly with **ranlib**)
- link shared library handled by **cc -shared**

11.4 `distutils.msvccompiler` — Microsoft Compiler

This module provides `MSVCCompiler`, an implementation of the abstract `CCompiler` class for Microsoft Visual Studio. Typically, extension modules need to be compiled with the same compiler that was used to compile Python. For Python 2.3 and earlier, the compiler was Visual Studio 6. For Python 2.4 and 2.5, the compiler is Visual Studio .NET 2003. The AMD64 and Itanium binaries are created using the Platform SDK.

`MSVCCompiler` will normally choose the right compiler, linker etc. on its own. To override this choice, the environment variables *DISTUTILS_USE_SDK* and *MSSdk* must be both set. *MSSdk* indicates that the current environment has been setup by the SDK's `SetEnv.Cmd` script, or that the environment variables had been registered when the SDK was installed; *DISTUTILS_USE_SDK* indicates that the distutils user has made an explicit choice to override the compiler selection by `MSVCCompiler`.

11.5 `distutils.bcppcompiler` — Borland Compiler

This module provides `BorlandCCompiler`, an subclass of the abstract `CCompiler` class for the Borland C++ compiler.

11.6 `distutils.cygwincompiler` — Cygwin Compiler

This module provides the `CygwinCCompiler` class, a subclass of `UnixCCompiler` that handles the Cygwin port of the GNU C compiler to Windows. It also contains the Mingw32CCompiler class which handles the mingw32 port of GCC (same as cygwin in no-cygwin mode).

11.7 `distutils.emxccompiler` — OS/2 EMX Compiler

This module provides the EMXCCompiler class, a subclass of `UnixCCompiler` that handles the EMX port of the GNU C compiler to OS/2.

11.8 `distutils.mwerkscompiler` — Metrowerks CodeWarrior support

Contains `MWerksCompiler`, an implementation of the abstract `CCompiler` class for MetroWerks CodeWarrior on the pre-Mac OS X Macintosh. Needs work to support CW on Windows or Mac OS X.

11.9 `distutils.archive_util` — Archiving utilities

This module provides a few functions for creating archive files, such as tarballs or zipfiles.

`make_archive`(*base_name, format*[, *root_dir=*`None`*, base_dir=*`None`*, verbose=*`0`*, dry_run=*`0`*])

Create an archive file (eg. `zip` or `tar`). *base_name* is the name of the file to create, minus any format-specific extension; *format* is the archive format: one of `zip`, `tar`, `ztar`, or `gztar`. *root_dir* is a directory that will be the root directory of the archive; ie. we typically `chdir` into *root_dir* before creating the archive. *base_dir* is the directory where we start archiving from; ie. *base_dir* will be the common prefix of all files and directories in the archive. *root_dir* and *base_dir* both default to the current directory. Returns the name of the archive file.

Warning: This should be changed to support bz2 files

make_tarball (*base_name, base_dir*[, *compress=*'gzip'*, verbose=0, dry_run=0*])
'Create an (optional compressed) archive as a tar file from all files in and under *base_dir*. *compress* must be 'gzip' (the default), 'compress', 'bzip2', or None. Both **tar** and the compression utility named by *compress* must be on the default program search path, so this is probably UNIX-specific. The output tar file will be named '*base_dir*.tar', possibly plus the appropriate compression extension ('.gz', '.bz2' or '.Z'). Return the output filename.

Warning: This should be replaced with calls to the `tarfile` module.

make_zipfile (*base_name, base_dir*[, *verbose=0, dry_run=0*])
Create a zip file from all files in and under *base_dir*. The output zip file will be named *base_dir* + '.zip'. Uses either the `zipfile` Python module (if available) or the InfoZIP 'zip' utility (if installed and found on the default search path). If neither tool is available, raises `DistutilsExecError`. Returns the name of the output zip file.

11.10 `distutils.dep_util` — Dependency checking

This module provides functions for performing simple, timestamp-based dependency of files and groups of files; also, functions based entirely on such timestamp dependency analysis.

newer (*source, target*)
Return true if *source* exists and is more recently modified than *target*, or if *source* exists and *target* doesn't. Return false if both exist and *target* is the same age or newer than *source*. Raise `DistutilsFileError` if *source* does not exist.

newer_pairwise (*sources, targets*)
Walk two filename lists in parallel, testing if each source is newer than its corresponding target. Return a pair of lists (*sources*, *targets*) where source is newer than target, according to the semantics of `newer()`

newer_group (*sources, target*[, *missing=*'error'])
Return true if *target* is out-of-date with respect to any file listed in *sources* In other words, if *target* exists and is newer than every file in *sources*, return false; otherwise return true. *missing* controls what we do when a source file is missing; the default ('error') is to blow up with an `OSError` from inside `os.stat()`; if it is 'ignore', we silently drop any missing source files; if it is 'newer', any missing source files make us assume that *target* is out-of-date (this is handy in "dry-run" mode: it'll make you pretend to carry out commands that wouldn't work because inputs are missing, but that doesn't matter because you're not actually going to run the commands).

11.11 `distutils.dir_util` — Directory tree operations

This module provides functions for operating on directories and trees of directories.

mkpath (*name*[, *mode=0777, verbose=0, dry_run=0*])
Create a directory and any missing ancestor directories. If the directory already exists (or if *name* is the empty string, which means the current directory, which of course exists), then do nothing. Raise `DistutilsFileError` if unable to create some directory along the way (eg. some sub-path exists, but is a file rather than a directory). If *verbose* is true, print a one-line summary of each mkdir to stdout. Return the list of directories actually created.

create_tree (*base_dir, files*[, *mode=0777, verbose=0, dry_run=0*])
Create all the empty directories under *base_dir* needed to put *files* there. *base_dir* is just the a name of a directory which doesn't necessarily exist yet; *files* is a list of filenames to be interpreted relative to *base_dir*. *base_dir* + the directory portion of every file in *files* will be created if it doesn't already exist. *mode*, *verbose* and *dry_run* flags are as for `mkpath()`.

copy_tree (*src, dst*[*preserve_mode=1, preserve_times=1, preserve_symlinks=0, update=0, verbose=0, dry_run=0*])

Copy an entire directory tree *src* to a new location *dst*. Both *src* and *dst* must be directory names. If *src* is not a directory, raise `DistutilsFileError`. If *dst* does not exist, it is created with `mkpath()`. The end result of the copy is that every file in *src* is copied to *dst*, and directories under *src* are recursively copied to *dst*. Return the list of files that were copied or might have been copied, using their output name. The return value is unaffected by *update* or *dry_run*: it is simply the list of all files under *src*, with the names changed to be under *dst*.

preserve_mode and *preserve_times* are the same as for `copy_file` in `distutils.file_util`; note that they only apply to regular files, not to directories. If *preserve_symlinks* is true, symlinks will be copied as symlinks (on platforms that support them!); otherwise (the default), the destination of the symlink will be copied. *update* and *verbose* are the same as for `copy_file()`.

remove_tree (*directory*[*verbose=0, dry_run=0*])

Recursively remove *directory* and all files and directories underneath it. Any errors are ignored (apart from being reported to `sys.stdout` if *verbose* is true).

****Some of this could be replaced with the shutil module?****

11.12 `distutils.file_util` — Single file operations

This module contains some utility functions for operating on individual files.

copy_file (*src, dst*[*preserve_mode=1, preserve_times=1, update=0, link=None, verbose=0, dry_run=0*])

Copy file *src* to *dst*. If *dst* is a directory, then *src* is copied there with the same name; otherwise, it must be a filename. (If the file exists, it will be ruthlessly clobbered.) If *preserve_mode* is true (the default), the file's mode (type and permission bits, or whatever is analogous on the current platform) is copied. If *preserve_times* is true (the default), the last-modified and last-access times are copied as well. If *update* is true, *src* will only be copied if *dst* does not exist, or if *dst* does exist but is older than *src*.

link allows you to make hard links (using `os.link`) or symbolic links (using `os.symlink`) instead of copying: set it to `'hard'` or `'sym'`; if it is `None` (the default), files are copied. Don't set *link* on systems that don't support it: `copy_file()` doesn't check if hard or symbolic linking is available. It uses `_copy_file_contents()` to copy file contents.

Return a tuple '`(dest_name, copied)`': *dest_name* is the actual name of the output file, and *copied* is true if the file was copied (or would have been copied, if *dry_run* true).

move_file (*src, dst*[*verbose, dry_run*])

Move file *src* to *dst*. If *dst* is a directory, the file will be moved into it with the same name; otherwise, *src* is just renamed to *dst*. Returns the new full name of the file. **Warning:** Handles cross-device moves on UNIX using `copy_file()`. What about other systems???

write_file (*filename, contents*)

Create a file called *filename* and write *contents* (a sequence of strings without line terminators) to it.

11.13 `distutils.util` — Miscellaneous other utility functions

This module contains other assorted bits and pieces that don't fit into any other utility module.

get_platform ()

Return a string that identifies the current platform. This is used mainly to distinguish platform-specific build directories and platform-specific built distributions. Typically includes the OS name and version and the architecture (as supplied by 'os.uname()'), although the exact information included depends on the OS; eg. for IRIX the architecture isn't particularly important (IRIX only runs on SGI hardware), but for Linux the kernel version isn't particularly important.

Examples of returned values:

- `linux-i586`
- `linux-alpha`
- `solaris-2.6-sun4u`
- `irix-5.3`
- `irix64-6.2`

For non-POSIX platforms, currently just returns `sys.platform`.

`convert_path`(*pathname*)
Return 'pathname' as a name that will work on the native filesystem, i.e. split it on '/' and put it back together again using the current directory separator. Needed because filenames in the setup script are always supplied in UNIX style, and have to be converted to the local convention before we can actually use them in the filesystem. Raises `ValueError` on non-UNIX-ish systems if *pathname* either starts or ends with a slash.

`change_root`(*new_root, pathname*)
Return *pathname* with *new_root* prepended. If *pathname* is relative, this is equivalent to '`os.path.join(new_root,pathname)`' Otherwise, it requires making *pathname* relative and then joining the two, which is tricky on DOS/Windows.

`check_environ`()
Ensure that 'os.environ' has all the environment variables we guarantee that users can use in config files, command-line options, etc. Currently this includes:

- HOME - user's home directory (UNIX only)
- PLAT - description of the current platform, including hardware and OS (see `get_platform()`)

`subst_vars`(*s, local_vars*)
Perform shell/Perl-style variable substitution on *s*. Every occurrence of `$` followed by a name is considered a variable, and variable is substituted by the value found in the *local_vars* dictionary, or in `os.environ` if it's not in *local_vars*. *os.environ* is first checked/augmented to guarantee that it contains certain values: see `check_environ()`. Raise `ValueError` for any variables not found in either *local_vars* or `os.environ`.

Note that this is not a fully-fledged string interpolation function. A valid `$variable` can consist only of upper and lower case letters, numbers and an underscore. No { } or style quoting is available.

`grok_environment_error`(*exc*[, *prefix*=`'error: '`])
Generate a useful error message from an `EnvironmentError` (`IOError` or `OSError`) exception object. Handles Python 1.5.1 and later styles, and does what it can to deal with exception objects that don't have a filename (which happens when the error is due to a two-file operation, such as `rename()` or `link()`). Returns the error message as a string prefixed with *prefix*.

`split_quoted`(*s*)
Split a string up according to UNIX shell-like rules for quotes and backslashes. In short: words are delimited by spaces, as long as those spaces are not escaped by a backslash, or inside a quoted string. Single and double quotes are equivalent, and the quote characters can be backslash-escaped. The backslash is stripped from any two-character escape sequence, leaving only the escaped character. The quote characters are stripped from any quoted string. Returns a list of words.

`execute`(*func, args*[, *msg*=`None`, *verbose*=`0`, *dry_run*=`0`])
Perform some action that affects the outside world (for instance, writing to the filesystem). Such actions are special because they are disabled by the *dry_run* flag. This method takes care of all that bureaucracy for you; all you have to do is supply the function to call and an argument tuple for it (to embody the "external action" being performed), and an optional message to print.

strtobool(*val*)
Convert a string representation of truth to true (1) or false (0).

True values are `y`, `yes`, `t`, `true`, `on` and `1`; false values are `n`, `no`, `f`, `false`, `off` and `0`. Raises `ValueError` if *val* is anything else.

byte_compile(*py_files*[, *optimize=0*, *force=0*, *prefix=None*, *base_dir=None*, *verbose=1*, *dry_run=0*, *direct=None*])
Byte-compile a collection of Python source files to either '.pyc' or '.pyo' files in the same directory. *py_files* is a list of files to compile; any files that don't end in '.py' are silently skipped. *optimize* must be one of the following:

- `0` - don't optimize (generate '.pyc')
- `1` - normal optimization (like '`python -O`')
- `2` - extra optimization (like '`python -OO`')

If *force* is true, all files are recompiled regardless of timestamps.

The source filename encoded in each bytecode file defaults to the filenames listed in *py_files*; you can modify these with *prefix* and *basedir*. *prefix* is a string that will be stripped off of each source filename, and *base_dir* is a directory name that will be prepended (after *prefix* is stripped). You can supply either or both (or neither) of *prefix* and *base_dir*, as you wish.

If *dry_run* is true, doesn't actually do anything that would affect the filesystem.

Byte-compilation is either done directly in this interpreter process with the standard `py_compile` module, or indirectly by writing a temporary script and executing it. Normally, you should let `byte_compile()` figure out to use direct compilation or not (see the source for details). The *direct* flag is used by the script generated in indirect mode; unless you know what you're doing, leave it set to `None`.

rfc822_escape(*header*)
Return a version of *header* escaped for inclusion in an RFC 822 header, by ensuring there are 8 spaces space after each newline. Note that it does no other modification of the string.

11.14 `distutils.dist` — The Distribution class

This module provides the `Distribution` class, which represents the module distribution being built/installed/distributed.

11.15 `distutils.extension` — The Extension class

This module provides the `Extension` class, used to describe C/C++ extension modules in setup scripts.

11.16 `distutils.debug` — Distutils debug mode

This module provides the DEBUG flag.

11.17 `distutils.errors` — Distutils exceptions

Provides exceptions used by the Distutils modules. Note that Distutils modules may raise standard exceptions; in particular, SystemExit is usually raised for errors that are obviously the end-user's fault (eg. bad command-line arguments).

This module is safe to use in '`from ... import *`' mode; it only exports symbols whose names start with `Distutils` and end with `Error`.

11.18 `distutils.fancy_getopt` — Wrapper around the standard getopt module

This module provides a wrapper around the standard `getopt` module that provides the following additional features:

- short and long options are tied together
- options have help strings, so `fancy_getopt` could potentially create a complete usage summary
- options set attributes of a passed-in object
- boolean options can have "negative aliases" — eg. if **--quiet** is the "negative alias" of **--verbose**, then **--quiet** on the command line sets *verbose* to false.

****Should be replaced with `optik` (which is also now known as `optparse` in Python 2.3 and later).****

`fancy_getopt`(*options, negative_opt, object, args*)
Wrapper function. *options* is a list of '`(long_option, short_option, help_string)`' 3-tuples as described in the constructor for `FancyGetopt`. *negative_opt* should be a dictionary mapping option names to option names, both the key and value should be in the *options* list. *object* is an object which will be used to store values (see the `getopt()` method of the `FancyGetopt` class). *args* is the argument list. Will use `sys.argv[1:]` if you pass `None` as *args*.

`wrap_text`(*text, width*)
Wraps *text* to less than *width* wide.

Warning: Should be replaced with `textwrap` (which is available in Python 2.3 and later).

class **`FancyGetopt`**([*option_table*=`None`])
The option_table is a list of 3-tuples: '`(long_option, short_option, help_string)`'

If an option takes an argument, its *long_option* should have `'='` appended; *short_option* should just be a single character, no `':'` in any case. *short_option* should be `None` if a *long_option* doesn't have a corresponding *short_option*. All option tuples must have long options.

The `FancyGetopt` class provides the following methods:

`getopt`([*args*=`None`, *object*=`None`])
Parse command-line options in args. Store as attributes on *object*.

If *args* is `None` or not supplied, uses `sys.argv[1:]`. If *object* is `None` or not supplied, creates a new `OptionDummy` instance, stores option values there, and returns a tuple '`(args, object)`'. If *object* is supplied, it is modified in place and `getopt()` just returns *args*; in both cases, the returned *args* is a modified copy of the passed-in *args* list, which is left untouched.

`get_option_order`()
Returns the list of '`(option, value)`' tuples processed by the previous run of `getopt()` Raises `RuntimeError` if `getopt()` hasn't been called yet.

`generate_help`([*header*=`None`])
Generate help text (a list of strings, one per suggested line of output) from the option table for this `FancyGetopt` object.

If supplied, prints the supplied *header* at the top of the help.

11.19 `distutils.filelist` — The FileList class

This module provides the `FileList` class, used for poking about the filesystem and building lists of files.

11.20 `distutils.log` — Simple PEP 282-style logging

Warning: Should be replaced with standard `logging` module.

11.21 `distutils.spawn` — Spawn a sub-process

This module provides the `spawn()` function, a front-end to various platform-specific functions for launching another program in a sub-process. Also provides `find_executable()` to search the path for a given executable name.

11.22 `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.prefix)`.

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

`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.

11.23 distutils.text_file — The TextFile class

This module provides the TextFile class, which gives an interface to text files that (optionally) takes care of stripping comments, ignoring blank lines, and joining lines with backslashes.

class **TextFile**([*filename*=None, *file*=None, ***options*])
This class provides a file-like object that takes care of all the things you commonly want to do when processing a text file that has some line-by-line syntax: strip comments (as long as # is your comment character), skip blank lines, join adjacent lines by escaping the newline (ie. backslash at end of line), strip leading and/or trailing whitespace. All of these are optional and independently controllable.

The class provides a warn() method so you can generate warning messages that report physical line number, even if the logical line in question spans multiple physical lines. Also provides unreadline() for implementing line-at-a-time lookahead.

TextFile instances are create with either *filename*, *file*, or both. RuntimeError is raised if both are None. *filename* should be a string, and *file* a file object (or something that provides readline() and close() methods). It is recommended that you supply at least *filename*, so that TextFile can include it in warning messages. If *file* is not supplied, TextFile creates its own using the open() built-in function.

The options are all boolean, and affect the values returned by readline()

option name	description	default
strip_comments	strip from '#' to end-of-line, as well as any whitespace leading up to the '#'—unless it is escaped by a backslash	true
lstrip_ws	strip leading whitespace from each line before returning it	false
rstrip_ws	strip trailing whitespace (including line terminator!) from each line before returning it.	true
skip_blanks	skip lines that are empty *after* stripping comments and whitespace. (If both lstrip_ws and rstrip_ws are false, then some lines may consist of solely whitespace: these will *not* be skipped, even if *skip_blanks* is true.)	true
join_lines	if a backslash is the last non-newline character on a line after stripping comments and whitespace, join the following line to it to form one logical line; if N consecutive lines end with a backslash, then N+1 physical lines will be joined to form one logical line.	false
collapse_join	strip leading whitespace from lines that are joined to their predecessor; only matters if '(join_lines and not lstrip_ws)'	false

Note that since *rstrip_ws* can strip the trailing newline, the semantics of readline() must differ from those of the builtin file object's readline() method! In particular, readline() returns None for end-of-file: an empty string might just be a blank line (or an all-whitespace line), if *rstrip_ws* is true but *skip_blanks* is not.

open(*filename*)
Open a new file *filename*. This overrides any *file* or *filename* constructor arguments.

close()
Close the current file and forget everything we know about it (including the filename and the current line number).

warn(*msg*[, *line*=`None`])
Print (to stderr) a warning message tied to the current logical line in the current file. If the current logical line in the file spans multiple physical lines, the warning refers to the whole range, such as '`"lines 3-5"`'. If *line* is supplied, it overrides the current line number; it may be a list or tuple to indicate a range of physical lines, or an integer for a single physical line.

readline()
Read and return a single logical line from the current file (or from an internal buffer if lines have previously been "unread" with `unreadline()`). If the *join_lines* option is true, this may involve reading multiple physical lines concatenated into a single string. Updates the current line number, so calling `warn()` after `readline()` emits a warning about the physical line(s) just read. Returns `None` on end-of-file, since the empty string can occur if *rstrip_ws* is true but *strip_blanks* is not.

readlines()
Read and return the list of all logical lines remaining in the current file. This updates the current line number to the last line of the file.

unreadline(*line*)
Push *line* (a string) onto an internal buffer that will be checked by future `readline()` calls. Handy for implementing a parser with line-at-a-time lookahead. Note that lines that are "unread" with `unreadline` are not subsequently re-cleansed (whitespace stripped, or whatever) when read with `readline`. If multiple calls are made to `unreadline` before a call to `readline`, the lines will be returned most in most recent first order.

11.24 `distutils.version` — Version number classes

11.25 `distutils.cmd` — Abstract base class for Distutils commands

This module supplies the abstract base class `Command`.

class Command(*dist*)
Abstract base class for defining command classes, the "worker bees" of the Distutils. A useful analogy for command classes is to think of them as subroutines with local variables called *options*. The options are declared in `initialize_options()` and defined (given their final values) in `finalize_options()`, both of which must be defined by every command class. The distinction between the two is necessary because option values might come from the outside world (command line, config file, ...), and any options dependent on other options must be computed after these outside influences have been processed — hence `finalize_options()`. The body of the subroutine, where it does all its work based on the values of its options, is the `run()` method, which must also be implemented by every command class.

The class constructor takes a single argument *dist*, a `Distribution` instance.

11.26 `distutils.command` — Individual Distutils commands

11.27 `distutils.command.bdist` — Build a binary installer

11.28 `distutils.command.bdist_packager` — Abstract base class for packagers

11.29 `distutils.command.bdist_dumb` — Build a “dumb” installer

11.30 `distutils.command.bdist_msi` — Build a Microsoft Installer binary package

11.31 `distutils.command.bdist_rpm` — Build a binary distribution as a Redhat RPM and SRPM

11.32 `distutils.command.bdist_wininst` — Build a Windows installer

11.33 `distutils.command.sdist` — Build a source distribution

11.34 `distutils.command.build` — Build all files of a package

11.35 `distutils.command.build_clib` — Build any C libraries in a package

11.36 `distutils.command.build_ext` — Build any extensions in a package

11.37 `distutils.command.build_py` — Build the .py/.pyc files of a package

11.38 `distutils.command.build_scripts` — Build the scripts of a package

11.39 `distutils.command.clean` — Clean a package build area

11.40 `distutils.command.config` — Perform package configuration

11.41 `distutils.command.install` — Install a package

11.42 `distutils.command.install_data` — Install data files from a package

11.43 `distutils.command.install_headers` — Install C/C++ header files from a package

11.44 `distutils.command.install_lib` — Install library files from a package

11.45 `distutils.command.install_scripts` — Install script files from a package

11.46 `distutils.command.register` — Register a module with the Python Package Index

The `register` command registers the package with the Python Package Index. This is described in more detail in PEP 301.

11.47 Creating a new Distutils command

This section outlines the steps to create a new Distutils command.

A new command lives in a module in the `distutils.command` package. There is a sample template in that directory called 'command_template'. Copy this file to a new module with the same name as the new command you're implementing. This module should implement a class with the same name as the module (and the command). So, for instance, to create the command `peel_banana` (so that users can run '`setup.py peel_banana`'), you'd copy 'command_template' to 'distutils/command/peel_banana.py', then edit it so that it's implementing the class `peel_banana`, a subclass of `distutils.cmd.Command`.

Subclasses of `Command` must define the following methods.

`initialize_options()`(*S*)
: et default values for all the options that this command supports. Note that these defaults may be overridden by other commands, by the setup script, by config files, or by the command-line. Thus, this is not the place to code dependencies between options; generally, `initialize_options()` implementations are just a bunch of '`self.foo = None`' assignments.

`finalize_options()`
: Set final values for all the options that this command supports. This is always called as late as possible, ie. after any option assignments from the command-line or from other commands have been done. Thus, this is the place to to code option dependencies: if *foo* depends on *bar*, then it is safe to set *foo* from *bar* as long as *foo* still has the same value it was assigned in `initialize_options()`.

`run()`
: A command's raison d'etre: carry out the action it exists to perform, controlled by the options initialized in `initialize_options()`, customized by other commands, the setup script, the command-line, and config files, and finalized in `finalize_options()`. All terminal output and filesystem interaction should be done by `run()`.

sub_commands formalizes the notion of a "family" of commands, eg. `install` as the parent with sub-commands `install_lib`, `install_headers`, etc. The parent of a family of commands defines *sub_commands* as a class attribute; it's a list of 2-tuples '`(command_name, predicate)`', with *command_name* a string and *predicate* an unbound method, a string or None. *predicate* is a method of the parent command that determines whether the corresponding command is applicable in the current situation. (Eg. we `install_headers` is only applicable if we have any C header files to install.) If *predicate* is None, that command is always applicable.

sub_commands is usually defined at the *end* of a class, because predicates can be unbound methods, so they must already have been defined. The canonical example is the `install` command.

MODULE INDEX

D

INDEX

E

F

G

H

I

L

M

N

O

P

R

S

T

U

W