Installing Python Modules

Release 2.4.2

Greg Ward

28 September 2005

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

Abstract

This document describes the Python Distribution Utilities ("Distutils") from the end-user's point-of-view, describing how to extend the capabilities of a standard Python installation by building and installing third-party Python modules and extensions.

Contents

1 Introduction

Although Python's extensive standard library covers many programming needs, there often comes a time when you need to add some new functionality to your Python installation in the form of third-party modules. This might be necessary to support your own programming, or to support an application that you want to use and that happens to be written in Python.

In the past, there has been little support for adding third-party modules to an existing Python installation. With the introduction of the Python Distribution Utilities (Distutils for short) in Python 2.0, this changed.

This document is aimed primarily at the people who need to install third-party Python modules: end-users and system administrators who just need to get some Python application running, and existing Python programmers who want to add some new goodies to their toolbox. You don't need to know Python to read this document; there will be some brief forays into using Python's interactive mode to explore your installation, but that's it. If you're looking for information on how to distribute your own Python modules so that others may use them, see the *Distributing Python Modules* manual.

1.1 Best case: trivial installation

In the best case, someone will have prepared a special version of the module distribution you want to install that is targeted specifically at your platform and is installed just like any other software on your platform. For example, the module developer might make an executable installer available for Windows users, an RPM package for users of RPM-based Linux systems (Red Hat, SuSE, Mandrake, and many others), a Debian package for users of Debian-based Linux systems, and so forth.

In that case, you would download the installer appropriate to your platform and do the obvious thing with it: run it if it's an executable installer, `rpm --install` it if it's an RPM, etc. You don't need to run Python or a setup script, you don't need to compile anything—you might not even need to read any instructions (although it's always a good idea to do so anyways).

Of course, things will not always be that easy. You might be interested in a module distribution that doesn't have an easy-to-use installer for your platform. In that case, you'll have to start with the source distribution released by the module's author/maintainer. Installing from a source distribution is not too hard, as long as the modules are packaged in the standard way. The bulk of this document is about building and installing modules from standard source distributions.

1.2 The new standard: Distutils

If you download a module source distribution, you can tell pretty quickly if it was packaged and distributed in the standard way, i.e. using the Distutils. First, the distribution's name and version number will be featured prominently in the name of the downloaded archive, e.g. 'foo-1.0.tar.gz' or 'widget-0.9.7.zip'. Next, the archive will unpack into a similarly-named directory: 'foo-1.0' or 'widget-0.9.7'. Additionally, the distribution will contain a setup script 'setup.py', and a file named 'README.txt' or possibly just 'README', which should explain that building and installing the module distribution is a simple matter of running

```
python setup.py install
```

If all these things are true, then you already know how to build and install the modules you've just downloaded: Run the command above. Unless you need to install things in a non-standard way or customize the build process, you don't really need this manual. Or rather, the above command is everything you need to get out of this manual.

2 Standard Build and Install

As described in section 1.2, building and installing a module distribution using the Distutils is usually one simple command:

```
python setup.py install
```

On UNIX, you'd run this command from a shell prompt; on Windows, you have to open a command prompt window ("DOS box") and do it there; on Mac OS X, you open a `Terminal` window to get a shell prompt.

2.1 Platform variations

You should always run the setup command from the distribution root directory, i.e. the top-level subdirectory that the module source distribution unpacks into. For example, if you've just downloaded a module source distribution 'foo-1.0.tar.gz' onto a UNIX system, the normal thing to do is:

```
gunzip -c foo-1.0.tar.gz | tar xf -    # unpacks into directory foo-1.0
cd foo-1.0
python setup.py install
```

On Windows, you'd probably download 'foo-1.0.zip'. If you downloaded the archive file to 'C:\Temp', then it would unpack into 'C:\Temp\foo-1.0'; you can use either a archive manipulator with a graphical user interface (such as WinZip) or a command-line tool (such as **unzip** or **pkunzip**) to unpack the archive. Then, open a command prompt window ("DOS box"), and run:

```
cd c:\Temp\foo-1.0
python setup.py install
```

2.2 Splitting the job up

Running `setup.py install` builds and installs all modules in one run. If you prefer to work incrementally—especially useful if you want to customize the build process, or if things are going wrong—you can use the setup script to do one thing at a time. This is particularly helpful when the build and install will be done by different users—for example, you might want to build a module distribution and hand it off to a system administrator for installation (or do it yourself, with super-user privileges).

For example, you can build everything in one step, and then install everything in a second step, by invoking the setup script twice:

```
python setup.py build
python setup.py install
```

If you do this, you will notice that running the `install` command first runs the `build` command, which—in this case—quickly notices that it has nothing to do, since everything in the 'build' directory is up-to-date.

You may not need this ability to break things down often if all you do is install modules downloaded off the 'net, but it's very handy for more advanced tasks. If you get into distributing your own Python modules and extensions, you'll run lots of individual Distutils commands on their own.

2.3 How building works

As implied above, the `build` command is responsible for putting the files to install into a *build directory*. By default, this is 'build' under the distribution root; if you're excessively concerned with speed, or want to keep the source tree pristine, you can change the build directory with the **--build-base** option. For example:

```
python setup.py build --build-base=/tmp/pybuild/foo-1.0
```

(Or you could do this permanently with a directive in your system or personal Distutils configuration file; see section 5.) Normally, this isn't necessary.

The default layout for the build tree is as follows:

```
--- build/ --- lib/
or
--- build/ --- lib.<plat>/
               temp.<plat>/
```

where `<plat>` expands to a brief description of the current OS/hardware platform and Python version. The first form, with just a 'lib' directory, is used for "pure module distributions"—that is, module distributions that include only pure Python modules. If a module distribution contains any extensions (modules written in C/C++), then the second form, with two `<plat>` directories, is used. In that case, the 'temp.*plat*' directory holds temporary files generated by the compile/link process that don't actually get installed. In either case, the 'lib' (or 'lib.*plat*') directory contains all Python modules (pure Python and extensions) that will be installed.

In the future, more directories will be added to handle Python scripts, documentation, binary executables, and whatever else is needed to handle the job of installing Python modules and applications.

2.4 How installation works

After the `build` command runs (whether you run it explicitly, or the `install` command does it for you), the work of the `install` command is relatively simple: all it has to do is copy everything under 'build/lib' (or 'build/lib.*plat*') to your chosen installation directory.

If you don't choose an installation directory—i.e., if you just run `setup.py install`—then the `install` command installs to the standard location for third-party Python modules. This location varies by platform and by how you built/installed Python itself. On UNIX (and Mac OS X, which is also Unix-based), it also depends on whether the module distribution being installed is pure Python or contains extensions ("non-pure"):

Platform	**Standard installation location**	**Default value**	**Notes**
UNIX (pure)	*prefix*/lib/python2.4/site-packages	/usr/local/lib/python2.4/site-packages	(1)
UNIX (non-pure)	*exec-prefix*/lib/python2.4/site-packages	/usr/local/lib/python2.4/site-packages	(1)
Windows	*prefix*	C:\Python	(2)

Notes:

(1) Most Linux distributions include Python as a standard part of the system, so *prefix* and *exec-prefix* are usually both '/usr' on Linux. If you build Python yourself on Linux (or any UNIX-like system), the default *prefix* and *exec-prefix* are '/usr/local'.

(2) The default installation directory on Windows was 'C:\Program Files\Python' under Python 1.6a1, 1.5.2, and earlier.

prefix and *exec-prefix* stand for the directories that Python is installed to, and where it finds its libraries at run-time. They are always the same under Windows, and very often the same under UNIXand Mac OS X. You can find out what your Python installation uses for *prefix* and *exec-prefix* by running Python in interactive mode and typing a few simple commands. Under UNIX, just type `python` at the shell prompt. Under Windows, choose Start > Programs > Python 2.4 > Python (command line). Once the interpreter is started, you type Python code at the prompt. For example, on my Linux system, I type the three Python statements shown below, and get the output as shown, to find out my *prefix* and *exec-prefix*:

```
Python 2.4 (#26, Aug  7 2004, 17:19:02)
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.prefix
'/usr'
>>> sys.exec_prefix
'/usr'
```

If you don't want to install modules to the standard location, or if you don't have permission to write there, then you need to read about alternate installations in section 3. If you want to customize your installation directories more heavily, see section 4 on custom installations.

3 Alternate Installation

Often, it is necessary or desirable to install modules to a location other than the standard location for third-party Python modules. For example, on a UNIX system you might not have permission to write to the standard third-party module directory. Or you might wish to try out a module before making it a standard part of your local Python installation. This is especially true when upgrading a distribution already present: you want to make sure your existing base of scripts still works with the new version before actually upgrading.

The Distutils `install` command is designed to make installing module distributions to an alternate location simple and painless. The basic idea is that you supply a base directory for the installation, and the `install` command picks a set of directories (called an *installation scheme*) under this base directory in which to install files. The details differ across platforms, so read whichever of the following sections applies to you.

3.1 Alternate installation: the home scheme

The idea behind the "home scheme" is that you build and maintain a personal stash of Python modules. This scheme's name is derived from the idea of a "home" directory on UNIX, since it's not unusual for a UNIX user to make their home directory have a layout similar to '/usr/' or '/usr/local/'. This scheme can be used by anyone, regardless of the operating system their installing for.

Installing a new module distribution is as simple as

```
python setup.py install --home=<dir>
```

where you can supply any directory you like for the **--home** option. On UNIX, lazy typists can just type a tilde (~); the `install` command will expand this to your home directory:

```
python setup.py install --home=~
```

The **--home** option defines the installation base directory. Files are installed to the following directories under the installation base as follows:

Type of file	Installation Directory	Override option
pure module distribution	*home*/lib/python	**--install-purelib**
non-pure module distribution	*home*/lib/python	**--install-platlib**
scripts	*home*/bin	**--install-scripts**
data	*home*/share	**--install-data**

Changed in version 2.4: The **--home** option used to be supported only on UNIX.

3.2 Alternate installation: UNIX (the prefix scheme)

The "prefix scheme" is useful when you wish to use one Python installation to perform the build/install (i.e., to run the setup script), but install modules into the third-party module directory of a different Python installation (or something that looks like a different Python installation). If this sounds a trifle unusual, it is—that's why the "home scheme" comes first. However, there are at least two known cases where the prefix scheme will be useful.

First, consider that many Linux distributions put Python in '/usr', rather than the more traditional '/usr/local'. This is entirely appropriate, since in those cases Python is part of "the system" rather than a local add-on. However, if you are installing Python modules from source, you probably want them to go in '/usr/local/lib/python2.*X*' rather than '/usr/lib/python2.*X*'. This can be done with

```
/usr/bin/python setup.py install --prefix=/usr/local
```

Another possibility is a network filesystem where the name used to write to a remote directory is different from the name used to read it: for example, the Python interpreter accessed as '/usr/local/bin/python' might search for modules in '/usr/local/lib/python2.*X*', but those modules would have to be installed to, say, '/mnt/@*server*/export/lib/python2.*X*'. This could be done with

```
/usr/local/bin/python setup.py install --prefix=/mnt/@server/export
```

In either case, the **--prefix** option defines the installation base, and the **--exec-prefix** option defines the platform-specific installation base, which is used for platform-specific files. (Currently, this just means non-pure module distributions, but could be expanded to C libraries, binary executables, etc.) If **--exec-prefix** is not supplied, it defaults to **--prefix**. Files are installed as follows:

Type of file	**Installation Directory**	**Override option**
pure module distribution	*prefix*/lib/python2.*X*/site-packages	**--install-purelib**
non-pure module distribution	*exec-prefix*/lib/python2.*X*/site-packages	**--install-platlib**
scripts	*prefix*/bin	**--install-scripts**
data	*prefix*/share	**--install-data**

There is no requirement that **--prefix** or **--exec-prefix** actually point to an alternate Python installation; if the directories listed above do not already exist, they are created at installation time.

Incidentally, the real reason the prefix scheme is important is simply that a standard UNIX installation uses the prefix scheme, but with **--prefix** and **--exec-prefix** supplied by Python itself as `sys.prefix` and `sys.exec_prefix`. Thus, you might think you'll never use the prefix scheme, but every time you run `python setup.py install` without any other options, you're using it.

Note that installing extensions to an alternate Python installation has no effect on how those extensions are built: in particular, the Python header files ('Python.h' and friends) installed with the Python interpreter used to run the setup script will be used in compiling extensions. It is your responsibility to ensure that the interpreter used to run extensions installed in this way is compatible with the interpreter used to build them. The best way to do this is to ensure that the two interpreters are the same version of Python (possibly different builds, or possibly copies of the same build). (Of course, if your **--prefix** and **--exec-prefix** don't even point to an alternate Python installation, this is immaterial.)

3.3 Alternate installation: Windows (the prefix scheme)

Windows has no concept of a user's home directory, and since the standard Python installation under Windows is simpler than under UNIX, the **--prefix** option has traditionally been used to install additional packages in separate locations on Windows.

```
python setup.py install --prefix="\Temp\Python"
```

to install modules to the '\Temp\Python' directory on the current drive.

The installation base is defined by the **--prefix** option; the **--exec-prefix** option is not supported under Windows. Files are installed as follows:

Type of file	Installation Directory	Override option
pure module distribution	*prefix*	**--install-purelib**
non-pure module distribution	*prefix*	**--install-platlib**
scripts	*prefix*\Scripts	**--install-scripts**
data	*prefix*\Data	**--install-data**

4 Custom Installation

Sometimes, the alternate installation schemes described in section 3 just don't do what you want. You might want to tweak just one or two directories while keeping everything under the same base directory, or you might want to completely redefine the installation scheme. In either case, you're creating a *custom installation scheme*.

You probably noticed the column of "override options" in the tables describing the alternate installation schemes above. Those options are how you define a custom installation scheme. These override options can be relative, absolute, or explicitly defined in terms of one of the installation base directories. (There are two installation base directories, and they are normally the same—they only differ when you use the UNIX "prefix scheme" and supply different **--prefix** and **--exec-prefix** options.)

For example, say you're installing a module distribution to your home directory under UNIX—but you want scripts to go in '~/scripts' rather than '~/bin'. As you might expect, you can override this directory with the **--install-scripts** option; in this case, it makes most sense to supply a relative path, which will be interpreted relative to the installation base directory (your home directory, in this case):

```
python setup.py install --home=~ --install-scripts=scripts
```

Another UNIX example: suppose your Python installation was built and installed with a prefix of '/usr/local/python', so under a standard installation scripts will wind up in '/usr/local/python/bin'. If you want them in '/usr/local/bin' instead, you would supply this absolute directory for the **--install-scripts** option:

```
python setup.py install --install-scripts=/usr/local/bin
```

(This performs an installation using the "prefix scheme," where the prefix is whatever your Python interpreter was installed with— '/usr/local/python' in this case.)

If you maintain Python on Windows, you might want third-party modules to live in a subdirectory of *prefix*, rather than right in *prefix* itself. This is almost as easy as customizing the script installation directory—you just have to remember that there are two types of modules to worry about, pure modules and non-pure modules (i.e., modules from a non-pure distribution). For example:

```
python setup.py install --install-purelib=Site --install-platlib=Site
```

The specified installation directories are relative to *prefix*. Of course, you also have to ensure that these directories are in Python's module search path, such as by putting a '.pth' file in *prefix*. See section 4.1 to find out how to modify Python's search path.

If you want to define an entire installation scheme, you just have to supply all of the installation directory options. The recommended way to do this is to supply relative paths; for example, if you want to maintain all Python module-related files under 'python' in your home directory, and you want a separate directory for each platform that you use your home directory from, you might define the following installation scheme:

```
python setup.py install --home=~ \
                        --install-purelib=python/lib \
                        --install-platlib=python/lib.$PLAT \
                        --install-scripts=python/scripts
                        --install-data=python/data
```

or, equivalently,

```
python setup.py install --home=~/python \
                        --install-purelib=lib \
                        --install-platlib='lib.$PLAT' \
                        --install-scripts=scripts
                        --install-data=data
```

$PLAT is not (necessarily) an environment variable—it will be expanded by the Distutils as it parses your command line options, just as it does when parsing your configuration file(s).

Obviously, specifying the entire installation scheme every time you install a new module distribution would be very tedious. Thus, you can put these options into your Distutils config file (see section 5):

```
[install]
install-base=$HOME
install-purelib=python/lib
install-platlib=python/lib.$PLAT
install-scripts=python/scripts
install-data=python/data
```

or, equivalently,

```
[install]
install-base=$HOME/python
install-purelib=lib
install-platlib=lib.$PLAT
install-scripts=scripts
install-data=data
```

Note that these two are *not* equivalent if you supply a different installation base directory when you run the setup script. For example,

```
python setup.py --install-base=/tmp
```

would install pure modules to */tmp/python/lib* in the first case, and to */tmp/lib* in the second case. (For the second case, you probably want to supply an installation base of '/tmp/python'.)

You probably noticed the use of $HOME and $PLAT in the sample configuration file input. These are Distutils configuration variables, which bear a strong resemblance to environment variables. In fact, you can use environment variables in config files on platforms that have such a notion but the Distutils additionally define a few extra variables that may not be in your environment, such as $PLAT. (And of course, on systems that don't have environment variables, such as Mac OS 9, the configuration variables supplied by the Distutils are the only ones you can use.) See section 5 for details.

4.1 Modifying Python's Search Path

When the Python interpreter executes an `import` statement, it searches for both Python code and extension modules along a search path. A default value for the path is configured into the Python binary when the interpreter is built. You can determine the path by importing the `sys` module and printing the value of `sys.path`.

```
$ python
Python 2.2 (#11, Oct  3 2002, 13:31:27)
[GCC 2.96 20000731 (Red Hat Linux 7.3 2.96-112)] on linux2
Type ''help'', ''copyright'', ''credits'' or ''license'' for more information.
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.3', '/usr/local/lib/python2.3/plat-linux2',
 '/usr/local/lib/python2.3/lib-tk', '/usr/local/lib/python2.3/lib-dynload',
 '/usr/local/lib/python2.3/site-packages']
>>>
```

The null string in `sys.path` represents the current working directory.

The expected convention for locally installed packages is to put them in the '.../site-packages/' directory, but you may want to install Python modules into some arbitrary directory. For example, your site may have a convention of keeping all software related to the web server under '/www'. Add-on Python modules might then belong in '/www/python', and in order to import them, this directory must be added to `sys.path`. There are several different ways to add the directory.

The most convenient way is to add a path configuration file to a directory that's already on Python's path, usually to the '.../site-packages/' directory. Path configuration files have an extension of '.pth', and each line must contain a single path that will be appended to `sys.path`. (Because the new paths are appended to `sys.path`, modules in the added directories will not override standard modules. This means you can't use this mechanism for installing fixed versions of standard modules.)

Paths can be absolute or relative, in which case they're relative to the directory containing the '.pth' file. Any directories added to the search path will be scanned in turn for '.pth' files. See *site module documentation* for more information.

A slightly less convenient way is to edit the 'site.py' file in Python's standard library, and modify `sys.path`. 'site.py' is automatically imported when the Python interpreter is executed, unless the **-S** switch is supplied to suppress this behaviour. So you could simply edit 'site.py' and add two lines to it:

```
import sys
sys.path.append('/www/python/')
```

However, if you reinstall the same major version of Python (perhaps when upgrading from 2.2 to 2.2.2, for example) 'site.py' will be overwritten by the stock version. You'd have to remember that it was modified and save a copy before doing the installation.

There are two environment variables that can modify `sys.path`. PYTHONHOME sets an alternate value for the prefix of the Python installation. For example, if PYTHONHOME is set to '`/www/python`', the search path will be set to `['', '/www/python/lib/python2.2/', '/www/python/lib/python2.3/plat-linux2', ...]`.

The PYTHONPATH variable can be set to a list of paths that will be added to the beginning of `sys.path`. For example, if PYTHONPATH is set to '`/www/python:/opt/py`', the search path will begin with `['/www/python', '/opt/py']`. (Note that directories must exist in order to be added to `sys.path`; the `site` module removes paths that don't exist.)

Finally, `sys.path` is just a regular Python list, so any Python application can modify it by adding or removing entries.

5 Distutils Configuration Files

As mentioned above, you can use Distutils configuration files to record personal or site preferences for any Distutils options. That is, any option to any command can be stored in one of two or three (depending on your platform) configuration files, which will be consulted before the command-line is parsed. This means that configuration files will override default values, and the command-line will in turn override configuration files. Furthermore, if multiple configuration files apply, values from "earlier" files are overridden by "later" files.

5.1 Location and names of config files

The names and locations of the configuration files vary slightly across platforms. On UNIXand Mac OS X, the three configuration files (in the order they are processed) are:

Type of file	**Location and filename**	**Notes**
system	*prefix*/lib/python*ver*/distutils/distutils.cfg	(1)
personal	$HOME/.pydistutils.cfg	(2)
local	setup.cfg	

affect only module distributions processed by you. And if it is used as the ‘setup.cfg’ for a particular module distribution, it affects only that distribution.

You could override the default “build base” directory and make the `build*` commands always forcibly rebuild all files with the following:

```
[build]
build-base=blib
force=1
```

which corresponds to the command-line arguments

```
python setup.py build --build-base=blib --force
```

except that including the `build` command on the command-line means that command will be run. Including a particular command in config files has no such implication; it only means that if the command is run, the options in the config file will apply. (Or if other commands that derive values from it are run, they will use the values in the config file.)

You can find out the complete list of options for any command using the **--help** option, e.g.:

```
python setup.py build --help
```

and you can find out the complete list of global options by using **--help** without a command:

```
python setup.py --help
```

See also the “Reference” section of the “Distributing Python Modules” manual.

6 Building Extensions: Tips and Tricks

Whenever possible, the Distutils try to use the configuration information made available by the Python interpreter used to run the ‘setup.py’ script. For example, the same compiler and linker flags used to compile Python will also be used for compiling extensions. Usually this will work well, but in complicated situations this might be inappropriate. This section discusses how to override the usual Distutils behaviour.

6.1 Tweaking compiler/linker flags

Compiling a Python extension written in C or C++ will sometimes require specifying custom flags for the compiler and linker in order to use a particular library or produce a special kind of object code. This is especially true if the extension hasn’t been tested on your platform, or if you’re trying to cross-compile Python.

In the most general case, the extension author might have foreseen that compiling the extensions would be complicated, and provided a ‘Setup’ file for you to edit. This will likely only be done if the module distribution contains many separate extension modules, or if they often require elaborate sets of compiler flags in order to work.

A ‘Setup’ file, if present, is parsed in order to get a list of extensions to build. Each line in a ‘Setup’ describes a single module. Lines have the following structure:

module `...` [*sourcefile* `...`] [*cpparg* `...`] [*library* `...`]

Let’s examine each of the fields in turn.

- *module* is the name of the extension module to be built, and should be a valid Python identifier. You can't just change this in order to rename a module (edits to the source code would also be needed), so this should be left alone.
- *sourcefile* is anything that's likely to be a source code file, at least judging by the filename. Filenames ending in '.c' are assumed to be written in C, filenames ending in '.C', '.cc', and '.c++' are assumed to be C++, and filenames ending in '.m' or '.mm' are assumed to be in Objective C.
- *cpparg* is an argument for the C preprocessor, and is anything starting with **-I**, **-D**, **-U** or **-C**.
- *library* is anything ending in '.a' or beginning with **-l** or **-L**.

If a particular platform requires a special library on your platform, you can add it by editing the 'Setup' file and running `python setup.py build`. For example, if the module defined by the line

```
foo foomodule.c
```

must be linked with the math library 'libm.a' on your platform, simply add **-lm** to the line:

```
foo foomodule.c -lm
```

Arbitrary switches intended for the compiler or the linker can be supplied with the **-Xcompiler** *arg* and **-Xlinker** *arg* options:

```
foo foomodule.c -Xcompiler -o32 -Xlinker -shared -lm
```

The next option after **-Xcompiler** and **-Xlinker** will be appended to the proper command line, so in the above example the compiler will be passed the **-o32** option, and the linker will be passed **-shared**. If a compiler option requires an argument, you'll have to supply multiple **-Xcompiler** options; for example, to pass `-x c++` the 'Setup' file would have to contain `-Xcompiler -x -Xcompiler c++`.

Compiler flags can also be supplied through setting the CFLAGS environment variable. If set, the contents of CFLAGS will be added to the compiler flags specified in the 'Setup' file.

6.2 Using non-Microsoft compilers on Windows

Borland C++

This subsection describes the necessary steps to use Distutils with the Borland C++ compiler version 5.5.

First you have to know that Borland's object file format (OMF) is different from the format used by the Python version you can download from the Python or ActiveState Web site. (Python is built with Microsoft Visual C++, which uses COFF as the object file format.) For this reason you have to convert Python's library 'python24.lib' into the Borland format. You can do this as follows:

```
coff2omf python24.lib python24_bcpp.lib
```

The 'coff2omf' program comes with the Borland compiler. The file 'python24.lib' is in the 'Libs' directory of your Python installation. If your extension uses other libraries (zlib,...) you have to convert them too.

The converted files have to reside in the same directories as the normal libraries.

How does Distutils manage to use these libraries with their changed names? If the extension needs a library (eg. 'foo') Distutils checks first if it finds a library with suffix '_bcpp' (eg. 'foo_bcpp.lib') and then uses this library. In the case it doesn't find such a special library it uses the default name ('foo.lib'.)[1]

To let Distutils compile your extension with Borland C++ you now have to type:

[1] This also means you could replace all existing COFF-libraries with OMF-libraries of the same name.

```
python setup.py build --compiler=bcpp
```

If you want to use the Borland C++ compiler as the default, you could specify this in your personal or system-wide configuration file for Distutils (see section 5.)

See Also:

C++Builder Compiler
(http://www.borland.com/bcppbuilder/freecompiler/)
Information about the free C++ compiler from Borland, including links to the download pages.

Creating Python Extensions Using Borland's Free Compiler
(http://www.cyberus.ca/~g_will/pyExtenDL.shtml)
Document describing how to use Borland's free command-line C++compiler to build Python.

GNU C / Cygwin / MinGW

This section describes the necessary steps to use Distutils with the GNU C/C++ compilers in their Cygwin and MinGW distributions.[2] For a Python interpreter that was built with Cygwin, everything should work without any of these following steps.

These compilers require some special libraries. This task is more complex than for Borland's C++, because there is no program to convert the library.

First you have to create a list of symbols which the Python DLL exports. (You can find a good program for this task at http://starship.python.net/crew/kernr/mingw32/Notes.html, see at PExports 0.42h there.)

```
pexports python24.dll >python24.def
```

Then you can create from these information an import library for gcc.

```
dlltool --dllname python24.dll --def python24.def --output-lib libpython24.a
```

The resulting library has to be placed in the same directory as 'python24.lib'. (Should be the 'libs' directory under your Python installation directory.)

If your extension uses other libraries (zlib,...) you might have to convert them too. The converted files have to reside in the same directories as the normal libraries do.

To let Distutils compile your extension with Cygwin you now have to type

```
python setup.py build --compiler=cygwin
```

and for Cygwin in no-cygwin mode[3] or for MinGW type:

```
python setup.py build --compiler=mingw32
```

If you want to use any of these options/compilers as default, you should consider to write it in your personal or system-wide configuration file for Distutils (see section 5.)

See Also:

Building Python modules on MS Windows platform with MinGW
(http://www.zope.org/Members/als/tips/win32_mingw_modules)
Information about building the required libraries for the MinGW environment.

http://pyopengl.sourceforge.net/ftp/win32-stuff/
Converted import libraries in Cygwin/MinGW and Borland format, and a script to create the registry entries

[2] Check http://sources.redhat.com/cygwin/ and http://www.mingw.org/ for more information
[3] Then you have no POSIX emulation available, but you also don't need 'cygwin1.dll'.

needed for Distutils to locate the built Python.