Installing Python Modules

Greg Ward

October 30, 2000

E-mail: gward@python.net

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 is starting to change. Not everything will change overnight, though, so while this document concentrates on installing module distributions that use the Distutils, we will also spend some time dealing with the old ways.

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 (Debian proper, Caldera, Corel, etc.), 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 'README.txt' (or possibly '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.

1.3 The old way: no standards

Before the Distutils, there was no infrastructure to support installing third-party modules in a consistent, standardized way. Thus, it's not really possible to write a general manual for installing Python modules that don't use the Distutils; the only truly general statement that can be made is, "Read the module's own installation instructions."

However, if such instructions exist at all, they are often woefully inadequate and targeted at experienced Python developers. Such users are already familiar with how the Python library is laid out on their platform, and know where to copy various files in order for Python to find them. This document makes no such assumptions, and explains how the Python library is laid out on three major platforms (UNIX, Windows, and MacOS), so that you can understand what happens when the Distutils do their job *and* know how to install modules manually when the module author fails to provide a setup script.

Additionally, while there has not previously been a standard installation mechanism, Python has had some standard machinery for building extensions on UNIX since Python ****version?****. This machinery (the 'Makefile.pre.in' file) is superseded by the Distutils, but it will no doubt live on in older module distributions for a while. This 'Makefile.pre.in' mechanism is documented in the *Extending & Embedding Python* manual, but that manual is aimed at module developers—hence, we include documentation for builders/installers here.

All of the pre-Distutils material is tucked away in section 7.

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 MacOS, things are a tad more complicated (see below).

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 GUI archive manipulator (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
```

On MacOS, you have to go through a bit more effort to supply command-line arguments to the setup script:

- hit option-double-click on the script's icon (or option-drop it onto the Python interpreter's icon)

- press the "Set unix-style command line" button
- set the "Keep stdio window open on termination" if you're interested in seeing the output of the setup script (which is usually voluminous and often useful)
- when the command-line dialog pops up, enter "install" (you can, of course, enter any Distutils command-line as described in this document or in *Distributing Python Modules*: just leave off the initial `python setup.py` and you'll be fine)

****this should change: every Distutils setup script will need command-line arguments for every run (and should probably keep stdout around), so all this should happen automatically for setup scripts****

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—e.g., 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 6.) 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 MacOS, 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/python2.0/site-packages	/usr/local/lib/python2.0/site-packages	(1)
UNIX (non-pure)	*exec-prefix*/lib/python2.0/site-packages	/usr/local/lib/python2.0/site-packages	(1)
Windows	*prefix*	C:\Python	(2)
MacOS (pure)	*prefix*:Lib:site-packages	Python:Lib:site-packages	
MacOS (non-pure)	*prefix*:Lib:site-packages	Python:Lib:site-packages	

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 MacOS, and very often the same under UNIX. 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, run "Python 2.0 (interpreter)" ****right?****; under MacOS, ****???****. 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 1.5.2 (#1, Apr 18 1999, 16:03:16)  [GCC pgcc-2.91.60 19981201 (egcs-1.1.1  on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> 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 4. If you want to customize your installation directories more heavily, see section 5 on custom installations.

3 Building Extensions: Tips and Tricks

(This is the section to read for people doing any sort of interesting build. Things to talk about:

- the 'Setup' file (any platform now, but UNIX-biased)
- CFLAGS and LDFLAGS (must implement them first!)
- using non-MS compilers on Windows (how to convert Python's library, ...)

3.1 Tweaking compiler/linker flags

3.2 Using non-Microsoft compilers on Windows

****One place to look: http://www.cyberus.ca/ g_will/pyExtenDL.shtml****

4 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 section applies to you.

4.1 Alternate installation: UNIX (the home scheme)

Under UNIX, there are two ways to perform an alternate installation. The "prefix scheme" is similar to how alternate installation works under Windows and MacOS, but is not necessarily the most useful way to maintain a personal Python library. Hence, we document the more convenient and commonly useful "home scheme" first.

The idea behind the "home scheme" is that you build and maintain a personal stash of Python modules, probably under your home directory. 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. 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**

4.2 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/python1.*X*' rather than '/usr/lib/python1.*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/python1.*X*', but those modules would have to be installed to, say, '/mnt/*@server*/export/lib/python1.*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/python1.*X*/site-packages	**--install-purelib**
non-pure module distribution	*exec-prefix*/lib/python1.*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 compatibile 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.)

4.3 Alternate installation: Windows

Since Windows has no conception of a user's home directory, and since the standard Python installation under Windows is simpler than that under UNIX, there's no point in having separate **--prefix** and **--home** options. Just use the **--prefix** option to specify a base directory, e.g.

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

to install modules to the '\Temp' 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**

4.4 Alternate installation: MacOS

Like Windows, MacOS has no notion of home directories (or even of users), and a fairly simple standard Python installation. Thus, only a **--prefix** option is needed. It defines the installation base, and files are installed under it as follows:

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

See section 2.1 for information on supplying command-line arguments to the setup script with MacPython.

5 Custom Installation

Sometimes, the alternate installation schemes described in section 4 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, e.g. by putting a '.pth' file in *prefix* (****should have a section describing .pth files and cross-ref it here****).

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 6):

```
[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, you can only use the configuration variables supplied by the Distutils on systems that don't have environment variables, such as MacOS (****true?****).) See section 6 for details.

****need some Windows and MacOS examples—when would custom installation schemes be needed on those platforms?****

6 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.

6.1 Location and names of config files

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

Type of file	Location and filename	Notes
system	*prefix*/lib/python*ver*/distutils/pydistutils.cfg	(1)
personal	$HOME/.pydistutils.cfg	(2)
local	setup.cfg	(3)

On Windows, the configuration files are:

Type of file	Location and filename	Notes
system	*prefix*\Lib\distutils\pydistutils.cfg	(4)
personal	%HOME\pydistutils.cfg	(5)
local	setup.cfg	(3)

And on MacOS, they are:

Type of file	Location and filename	Notes
system	*prefix*:Lib:distutils:pydistutils.cfg	(6)
personal	N/A	
local	setup.cfg	(3)

Notes:

(1) Strictly speaking, the system-wide configuration file lives in the directory where the Distutils are installed; under Python 1.6 and later on UNIX, this is as shown. For Python 1.5.2, the Distutils will normally be installed to '*prefix*/lib/site-packages/python1.5/distutils', so the system configuration file should be put there under Python 1.5.2.

(2) On UNIX, if the $HOME environment variable is not defined, the user's home directory will be determined with the `getpwuid()` function from the standard `pwd` module.

(3) I.e., in the current directory (usually the location of the setup script).

(4) (See also note (1).) Under Python 1.6 and later, Python's default "installation prefix" is 'C:\Python', so the system configuration file is normally 'C:\Python\Lib\distutils\pydistutils.cfg'. Under Python 1.5.2, the default prefix was 'C:\Program Files\Python', and the Distutils were not part of the standard library—so the system configuration file would be 'C:\Program Files\Python\distutils\pydistutils.cfg' in a standard Python 1.5.2 installation under Windows.

(5) On Windows, if the $HOME environment variable is not defined, no personal configuration file will be found or used. (In other words, the Distutils make no attempt to guess your home directory on Windows.)

(6) (See also notes (1) and (4).) The default installation prefix is just 'Python:', so under Python 1.6 and later this is normally'Python:Lib:distutils:pydistutils.cfg'. (The Distutils don't work very well with Python 1.5.2 under MacOS. ****true?****)

6.2 Syntax of config files

The Distutils configuration files all have the same syntax. The config files are grouped into sections; there is one section for each Distutils command, plus a `global` section for global options that affect every command. Each section consists of one option per line, specified like `option=value`.

For example, the following is a complete config file that just forces all commands to run quietly by default:

```
[global]
verbose=0
```

If this is installed as the system config file, it will affect all processing of any Python module distribution by any user on the current system. If it is installed as your personal config file (on systems that support them), it will 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.

7 Pre-Distutils Conventions

7.1 The Makefile.pre.in file

7.2 Installing modules manually