The Python Module Distribution Utilities: An Introduction to the Distutils

Gregory P. Ward
Corporation for National Research Initiatives
`gward@python.net`

Abstract

The Python Module Distribution Utilities, or Distutils for short, are being developed to address a long-standing need in the Python community: a standard mechanism for building, distributing, and installing Python modules—or, more realistically, multi-module distributions. The Distutils will address this need by providing a set of classes to implement the normal tasks involved in such work—build C extensions, process documentation, install to library directory, compile Python files to bytecode, etc.—and a standard way for module developers to give users access to these classes through a simple Python script included with every distribution. All of this will work cross-platform, with Unix and DOS/Windows support included from the beginning, and plans for Mac OS support in the future.

1 Introduction and Motivation

The need for such a project is clear to anyone who has installed more than one large module distribution for Python: they all have their own build mechanisms, requiring users to read and understand different (possibly complex) instructions for every new distribution, with no guarantee of working on any platform other than that used by the developer. The need is even greater for module developers, who duplicate each others' efforts and spend time working on code that is ancillary to their main development effort.

The potential payoffs of the Distutils are great, as anyone who has used Perl over the last couple of years can attest. One of the major reasons for the success of CPAN (the Comprehensive Perl Archive Network) is the standard build mechanism used by all Perl module distributions. No Perl developer is taken seriously unless any user can download his distribution, run a few simple commands:

```
perl Makefile.PL
make
make test
make install
```

and be assured of tested, documented new functionality for their Perl system. This system works identically from the tiniest one-module distribution to the mammoth Perl/Tk.

The principle drawback to Perl's MakeMaker system (ExtUtils::MakeMaker is the module that underlies the **`Makefile.PL`** script) is that it generates a makefile, rather than doing its work directly. This reduces portability (not many Windows users have a Unix-compatible **`make`**, even if they do have a C compiler—and MakeMaker requires **`make`** even to build pure-Perl module distributions) and makes customizing the build/install steps awkward and unnecessarily complicated.

2 Enter the Distutils

The Distutils interface is strongly influenced by MakeMaker's. Superficially, Distutils relies on a script **`setup.py`** instead of **`Makefile.PL`**; instead of a call to `WriteMakefile`, that script consists mainly of a call to `setup`. However, the machinery behind that call is all implemented directly in Python, with no intervening **`make`** step. This means that the action of building or installing (or whatever else is required) is done immediately; thus, the user has to specify what he wants to do on the **`setup.py`** command line.

The canonical example of running a Distutils setup script is

```
python setup.py -v install
```

which takes care of everything required to install a workable module distribution. This is planned eventually to include:

- finding pure Python modules
- compiling and linking C/C++ extension modules
- compiling Java extension modules (for JPython-friendly module distributions)

- processing documentation to one or more standard formats
- running a test suite
- installing everything in sensible places (according to the Python installation and user-supplied preferences)

Each of these steps is independantly selectable and controllable using arguments to the setup script. Currently (version 0.1.1), the Distutils only deal with building and installing pure Python modules and C extensions.

The canonical example of an actual setup script is that used by the Distutils itself, shown in Figure 1. Note that most of the arguments to `setup` are distribution meta-data—the only "interesting" data (depending on your point of view) is the list of packages in the distribution, which implicitly describes the set of modules to be built and installed (in this case, distutils/*.py and distutils/command/*.py). More complex module distributions—those involving extensions, or listing individual modules rather than whole packages—take a bit more space to describe, as outlined in section 4 below.

3 Using the Distutils

There are two group of users currently addressed by the Distutils: module developers and module installers (sysadmins or end-users). Generally, writing and editing the setup.py script is the domain of the developer, while running it—and using its extensive command-line options—is for installers. When the Distutils have the ability to generate "built distributions" (ready-to-install distributions, with extensions compiled), then a third community will be added: distributors who take a source distribution provided by the module developer, and create a built distribution for greater convenience to installers.

Of course, developers will need to run the script too, both to test it and to create a source distribution. Eventually, distributors will use the setup script to create built distributions from source distributions. Thus, we first cover the Distutils from the installer's point of view, and deal with the developer's problem of writing the setup script later, in section 3.2.

3.1 Running the setup script

3.1.1 Basic syntax and global options

The standard incantation for installing a Distutils-based module distribution is

```
python setup.py -v install
```

More generally, the command-line syntax is

```
setup.py [global-options]
         cmd [cmd-options ...]
         [cmd [cmd-options ...]] ...
```

Global options affect the actions of all commands; currently, there are only two: `--verbose` (`-v`) and `--dry-run` (`-n`). By default the Distutils are silent, issuing no output except for errors and warnings. `--verbose` countermands this, printing a message for every action that affects (or would affect, if `--dry-run` is supplied) the filesystem. The two are independent, so `-n` without `-v` is pointless: "Don't do anything, and don't tell me what you would have done."

Another caveat is that some Distutils commands base their action on the state of the filesystem, which is often determined by the action of earlier commands. For example, if you run

```
setup.py -nv build install
```

on a fresh module distribution, you'll get a detailed report of what would happen in the `build` phase—but since the effects of the 'build' phase control the `install` phase, you won't see anything relating to installation, because it doesn't see any files to install.

3.1.2 The Distutils commands

After the global options come a series of one or more commands, each with optional command-specific options. The commands implemented as of this writing are:

`build_py`	"build" pure Python modules
`build_ext`	build C/C++ extension modules
`build`	build everything
`install_py`	install pure Python modules
`install_ext`	install extension modules
`install`	install everything
`dist`	create source distribution

Commands may run other commands: the `build` command runs `build_py` and then `build_ext`, because that is what "everything" currently means; the `install` command first runs `build` (because you can't install what you haven't built), and then `install_py` and `install_ext`. Distutils is smart enough not to run the same command twice, and the commands are generally smart enough not to do redundant work in their run.

This illustrates why, if all you want to do is install a module distribution, all you have to do is

```
python setup.py -v install
```

—the `install` command runs `build`, which runs `build_py` and `build_ext` to build everything, and then installs everything with `install_py` and `install_ext`. (Actually, both `build` and `install` are smart

```
from distutils.core import setup

setup (name = "Distutils",
       version = "0.1.1",
       description = "Python Module Distribution Utilities",
       author = "Greg Ward",
       author_email = "gward@python.net",
       url = "http://www.python.org/sigs/distutils-sig/",

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

Figure 1: The setup.py script distributed with the Distutils.

enough that they only run `build_py` or `install_py` if there are pure Python modules in the distribution, and likewise with `build_ext` and `install_ext`. That's a minor optimization, though, as it mainly saves the unnecessary import of all the code for that operation—which, in the case of building extensions, is a fair chunk.)

In the absence of command options (described momentarily), the following are all nearly equivalent:

```
setup.py -v install
setup.py -v build install
setup.py -v build_py build_ext install
setup.py -v build_py build_ext \
            install_py install_ext
```

(The main difference is that when you explicitly specify a command, the module that implements it will be imported—which can cause a noticeable delay in the case of `build_ext`.)

3.1.3 Command options

The actions of every command are controlled by a set of options. Some command options can be specified on the command-line of the setup script; others may only be supplied in the setup script itself. This section only covers the command options that may be supplied by the installer, i.e. on the command-line—the remaining command options are generally the domain of the module developer, so will be covered in section 3.2 below.

The vast majority of command options deal with where to put various files. For instance, the `build*` commands let you specify where to build to, and the `install*` commands let you specify where to install to.

Command option names are only unique within their command, so multiple commands may have (e.g.) a `build_base` option. All command options have a long name (e.g. `build_base` in the setup script or `--build-base` on the command line), and many have a one-letter form that appears only on the command line. There's no guarantee that the same option name has the same one-letter form (or even the same meaning) across different commands, although this is certainly a desirable goal for command implementors.

An exhaustive reference of all command options is included with the Distutils documentation; here we'll concentrate on examples to illustrate some common cases. First, the `build_base` option: by default, the Distutils `build*` commands put ready-to-install modules and extensions into the build/ subdirectory of the distribution root. (The *distribution root* is the directory where **`setup.py`** exists and is run from, and most files and directories referenced by the Distutils are relative to this directory. Thus, you can generally read "distribution root" as "current directory".) If you want these files put somewhere else, use the `build_base` option to `build`:

```
setup.py -v build \
  --build-base=/tmp/pybuild
```

In this case, pure Python modules will be put in /tmp/pybuild/lib, and extension modules in /tmp/pybuild/platlib. If you want exact control over these two directories, you can specify them individually to the `build` command:

```
setup.py -v build \
  --build-lib=/tmp/pybuild.shared \
  --build-platlib=/tmp/pybuild.plat
```

In this case, you don't need to specify `build_base`, since it is only used to generate `build_lib` and `build_plat`.

Of course, if you then attempt to install the module distribution with `setup.py -v install`, it won't work as intended: `install` looks in the default build directory (./build), but the ready-to-install built files aren't there. So your files will be re-built to ./build instead of fetched from /tmp/pybuild (or wherever). Since you're running `install` separately from `build`, you have to tell it the build directories separately. For instance, the `install` command to go with the first `build` command above would be

```
setup.py -v install \
  --build-base=/tmp/pybuild
```

and to go with the second `build` command:

```
setup.py -v install \
  --build-lib=/tmp/pybuild.shared \
  --build-platlib=/tmp/pybuild.plat
```

(Note that the great similarity of these commands is due not to some conspiracy within the Distutils, but to the deliberate choice of the same option names for the same purposes across the two commands. Nothing enforces this design principle except common sense!)

Obviously, it's preferable to supply the build directory only once, as in:

```
setup.py build \
  --build-base=/tmp/pybuild install
```

or, since `install` implies `build`,

```
setup.py install
  --build-base=/tmp/pybuild
```

but note that this:

```
setup.py build install \
  --build-base=/tmp/pybuild
```

does not work—when you explicitly specify the `build` command in this way, its options are decided *before* the options for the `install` command are parsed. (This might be considered a bug—or a design flaw that would be nice, but tricky, to fix.)

Of course, you may be perfectly happy with the default build directories, but want to install elsewhere—e.g., if you don't have superuser privileges on a Unix system, you'll probably want to install to your home directory. Naturally, this is an option—well, several options—to the `install` command:

```
setup.py -v install
  --prefix=/home/greg \
  --exec-prefix=/home/greg
```

will install both pure Python modules and extension modules under /home/greg/lib/python1.5 (assuming Python 1.5, of course).

If you want more precise control, you can specify the `install_site_lib` and `install_site_platlib` options directly:

```
setup.py -v install \
  --install-site-lib=\
    /home/greg/lib/python \
  --install-site-platlib=\
    /home/greg/lib/python.plat
```

Note that the "lib" and "platlib" directories are completely independent; if you want to control both of them, you must specify both (or both `prefix` and `exec_prefix`). This is a feature (I think).

3.2 Writing the setup script

In order to understand how to write setup scripts, you have to understand how they're run—so if you skipped straight down here, you should skip right back up again and read section 3.1.

Now that you know about commands and command options, it is time to reveal more of the truth. (The full truth cannot be revealed since the Distutils are a work-in-progress, and no one yet knows the full truth.) Specifically, the setup script command-line is only one possible source for command options. Two other possible sources are in the setup script itself, as keyword arguments to `setup()`, and as elements in the `options` argument to `setup()`. (The distinction may seem pedantic, but it matters in some circumstances.) Undoubtedly more option sources will be revealed as work on the Distutils progresses.

3.2.1 Command options—the theory

The first option source is, as described above, the command line to the setup script. Whatever happens, this option source is the last word; options supplied here should override all other sources (so that the developer, distributor, or installer running the setup script has the final word). However, not all options can appear on the command line: of course we allow the installer to specify the build and installation directories, but they don't get to specify the extension modules to build or what package they belong to.

That sort of information—what's included in the module distribution and where it belongs in the space of Python modules—goes in the setup script. However, there are *two* sources of options in the setup script, and to understand why you need to understand the difference between *distribution options* and *command options*. The setup script in Figure 2 illustrates a microcosm of the Distutils option universe. First, (almost) all keyword arguments to `setup()` are distribution options, which wind up as attributes of the Distribution object that underlies everything. (But that's an implementation detail which should be reserved for section 4 below.) A few special arguments to `setup()` are, well, special and are not treated as distribution options: `options` is one of these.

The other keyword arguments here—`name`, `version`, `description`, and `py_modules`—are all distribution options. `py_modules` is not a *pure* distribution option, though, as it becomes a command option (`modules`) to guide the actions of the `build_py` command. `py_modules` is thus called an *alias option*, because it is a stand-in for a command option, `build_py.modules`.

Finally, the `options` argument lets you supply any

```
setup (name = "foobar"
       version = "1.1",
       description = "Modules to foo bars and bar foos",

       py_modules = ['foobar.foo', 'foobar.bar'],
       options = { 'build': { 'build_base': 'blib' } })
```

Figure 2: A hypothetical setup script that illustrates several flavours of Distutils command options.

arbitrary command option. The example shown here makes things look more familiar to Perl refugees by building to ./blib instead of ./build. Note that this sort of mischief is anti-social and officially Frowned Upon, but it is possible. Setting options in the `options` dictionary is deliberately awkward: it should not be needed often, and if a particular command option needs to be set frequently for many module distributions, an alias option should be created for it. Also, note that options from the `options` dictionary generally override alias options (depending on how the particular command is implemented), so before you go setting the list of extension modules or packages this way, you had better know exactly what you are doing and what ramifications it will have.

3.2.2 Command options in practice

Using the Distutils boils down to specifying the commands to run and the options that guide them. The developer's job in writing the setup script is to supply the options that the installer cannot know: distribution meta-data, which modules and extensions are present in the distribution, and where they go in the space of Python modules.

All of these were illustrated in the example in section 2: the distribution meta-data is contained in the `name`, `version`, `description`, `author`, `author_email`, and `url` distribution options. (`maintainer` and `maintainer_email` may be supplied in place of, or in addition to, the `author` options.)

In the case of a fully "package-ized," pure-Python distribution like the Distutils, the rest is simple: the `packages` option (an alias for `build_py.packages`) lists the packages in which pure Python modules can be found. The `build_py` command assumes the most sensible directory layout, namely that modules in the distutils package can be found in the distutils subdirectory of the distribution root. Overriding this is easy with the `package_dir` option, for example:

```
setup (
  ...
  packages = ['foo', 'foo.bar']
  package_dir = { 'foo': 'src' }
)
```

tells the Distutils to look for foo modules in src (under the distribution root), and foo.bar modules in src/bar.

If a module distribution isn't yet distributed in package form, just use the empty package name: for example,

```
setup (...
       packages = [''])
```

will install ∗.py from the current directory. Of course, it's not good form to put a bunch of modules right into the installation directory (*prefix*/lib/python1.5/site-packages on Unix or *prefix* on DOS/Windows)—collections of top-level module distributions should get their own directory, and a path configuration (.pth) file to add it to `sys.path` at run-time. This is handled by the `install_path` option (an alias for `install.install_path`). Consider a distribution that ships a bunch of top-level modules in src, and wants to install them to site-packages/foo with foo.pth:

```
setup (
  ...
  packages = [''],
  package_dir = {'': 'src'}
  install_path = 'foo'
)
```

If for some reason the installation directory and .pth file should have different names, `install_path` can be a tuple or comma-delimited string (the tuple/string duality is necessary to allow this option to be set on the command-line as well):

```
setup (
  ...
  packages = [''],
  package_dir = {'': 'src'}
  install_path = ('foo', 'foo/bar/baz')
)
```

will put the distribution's modules all the way down in site-packages/foo/bar/baz, and put foo.pth—referencing foo/bar/baz—right in site-packages.

(Actually, all of these references to site-packages should be to the installation option that controls the base installation directory—currently `install_site_lib` for pure Python module distributions—but you get the idea.)

If you need to specify pure Python modules explicitly, then you can't use the `packages` option at all—you need `py_modules` (an alias for `build_py.modules`). For example, if your source distribution includes modules foo.mod1 and foo.mod2 in the foo directory, use this:

```
setup (
  ...
  py_modules = ['foo.mod1', 'foo.mod2']
)
```

with the usual addition of `package_dirs` if you have a non-standard directory layout.

Whether you use `packages` or `py_modules`, the Distutils will take care of finding and installing the __init__.py file for each package, and will warn at installation time if any are missing. It will *not* generate an __init__.py for you if you forget it.

Things get a bit more interesting when you throw extension modules into the mix. First of all, keep in mind that `packages` affects *only* pure Python modules; extension modules are listed separately and have their own way of being put into packages. Extension modules must always be listed explicitly in the `ext_modules` option. This option is a list of tuples, where each tuple supplies an extension name and the information necessary to build the extension. For example:

```
name = 'DateTime.mxDateTime.mxDateTime'
src = 'mxDateTime/mxDateTime.c'
setup (
  ...
  ext_modules =
    [(name, { 'sources': [src] }
    )]
)
```

is a first approximation at building the mxDateTime extension from the **mxDateTime** distribution. Note that the extension name is a fully-qualified module name; if your distribution has many extensions all in the same package, it might be more convenient to specify that package separately with `ext_package`:

```
pkg = 'DateTime.mxDateTime'
name = 'mxDateTime'
src = 'mxDateTime/mxDateTime.c'
setup (
  ...
  ext_package = pkg,
  ext_modules =
    [(name, { 'sources': [src] }
    )]
)
```

In many cases, it will be necessary to supply extra information to the C compiler. For instance, the mxDateTime extension won't compile unless the compiler knows where to find its header file. This is one of many extra bits of optional information that can be supplied in the "build info" dictionary that must always contain `sources`:

```
name = 'DateTime.mxDateTime.mxDateTime'
src = 'mxDateTime/mxDateTime.c'
setup (
  ...
  ext_modules =
    [(name,
     { 'sources': [src]
       'include_dirs': ['mxDateTime'] }
    )]
)
```

In this case, we simply instruct the compiler to look for header files in the mxDateTime subdirectory.

Of course, extension modules can depend on multiple source files—which is why `sources` is a list of filenames. For example, the Python Imaging Library (PIL) setup script might include

```
ext_modules =
  [('_imaging',
    { 'sources':
       ['_imaging.c',
        'decode.c',
        ... ,
        'path.c']
    }
  )]
```

(omitting several source files). This isn't actually enough, though: the header files must be found, and additionally the C `imaging` library must be linked in:

```
ext_modules =
  [('_imaging',
    { 'sources':
       ['_imaging.c',
        'decode.c',
        ... ,
        'path.c']
      'include_dirs': ['libImaging'],
      'library_dirs': ['libImaging'],
      'libraries': ['Imaging']
    }
  )]
```

(This ignores the possibility of linking PIL with Tcl/Tk, the IJG JPEG library, and the zlib compression library.)

4 Inside the Distutils

4.1 Core Classes

The core Distutils classes are Distribution and Command, both found in the distutils.core module. Together, these two classes coordinate everything that happens in building, distributing, and installing a Python module distribution.

The details of how they are deployed differ considerably. Distribution has a sole instance, created either in the `setup()` function on behalf of the setup script, or directly in the setup script (when more customization is needed than is convenient in a single call to `setup()`).

Command is an abstract class that is never directly instantiated; rather, a number of subclasses—the *command classes*—are defined in the distutils.command.* modules, each of which is instantiated in a controlled fashion by Distribution. In fact, creating *command objects* (instances of command classes) is so tightly controlled that each command class is effectively a singleton; a Distribution instance will never create multiple instances of any given command class.

4.1.1 The Distribution class

The Distribution object represents the module distribution being operated on. Normally, the setup script calls the `setup()` function (also located in distutils.core), which then creates the Distribution instance and starts working on it. More complex module distributions might prefer to instantiate Distribution directly, which is not covered here.

Thus, the simplest possible setup script is

```
from distutils.core import setup
setup ()
```

which does nothing interesting, except possibly die from lack of arguments to the script. (Lack of arguments to `setup()` is not necessarily an error; an empty command-line is.) However, it is instructive to consider what happens in this simplest possible case.

First, of course, the Distribution instance is created. A key feature of this creation is that (almost) all of the arguments passed to `setup()`—which are all keyword arguments—are passed to the Distribution constructor, again as keyword arguments.

The next step should be to find and parse any configuration files relating to this module distribution. However, the question of configuration files for the Distutils has not yet been adequately considered, so the feature isn't there.

Thus, the next step is really to parse the command line. (Note the order of operations, intended so that command line options will override config file(s), which in turn override the arguments hard-coded in **`setup.py`**.) The Distutils command-line syntax—or, if you prefer, the command line syntax of the setup script—was covered in section 3.1.1; to review:

```
setup.py [global-options]
         cmd [cmd-options ...]
         [cmd [cmd-options ...]] ...
```

Parsing the Distutils command line is an incremental job: first, parse global options; then, get the first command, and parse its options; continue until all arguments are consumed. If no commands are found, that's a fatal error (hence the likelihood of dying from lack of arguments above).

First, global options are easy: there is a known, fixed set of them determined by the Distribution class (see section 3.1.1). Knowing which options are valid for a particular command is a bit trickier. Since command objects can actually be of any Command subclass (a key to the Distutils' extensibility), there's no global registry of command-specific command-line options. Thus, each command class has to supply its valid command-line options; in fact, as we traverse the script arguments, we create each command object as we come upon its corresponding argument on the command-line. We then query that command object for its valid set of command-line options, using the `options` attribute.

4.1.2 The Command class

While there is only one Distribution instance in a given Distutils run, there will be many Command instances: or rather, there will be instances of many Command subclasses, as Command is not meant to be directly instantiated. It defines the interface that must be implemented by concrete command classes, and provides utility methods to help them do their job consistently and with minimal duplicated code.

Command objects exist mainly to be run, i.e. to have their `run()` method invoked. This is where the real work of a command is done: for instance, this is where the `build_ext` command compiles and links extension modules, or where the `install_py` command copies pure Python modules to the installation directory. Usually, though, some initial setup work is required before the command can be run: the command has to know which extensions to build and how to build them, which modules to install and where to install them, etc.

This, of course, is all controlled by the *command options* explained in sections 3.1.3 and 3.2.2. Setting the options for each command object is the shared job of two methods that must be implemented by every command class: `set_default_options()` and `set_final_options()`. `set_default_options()` sets the default values for all options; generally, it consists of a series of `self.foo = None` assignments (for each command option `foo`). `set_final_options()` is called only after all external sources of options (currently just the command-line, someday configuration files as well) have been processed. Its responsibility is to set the value of any options that *weren't* set from an external source.

Command instances can be created in a variety of ways, one of which was described above (create a command object when the command is mentioned in the setup script command-line). Commands not mentioned on the command line might still be instantiated; it just happens later, and only on demand. For example, the `install` command always runs `build` before attempting to install. It does so by invoking its own `run_peer()` method, which (unless `build` has already been run) looks up the `build` command object, creating it if necessary, and invokes its `run()` method.

Similarly, `install` also runs `install_py` (but only if there are any pure Python modules) and `install_ext` (but only if there are any extension modules).

Commands can also indirectly cause the instantiation of other commands by attempting to get options from those commands. Again referring to the `install` command, before it runs it must figure out the build directories—where the `build_*` commands put their output files. This is done by looking up the relevant options in the `build` command object, which must be created if it didn't already exist. (This means that the `build` command object will always exist when `install` gets around to running it, since it is referred to for option values prior to running.)

4.2 Command classes

Each Distutils command is implemented by a command class, which must implement the interface described by Command—usually by subclassing it. The standard Distutils commands (listed in section 3.1.2) are named and organized in a predictable way, so that it's easy to import the module and instantiate the class without having a global registry of Distutils commands.

In particular, the standard command `foo_bar` (if there were such a command) would be implemented by the class FooBar in the distutils.command.foo_bar module. The ability to extend the system with non-standard command classes is built-in via the distribution option `cmdclass`: this is a dictionary that maps command names to command classes (*not* class names, but actual class objects). This will allow developers to override the default behaviour of standard commands as well as to add their own custom commands to the Distutils.

4.3 Compiler abstraction model

A key to the Distutils' portability and ability to compile extensions on multiple platforms is its *compiler abstraction model*. This is simply an abstract base class, CCompiler (in the distutils.ccompiler module) that implements an object-oriented interface to an idealized C/C++ compiler. The model handles all the usual bureaucracy involved in instructing a C/C++ compiler, such as specifying header search directories, macros to define/undefine, library search directories, and libraries to link in—but it handles them via method invocations and instance attributes rather than command-line options to a separate program. Of course, there are also methods to compile a set of source files and to link them to shared or static libraries; there will soon have to be a method to link to a binary executable, as well (in order to build a new static Python interpreter).

Currently, this abstract base class has two concrete subclasses, UnixCCompiler and MSVCCompiler. Both of these translate attribute values supplied by method calls into command-line arguments for, respectively, the traditional Unix compiler command-line and Microsoft Visual C++'s command-line interface. The compiler model should extend to platforms that don't even have a command-line, such as the Macintosh, but this remains to be demonstrated.

4.4 Utility modules and classes

Finally, the Distutils includes a number of modules of possibly broader utility, all in the distutils package. These include:

fancy_getopt	front-end to the standard getopt module, driven by a table (list of tuples) that ties long and short options together with help text
spawn	cross-platform (Unix and Windows at least) mechanism for running external programs: a front-end to `fork()` and `execvp()` on Unix, and to `spawnvp()` on Windows
text_file	provides TextFile, a file-like object that returns lines from a file after dealing with comments, "backslash joining", stripping trailing and/or leading whitespace, etc.
util	various filesystem utilities: copy files, copy directory trees, move files, compare timestamps on individual files and groups of files, create directories "deeply", etc.

These will certainly be useful to developers writing fancy setup scripts (or new Distutils commands), and could well be handy in other contexts.

5 Availability

Up-to-date information on the Distutils, including mailing list archives, access to the CVS repository, and downloads of the latest version, can all be found at

`http://www.python.org/sigs/distutils-sig/`

6 Conclusions and Future Plans

The Python Module Distribution Utilities are well on the way to being a general, powerful, extensible framework for building, distributing, and installing Python module distributions. The current release as of this writing (version 0.1.1) has the demonstrated ability to build and install several real-world Python module distributions (Numeric Python, mxDateTime, PIL, and of course the Distutils themselves).

Development is ongoing. A number of features are needed before the Distutils is ready to take over the world, including support for building external C/C++ libraries, creation and maintenance of a database of modules installed on a given system, checking developer-supplied prerequisites for a distribution, and a standard mechanism for running test suites (and a methodology for writing them).

Once these features are a reality, the Python world will have made a major step forward to achieving "plug and play" reusability, a goal that many language communities strive for, but few have achieved.