3. Writing the Setup Configuration File

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

The setup configuration file is a useful middle-ground between the setup script—which, ideally, would be opaque to installers [1]—and the command-line to the setup script, which is outside of your control and entirely up to the installer. In fact, `setup.cfg` (and any other Distutils configuration files present on the target system) are processed after the contents of the setup script, but before the command-line. This has several useful consequences:

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

The basic syntax of the configuration file is simple:

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

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

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

```
$ python setup.py --help build_ext
[...]
Options for 'build_ext' command:
  --build-lib (-b)     directory for compiled extension modules
```

```
  --build-temp (-t)    directory for temporary files (build by-product
  --inplace (-i)       ignore build-lib and put compiled extensions ir
                       source directory alongside your pure Python moc
  --include-dirs (-I)  list of directories to search for header files
  --define (-D)        C preprocessor macros to define
  --undef (-U)         C preprocessor macros to undefine
  --swig-opts          list of SWIG command line options
[...]
```

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

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

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

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

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

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

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

```
[bdist_rpm]
release = 1
```

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

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

See also:

Syntax of config files in "Installing Python Modules"
More information on the configuration files is available in the manual for system administrators.

Footnotes

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