25. Distributing Extensions and Programs

Distribution contents

A distribution can contain a mix of Python source files, C-coded extensions, and data files. `setup` accepts optional named arguments that detail which files to put in the distribution. Whenever you specify file paths, the paths must be relative to the distribution root directory and use `/` as the path separator. `setuptools` adapts location and separator appropriately when it installs the distribution. Wheels, in particular, do not support absolute paths: all paths are relative to the top-level directory of your package.

Note

The named arguments `packages` and `py_modules` do not list file paths, but rather Python packages and modules, respectively. Therefore, in the values of these named arguments, don't use path separators or file extensions. If you list subpackage names in argument `packages`, use Python dot syntax instead (e.g., `top_package.sub_package`).

Python source files

By default, `setup` looks for Python modules (listed in the value of the named argument `py_modules`) in the distribution root directory, and for Python packages (listed in the value of the named argument `packages`) as subdirectories of the distribution root directory.

Here are the `setup` named arguments you will most frequently use to detail which Python source files are part of the distribution:

entry_points `entry_points={'group':['name=P.m:obj',],}`

`entry_points` is a `dict` holding one or more *groups*; each `group` has a list of `name`=`value` strings. `name` is an identifier, while `value` is a Python module `P.m` (the package part `P.` is optional), followed by a function, class, or other object, `obj` within `m`, with a colon `:` as the separator between module and object.

The `group` may be a plug-in, parser, or other service. See Dynamic Discovery of Services and Plugins for further information on this use. However, the most common use of `entry_points` is to create executable scripts: `console_scripts` and `gui_scripts` are the most common *group* arguments. See "What are entry_points?".

packages `packages=find_packages()|[list of package name strings]`

You can import and use `find_packages` from `setuptools` to automatically locate and include packages and subpackages in your distribution root directory. Alternatively, you may provide a list of packages. For each package name string `p` in the list, `setup` expects to find a subdirectory `p` in the distribution root directory and includes in the distribution the file *p/__init__.py*, which must be present, as well as any other file *p/*.py* (i.e., all the modules of package `p`). `setup` does not search for subpackages of `p`: unless you use `find_packages`, you must explicitly list all subpackages, as well as top-level packages, in the value of the named argument `packages`. We recommend using `find_packages`, to avoid having to update `packages` (and potentially miss a package) as your distribution grows:

find_packages `find_packages(where='.', exclude=())`

`where` indicates the directory which `find_packages` walks (subdirectories included) to find and include all packages (and modules within those packages); by default, it's `'.'`, the usual notation for "current directory," meaning, in this case, the distributions's root directory. `exclude` lists names and wildcards (e.g., `'tests'` or `'*.test'`) to be removed from the list of packages to be returned (note that `exclude` is executed last).

py_modules `py_modules=[list of module name strings]`

For each module name string `m` in the list, `setup` expects to find the file *m.py* in the distribution root directory and includes *m.py* in the distribution. Use `py_modules`, instead of `find_packages`, when you have a very simple package with only a few modules and no subdirectories.

What are entry_points?

`entry_points` are a way to tell the installer (usually `pip`) to register plug-ins, services, or scripts with the OS and, if appropriate, to create a platform-specific executable. The primary `entry_points` `group` arguments used are `console_scripts` (replaces the named argument `scripts`, which is deprecated) and `gui_scripts`. Other plug-ins and services (e.g., parsers), are also supported, but we do not cover them further in this book; see the Python Packaging User Guide for more detailed information.

When `pip` installs a package, it registers each entry point *`name`* with the OS and creates an appropriate executable (including an *.exe* launcher on Windows), which you can then run by simply entering `name` at the terminal prompt, rather than, for example, having to type `python -m mymodule`.

Scripts are Python source files that are meant to be run as main programs (see "The Main Program"), generally from the command line. Each script file should have as its first line a shebang line—that is, a line starting with `#!` and containing the substring `python`. In addition, each script should end with the following code block:

```
if __name__ == '__main__':     mainfunc()
```

To have `pip` install your script as an executable, list the script in `entry_points` under `console_scripts` (or `gui_scripts`, as appropriate). In addition to, or instead of, the main function of your script, you can use `entry_points` to register other functions as script interfaces. Here's what `entry_points` with both `console_scripts` and `gui_scripts` defined might look like:

```
entry_points={     'console_scripts': ['example=example:mainfunc',
                       'otherfunc=example:anotherfunc',                      ],    '
gui_scripts': ['mygui=mygui.gui_main:run',                  ],    },
```

After installation, type `example` at the terminal prompt to execute `mainfunc` in the module `example`. If you type `otherfunc`, the system executes `anotherfunc`, also in the module `example`.

Data and other files

To put files of any kind in the distribution, supply the following named arguments. In most cases, you'll want to use `package_data` to list your data files. The named argument `data_files` is used for listing files that you want to install to directories *outside* your package; however, we do not recommend you use it, due to complicated and inconsistent behavior, as described here:

data_files

`data_files=[`*`list of pairs(target_directory, list_of_files)`*`]`

The value of named argument `data_files` is a list of pairs. Each pair's first item is a string and names a *target directory* (i.e., a directory where `setuptools` places data files when installing the distribution); the second item is the list of file path strings for files to put in the target directory.

At installation time, installing from a wheel places each target directory as a subdirectory of Python's `sys.prefix` for a pure distribution, or of Python's `sys.exec_prefix` for a nonpure distribution; installing from *sdist* with `pip` uses `setuptools` to place target directories relative to `site_packages`, but installing without `pip` and with `distutils` has the same behavior as wheels. Because of such inconsistencies, we do not recommend you use `data_files`.

package_data

`package_data={`*`k:list_of_globs`*`, ...}`

The value of named argument `package_data` is a dict. Each key is a string and names a *package* in which to find the data files; the corresponding value is a list of glob patterns for files to include. The patterns may include subdirectories (using relative paths separated by a forward slash, `/`, even on Windows). An empty package string, `''`, recursively includes all files in any subdirectory that matches the pattern—for example, `['*.txt']` includes all *.txt* files anywhere in the top-level directory or subdirectories. At installation time, `setuptools` places each file in appropriate subdirectories relative to `site_packages`.

C-coded extensions

To put C-coded extensions in the distribution, supply the following named argument:

ext_modules

`list of instances of Extension`

All the details about each extension are supplied as arguments when instantiating the `setuptools.Extension` class. `Extension`'s constructor accepts two mandatory arguments and many optional named arguments. The simplest possible example looks something like this:

```
ext_modules=[Extension('x',sources=['x.c'])]
```

The `Extension` class constructor is:

Extension

```
class Extension(name, sources, **kwds)
```

`name` is the module name string for the C-coded extension. `name` may include dots to indicate that the extension module resides within a package. `sources` is the list of C source files that must be compiled and linked in order to build the extension. Each item of `sources` is a string that gives a source file's path relative to the distribution root directory, complete with the file extension *.c*. `kwds` lets you pass other, optional named arguments to `Extension`, as covered later in this section.

The `Extension` class also supports other file extensions besides *.c*, indicating other languages you may use to code Python extensions. On platforms having a C++ compiler, the file extension *.cpp* indicates C++ source files. Other file extensions that may be supported, depending on the platform and on various add-ons to `setuptools`, include *.f* for Fortran, *.i* for SWIG, and *.pyx* for Cython files. See "Extending Python Without Python's C API" for information about using different languages to extend Python.

In most cases, your extension needs no further information besides mandatory arguments `name` and `sources`. Note that you need to list any *.h* headers in your *MANIFEST.in* file. `setuptools` performs all that is necessary to make the Python headers directory and the Python library available for your extension's compilation and linking, and provides whatever compiler or linker flags or options are needed to build extensions on a given platform.

When additional information is required to compile and link your extension correctly, you can supply such information via the named arguments of the class `Extension`. Such arguments may potentially interfere with the cross-platform portability of your distribution. In particular, whenever you specify file or directory paths as the values of such arguments, the paths should be relative to the distribution root directory. However, when you plan to distribute your extensions to other platforms, you should examine whether you really need to provide build information via named arguments to `Extension`. It is sometimes possible to bypass such needs by careful coding at the C level.

Here are the named arguments that you may pass when calling `Extension`:

```
define_macros = [                    ) ...
(                 macro_name,macro_value]
```

Each of the items `macro_name` and `macro_value` is a string, respectively the name and value of a C preprocessor macro definition, equivalent in effect to the C preprocessor directive: `#define macro_name macro_value`.

`macro_value` can also be `None`, to get the same effect as the C preprocessor directive: `#define macro_name`.

```
extra_compile_args =  list of
[                     compile_arg        strings ]
```

Each of the strings listed as the value of `extra_compile_args` is placed among the command-line arguments for each invocation of the C compiler.

```
extra_link_args =  list of
[                  link_arg        strings ]
```

Each of the strings listed as the value of `extra_link_args` is placed among the command-line arguments for the linker.

```
extra_objects   list of
=              [object_name        strings ]
```

Each of the strings listed as the value of `extra_objects` names an object file to link in. Do not specify the file extension as part of the object name: `distutils` adds the platform-appropriate file extension (such as *.o* on Unix-like platforms and *.obj* on Windows) to help you keep cross-platform portability.

```
include_dirs =  list of
[               directory_path        strings ]
```

Each of the strings listed as the value of `include_dirs` identifies a directory to supply to the compiler as one where header files are found.

```
libraries =  list of
[            library_name        strings ]
```

Each of the strings listed as the value of `libraries` names a library to link in. Do not specify the file extension or any prefix as part of the library name: `distutils`, in cooperation with the linker, adds the platform-appropriate file extension and prefix (such as *.a*, and a prefix *lib*, on Unix-like platforms, and *.lib* on Windows) to help you keep cross-platform portability.

```
library_dirs =  list of
[               directory_path        strings ]
```

Each of the strings listed as the value of `library_dirs` identifies a directory to supply to the linker as one where library files are found.

```
runtime_library_dirs =  list of
[                       directory_path        strings ]
```

Each of the strings listed as the value of `runtime_library_dirs` identifies a directory where dynamically loaded libraries are found at runtime.

```
undef_macros =  list of
[               macro_name        strings ]
```

Each of the strings `macro_name` listed as the value of `undef_macros` is the name for a C preprocessor macro definition, equivalent in effect to the C preprocessor directive: `#undef` *`macro_name`*.