

4. Building C and C++ Extensions ¶

A C extension for CPython is a shared library (e.g. a `.so` file on Linux, `.pyd` on Windows), which exports an *initialization function*.

To be importable, the shared library must be available on `PYTHONPATH`, and must be named after the module name, with an appropriate extension. When using distutils, the correct filename is generated automatically.

The initialization function has the signature:

`PyObject*` **PyInit_**`module_name`(void)

It returns either a fully-initialized module, or a `PyModuleDef` instance. See [Initializing C modules](#) for details.

For modules with ASCII-only names, the function must be named `PyInit_<module_name>`, with `<module_name>` replaced by the name of the module. When using [Multi-phase initialization](#), non-ASCII module names are allowed. In this case, the initialization function name is `PyInitU_<module_name>`, with `<module_name>` encoded using Python's *punycode* encoding with hyphens replaced by underscores. In Python:

```
def initfunc_name(name):
    try:
        suffix = b'_' + name.encode('ascii')
    except UnicodeEncodeError:
        suffix = b'U_' + name.encode('punycode').replace(b'-', b'_')
    return b'PyInit' + suffix
```

It is possible to export multiple modules from a single shared library by defining multiple initialization functions. However, importing them requires using symbolic links or a custom importer, because by default only the function corresponding to the filename is found. See the “*Multiple modules in one library*” section in [PEP 489](#) for details.

4.1. Building C and C++ Extensions with distutils

Extension modules can be built using distutils, which is included in Python. Since distutils also supports creation of binary packages, users don't necessarily need a compiler and distutils to install the extension.

A distutils package contains a driver script, `setup.py`. This is a plain Python file, which, in the most simple case, could look like this:

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    sources = ['demo.c'])

setup (name = 'PackageName',
      version = '1.0',
      description = 'This is a demo package',
      ext_modules = [module1])
```

With this `setup.py`, and a file `demo.c`, running

```
python setup.py build
```

will compile `demo.c`, and produce an extension module named `demo` in the `build` directory. Depending on the system, the module file will end up in a subdirectory `build/lib.system`, and may have a name like `demo.so` or `demo.pyd`.

In the `setup.py`, all execution is performed by calling the `setup` function. This takes a variable number of keyword arguments, of which the example above uses only a subset. Specifically, the example specifies meta-information to build packages, and it specifies the contents of the package. Normally, a package will contain additional modules, like Python source modules, documentation, subpackages, etc. Please refer to the [distutils documentation in Distributing Python Modules \(Legacy version\)](#) to learn more about the features of `distutils`; this section explains building extension modules only.

It is common to pre-compute arguments to `setup()`, to better structure the driver script. In the example above, the `ext_modules` argument to `setup()` is a list of extension modules, each of which is an instance of the `Extension`. In the example, the instance defines an extension named `demo` which is build by compiling a single source file, `demo.c`.

In many cases, building an extension is more complex, since additional preprocessor defines and libraries may be needed. This is demonstrated in the example below.

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    define_macros = [('MAJOR_VERSION', '1'),
                                    ('MINOR_VERSION', '0')],
                    include_dirs = ['/usr/local/include'],
                    libraries = ['tcl83'],
                    library_dirs = ['/usr/local/lib'],
                    sources = ['demo.c'])

setup (name = 'PackageName',
      version = '1.0',
```

```
description = 'This is a demo package',
author = 'Martin v. Loewis',
author_email = 'martin@v.loewis.de',
url = 'https://docs.python.org/extending/building',
long_description = '''
This is really just a demo package.
''',
ext_modules = [module1])
```

In this example, `setup()` is called with additional meta-information, which is recommended when distribution packages have to be built. For the extension itself, it specifies preprocessor defines, include directories, library directories, and libraries. Depending on the compiler, `distutils` passes this information in different ways to the compiler. For example, on Unix, this may result in the compilation commands

```
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC -DMAJOR_VERSION=1 -DMI
gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/local/lib -ltcl83 -o b
```

These lines are for demonstration purposes only; `distutils` users should trust that `distutils` gets the invocations right.

4.2. Distributing your extension modules

When an extension has been successfully built, there are three ways to use it.

End-users will typically want to install the module, they do so by running

```
python setup.py install
```

Module maintainers should produce source packages; to do so, they run

```
python setup.py sdist
```

In some cases, additional files need to be included in a source distribution; this is done through a `MANIFEST.in` file; see [Specifying the files to distribute](#) for details.

If the source distribution has been built successfully, maintainers can also create binary distributions. Depending on the platform, one of the following commands can be used to do so.

```
python setup.py bdist_wininst
python setup.py bdist_rpm
python setup.py bdist_dumb
```