Packaging binary extensions

Page Status: Incomplete
Last Reviewed: 2013-12-08

One of the features of the CPython reference interpreter is that, in addition to allowing the execution of Python code, it also exposes a rich C API for use by other software. One of the most common uses of this C API is to create importable C extensions that allow things which aren't always easy to achieve in pure Python code.

Contents

An overview of binary extensions

Use cases

The typical use cases for binary extensions break down into just three conventional categories:

- **accelerator modules**: these modules are completely self-contained, and are created solely to run faster than the equivalent pure Python code runs in CPython. Ideally, accelerator modules will always have a pure Python equivalent to use as a fallback if the accelerated version isn't available on a given system. The CPython standard library makes extensive use of accelerator modules. *Example*: Wh `datetime`, Python falls back to the datetime.py module if the C impl (_datetimemodule.c) is not available.

- **wrapper modules**: these modules are created to expose existing C interfaces to Python code. They may either expose the underlying C interface directly, or else expose a more “Pythonic” API that makes use of Python language features to make the API easier to use. The CPython standard library makes extensive use of wrapper modules. *Example*: functools.py is a Python module wrapper for _functoolsmodule.c.

- **low-level system access**: these modules are created to access lower level features of the CPython runtime, the operating system, or the underlying hardware. Through platform specific code, extension modules may achieve things that aren’t possible in pure Python code. A number of CPython standard library modules are written in C in order to access interpreter internals that aren’t exposed at the language level. *Example*: `sys`, which comes from sysmodule.c.

  One particularly notable feature of C extensions is that, when they don’t need to call back into the interpreter runtime, they can release CPython’s global interpreter lock around long-running operations (regardless of whether those operations are CPU or IO bound).

Not all extension modules will fit neatly into the above categories. The extension modules included with NumPy, for example, span all three use cases - they move inner loops to C for speed reasons, wrap external libraries written in C, FORTRAN and other languages, and use low level system interfaces for both CPython and the underlying operation system to support concurrent execution of vectorised operations and to tightly control the exact memory layout of created objects.

Disadvantages

The main disadvantage of using binary extensions is the fact that it makes subsequent distribution of the software more difficult. One of the advantages of using Python is that it is largely cross platform, and the languages used to write extension modules (typically C or C++, but really any language that can bind to the CPython C API) typically require that custom binaries be created for different platforms.

This means that binary extensions:

- require that end users be able to either build them from source, or else that someone publish pre-built binaries for common platforms
- may not be compatible with different builds of the CPython reference interpreter
- often will not work correctly with alternative interpreters such as PyPy, IronPython or Jython
- if handcoded, make maintenance more difficult by requiring that maintainers be familiar not only with Python, but also with the language used to crea v: latest extension, as well as with the details of the CPython C API.

- if a pure Python fallback implementation is provided, make maintenance more difficult by requiring that changes be implemented in two places, and introducing additional complexity in the test suite to ensure both versions are always executed.

Another disadvantage of relying on binary extensions is that alternative import mechanisms (such as the ability to import modules directly from zipfiles) often won't work for extension modules (as the dynamic loading mechanisms on most platforms can only load libraries from disk).

Alternatives to handcoded accelerator modules

When extension modules are just being used to make code run faster (after profiling has identified the code where the speed increase is worth additional maintenance effort), a number of other alternatives should also be considered:

- look for existing optimised alternatives. The CPython standard library includes a number of optimised data structures and algorithms (especially in the builtins and the `collections` and `itertools` modules). The Python Package Index also offers additional alternatives. Sometimes, the appropriate choice of standard library or third party module can avoid the need to create your own accelerator module.
- for long running applications, the JIT compiled PyPy interpreter may offer a suitable alternative to the standard CPython runtime. The main barrier to adopting PyPy is typically reliance on other binary extension modules - while PyPy does emulate the CPython C API, modules that rely on that cause problems for the PyPy JIT, and the emulation layer can often expose latent defects in extension modules that CPython currently tolerates (frequently around reference counting errors - an object having one live reference instead of two often won't break anything, but no references instead of one is a major problem).
- Cython is a mature static compiler that can compile most Python code to C extension modules. The initial compilation provides some speed increases (by bypassing the CPython interpreter layer), and Cython's optional static typing features can offer additional opportunities for speed increases. Using Cython still has the disadvantage of increasing the complexity of distributing the resulting application, but has the benefit of having a reduced barrier to entry for Python programmers (relative to other languages like C or C++).
- Numba is a newer tool, created by members of the scientific Python community, that aims to leverage LLVM to allow selective compilation of pieces of a Python application to native machine code at runtime. It requires that LLVM be available on the system where the code is running, but can provide significant speed increases, especially for operations that are amenable to vectorisation.

Alternatives to handcoded wrapper modules

v: latest

The C ABI (Application Binary Interface) is a common standard for sharing functionality between multiple applications. One of the strengths of the CPython C API (Application Programming Interface) is allowing Python users to tap into that functionality. However, wrapping modules by hand is quite tedious, so a number of other alternative approaches should be considered.

The approaches described below don't simplify the distribution case at all, but they *can* significantly reduce the maintenance burden of keeping wrapper modules up to date.

- In addition to being useful for the creation of accelerator modules, Cython is also useful for creating wrapper modules. It still involves wrapping the interfaces by hand, however, so may not be a good choice for wrapping large APIs.
- cffi is a project created by some of the PyPy developers to make it straightforward for developers that already know both Python and C to expose their C modules to Python applications. It also makes it relatively straightforward to wrap a C module based on its header files, even if you don't know C yourself.

  One of the key advantages of `cffi` is that it is compatible with the PyPy JIT, allowing CFFI wrapper modules to participate fully in PyPy's tracing JIT optimisations.
- SWIG is a wrapper interface generator that allows a variety of programming languages, including Python, to interface with C *and C++* code.
- The standard library's `ctypes` module, while useful for getting access to C level interfaces when header information isn't available, suffers from the fact that it operates solely at the C ABI level, and thus has no automatic consistency checking between the interface actually being exported by the library and the one declared in the Python code. By contrast, the above alternatives are all able to operate at the C *API* level, using C header files to ensure consistency between the interface exported by the library being wrapped and the one expected by the Python wrapper module. While `cffi` *can* operate directly at the C ABI level, it suffers from the same interface inconsistency problems as `ctypes` when it is used that way.

Alternatives for low level system access

For applications that need low level system access (regardless of the reason), a binary extension module often *is* the best way to go about it. This is particularly true for low level access to the CPython runtime itself, since some operations (like releasing the Global Interpreter Lock) are simply invalid when the interpreter is running code, even if a module like `ctypes` or `cffi` is used to obtain access to the relevant C API interfaces.

For cases where the extension module is manipulating the underlying oper[illegible] or hardware (rather than the CPython runtime), it may sometimes be better [illegible] an ordinary C library (or a library in another systems programming language like C++ or

Rust that can export a C compatible ABI), and then use one of the wrapping techniques described above to make the interface available as an importable Python module.

Implementing binary extensions

The CPython Extending and Embedding guide includes an introduction to writing a custom extension module in C.

```
mention the stable ABI (3.2+, link to the CPython C API docs)
mention the module lifecycle
mention the challenges of shared static state and subinterpreters
mention the implications of the GIL for extension modules
mention the memory allocation APIs in 3.4+

mention again that all this is one of the reasons why you probably
*don't* want to handcode your extension modules :)
```

Building binary extensions

Binary extensions for Windows

Before it is possible to build a binary extension, it is necessary to ensure that you have a suitable compiler available. On Windows, Visual C is used to build the official CPython interpreter, and should be used to build compatible binary extensions.

Python 2.7 used Visual Studio 2008, Python 3.3 and 3.4 used Visual Studio 2010, and Python 3.5+ uses Visual Studio 2015 or later. Unfortunately, older versions of Visual Studio are no longer easily available from Microsoft, so for versions of Python prior to 3.5, the compilers must be obtained differently if you do not already have a copy of the relevant version of Visual Studio.

To set up a build environment for binary extensions, the steps are as follows:

For Python 2.7

1. Install “Visual C++ Compiler Package for Python 2.7”, which is available from Microsoft’s website.
2. Use (a recent version of) setuptools in your setup.py (pip will do this for you, in any case).
3. Done.

For Python 3.4

1. Install “Windows SDK for Windows 7 and .NET Framework (v7.1), which is available from Microsoft’s website.

v: latest

2. Work from an SDK command prompt (with the environment variables set, and the SDK on PATH).
3. Set DISTUTILS_USE_SDK=1
4. Done.

For Python 3.5

1. Install Visual Studio 2015 Community Edition (or any later version, when these are released).
2. Done.

Note that from Python 3.5 onwards, Visual Studio works in a backward compatible way, which means that any future version of Visual Studio will be able to build Python extensions for all Python versions from 3.5 onwards.

Building with the recommended compiler on Windows ensures that a compatible C library is used throughout the Python process.

Binary extensions for Linux

Linux binaries must use a sufficiently old glibc to be compatible with older distributions. The manylinux Docker images provide a build environment with a glibc old enough to support most current Linux distributions on common architectures.

Binary extensions for macOS

Binary compatibility on macOS is determined by the target minimum deployment system, e.g. *10.9*, which is often specified with the `MACOSX_DEPLOYMENT_TARGET` environmental variable when building binaries on macOS. When building with setuptools / distutils, the deployment target is specified with the flag `--plat-name`, e.g. `macosx-10.9-x86_64`. For common deployment targets for macOS Python distributions, see the MacPython Spinning Wheels wiki.

Publishing binary extensions

For interim guidance on this topic, see the discussion in this issue.

```
FIXME

cover publishing as wheel files on PyPI or a custom index server
cover creation of Windows and macOS installers
cover weak linking
mention the fact that Linux distros have a requirement to build
source in their own build systems, so binary-only releases are strongly
discouraged
```

v: latest

Additional resources

Cross-platform development and distribution of extension modules is a complex topic, so this guide focuses primarily on providing pointers to various tools that automate dealing with the underlying technical challenges. The additional resources in this section are instead intended for developers looking to understand more about the underlying binary interfaces that those systems rely on at runtime.

Cross-platform wheel generation with scikit-build

The scikit-build package helps abstract cross-platform build operations and provides additional capabilities when creating binary extension packages. Additional documentation is also available on the C runtime, compiler, and build system generator for Python binary extension modules.

Introduction to C/C++ extension modules

For a more in depth explanation of how extension modules are used by CPython on a Debian system, see the following articles:

- What are (c)python extension modules?
- Releasing the gil
- Writing cpython extension modules using C++

v: latest