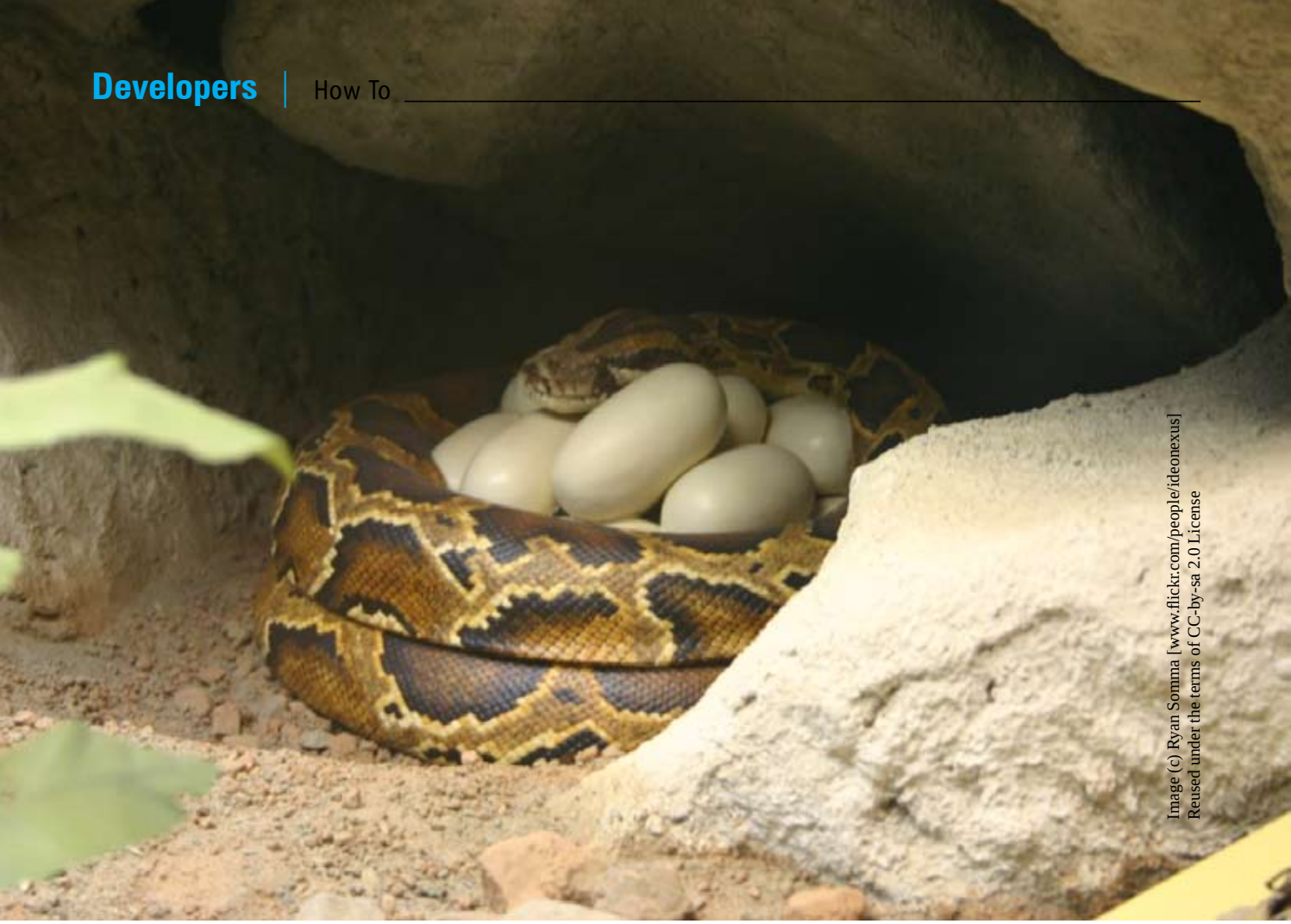

Extending Python via Shared Libraries

Python is one of the most popular programming languages ever—its great productivity, flexibility and general-purpose nature efficiently address areas ranging from Internet applications to system uses. Much of Python's power comes from extensions written in languages like C and C++. In this article, we look at extending Python with Ctypes, SWIG, Pyrex and Cython, and the pros and cons of each.

The ability to call C/C++ functions from Python code, through extensions, exposes lower-level 'system' functionality that would otherwise not be available in Python. Also, you can optimise the performance-critical parts of your applications by creating those as C/C++ extension modules, which will be compiled to native code, yielding better performance than the interpreted Python byte-code.

How can we extend Python?

The standard and most widely used implementation of Python is CPython, which is programmed in C. CPython has a C API to create extensions, but this approach of extending Python is complicated, tedious, and error-prone. You have to manually code everything from data conversion to garbage collection. Leaving even a small detail unattended increases the likelihood of crashes.

Given these problems, I will not discuss this approach to extending Python. If you're interested in it, then you could browse through the extension examples provided in the Python source directory.

The preferred way of extending Python is through extension modules created in the form of shared libraries. This way, you don't bloat Python itself with C/C++ chunks added to it, but load only the required extension module, on demand. Also, separate extension modules provide better modularity in the code of complicated or large applications. We will look at Ctypes, SWIG, Pyrex and Cython below; all of which allow you to create extensions in this manner.

Note: I used Puppy Linux 4.2.1 and Ubuntu 9.10 64-bit desktop edition to test the source code presented in the article, and to generate the screenshots.

Ctypes, a Python "insider"

Ctypes is a *foreign function interface* Python package, and part of the official Python distribution from version 2.5 onwards. Your Python code can use the Ctypes module to invoke C functions residing in shared libraries. Ctypes also does the data conversion between C and Python data types, and provides some general-purpose functions for working with shared libraries. You can create Python wrappers over your C shared libraries very easily, and code using Ctypes stands a better chance of being portable across all platforms on which Python runs.

To see Ctypes in action, type the following code in a Python file and run it:

```
from ctypes import CDLL

slibc = 'libc.so.6'
hlibc = CDLL(slibc)

iret = hlibc.abs(-7)
print iret
```

This Python code calls the GNU/Linux standard C library's *abs()* function. You can dynamically load any C shared library that contains functions following the cdecl calling convention (which is the default C calling convention on the x86 architecture), by passing its name to the *CDLL* class constructor. Alternately, use the *LoadLibrary()* method of the *CDLL* class. Ctypes provides other classes to load shared libraries that export functions under other calling conventions, like *stdcall* and *thiscall*. I encourage you to explore those on your own if you're interested—I'm focusing on the *cdecl* calling convention in this article. If you use the wrong Ctypes library-loading class for the calling convention used in functions exported by a shared library, you will encounter an exception. To learn more about calling conventions, follow the Wikipedia link in the *References* section.

Loading the shared library returns a handle to the library in the form of a *CDLL* instance. You can then invoke a function in the shared library as a method of the class instance. Simple, right? Try to call other standard C library functions like *sleep()*, *time()*, etc. You can also experiment with other *libc* functions that accept *None* or a single integer/long parameter. API documentation for *libc* is linked to in the *References* section.

The Ctypes utility function *find_library()* (in the *util* sub-module) finds the full name of any shared library given a library base name string (without the prefix "lib" and the extension and version number). The function returns *None* if it can't find a matching library. Run the following code:

```
from ctypes.util import find_library

llibs = ('bz2', 'c', 'm',)

for s in llibs:
    print (s + ':  ' + str(find_library(s)))
```

The code finds the full names of the bzip2, standard C and math libraries, respectively.

Though using Ctypes seems simple, there are enough ways to crash Python with it—in particular, passing unsupported Python data types in a Ctypes call to a library function. *None*, integers, longs, byte strings and Unicode strings are the only native Python data types/objects that you can directly pass as parameters in Ctypes function calls. (This was the reason we could pass an integer directly in the *abs()* call shown earlier.) *None* is passed as a C *NULL* pointer, byte strings and Unicode strings are passed as pointers to the memory block that contains their data; integers and longs are passed as the platform's default C *int* type.

To pass other data types into functions, you have to construct a proper data object for each variable that may not be directly passed. Ctypes' fundamental data type objects, which correspond to their counterparts in C, are: *c_char*, *c_wchar*, *c_byte*, *c_short*, *c_ushort*, *c_int*, *c_uint*, *c_long*, *c_ulong*, *c_float*, *c_double*, etc. Ctypes also has the basic pointer types, like *c_char_p*, *c_wchar_p* and *c_void_p*. You can instantiate these data objects, optionally passing an initialiser value of the corresponding Python data type to the constructor. For example, *hw=c_char_p("Hello, World")* and *us=c_ushort(-3)* creates the *hw* and *us* Ctypes data type instances.

Let's move on from calling functions in existing libraries to making a shared library of our own, and then using its exported functions in a small Python application.

Note: To keep article length down, some source code is not included in-line; you'll need to download the zip archive containing the files from *www.linuxforu.com/article_source_code/extending-python.zip*. Extract the contents of the archive in a folder you set aside for this experimentation.

Compile the file *testlib.c* to generate a shared library, with the following command:

```
gcc -shared -fPIC -o testlib.so testlib.c
```

You should find the new shared library *testlib.so* in the current working directory. To make a call to the C function in the library from Python, run the Python file *testlib.py*.

Note: Make sure that the dynamic runtime linker finds the shared libraries created in the various code examples shown in the article—otherwise the programs will throw errors. Open a terminal and navigate (*cd*) to the downloaded source code folder to make it your current working directory. Now run the following command:

```
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
```

The above command will add your current working directory to the *LD_LIBRARY_PATH*.

Note how we passed the integer data from Python to the C function: for each integer, we used *c_int()* to create the integer object, and passed a pointer to it using *byref()*. (That is because the shared library expects a pointer to an integer, because it modifies the value stored in it.) We use the *value* attribute of the integer object to retrieve the value set by the *dataModel()* function in *testlib.c*. You can follow the same method for any supported data object/type.

You can also manipulate C arrays, structures, unions, pointers, etc, through Ctypes. To explore more of these, follow the Ctypes link in the *References* section.

SWIG—a wrapper generator for Python extension modules

Ctypes is limited in some ways: you can extend Python only with functions written in C. Also, you need to create Python 'wrapper' code around the call to the function, with the necessary conversions between Python types and Ctypes data objects.

SWIG, the Simplified Wrapper and Interface Generator, is another way to extend Python through shared libraries. It's a powerful tool that can automatically generate Python bindings for C and C++ sources. Google extensively uses SWIG, which is proof of its capabilities. SWIG is not limited to Python use only: it can wrap C and C++ functionality for use in more than a dozen programming languages, including Ruby, Perl, PHP, Lua, Tcl and Java.

The easiest way to install SWIG on Ubuntu is to run the following command:

```
sudo apt-get install swig
```

To install SWIG from source, you need GCC and g++ (install the *build-essential* metapackage). Download the SWIG source tarball from its homepage (see *References*) and then run these commands:

```
tar -zxvf swig-version.tar.gz && cd swig-version
./configure && make && sudo make install
```

You also need Python development headers installed to create extension modules with SWIG. Now run the following:

```
sudo apt-get install python-dev
```

SWIG follows a layered approach to generate Python extension modules from C/C++ sources: it generates a C file that contains the lower-level code required for the extension module, and a Python file that contains the higher-level code. To generate an extension module from your C/C++ source, first you need to prepare an *interface file* that provides SWIG with information about the declaration and definitions of your data structures and routines. SWIG generates layered wrapper files from the interface file. Then you turn the generated C wrapper file and your source file into a shared library, and use your extension library through the generated Python wrapper (see the example session below). You could also automate generation of the shared library using Python's inbuilt *distutils* package.

Let's see SWIG in action, with a trivial C++ extension module. In the working directory where you extracted the source files archive, take a look at *testmodule.i*, *testmodule.hpp* and *test.py*.

The SWIG interface file *testmodule.i* has the directive *%module* that specifies the extension module you intend to generate. You could also provide the module name with the *-module* switch to SWIG if you don't provide this directive in the interface file. The text between *%{* and *%}* is put in the generated C/C++ wrapper file verbatim, so it is the place for all macros, headers, etc., required to build the module library. Declaration of module data structures and routines is done below this section. SWIG generates the wrapper files based upon these signatures.

Run the command swig -python -c++ testmodule.i to generate the wrapper files testmodule_wrap.cxx and testmodule.py in the working directory. Run g++ -shared -fPIC -o testmodule.so testmodule.cpp testmodule_wrap.cxx -I(location of Python headers) to build the shared extension library. Finally, execute test.py to access the C++ routine helloSwig() from Python.

The *-c++* and *-python* switches instruct SWIG to generate a C++ wrapper for Python. You can change the wrapper filename from the default *modulename_wrap* with the *-o* switch. Also note that the name of the generated shared library is *_modulename.so*, as this is the Python naming convention for extension modules, and the generated Python wrapper module looks for it under this name.

Now we try something more serious. Look at the files *testclass.i*, *testclass.hpp*, *testclass.cpp* and *testoop.py*. Generate the extension shared library as described for the previous example. Run *testoop.py* to instantiate and access a simple C++ class from Python code.

As you have seen, to create extension modules with

SWIG, you need the C/C++ source and header files of the code you want to wrap as a Python extension. You also need the Python header files to compile the wrappers. You need to write the interface file, too, but SWIG's capabilities can balance the extra effort required.

These examples should give you enough of a boost to start with SWIG. SWIG is a very powerful tool that lets you use almost all the advanced features of C/C++ to create extension modules, and also has many powerful features of its own. Covering SWIG in detail would require a book in itself; you can explore it via the documentation provided on its home page.

Pyrex and Cython—creating extension modules with Python itself

Both Ctypes and SWIG wrap existing C/C++ code that you might have to write yourself. SWIG's interface file is extra work as well. Besides, you can't create new Python types with SWIG—and if something goes wrong, then debugging SWIG-generated C/C++ code is a daunting task. Creating new functionality and debugging it in C/C++ is complicated and tedious.

Instead of all this, here's a different approach: create Python modules by writing your extension's code in Python itself! This code is then compiled to the equivalent C code before building into a shared library.

Pyrex is a specialised language very similar to Python. It allows you to create C data types and functions in Python-like code. Cython is inspired by Pyrex; it is more feature-rich and optimised than Pyrex, is under very active development and is more frequently updated than Pyrex. Therefore, this section mainly explores Cython; you can apply working knowledge from here to Pyrex without much extra effort, should you choose to do so.

To install Pyrex, if you wish to, download the latest source tarball from its home page (see *References*) and run the following commands:

```
tar -zxvf Pyrex-version.tar.gz && cd Pyrex-version && sudo python setup.py install
```

To install Cython, you again require the Python development headers (install with the command *sudo apt-get install python-dev*). Download the latest source tarball of Cython from its home page, and run:

```
tar -zxvf Cython-version.tar.gz && cd Cython-version && sudo python setup.py install
```

To get a glimpse of Cython in action, compile the *primes.pyx* example file taken from the Cython official documentation with *cython primes.pyx*. To compile the generated *primes.c*, run *gcc -shared -fPIC -o primes.so primes.c -I(path of Python headers)*. To test the created extension module, run *testprimes.py*.

Unlike SWIG, it doesn't matter if you name the shared library with or without an underscore before the module name: *_primes.so or primes.so*. The extension module in Python works for both these names.

If you look at the source of *primes.pyx*, you will notice that we are mixing C and Python types in the Python routine to calculate prime numbers. In Cython, you declare C data types, and C struct, *union* or *enum* types with the *cdef* keyword. You can also declare functions with *cdef*—see below.

There are two kinds of function definitions in Cython—Python functions, defined using the familiar Python *def* statement, and C functions, defined using the *cdef* statement. The first take Python objects as parameters and return Python objects, while Cython C functions can take either Python objects or C values as parameters, and can return either Python objects or C values. The small catch is that within a Cython module, Python functions and C functions can call each other freely—but you can only call Python functions from outside the module (from interpreted Python code). So, any functions that you want to 'export' from your Cython module should be declared as Python functions using *def*—or use the hybrid *cpdef*, which can be called from both outside the module and from within, but which uses the faster C calling conventions only when being called from other Cython code.

Cython's source code compiler translates Python code to the equivalent C code that is executed within the CPython runtime environment, but at the speed of compiled C, and with the ability to call directly into C libraries. Yet, it keeps the original interface of the Python source code, making it directly usable from Python code. These characteristics enable Cython to extend the CPython interpreter with fast binary modules, and also to interface Python code with external C libraries.

The sample code was just an introduction to Cython; you can learn more about it (and Pyrex) from the documentation provided on the home pages.

To sum up, extension modules greatly enhance the functionality and power of Python. You can create very innovative Python applications using the tools we've covered in this article—being limited only by your imagination. END

References

- GNU libc documentation: *http://www.gnu.org/software/libc/manual/*
- X86 calling conventions: *http://en.wikipedia.org/wiki/X86_calling_conventions*
- Python library documentation on Ctypes: *http://docs.python.org/library/ctypes.html*
- SWIG home page: *http://swig.org/*
- Pyrex home page: *http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex*
- Cython home page: *http://cython.org/*

By: Ankur Kumar Sharma

The author is a software developer and researcher with experience of more than eight years in various areas like kernel/hardware programming, systems development, automation tools development, R&D, etc. He is now the founder of RichNusGeeks Consulting, an open source consulting and training start-up. He likes to play classic rock music on his guitar and croon along, read self-help books, write, and explore stuff that interests him in his spare time. He blogs at http://www.richnusgeeks.wordpress.com.