Embed Python scripting in C applications

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Before you start

About this tutorial

Embedding a scripting language into your applications offers many benefits, such as the ability to use functionality from an embedded language that would otherwise be a difficult or at least a lengthy process from within C. Typical examples of such functionality include the regular expression and text-handling functionality that languages such as Python use for parsing documents and other information. Other examples include adding scripting ability to your applications (often used by game programmers to build the components of a game, from characters to objects) or providing a user-extensible component to the language, much like the macro systems in many desktop applications.

This tutorial shows you how to embed Python scripts in your C applications and explains how you can use the functionality of the Python scripting language from within C to extend the capabilities of the host language. We first look at the basics -- that is, the conversion of information between the two languages -- and then at the different methods for embedding Python scripts into your C applications.

You should have a basic knowledge of Python programming and some experience in compiling C programs.

Prerequisites

The examples in this tutorial rely on your having access to a recent version of Python (Python 2.1 or later). You can *download Python from the Python home page* (http://www.python.org) at Python.org. You also need access to a C compiler and a suitable `make` tool, such as `make, gmake,` or `dmake`. Both the compiler and the tool should be standard with most UNIX/Linux installations.

About the author

Martin C. Brown is a former IT Director with experience in cross-platform integration. A keen developer, he has produced dynamic sites for blue-chip customers, including HP and Oracle, and is the Technical Director of Foodware.net. Now a freelance writer and consultant, MC (as he is better known) works closely with Microsoft as an SME, is the LAMP Technologies Editor for *LinuxWorld* magazine, is a core member of the AnswerSquad.com team, and has written several books on topics as diverse as Microsoft certification, iMacs, and open source programming. Despite his best attempts, he remains a regular and voracious programmer on many platforms and numerous environments.

For technical questions or comments about the contents of this tutorial, contact MC at *questions@mcslp.com* or through his *Web site* (http://www.mcslp.com/contact) , or click **Feedback** at the top of any panel.

Section 2. Introduction to how Python embedding works

Why use embedding?

The problem with most compiled languages is that the resulting application is fixed and inflexible. Long before Python was developed, I worked with free-text databases in which I would parse and reformat a dump of the data and de-dupe keyword lists and reformat addresses and other information. At the time, C was all I had available. Developing a cleaner application was a long and complicated process -- not because the data were particularly complex, but because each time the database changed, I had to add new functions to handle different types of information and to parse the data in new ways to cope with the new fields.

Today, I'd probably use Python embedded in a C application to perform these same tasks. The C application would rarely change; it would take in information from the Python component and reprocess the data for the desired output format. The Python component would parse the source text, and I'd be able to handle any type of database format I needed just by rewriting the Python component embedded in the C application. I could do all this without recompiling the C application, yet I'd still have all the speed of a native C application.

In short, by embedding the Python interpreter into my application, I can combine the best of both worlds and use the best components of each language and environment to my advantage. In this case, it's the text-parsing capability, but you could just as easily use the same methods to support all sorts of applications that could benefit from the combination of the two languages.

Principles of Python embedding

You can execute Python code from within a C application in three ways: by using an arbitrary Python string, by calling a Python object, or by integrating with a Python module. The Python string is just a piece of text that you might otherwise have executed from within Python by using the `exec` statement of the `eval` function.

Beyond the different execution methods, the basic sequence is straightforward:

1. Initialize an instance of the Python interpreter.
2. Execute your Python code (string, object, or module).
3. Close (finalize) the Python interpreter.

The actual embedding process is easy, but getting the process and integration between the different areas correct is more difficult. This process is the main focus of this tutorial.

There are different ways in which you can integrate between the different components to allow for more effective communication. Let's start by looking at the embedding API and how to translate between C and Python datatypes.

Section 3. The fundamentals of embedding

The Python Embedding API

The Python Embedding API is surprisingly simple. The functions are a combination of those that execute code, such as `PyRun_String`, those that control the environment for the code, such as `PyImport_ImportModule`, or those that supply and/or access information and objects from the environment. Table 1 summarizes the main functions.

C API call	**Python equivalent**	**Description**
`PyImport_ImportModule`	`import module`	Imports a module into the Python insta
`PyImport_ReloadModule`	`reload(module)`	Reloads the specified module
`PyImport_GetModuleDict`	`sys.modules`	Returns a dictionary object containing of loaded modules
`PyModule_GetDict`	`module.__dict__`	Returns the dictionary for a given obje
`PyDict_GetItemString`	`dict[key]`	Gets the value for a corresponding dic key
`PyDict_SetItemString`	`dict[key] = value`	Sets a dictionary key’s value
`PyDict_New`	`dict = {}`	Creates a new dictionary object
`PyObject_GetAttrString`	`getattr(obj, attr)`	Gets the attribute for a given object
`PyObject_SetAttrString`	`setattr(obj, attr, val)`	Sets the value for a given attribute in a object
`PyEval_CallObject`	`apply(function, args)`	Calls a function with arguments in `arg`
`PyRunString`	`eval(expr), exec expr`	Executes `expr` as a Python statement
`PyRun_File`	`execfile(filename)`	Executes the file filename
`PySetProgramName`	`sys.argv[0] = name`	Changes the name of the Python prog typically set on the command line
`PyGetProgramName`	`sys.argv[0]`	Returns the name of the Python progra name set through `PySetProgramNam`
`PySys_SetArgv`	`sys.argv = list`	Sets the arguments typically supplied command line; should be supplied with arguments (`argc` and `argv`), the numb arguments, and a list of strings, startin 0

You'll use many of these functions as you start building embedded Python applications. Also useful is an additional suite of functions that provide information about the Python interpreter itself. I cover this suite later.

Getting embedded instance information

Getting information about the Python interpreter is important because you can use it when you execute an application on another platform to determine whether you support the target environment. The functions supported are all part of the Python C API; you can use them just as you could any other part of the C API.

Table 2 lists the main functions in this section of the API. Using the functions here, you can gather information about the installation, including version numbers, module paths, and compiler and build information for the Python library used when building the application.

Function	Description
`char* Py_GetPrefix()`	Returns the prefix for the platform-independent files
`char* PyGetExecPrefix()`	Returns the execution prefix for the installed Python files
`char* Py_GetPath()`	Returns the list of directories that Python searches for modules (The return value is a string, so directories are separated by colons under UNIX variants and semicolons under Windows.)
`char *Py_GetProgramFullPath()`	Returns the full default path to the Python interpreter (Obviously, if you moved interpreter from this location after installation, you won't get the answer you expect.)
`const char* Py_GetVersion()`	Returns a string for the current version of the Python library
`const char* Py_GetPlatform()`	Returns the platform identifier for the current platform
`const char* Py_GetCopyright()`	Returns the copyright statement
`const char* Py_GetCompiler()`	Returns the compiler string (name and version of the compiler) used to build the Python library
`const char* Py_GetBuilderInfo()`	Returns the build information (version and date) of the interpreter

You can put all these data together to generate a useful application that reports the information. I show you how to do so in the next panel.

Create a Python information application

Using the API, both for full embedding and for the information application you're going to build here, is straightforward. Begin by using `Py_Initialize()` to start the interpreter. I've included a test in the application (by means of `Py_IsInitialized()`) to ensure that the Python interpreter is ready, then I simply call each information-gathering function to show information about the interpreter.

Next, I call `Py_Finalize()` to close the interpreter and free up the memory before terminating. Technically, this step should occur when the application ends, but it's good practice to close the interpreter deliberately. Here's the final code:

```
#include <Python.h>

int main()
{
  printf("Getting Python information\n");
  Py_Initialize();
  if( !Py_IsInitialized() ) {
    puts('Unable to initialize Python interpreter.');
    return 1;
  }

  printf("Prefix: %s\nExec Prefix: %s\nPython Path: %s\n",
         Py_GetPrefix(),
         Py_GetExecPrefix(),
         Py_GetProgramFullPath());
  printf("Module Path: %s\n",
         Py_GetPath());
  printf("Version: %s\nPlatform: %s\nCopyright: %s\n",
         Py_GetVersion(),
         Py_GetPlatform(),
         Py_GetCopyright());
  printf("Compiler String: %s\nBuild Info: %s\n",
         Py_GetCompiler(),
         Py_GetBuildInfo());

  Py_Finalize();
  return 0;
}
```

Now, let's look at the process of building the application.

Report the information

Here is the result of running of the application from the previous panel:

```
$ ./pyinfo
Getting Python information
Prefix: /usr
Exec Prefix: /usr
Python Path: /usr/bin/python
Module Path: /usr/lib/python2.2/:/usr/lib/python2.2/plat-linux2:\
/usr/lib/python2.2/lib-tk:/usr/lib/python2.2/lib-dynload
```

```
Version: 2.2.3 (#1, Oct 15 2003, 23:33:35)
[GCC 3.3.1 20030930 (Red Hat Linux 3.3.1-6)]
Platform: linux2
Copyright: Copyright (c) 2001, 2002 Python Software Foundation.
All Rights Reserved.

Copyright (c) 2000 BeOpen.com.
All Rights Reserved.

Copyright (c) 1995-2001 Corporation for National Research Initiatives.
All Rights Reserved.

Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
Compiler String:
[GCC 3.3.1 20030930 (Red Hat Linux 3.3.1-6)]
Build Info: #1, Oct 15 2003, 23:33:35
```

Although the application is a simple example of how you can use the Python API, it does show you the basics. To really use the API, however, you need to understand how to translate between different data types. In the next panel, I show you the basic mechanics of translating between Python data types and C data types so that you can exchange information.

Translating between data types

Converting between C and Python formats relies on a solution similar to what C uses for printing and "reading" variables through the `printf` and `scanf` functions. Within the Python API, the `PyArg_Parse*` suite of functions extracts information from objects within the Python original into local C types.

For example, you can use the `PyArg_ParseTuple` function to extract information from a tuple of values (that is, a fixed list of values) -- a frequent return type from many functions. You could extract a couple of values from such a tuple by using:

```
float radius, height;
PyArg_ParseTuple(args,"ff",&radius,&height);
```

The first argument is the Python tuple object from which you want to extract values. The second argument is the format to use when extracting values. In this case, I've used the letter *f* twice to signify two floating point values. The last two arguments are the pointers to the C variables into which I want the data loaded. You can also compose Python objects by using the `Py_BuildValue` function. Try creating a Python floating point variable from a C `float` variable. The code looks like this:

```
PyObject pyfloat;
float pi = 3.141592654;
pyfloat = Py_BuildValue("f",pi);
```

Note how I've used the same letter *f* to signify a floating point value. And instead of extracting the data, the function returns a Python object (using the special C type `PyObject`) made up from the supplied value.

The format string is used when creating and extracting values; it follows a format similar to `printf`. You can get a more detailed list from the Python documentation (see the *Resources* (#resources) section at the end of this tutorial for more information), but the main format types and their corresponding data type are listed in Table 3.

Format	**Python Type**	**C type**	**Notes**
s	String	char *	Null terminates string
i	Integer	int	
f	Float	float	
(items)	Tuple	Variable list	Where *items* is a list of appropriate format specifiers
[items]	Array	Variable list	C to Python only, where *items* is a list of format specifiers
[items]	Dictionary	Variable list	C to Python only, where *items* is a list of format specifiers (Specify the key first, then the value.)

Section 4. Build embedded applications

Use Distutils to build an application

Compiling an embedded application is complicated -- not because there are many steps, but because you need to tell your C compiler where to find the necessary header files and libraries required to build the final application. The problem is that this information isn't standardized across platforms or even across individual machines of the same platform: The user or administrator can install Python anywhere. And as if that weren't complicated enough, individual installations can use a range of command-line options during the build process that must be duplicated when building any application that relies on the Python libraries.

Fortunately, help is at hand in the guise of the `distutils` system and, in particular, `distutils.sysconfig`, which provides information about the current Python installation. You can use these tools to get information about the Python header files and libraries necessary for actually building the application.

You need three main pieces of information:

- The options to the C compiler
- The list of include directories for header files
- The list of libraries

You can extract the information by using the following statement:

```
distutils.sysconfig.get_config_var(OPT)
```

For example, the list of libraries that you need to build into an application is stored in several different options. The system libraries reside in SYSLIBS, the libraries that the core Python interpreter requires reside in LIBS, and those libraries that many Python modules require reside in MODLIBS. Because these libraries can also appear in locations other than the default location, you need to determine the library directories, which reside in LIBDIR. This last option is actually a space-separated list of library directories; for inclusion on the command line, you need to prefix each directory with *-L.* The include directories, which reside in INCLDIRSTOMAKE, need the same treatment but with an *-I* prefix to identify them as *include* rather than *library* directories.

Daunted? There's a simpler way.

Write a script to create a Makefile

To make the build process easier, I've written a script (which I've been using for many years) to build a simple Makefile using the information that the `distutils` tool provides. You can use the resulting Makefile in combination with a suitable `make` tool to compile nearly any embedded application. Here's my simple Python script, which I call `pymkfile.py`:

```
#!/usr/local/bin/python

import distutils.sysconfig
import string, sys

configopts = {}

maketemplate = """
PYLIB=%(pythonlib)s
PYINC=-I%(pythoninc)s
LIBS=%(pylibs)s
OPTS=%(pyopt)s
PROGRAMS=%(programs)s

all: $(PROGRAMS)

"""

configopts['pythonlib'] = distutils.sysconfig.get_config_var('LIBPL') \
                          + '/' + \
                          distutils.sysconfig.get_config_var('LIBRARY')
configopts['pythoninc'] = ''
configopts['pylibs'] = ''
for dir in string.split(distutils.sysconfig.get_config_var('INCLDIRSTOMAKE')):
        configopts['pythoninc'] += '-I%s ' % (dir,)
for dir in string.split(distutils.sysconfig.get_config_var('LIBDIR')):
        configopts['pylibs'] += '-L%s ' % (dir,)

configopts['pylibs'] += distutils.sysconfig.get_config_var('MODLIBS') \
                        + ' ' + \
                        distutils.sysconfig.get_config_var('LIBS') \
                        + ' ' + \
                        distutils.sysconfig.get_config_var('SYSLIBS')
configopts['pyopt'] = distutils.sysconfig.get_config_var('OPT')

targets = ''
for arg in sys.argv[1:]:
        targets += arg + ' '
configopts['programs'] = targets

print maketemplate % configopts

for arg in sys.argv[1:]:
        print "%s: %s.o\n\tgcc %s.o $(LIBS) $(PYLIB) -o %s" \
              % (arg, arg, arg, arg)
        print "%s.o: %s.c\n\tgcc %s.c -c $(PYINC) $(OPTS)" \
              % (arg, arg, arg)

print "clean:\n\trm -f $(PROGRAMS) *.o *.pyc core"
```

It's probably best not to worry too much about the content of this script. Aside from a small amount of initial preamble to extract, then build up some information, the bulk of the script is given over to writing out a Makefile with the various pieces of information embedded at suitable places. I show you how to use this script in the next panel.

Create a Makefile

To use the `pymkfile.py` script, provide the name of the source file you want to build (without its extension). For example, to build a Makefile to compile the Python information application I created in the previous section, you would run the command:

```
$ ./pymkfile.py pyinfo >Makefile
```

The result, on the Linux x86 host that I'm using, is the Makefile below:

```
PYLIB=/usr/lib/python2.2/config/libpython2.2.a
PYINC=-I-I/usr/include -I/usr/include -I/usr/include/python2.2 -I/usr/include/python2.2
LIBS=-L/usr/lib  -ldl  -lpthread -lutil -lm
OPTS=-DNDEBUG -O2 -g -pipe -march=i386 -mcpu=i686 -D_GNU_SOURCE -fPIC
PROGRAMS=pyinfo

all: $(PROGRAMS)


pyinfo: pyinfo.o
        gcc pyinfo.o $(LIBS) $(PYLIB) -o pyinfo
pyinfo.o: pyinfo.c
        gcc pyinfo.c -c $(PYINC) $(OPTS)
clean:
        rm -f $(PROGRAMS) *.o *.pyc core
```

To use this Makefile, simply run the `make` command:

```
$  make
gcc pyinfo.c -c -I-I/usr/local/include -I/usr/local/include \
    -I/usr/local/include/python2.1 -I/usr/local/include/python2.1 \
    -g -O2 -W
all -Wstrict-prototypes
gcc pyinfo.o -L/usr/local/lib  -lpthread -lsocket -lnsl -ldl \
    -lthread -lm /usr/local/lib/python2.1/config/libpython2.1.a \
    -o pyinfo
```

Incidentally, you can provide the names of multiple sources to the `pymkfile.py` script. The script will produce multiple targets to build each source individually. For example, to build a Makefile for the `pyinfo` example and the upcoming `pystring` example, you would type:

```
$ ./pymkfile.py pyinfo pystring>Makefile
```

To compile either application individually, specify the name of the application (target) to `make`:

```
$ make pystring
```

To simply rebuild all the applications whose source files have changed, type `make`. Now, let's look at a real embedded Python example.

Execute an arbitrary string

The simplest method of embedding Python code is to use the `PyRun_SimpleString()` function. This function sends a single line of code to the interpreter. Because the interpreter is persistent, it's like typing the individual lines in an interactive Python session, which means that you can execute new lines to the interpreter by calling `PyRun_SimpleString` with each line of code. For example, the following code runs a simple Python program to split and reassemble a string:

```
#include <Python.h>

int main()
{
  printf("String execution\n");
  Py_Initialize();
  PyRun_SimpleString("import string");
  PyRun_SimpleString("words = string.split('rod jane freddy')");
  PyRun_SimpleString("print string.join(words,', ')");
  Py_Finalize();
  return 0;
}
```

You can see that the basic structure is simple. You have the initialize and finalize steps, and embedded between are the calls to the Python interpreter. To compile the program, copy the code and save it to a file called *pystring.c*. First, use the script to build a suitable Makefile:

```
$ pymkfile.py pystring > Makefile
```

Now, run `make` to actually build the code. After the final application has been created, run the application to see the results:

```
$  ./pystring
String execution
rod, jane, freddy
```

You've just executed a simple bit of Python code by using a C wrapper. You could use the same basic technique to perform many different tasks that use Python embedded in an application. The only obvious limitation is that, in essence, the two components are separate. There's no communication between the embedded Python interpreter and the host C application.

So, you gave the interpreter some code and it executed and printed it out; note how you didn't have to take the output from the interpreter, then print it: The interpreter sent the information directly to the standard output. Now, let's move on to more realistic embedding situations by integrating with the Python object and class system to execute arbitrary code from external modules.

Section 5. Advanced embedding techniques

Integrating with objects

In the previous section, you learned how you can execute strings through an embedded interpreter. These strings are executed without any further intervention or integration. In fact, you could have either run the code directly through Python or by used the `PyRun_File()` function: You wouldn't have recognized the difference.

However, if you want to work with Python code and pass information to and from the Python interpreter to process information from the host application, you need to work with the interpreter in a different way. You do so by working with Python objects. In essence, everything within Python is an object of some kind. Through objects, you can extract attributes and use a consistent referencing format for everything from modules to variables. The object interface within embedded code uses the object structure to access entities in the system directly and execute code. For the system to work, you need to use the Python/C data-exchange system you saw in the first section as well as additional functions from the API that will create a suitable environment and provide an interface to the interpreter in a way that enables you to use objects directly.

First, you need to start with a module. Use a simple module that can reverse a variety of data types, such as strings or lists, or create the inverse of numbers and swap keys for values in a dictionary. Here, you can see the `reverse` module:

```
def rstring(s):
    i = len(s)-1
    t = ''
    while(i > -1):
        t += s[i]
        i -= 1
    return t

def rnum(i):
    return 1.0/float(i)

def rlist(l):
    l.reverse()
    return l

def rdict(d):
    e = {}
    for k in d.keys():
        e[d[k]] = k
    return e
```

To use a single function from within an embedded application, you need to access a reference to an object. The `PyObject_GetAttrString()` function gets an objects attribute by name by accessing the appropriate attribute table. All you have to do is supply the host object and the name of the attribute. In this case, you want to access a function, so choose `rstring`, which is actually an attribute of the `reverse` module.

Call an object

You're going to use the `rstring` function by supplying it with a string to be reversed, then print the result. Both the source string and the return value will be supplied and printed by the host C application. To exchange information with the function in this way, perform the following steps:

1. Import the module that contains the function. (Remember the loaded module object.)
2. Get the reference to the function by accessing the attribute of the module.
3. Build the Python variable that you'll supply to the function.
4. Call the function.
5. Convert the returned Python object back into a C variable.

Use this sequence to start building the final code. First, load the module. Rather than using a `PyRun_*` statement, use the `PyImport_ImportModule()` function. This function accepts a module name and returns a Python object (in this case, a module object). You'll need this object when you want to pick a function from that module. The fragment of code looks like this:

```
PyObject *mymod = NULL;
mymod = PyImport_ImportModule("reverse");
```

Now, using that module, pick the `rstring` function from the module by accessing the corresponding attribute that points to that function. Again, the return value will be a Python object. To get your function object, you use the following statement:

```
PyObject *strfunc = NULL;
strfunc = PyObject_GetAttrString(mymod, "rstring");
```

The resulting object is a function reference that you can use to call the function from within the C wrapper. Now, you need to build the argument list. There's only one argument to this function, so you can simply call `Py_BuildValue` to convert a string. Using a static string, the code looks like this:

```
PyObject *strargs = NULL;
strargs = PyBuildValue("(s)", "Hello World");
```

Note that you could just as easily have used an argument from the command line by supplying this fragment, instead:

```
strargs = PyBuildValue("(s)", argv[1]);
```

The call to the function uses `PyEval_CallObject`, which accepts a suitable code object (the object you extracted through attributes earlier in this section) and the arguments (which you've just built):

```
PyObject *strret = NULL;
PyEval_CallObject(strfunc, strargs);
```

Finally, the `rstring` function returns the string in its reversed format. You need to trap that return value, convert it back to a C variable, and print it out:

```
char *cstrret = NULL;
PyArg_Parse(strret, "s", &cstrret);
printf("Reversed string: %s\n",cstrret);
```

Putting all that together by hand is challenging, so let's look at the final C source.

Calling the reversal functions

Here's the final code.

```
#include <Python.h>

int main()
{
  PyObject *strret, *mymod, *strfunc, *strargs;
  char *cstrret;

  Py_Initialize();

  mymod = PyImport_ImportModule("reverse");

  strfunc = PyObject_GetAttrString(mymod, "rstring");

  strargs = Py_BuildValue("(s)", "Hello World");

  strret = PyEval_CallObject(strfunc, strargs);
  PyArg_Parse(strret, "s", &cstrret);
  printf("Reversed string: %s\n", cstrret);

  Py_Finalize();
  return 0;
}
```

If you compile the code, then run the application, you should get a suitably reversed string:

```
$ ./pyreverse
Reversed string: dlroW olleH
```

If you have problems getting the example to work, make sure that the `reverse.py` module is in the Python interpreter's path. (This path is probably the same -- that is, the current -- directory.) Make sure you update the value of the `PYTHONPATH` environment variable accordingly.

The final method for integrating with Python from within C is through the class/object system, a common requirement with an object-based language.

Section 6. Working with Python classes

The basic class

Because Python is an object-oriented language, most of the interaction with Python will often be with Python classes and objects. To use a class within an embedded application, you follow more or less the same sequence as before: You load the module, create a suitable object, use `PyObject_GetAttrString` to get the reference to the object or its methods, then execute the method as before. For the example I provide here, you're going to use a simple module for converting a given Celsius value into Fahrenheit:

```
class celsius:
    def __init__(self, degrees):
        self.degrees = degrees
    def farenheit(self):
        return ((self.degrees*9.0)/5.0)+32.0
```

A simple use of this class within Python might look like this:

```
import celsius
temp = celsius.celsius(100)
print temp.farenheit()
```

Now, let's look at an embedded Python example that performs the same basic sequence.

The embedded version

The embedded version performs the same operation and follows the same basic sequence as the earlier object example. This example also includes error checking at each stage so that you can see how to verify the progress of each step.

```
#include <Python.h>

/* Create a function to handle errors when they occur */
void error(char errstring)
{
  printf("%s\n",errstring);
  exit(1);
}

int main()
{
/* Set up the variables to hold methods, functions and class
   instances. farenheit will hold our return value */
  PyObject *ret, *mymod, *class, *method, *args, *object;
  float farenheit;

  Py_Initialize();
```

```
/* Load our module */
  mymod = PyImport_ImportModule("celsius");

/* If we dont get a Python object back there was a problem */
  if (mymod == NULL)
    error("Can't open module");

/* Find the class */
  class = PyObject_GetAttrString(mymod, "celsius");

/* If found the class we can dump mymod, since we won't use it
   again */
  Py_DECREF(mymod);

/* Check to make sure we got an object back */
  if (class == NULL)
    {
      Py_DECREF(class);
      error("Can't find class");
    }

/* Build the argument call to our class - these are the arguments
   that will be supplied when the object is created */
  args = Py_BuildValue("(f)", 100.0);

  if (args == NULL)
    {
      Py_DECREF(args);
      error("Can't build argument list for class instance");
    }

/* Create a new instance of our class by calling the class
   with our argument list */
  object = PyEval_CallObject(class, args);
  if (object == NULL)
    {
      Py_DECREF(object);
      error("Can't create object instance");
    }

/* Decrement the argument counter as we'll be using this again */
  Py_DECREF(args);

/* Get the object method - note we use the object as the object
   from which we access the attribute by name, not the class */
  method = PyObject_GetAttrString(object, "farenheit");

  if (method == NULL)
    {
      Py_DECREF(method);
      error("Can't find method");
    }

/* Decrement the counter for our object, since we now just need
   the method reference */
  Py_DECREF(object);
```

```
/* Build our argument list - an empty tuple because there aren't
   any arguments */
  args = Py_BuildValue("()");

  if (args == NULL)
    {
      Py_DECREF(args);
      error("Can't build argument list for method call");
    }

/* Call our object method with arguments */
  ret = PyEval_CallObject(method,args);

  if (ret == NULL)
    {
      Py_DECREF(ret);
      error("Couldn't call method");
    }

/* Convert the return value back into a C variable and display it */
  PyArg_Parse(ret, "f", &farenheit);
  printf("Farenheit: %f\n", farenheit);

/* Kill the remaining objects we don't need */
  Py_DECREF(method);
  Py_DECREF(ret);

/* Close off the interpreter and terminate */
  Py_Finalize();
  return 0;
}
```

One key difference between this example and the previous examples is that in addition to the error check, you also free up Python objects that you've created but no longer need. However, you don't have to free the objects directly: Instead, you tell Python through the `Py_DECREF` function to decrement the reference count for the object. In Python (and other languages), when the reference count to a variable reaches zero, the object is automatically deleted as part of the garbage-collection process.

If you compile and run the example, you should get output identical to the earlier, Python-based version:

```
$ ./pycelsius
Farenheit: 212.000000
```

Section 7. Summary and resources

Summary

In this tutorial, you've seen the basic process for embedding Python scripts in your C applications. You can use various embedding methods shown here for different types of solutions. The simple string execution isn't that useful unless you want to be able to run a piece of Python code from within your C application that performs a specific task without using `exec` or a similar command -- a common requirement for embedded hardware, for example.

With the object- and class-based versions, you have more detailed connectivity between the Python and C components of the application. Using these methods, you can make the best of both C and Python in a final application -- for example, using Python for the mathematical components (something it's very good at) but with the fixed distributable format of a C-based application. You should be able to adapt the examples in this tutorial to your own needs and requirements.

Resources

- You can find more information about embedding Python in the online documentation at *Embedding Python in Another Application* (http://docs.python.org/ext/embedding.html) .
- For more information about the Python-to-C translation functions for converting data types and the formats available, check out the *The Py_BuildValue() Function* documentation page on the Python Web site.
- The examples in this tutorial were taken from Chapter 27 of *Python: The Complete Reference* (http://www.mcslp.com/projects/books/pytcr) by Martin C. Brown.
- You can also find the source code for all the examples in the book, including those used here, on my homepage, *MCslp.com* (http://www.mcslp.com)

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like developerWorks to cover.

For questions about the content of this tutorial, contact the author, Martin C. Brown, at *questions@mcslp.com*.

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

For more information about the Toot-O-Matic, visit www-106.ibm.com/developerworks/xml/library/x-toot/ .