CHAPTER 17

Extending Python

You can implement anything in Python, really; it's a powerful language, but sometimes it can get a bit too slow. For example, if you're writing a scientific simulation of some form of nuclear reaction, or you're rendering the graphics for the next *Star Wars* movie (wait—there won't be any more now, will there?), writing the high-performance code in Python will probably not be a good choice. Python is meant to be easy to work with and to help make the development fast. The flexibility needed for this comes with a hefty price in terms of efficiency. It's certainly fast enough for most common programming tasks, but if you need real speed, languages such as C, C++, and Java can usually beat it by several orders of magnitude.

The Best of Both Worlds

Now, I don't want to encourage the speed freaks among you to start developing exclusively in C. Although this may speed up the program itself, it will most certainly slow down your programming. So you need to consider what is most important: getting the program done quickly, or eventually (in the distant future) getting a program that runs *really, really* fast. If Python is *fast enough*, the extra pain involved will make using a low-level language such as C something of a meaningless choice (unless you have other requirements, such as running on an embedded device that doesn't have room for Python, or something like that).

This chapter deals with the cases where you *do* need extra speed. The best solution then probably isn't to switch entirely to C (or some other low- or mid-level language); instead, I recommend the following approach, which has worked for plenty of industrial-strength speed freaks out there (in one form or another):

1. Develop a prototype in Python. (See Chapter 19 for some material on prototyping.)
2. Profile your program and determine the bottlenecks. (See Chapter 16 for some material on testing.)
3. Rewrite the bottlenecks as a C (or C++, C#, Java, Fortran,[1] and so on) extension.

1. Fortran was the first "real" programming language (originally developed in 1954). In some areas, Fortran is still the language of choice for high-performance computing. If you want to (or, perhaps more likely, have to) use Fortran for your extensions, you should check out Pyfort (`http://pyfortran.sf.net`) and F2PY (`http://cens.ioc.ee/projects/f2py2e`).

The resulting architecture—a Python framework with one or more C components—is a very powerful one, because it combines the best of two worlds. It's a matter of choosing the right tools for each job. It affords you the benefits of developing a complex system in a high-level language (Python), and it lets you develop your smaller (and presumably simpler) speed-critical components in a low-level language (C).

Note There are other reasons for reaching for C. For example, if you want to write low-level code for interfacing with a strange piece of hardware, you really have no alternative.

If you have some knowledge of what the bottlenecks of your system will be even before you begin, you can (and probably should) design your prototype so that replacing the critical parts is easy. I think I might as well state this in the form of a tip:

Tip Encapsulate potential bottlenecks.

You may find that you don't need to replace the bottlenecks with C extensions (perhaps you suddenly got hold of a faster computer), but at least the option is there.

There is another situation that is a common use case for extensions as well: legacy code. You may want to use some code that exists only in, say, C. You can then "wrap" this code (write a small C library that gives you a proper interface) and create a Python extension library from your wrapper.

In the following sections, I give you some starting points for extending both the classic C implementation of Python, either by writing all the code yourself or by using a tool called SWIG, and for extending two other implementations: Jython and IronPython. You will also find some hints about other options for accessing external code. Read on . . .

THE OTHER WAY AROUND

In this chapter, I focus on writing extensions to your Python programs in a compiled language. But turning this on its head—writing a program in a compiled language and embedding a Python interpreter for minor scripting and extensions—can have its uses. In that case, what you're after when embedding Python isn't speed—it's flexibility. In many ways, it's the same "best of both worlds" argument that is used for writing compiled extensions; it's just that the focus is shifted.

The embedding approach is used in many real-world systems. For example, many computer games (which are almost invariably written in compiled languages, with a code base primarily developed for maximum speed) use dynamic languages such as Python for describing high-level behavior (such as the "intelligence" of the characters in the game), while the main code engine takes care of graphics and the like.

The documentation referenced in the main text (for CPython, Jython, and IronPython) also discusses the embedding option, in case you wish to go that route.

The Really Easy Way: Jython and IronPython

If you happen to be running Jython or IronPython (both mentioned in Chapter 1), extending Python with native modules is quite easy. The reason for this is that Jython and IronPython give you direct access to modules and classes from the underlying languages (Java for Jython, and C# and other .NET languages for IronPython), so you don't need to conform to some specific API (as you must when extending CPython). You simply implement the functionality you need, and, as if by magic, it will work in Python. As a case in point, you can access the Java standard libraries directly in Jython and the C# standard libraries directly in IronPython.

Listing 17-1 shows a simple Java class.

Listing 17-1. *A Simple Java Class (JythonTest.java)*

```
public class JythonTest {

   public void greeting() {
      System.out.println("Hello, world!");
   }

}
```

You can compile this with some Java compiler, such as `javac` (freely downloadable from `http://java.sun.com`):

```
$ javac JythonTest.java
```

■Tip If you're working with Java, you can also use the command `jythonc` to compile your Python classes into Java classes, which can then be imported into your Java programs.

Once you have compiled the class, you fire up Jython (and put the `.class` file either in your current directory or somewhere in your Java `CLASSPATH`):

```
$ CLASSPATH=JythonTest.class jython
```

You can then import the class directly:

```
>>> import JythonTest
>>> test = JythonTest()
>>> test.greeting()
Hello, world!
```

See? There's nothing to it.

JYTHON PROPERTY MAGIC

Jython has several nifty tricks up its sleeve when it comes to interacting with Java classes. One of the most obviously useful is that it gives you access to so-called JavaBean properties through ordinary attribute access. In Java, you use accessor methods to read or modify these. What this means is that if the Java instance `foo` has a method called `setBar`, you can simply use `foo.bar = baz` instead of `foo.setBar(baz)`. Similarly, if the instance has a method called either `getBar` or `isBar` (for Boolean properties), you can access the value using `foo.bar`. Using an example from the Jython documentation, instead of this:

```
b = awt.Button()
b.setEnabled(False)
```

you could use this:

```
b = awt.Button()
b.enabled = False
```

In fact, all properties can be set through keyword arguments in constructors as well. So you could, in fact, simply write this:

```
b = awt.Button(enabled=False)
```

This works with tuples for multiple arguments and even function arguments for Java idioms such as event listeners:

```
def exit(event):
    java.lang.System.exit(0)
b = awt.Button("Close Me!", actionPerformed=exit)
```

In Java, you would need to implement a separate class with the proper `actionPerformed` method, and then add that using `b.addActionListener`.

Listing 17-2 shows a similar class in C#.

Listing 17-2. *A Simple C# Class (IronPythonTest.cs)*

```
using System;
namespace FePyTest {
   public class IronPythonTest {

      public void greeting() {
         Console.WriteLine("Hello, world!");
      }

   }
}
```

Compile this with your compiler of choice (free software is available from `http://www.mono-project.com`). For Microsoft .NET, the command is as follows:

```
csc.exe /t:library IronPythonTest.cs
```

One way of using this in IronPython would be to compile the class to a dynamic link library (DLL; see the documentation for your C# installation for details) and update the relevant environment variables (such as `PATH`) as needed. Then you should be able to use it as in the following (using the IronPython interactive interpreter):

```
>>> import clr
>>> clr.AddReferenceToFile("IronPythonTest.dll")
>>> import FePyTest
>>> f = FePyTest.IronPythonTest()
>>> f.greeting()
```

For more details on these implementations of Python, visit the Jython web site (`http://www.jython.org`) and the IronPython web site (`http://www.codeplex.com/Wiki/View.aspx?ProjectName=IronPython`).

Writing C Extensions

This is what it's all about, really. Extending Python normally means extending CPython, the standard version of Python, implemented in the programming language C.

■ **Tip** For a basic introduction and some background material, see the Wikipedia article on C, `http://en.wikipedia.org/wiki/C_programming_language`. For more information, check out Ivor Horton's book *Beginning C: From Novice to Professional*, *Fourth Edition* (Apress, 2006). A really authoritative source of information is the all-time classic by Brian Kernighan and Dennis Ritchie, the inventors of the language: *C Programming Language, Second Edition* (Prentice-Hall, 1988).

C isn't quite as dynamic as Java or C#, and it's not as easy for Python to figure out things for itself if you just supply it with your compiled C code. Therefore, you need to adhere to a strict API when writing C extensions for Python. I discuss this API a bit later, in the section "Hacking It on Your Own." Several projects try to make the process of writing C extensions easier, though, and one of the better-known projects is SWIG, which I discuss in the following section. (See the sidebar "Other Approaches" for some . . . well . . . other approaches.)

OTHER APPROACHES

If you're using CPython, plenty of tools are available to help you speed up your programs, either by generating and using C libraries or by actually speeding up your Python code. Here is an overview of some options:

- **Psyco** (`http://psyco.sf.net`): A specialized just-in-time compiler for Python, which can speed up certain kinds of code (especially low-level code dealing with lists of numbers) by an order of magnitude or more. It won't help in all cases, and does need quite a bit of memory to do its job well. It's very easy to use. In the simplest case, just import it and call `psyco.full()`. One of the interesting things about Psyco is that it actually analyzes what goes on while the program is running, so it may, in fact, speed up some Python code beyond what you could achieve by writing a C extension! (Perhaps it's worth a try, before you dive into the nearest C textbook?)

- **Pyrex** (`http://www.cosc.canterbury.ac.nz/~greg/python/Pyrex`): A Python "dialect"—sort of. It's a language specifically designed for writing extension modules for Python. The Pyrex language combines Python (or a subset of it) with optional static typing as in C. Once you've written a module in Pyrex, you can translate it into C code using the `pyrexc` program. The resulting C code will have been constructed to conform to the Python C API, so after compiling it (as described in the main text), you should be able to use it in your Python program without problems. Pyrex can certainly take much of the drudgery out of writing C extensions, while still letting you control the details you might care about, such as the exact C data types for some of your variables.

- **PyPy** (`http://codespeak.net/pypy`): This is an ambitious and forward-looking implementation of Python—*in Python*. While this might sound super-slow, the hope is that eventually, through quite advanced code analysis and compilation, it will outperform CPython. According to the web site, "Rumors have it that the secret goal is being faster-than-C, which is nonsense, isn't it?" At the core of PyPy lies RPython, which is a restricted dialect of Python. RPython is suited for automated type inference and the like, permitting translation into static languages or native machine code, or to other dynamic languages (such as JavaScript), for that matter.

- **Weave** (`http://www.scipy.org/Weave`). Part of the SciPy distribution, but also available separately, Weave is a tool for including C or C++ code directly in your Python code (as strings) and having the code compiled and executed seamlessly. If you have certain mathematical expressions you want to compute quickly, for example, then this might be the way to go. Weave can also speed up expressions using numeric arrays (see the next item).

- **NumPy** (`http://numeric.scipy.org`): NumPy gives you access to numeric arrays, which are very useful for analyzing many forms of numeric data (from stock values to astronomical images). One advantage is the simple interface, which relieves the need to explicitly specify many low-level operations. The main advantage, however, is speed. Performing many common operations on every element in a numeric array is much, much faster than doing something equivalent with lists and `for` loops, because the implicit loops are implemented directly in C. Numeric arrays work well with both Pyrex and Weave.

- **ctypes** (`http://python.net/crew/theller/ctypes`): The ctypes library takes a very direct approach—it simply lets you import existing (shared) C libraries. While there are some restrictions, this is, perhaps, one of the simplest ways of accessing C code. There is no need for wrappers or special APIs. You just import the library and use it. As of Python 2.5, ctypes is part of the Python standard library.

- **subprocess** (`http://docs.python.org/lib/module-subprocess.html`): Okay, this one is a bit different. The `subprocess` module can be found in the standard library, along with the older modules and functions with similar functionality. It allows you to have Python run external programs, and communicate with them through command-line arguments and the standard input, output, and error streams. If your speed-critical code can do much of its work in a few long-running batch jobs, little time will be lost starting the program and communicating with it. In that case, simply placing your C code in a completely separate program and running it as a subprocess could well be the cleanest solution of all.
- **modulator**: Found in the `Tools` directory of your Python distribution, this script can be used to generate some of the boilerplate code needed for C extensions.
- **PyCXX** (`http://cxx.sourceforge.net`): Previously known as CXX, or CXX/Objects, this is a set of C++ facilities for writing Python extensions. For example, it includes a good deal of support for reference counting, to reduce the chances of making errors.
- **SIP** (`http://www.riverbankcomputing.co.uk/software/sip`): SIP (a pun on SWIG?) was originally created as a tool for the development of the GUI package PyQt and consists of a code generator and a Python module. It uses specification files in a manner similar to SWIG.
- **Boost.Python** (`http://www.boost.org/libs/python/doc`): Boost.Python is designed to enable seamless interoperability between Python and C++, and can give you great help with issues such as reference counting and manipulating Python objects in C++. One of the main ways of using it is to write C++ code in a rather Python-like style (enabled by Boost.Python's macros), and then compile that directly into Python extensions using your favorite C++ compiler. As a rather different yet very solid alternative to SWIG, this might certainly be worth a look.

A Swig of . . . SWIG

SWIG (`http://www.swig.org`), short for Simple Wrapper and Interface Generator, is a tool that works with several languages. On the one hand, it lets you write your extension code in C or C++; on the other hand, it automatically wraps these so that you can use them in several high-level languages such as Tcl, Python, Perl, Ruby, and Java. This means that if you decide to write some of your system as a C extension, rather than implement it directly in Python, the C extension library can also be made available (using SWIG) to a host of *other* languages. This can be very useful if you want several subsystems written in different languages to work together; your C (or C++) extension can then become a hub for the cooperation.

Installing SWIG follows the same pattern as installing other Python tools:

- You can get SWIG from the web site, `http://www.swig.org`.
- Many UNIX/Linux distributions come with SWIG. Many package managers will let you install it directly.
- There is a binary installer for Windows.
- Compiling the sources yourself is again simply a matter of calling `configure` and `make install`.

If you have problems installing SWIG, you should be able to find helpful information on the web site.

What Does It Do?

Using SWIG is a simple process, provided that you have some C code:

1. Write an *interface file* for your code. This is quite similar to C header files (and, for simple cases, you can use your header file directly).
2. Run SWIG on the interface file, in order to automatically produce some more C code (*wrapper code*).
3. Compile the original C code together with the generated wrapper code in order to generate a shared library.

In the following, I discuss each of these steps, starting with a bit of C code.

I Prefer Pi

A palindrome (such as the title of this section) is a sentence that is the same when read backwards, if you ignore spaces and punctuation and the like. Let's say you want to recognize huge palindromes, without the allowance for whitespace and friends. (Perhaps you need it for analyzing a protein sequence or something.) Of course, the string would have to be really big for this to be a problem for a pure Python program, but let's say the strings are really big, and that you need to do a whole lot of these checks. You decide to write a piece of C code to deal with it (or perhaps you find some finished code—as mentioned, using existing C code in Python is one of the main uses of SWIG). Listing 17-3 shows a possible implementation.

Listing 17-3. *A Simple C Function for Detecting a Palindrome (palindrome.c)*

```
#include <string.h>

int is_palindrome(char *text) {
  int i, n=strlen(text);
  for (i=0; i<=n/2; ++i) {
    if (text[i] != text[n-i-1]) return 0;
  }
  return 1;
}
```

Just for reference, an equivalent pure Python function is shown in Listing 17-4.

Listing 17-4. *Detecting Palindromes in Python*

```
def is_palindrome(text):
    n = len(text)
    for i in range(len(text)//2):
        if text[i] != text[n-i-1]:
            return False
    return True
```

You'll see how to compile and use the C code in a bit.

The Interface File

Assuming that you put the code from Listing 17-3 in a file called `palindrome.c`, you should now put an interface description in a file called `palindrome.i`. In many cases, if you define a header file (that is, `palindrome.h`), SWIG may be able to get the information it needs from that. So if you have a header file, feel free to try to use it. One of the reasons for explicitly writing an interface file is that you can tweak how SWIG actually wraps the code; the most important tweak is excluding things. For example, if you're wrapping a huge C library, perhaps you just want to export a couple of functions to Python. In that case, you put only the functions you want to export in the interface file.

In the interface file, you simply declare all the functions (and variables) you want to export, just like in a header file. In addition, there is a section at the top (delimited by `%{` and `%}`) where you specify included header files (such as `string.h` in our case) and before even that, a `%module` declaration, giving the name of the module. (Some of this is optional, and there is a lot more you can do with interface files; see the SWIG documentation for more information.) Listing 17-5 shows this interface file.

Listing 17-5. *Interface to the Palindrome Library (palindrome.i)*

```
%module palindrome

%{
#include <string.h>
%}

extern int is_palindrome(char *text);
```

Running SWIG

Running SWIG is probably the easiest part of the process. Although many command-line switches are available (try running `swig -help` for a list of options), the only one needed is the `-python` option, to make sure SWIG wraps your C code so you can use it in Python. Another option you may find useful is `-c++`, which you use if you're wrapping a C++ library. You run SWIG with the interface file (or, if you prefer, a header file) like this:

```
$ swig -python palindrome.i
```

After this, you should have two new files: one called `palindrome_wrap.c` and one called `palindrome.py`.

Compiling, Linking, and Using

Compiling is, perhaps, the trickiest part (at least I think so). In order to compile things properly, you need to know where you keep the source code of your Python distribution (or, at least, the header files called `pyconfig.h` and `Python.h`; you will probably find these in the root directory of your Python installation, and in the `Include` subdirectory, respectively). You also need to figure out the correct switches to compile your code into a shared library with your C compiler of choice. If you're having trouble finding the right combination of arguments and switches, take a look at the next section "A Shortcut Through the Magic Forest of Compilers."

Here is an example for Solaris using the `cc` compiler, assuming that `$PYTHON_HOME` points to the root of Python installation:

```
$ cc -c palindrome.c
$ cc -I$PYTHON_HOME -I$PYTHON_HOME/Include -c palindrome_wrap.c
$ cc -G palindrome.o palindrome_wrap.o -o _palindrome.so
```

Here is the sequence for using the `gcc` compiler in Linux:

```
$ gcc -c palindrome.c
$ gcc -I$PYTHON_HOME -I$PYTHON_HOME/Include -c palindrome_wrap.c
$ gcc -shared palindrome.o palindrome_wrap.o -o _palindrome.so
```

It may be that all the necessary include files are found in one place, such as `/usr/include/python2.5` (update the version number as needed). In this case, the following should do the trick:

```
$ gcc -c palindrome.c
$ gcc -I/usr/include/python2.5 -c palindrome_wrap.c
$ gcc -shared palindrome.o palindrome_wrap.o -o _palindrome.so
```

In Windows (again assuming that you're using `gcc` on the command line), you could use the following command as the last one, for creating the shared library:

```
$ gcc -shared palindrome.o palindrome_wrap.o C:/Python25/libs/libpython25.a -o
  _palindrome.dll
```

In Mac OS X, you could do something like the following (where `PYTHON_HOME` would be `/Library/Frameworks/Python.framework/Versions/Current` if you're using the official Python installation):

```
$ gcc -dynamic -I$PYTHON_HOME/include/python2.5 -c palindrome.c
$ gcc -dynamic -I$PYTHON_HOME/include/python2.5 -c palindrome_wrap.c
$ gcc -dynamiclib palindrome_wrap.o palindrome.o -o _palindrome.so -Wl, -undefined,
  dynamic_lookup
```

■ **Note** If you use `gcc` on Solaris, add the flag `-fPIC` to the first two command lines (right after the command `gcc`). Otherwise, the compiler will become mighty confused when you try to link the files in the last command. Also, if you're using a package manager (as is common in many Linux platforms), you may need to install a separate package (called something like `python-dev`) to get the header files needed to compile your extensions.

After these darkly magical incantations, you should end up with a highly useful file called _palindrome.so. This is your *shared library*, which can be imported directly into Python (if it's put somewhere in your PYTHONPATH, such as in the current directory):

```
>>> import _palindrome
>>> dir(_palindrome)
['__doc__', '__file__', '__name__', 'is_palindrome']
>>> _palindrome.is_palindrome('ipreferpi')
1
>>> _palindrome.is_palindrome('notlob')
0
```

In older versions of SWIG, that would have been all there was to it. Recent versions of SWIG, however, generate some wrapping code in Python as well (the file palindrome.py, remember?). This wrapper code imports the _palindrome module and takes care of a bit of checking. If you would rather skip that, you could just remove the palindrome.py file and link your library directly into a file named palindrome.so.

Using the wrapper code works just as well as using the shared library:

```
>>> import palindrome
>>> from palindrome import is_palindrome
>>> if is_palindrome('abba'):
...     print 'Wow -- that never occurred to me...'
...
Wow -- that never occurred to me...
```

A Shortcut Through the Magic Forest of Compilers

If you think the compilation process can be a bit arcane, you're not alone. If you automate the compilation (say, using a makefile), users will need to configure the setup by specifying where their Python installation is, which specific options to use with their compiler, and, not the least, which compiler to use. You can avoid this elegantly by using Distutils. In fact, it has direct support for SWIG, so you don't even need to run that manually. You just write the code and the interface file, and run your Distutils script. For more information about this magic, see the section "Compiling Extensions" in Chapter 18.

Hacking It on Your Own

SWIG does quite a bit of magic behind the scenes, but not all of it is strictly necessary. If you want to get close to the metal and grind your teeth on the processor, so to speak, you can certainly write your wrapper code yourself, or simply write your C code so that it uses the Python C API directly.

The Python C API is described in the documents "Extending and Embedding the Python Interpreter" (a tutorial) and "Python/C API Reference Manual" (a reference), both by Guido van Rossum and available from http://python.org/doc. There is quite a bit of information to swallow in these documents, but if you know some C programming, the tutorial includes a fairly gentle introduction. I'll try to be even gentler (and briefer) here. If you're curious about what I'm leaving out (which is rather a lot), you should take a look at the documents on the Python site.

Reference Counting

If you haven't worked with it before, reference counting will probably be one of the most foreign concepts you'll encounter in this section, although it's not really all that complicated. In Python, memory used is dealt with automatically—you just create objects, and they disappear when you no longer use them. In C, this isn't the case. You must explicitly *deallocate* objects (or, rather, chunks of memory) that you're no longer using. If you don't, your program may start hogging more and more memory, and you have what's called a *memory leak*.

When writing Python extensions, you have access to the same tools Python uses "under the hood" to manage memory, one of which is reference counting. The idea is that as long as some parts of your code have references to an object (that is, in C-speak, pointers pointing to it), it should not be deallocated. However, once the number of references to an object hits zero, the number can no longer increase—there is no code that can create new references to it, and it's just "free floating" in memory. At this point, it's safe to deallocate it. Reference counting automates this process. You follow a set of rules where you increment or decrement the reference count for an object under various circumstances (through a part of the Python API), and if the count ever goes to zero, the object is automatically deallocated. This means that no single piece of code has the sole responsibility for managing an object. You can create an object, return it from a function, and forget about it, safe in the knowledge that it will disappear when it is no longer needed.

You use two macros, called `Py_INCREF` and `Py_DECREF`, to increment and decrement the reference count of an object, respectively. You can find detailed information about how to use these in the Python documentation (`http://python.org/doc/ext/refcounts.html`). Here is the gist of it:

- You can't *own* an object, but you can own a *reference* to it. The reference count of an object is the number of owned references to that object.
- If you own a reference, you are responsible for calling `Py_DECREF` when you no longer need the reference.
- If you *borrow* a reference temporarily, you should *not* call `Py_DECREF` when you're finished with the object; that's the responsibility of the owner.

Caution You should certainly *never* use a borrowed reference after the owner has disposed of it. See the "Thin Ice" sections in the documentation for some more advice on staying safe.

- You can turn a borrowed reference into an owned reference by calling `Py_INCREF`. This creates a *new* owned reference; the original owner still owns the original reference.
- When you receive an object as a parameter, it's up to you whether you want the ownership of its reference transferred (for example, if you're going to store it somewhere) or you simply want to borrow it. This should be documented clearly. If your function is called from Python, it's safe to simply borrow—the object will live for the duration of the function call. If, however, your function is called from C, this cannot be guaranteed, and you might want to create an owned reference, and then release it when you're finished.

Hopefully, this will all seem clearer when we get down to a concrete example in a little while.

MORE GARBAGE COLLECTION

Reference counting is a form of *garbage collection*, where the term *garbage* refers to objects that are no longer of use to the program. Python also uses a more sophisticated algorithm to detect *cyclic* garbage; that is, objects that refer only to each other (and thus have nonzero reference counts), but have no other objects referring to them.

You can access the Python garbage collector in your Python programs, through the `gc` module. You can find more information about it in the Python Library Reference (`http://python.org/doc/lib/module-gc.html`).

A Framework for Extensions

Quite a lot of cookie-cutter code is needed to write a Python C extension, which is why tools such as SWIG, Pyrex, and modulator are so nice. Automating cookie-cutter code is the way to go. Doing it by hand can be a great learning experience, though. You do have quite some leeway in how you structure your code, really. I'll just show you a way that works.

The first thing to remember is that the `Python.h` header file must be included *first*, before other standard header files. That is because it may, on some platforms, perform some redefinitions that should be used by the other headers. So, for simplicity, just place this:

```
#include <Python.h>
```

as the first line of your code.

Your function can be called anything you want. It should be `static`, return a pointer (an owned reference) to an object of the `PyObject` type, and take two arguments, both also pointers to `PyObject`. The objects are conventionally called `self` and `args` (with `self` being the self-object, or `NULL`, and `args` being a tuple of arguments). In other words, the function should look something like this:

```
static PyObject *somename(PyObject *self, PyObject *args) {
    PyObject *result;
    /* Do something here, including allocating result. */

    Py_INCREF(result); /* Only if needed! */
    return result;
}
```

The self argument is actually used only in bound methods. In other functions, it will simply be a NULL pointer.

Note that the call to Py_INCREF may not be needed. If the object is created in the function (for example, using a utility function such as Py_BuildValue), the function will *already* own a reference to it, and can simply return it. If, however, you wish to return None from your function, you should use the existing object Py_None. In this case, however, the function does *not* own a reference to Py_None, and so should call Py_INCREF(Py_None) before returning it.

The args parameter contains all the arguments to your function (except, if present, the self argument). In order to extract the objects, you use the function PyArg_ParseTuple (for positional arguments) and PyArg_ParseTupleAndKeywords (for positional and keyword arguments). I'll stick to positional arguments here.

The function PyArg_ParseTuple has the following signature:

```
int PyArg_ParseTuple(PyObject *args, char *format, ...);
```

The format string describes the arguments you're expecting, and then you supply the addresses of the variables you want populated at the end. The return value is a Boolean value. If it's true, everything went well; otherwise, there was an error. If there was an error, the proper preparations for raising an exception will have been made (you can learn more about that in the documentation), and all you need to do is to return NULL to set it off. So, if you're not expecting any arguments (an empty format string), the following is a useful way of handling arguments:

```
    if (!PyArg_ParseTuple(args, "")) {
        return NULL;
    }
```

If the code proceeds beyond this statement, you know you have your arguments (in this case, no arguments). Format strings can look like "s" for a string, "i" for an integer, "o" for a Python object, with possible combinations such as "iis" for two integers and a string. There are many more format string codes. A full reference of how to write format strings can be found in the Python/C API Reference Manual (http://python.org/doc/api/arg-parsing.html).

■ **Note** You can create your own built-in types and classes in extension modules, too. It's not too hard, really, but still a rather involved subject. If you mainly need to factor out some bottleneck code into C, using functions will probably be enough for most of your needs anyway. If you want to learn how to create types and classes, the Python documentation is a good source of information.

Once you have your function in place, some extra wrapping is still needed to make your C code act as a module. But let's get back to that once we have a real example to work with, shall we?

Palindromes, Detartrated[2] for Your Pleasure

Without further ado, I give you the hand-coded Python C API version of the palindrome module (with some interesting new stuff added) in Listing 17-6.

Listing 17-6. *Palindrome Checking Again (palindrome2.c)*

```
#include <Python.h>

static PyObject *is_palindrome(PyObject *self, PyObject *args) {
    int i, n;
    const char *text;
    int result;
    /* "s" means a single string: */
    if (!PyArg_ParseTuple(args, "s", &text)) {
        return NULL;
    }
    /* The old code, more or less: */
    n=strlen(text);
    result = 1;
    for (i=0; i<=n/2; ++i) {
        if (text[i] != text[n-i-1]) {
            result = 0;
            break;
        }
    }
    /* "i" means a single integer: */
    return Py_BuildValue("i", result);
}

/* A listing of our methods/functions: */
static PyMethodDef PalindromeMethods[] = {
    /* name, function, argument type, docstring */
    {"is_palindrome", is_palindrome, METH_VARARGS, "Detect palindromes"},
    /* An end-of-listing sentinel: */
    {NULL, NULL, 0, NULL}
};


/* An initialization function for the module (the name is
   significant): */
PyMODINIT_FUNC initpalindrome() {
    Py_InitModule("palindrome", PalindromeMethods);
}
```

2. That is, the tartrates have been removed. Okay, so the word is totally irrelevant to the code (and more relevant to fruit juices), but at least it's a palindrome.

Most of the added stuff in Listing 17-6 is total boilerplate. Where you see `palindrome`, you could insert the name of your module. Where you see `is_palindrome`, insert the name of your function. If you have more functions, simply list them all in the `PyMethodDef` array. One thing is worth noting, though: the name of the initialization function must be `initmodule`, where `module` is the name of your module; otherwise, Python won't find it.

So, let's compile! You do this just as described in the section on SWIG, except that there is only one file to deal with now. Here is an example using `gcc` (remember to add `-fPIC` in Solaris):

```
$ gcc -I$PYTHON_HOME -I$PYTHON_HOME/Include -shared palindrome2.c -o palindrome.so
```

Again, you should have a file called `palindrome.so`, ready for your use. Put it somewhere in your `PYTHONPATH` (such as the current directory) and away we go:

```
>>> from palindrome import is_palindrome
>>> is_palindrome('foobar')
0
>>> is_palindrome('deified')
1
```

And that's it. Now go play. (But be careful; remember the Waldi Ravens quote from this book's Introduction.)

A Quick Summary

Extending Python is a huge subject. The tiny glimpse provided by this chapter included the following:

Extension philosophy: Python extensions are useful mainly for two things: for using existing (legacy) code or for speeding up bottlenecks. If you're writing your own code from scratch, try to prototype it in Python, find the bottlenecks, and factor them out as extensions *if needed*. Encapsulating potential bottlenecks beforehand can be useful.

Jython and IronPython: Extending these implementations of Python is quite easy. You simply implement your extension as a library in the underlying implementation (Java for Jython and C# or some other .NET language for IronPython) and the code is immediately usable from Python.

Extension approaches: Plenty of tools are available for extending or speeding up your code. You can find tools for making the incorporation of C code into your Python program easier, for speeding up common operations such as numeric array manipulation, and for speeding up Python itself. Such tools include SWIG, Psyco, Pyrex, Weave, NumPy, ctypes, subprocess, and modulator.

SWIG: SWIG is a tool for automatically generating wrapper code for your C libraries. The wrapper code takes care of the Python C API so you don't have to deal with it. SWIG is one of the easiest and most popular ways of extending Python.

Using the Python/C API: You can write C code yourself that can be imported directly into Python as shared libraries. To do this, you must adhere to the Python/C API. Things you need to take care of for each function include reference counting, extracting arguments, and building return values. There is also a certain amount of code needed to make a C library work as a module, including listing the functions in the module and creating a module initialization function.

New Functions in This Chapter

Function	Description
`Py_INCREF(obj)`	Increments reference count of `obj`
`Py_DECREF(obj)`	Decrements reference count of `obj`
`PyArg_ParseTuple(args, fmt, ...)`	Extracts positional arguments
`PyArg_ParseTupleAndKeywords(args, kws, fmt, kwlist)`	Extracts positional *and* keyword arguments
`PyBuildValue(fmt, value)`	Builds a `PyObject` from a C value

What Now?

Now you should either have some really cool programs or at least some really cool program ideas. Once you have something you want to share with the world (and you *do* want to share your code with the world, don't you?), the next chapter can be your next step.