Extending and Embedding the Python Interpreter

Release 1.5.2

Guido van Rossum

March 22, 2000

Corporation for National Research Initiatives
1895 Preston White Drive, Reston, VA 20191, USA
E-mail: guido@python.org

Copyright © 1991-1995 by Stichting Mathematisch Centrum, Amsterdam, The Netherlands.

All Rights Reserved

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the names of Stichting Mathematisch Centrum or CWI or Corporation for National Research Initiatives or CNRI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

While CWI is the initial source for this software, a modified version is made available by the Corporation for National Research Initiatives (CNRI) at the Internet address ftp://ftp.python.org.

STICHTING MATHEMATISCH CENTRUM AND CNRI DISCLAIM ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM OR CNRI BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Acknowledgements

The following people have contributed sections to this document: Jim Fulton, Konrad Hinsen, Chris Phoenix, and Neil Schemenauer.

Abstract

Python is an interpreted, object-oriented programming language. This document describes how to write modules in C or C++ to extend the Python interpreter with new modules. Those modules can define new functions but also new object types and their methods. The document also describes how to embed the Python interpreter in another application, for use as an extension language. Finally, it shows how to compile and link extension modules so that they can be loaded dynamically (at run time) into the interpreter, if the underlying operating system supports this feature.

This document assumes basic knowledge about Python. For an informal introduction to the language, see the *Python Tutorial*. The *Python Reference Manual* gives a more formal definition of the language. The *Python Library Reference* documents the existing object types, functions and modules (both built-in and written in Python) that give the language its wide application range.

For a detailed description of the whole Python/C API, see the separate *Python/C API Reference Manual*.

CONTENTS

1 Extending Python with C or C++ 1
- 1.1 A Simple Example 1
- 1.2 Intermezzo: Errors and Exceptions 2
- 1.3 Back to the Example 4
- 1.4 The Module's Method Table and Initialization Function 4
- 1.5 Compilation and Linkage 6
- 1.6 Calling Python Functions from C 6
- 1.7 Format Strings for `PyArg_ParseTuple()` 8
- 1.8 Keyword Parsing with `PyArg_ParseTupleAndKeywords()` 11
- 1.9 The `Py_BuildValue()` Function 12
- 1.10 Reference Counts 14
- 1.11 Writing Extensions in C++ 17
- 1.12 Providing a C API for an Extension Module 17

2 Building C and C++ Extensions on Unix 23
- 2.1 Building Custom Interpreters 24
- 2.2 Module Definition Options 24
- 2.3 Example 25
- 2.4 Distributing your extension modules 25

3 Building C and C++ Extensions on Windows 27
- 3.1 A Cookbook Approach 27
- 3.2 Differences Between Unix and Windows 27
- 3.3 Using DLLs in Practice 28

4 Embedding Python in Another Application 29
- 4.1 Embedding Python in C++ 29

CHAPTER

ONE

Extending Python with C or C++

It is quite easy to add new built-in modules to Python, if you know how to program in C. Such *extension modules* can do two things that can't be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.

To support extensions, the Python API (Application Programmers Interface) defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The Python API is incorporated in a C source file by including the header `"Python.h"`.

The compilation of an extension module depends on its intended use as well as on your system setup; details are given in later chapters.

1.1 A Simple Example

Let's create an extension module called '`spam`' (the favorite food of Monty Python fans...) and let's say we want to create a Python interface to the C library function `system()`.[1] This function takes a null-terminated character string as argument and returns an integer. We want this function to be callable from Python as follows:

```
>>> import spam
>>> status = spam.system("ls -l")
```

Begin by creating a file 'spammodule.c'. (Historically, if a module is called '`spam`', the C file containing its implementation is called 'spammodule.c'; if the module name is very long, like '`spammify`', the module name can be just 'spammify.c'.)

The first line of our file can be:

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

which pulls in the Python API (you can add a comment describing the purpose of the module and a copyright notice if you like).

All user-visible symbols defined by `"Python.h"` have a prefix of '`Py`' or '`PY`', except those defined in standard header files. For convenience, and since they are used extensively by the Python interpreter, `"Python.h"` includes a few standard header files: `<stdio.h>`, `<string.h>`, `<errno.h>`, and `<stdlib.h>`. If the latter header file does not exist on your system, it declares the functions `malloc()`, `free()` and `realloc()` directly.

The next thing we add to our module file is the C function that will be called when the Python expression '`spam.system(`*string*`)`' is evaluated (we'll see shortly how it ends up being called):

---

[1] An interface for this function already exists in the standard module `os` — it was chosen as a simple and straightfoward example.

```
static PyObject *
spam_system(self, args)
    PyObject *self;
    PyObject *args;
{
    char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```

There is a straightforward translation from the argument list in Python (e.g. the single expression `"ls -l"`) to the arguments passed to the C function. The C function always has two arguments, conventionally named *self* and *args*.

The *self* argument is only used when the C function implements a built-in method, not a function. In the example, *self* will always be a `NULL` pointer, since we are defining a function, not a method. (This is done so that the interpreter doesn't have to understand two different types of C functions.)

The *args* argument will be a pointer to a Python tuple object containing the arguments. Each item of the tuple corresponds to an argument in the call's argument list. The arguments are Python objects — in order to do anything with them in our C function we have to convert them to C values. The function `PyArg_ParseTuple()` in the Python API checks the argument types and converts them to C values. It uses a template string to determine the required types of the arguments as well as the types of the C variables into which to store the converted values. More about this later.

`PyArg_ParseTuple()` returns true (nonzero) if all arguments have the right type and its components have been stored in the variables whose addresses are passed. It returns false (zero) if an invalid argument list was passed. In the latter case it also raises an appropriate exception so the calling function can return `NULL` immediately (as we saw in the example).

1.2 Intermezzo: Errors and Exceptions

An important convention throughout the Python interpreter is the following: when a function fails, it should set an exception condition and return an error value (usually a `NULL` pointer). Exceptions are stored in a static global variable inside the interpreter; if this variable is `NULL` no exception has occurred. A second global variable stores the "associated value" of the exception (the second argument to `raise`). A third variable contains the stack traceback in case the error originated in Python code. These three variables are the C equivalents of the Python variables `sys.exc_type`, `sys.exc_value` and `sys.exc_traceback` (see the section on module `sys` in the *Python Library Reference*). It is important to know about them to understand how errors are passed around.

The Python API defines a number of functions to set various types of exceptions.

The most common one is `PyErr_SetString()`. Its arguments are an exception object and a C string. The exception object is usually a predefined object like `PyExc_ZeroDivisionError`. The C string indicates the cause of the error and is converted to a Python string object and stored as the "associated value" of the exception.

Another useful function is `PyErr_SetFromErrno()`, which only takes an exception argument and constructs the associated value by inspection of the global variable `errno`. The most general function is `PyErr_SetObject()`, which takes two object arguments, the exception and its associated value. You don't need to `Py_INCREF()` the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with `PyErr_Occurred()`. This returns the current exception object, or `NULL` if no exception has occurred. You normally don't need to call

`PyErr_Occurred()` to see whether an error occurred in a function call, since you should be able to tell from the return value.

When a function *f* that calls another function *g* detects that the latter fails, *f* should itself return an error value (e.g. `NULL` or `-1`). It should *not* call one of the `PyErr_*()` functions — one has already been called by *g*. *f*'s caller is then supposed to also return an error indication to *its* caller, again *without* calling `PyErr_*()`, and so on — the most detailed cause of the error was already reported by the function that first detected it. Once the error reaches the Python interpreter's main loop, this aborts the currently executing Python code and tries to find an exception handler specified by the Python programmer.

(There are situations where a module can actually give a more detailed error message by calling another `PyErr_*()` function, and in such cases it is fine to do so. As a general rule, however, this is not necessary, and can cause information about the cause of the error to be lost: most operations can fail for a variety of reasons.)

To ignore an exception set by a function call that failed, the exception condition must be cleared explicitly by calling `PyErr_Clear()`. The only time C code should call `PyErr_Clear()` is if it doesn't want to pass the error on to the interpreter but wants to handle it completely by itself (e.g. by trying something else or pretending nothing happened).

Every failing `malloc()` call must be turned into an exception — the direct caller of `malloc()` (or `realloc()`) must call `PyErr_NoMemory()` and return a failure indicator itself. All the object-creating functions (for example, `PyInt_FromLong()`) already do this, so this note is only relevant to those who call `malloc()` directly.

Also note that, with the important exception of `PyArg_ParseTuple()` and friends, functions that return an integer status usually return a positive value or zero for success and `-1` for failure, like UNIX system calls.

Finally, be careful to clean up garbage (by making `Py_XDECREF()` or `Py_DECREF()` calls for objects you have already created) when you return an error indicator!

The choice of which exception to raise is entirely yours. There are predeclared C objects corresponding to all built-in Python exceptions, e.g. `PyExc_ZeroDivisionError`, which you can use directly. Of course, you should choose exceptions wisely — don't use `PyExc_TypeError` to mean that a file couldn't be opened (that should probably be `PyExc_IOError`). If something's wrong with the argument list, the `PyArg_ParseTuple()` function usually raises `PyExc_TypeError`. If you have an argument whose value must be in a particular range or must satisfy other conditions, `PyExc_ValueError` is appropriate.

You can also define a new exception that is unique to your module. For this, you usually declare a static object variable at the beginning of your file, e.g.

```
static PyObject *SpamError;
```

and initialize it in your module's initialization function (`initspam()`) with an exception object, e.g. (leaving out the error checking for now):

```
void
initspam()
{
    PyObject *m, *d;

    m = Py_InitModule("spam", SpamMethods);
    d = PyModule_GetDict(m);
    SpamError = PyErr_NewException("spam.error", NULL, NULL);
    PyDict_SetItemString(d, "error", SpamError);
}
```

Note that the Python name for the exception object is `spam.error`. The `PyErr_NewException()` function may create either a string or class, depending on whether the '`-X`' flag was passed to the interpreter.

If '`-X`' was used, `SpamError` will be a string object, otherwise it will be a class object with the base class being `Exception`, described in the *Python Library Reference* under "Built-in Exceptions."

1.3 Back to the Example

Going back to our example function, you should now be able to understand this statement:

```
if (!PyArg_ParseTuple(args, "s", &command))
    return NULL;
```

It returns `NULL` (the error indicator for functions returning object pointers) if an error is detected in the argument list, relying on the exception set by `PyArg_ParseTuple()`. Otherwise the string value of the argument has been copied to the local variable `command`. This is a pointer assignment and you are not supposed to modify the string to which it points (so in Standard C, the variable `command` should properly be declared as '`const char *command`').

The next statement is a call to the UNIX function `system()`, passing it the string we just got from `PyArg_ParseTuple()`:

```
sts = system(command);
```

Our `spam.system()` function must return the value of `sts` as a Python object. This is done using the function `Py_BuildValue()`, which is something like the inverse of `PyArg_ParseTuple()`: it takes a format string and an arbitrary number of C values, and returns a new Python object. More info on `Py_BuildValue()` is given later.

```
return Py_BuildValue("i", sts);
```

In this case, it will return an integer object. (Yes, even integers are objects on the heap in Python!)

If you have a C function that returns no useful argument (a function returning `void`), the corresponding Python function must return `None`. You need this idiom to do so:

```
Py_INCREF(Py_None);
return Py_None;
```

`Py_None` is the C name for the special Python object `None`. It is a genuine Python object rather than a `NULL` pointer, which means "error" in most contexts, as we have seen.

1.4 The Module's Method Table and Initialization Function

I promised to show how `spam_system()` is called from Python programs. First, we need to list its name and address in a "method table":

```
static PyMethodDef SpamMethods[] = {
    ...
    {"system",  spam_system, METH_VARARGS},
    ...
    {NULL,      NULL}        /* Sentinel */
};
```

Note the third entry (‘`METH_VARARGS`’). This is a flag telling the interpreter the calling convention to be used for the C function. It should normally always be ‘`METH_VARARGS`’ or ‘`METH_VARARGS | METH_KEYWORDS`’; a value of `0` means that an obsolete variant of `PyArg_ParseTuple()` is used.

When using only ‘`METH_VARARGS`’, the function should expect the Python-level parameters to be passed in as a tuple acceptable for parsing via `PyArg_ParseTuple()`; more information on this function is provided below.

The `METH_KEYWORDS` bit may be set in the third field if keyword arguments should be passed to the function. In this case, the C function should accept a third ‘`PyObject *`’ parameter which will be a dictionary of keywords. Use `PyArg_ParseTupleAndKeywords()` to parse the arguments to such a function.

The method table must be passed to the interpreter in the module’s initialization function (which should be the only non-`static` item defined in the module file):

```
void
initspam()
{
    (void) Py_InitModule("spam", SpamMethods);
}
```

When the Python program imports module `spam` for the first time, `initspam()` is called. (See below for comments about embedding Python.) It calls `Py_InitModule()`, which creates a “module object” (which is inserted in the dictionary `sys.modules` under the key `"spam"`), and inserts built-in function objects into the newly created module based upon the table (an array of `PyMethodDef` structures) that was passed as its second argument. `Py_InitModule()` returns a pointer to the module object that it creates (which is unused here). It aborts with a fatal error if the module could not be initialized satisfactorily, so the caller doesn’t need to check for errors.

When embedding Python, the `initspam()` function is not called automatically unless there’s an entry in the `_PyImport_Inittab` table. The easiest way to handle this is to statically initialize your statically-linked modules by directly calling `initspam()` after the call to `Py_Initialize()` or `PyMac_Initialize()`:

```
int main(int argc, char **argv)
{
    /* Pass argv[0] to the Python interpreter */
    Py_SetProgramName(argv[0]);

    /* Initialize the Python interpreter.  Required. */
    Py_Initialize();

    /* Add a static module */
    initspam();
```

And example may be found in the file ‘Demo/embed/demo.c’ in the Python source distribution.

Note: Removing entries from `sys.modules` or importing compiled modules into multiple interpreters within a process (or following a `fork()` without an intervening `exec()`) can create problems for some extension modules. Extension module authors should exercise caution when initializing internal data structures.

A more substantial example module is included in the Python source distribution as ‘Modules/xxmodule.c’. This file may be used as a template or simply read as an example. The **modulator.py** script included in the source distribution or Windows install provides a simple graphical user interface for declaring the functions and objects which a module should implement, and can generate a template which can be filled in. The script lives in the ‘Tools/modulator/’ directory; see the ‘README’ file there for more information.

1.5 Compilation and Linkage

There are two more things to do before you can use your new extension: compiling and linking it with the Python system. If you use dynamic loading, the details depend on the style of dynamic loading your system uses; see the chapters about building extension modules on UNIX (chapter 2) and Windows (chapter 3) for more information about this.

If you can't use dynamic loading, or if you want to make your module a permanent part of the Python interpreter, you will have to change the configuration setup and rebuild the interpreter. Luckily, this is very simple: just place your file ('spammodule.c' for example) in the 'Modules/' directory of an unpacked source distribution, add a line to the file 'Modules/Setup.local' describing your file:

```
spam spammodule.o
```

and rebuild the interpreter by running **make** in the toplevel directory. You can also run **make** in the 'Modules/' subdirectory, but then you must first rebuild 'Makefile' there by running '**make** Makefile'. (This is necessary each time you change the 'Setup' file.)

If your module requires additional libraries to link with, these can be listed on the line in the configuration file as well, for instance:

```
spam spammodule.o -lX11
```

1.6 Calling Python Functions from C

So far we have concentrated on making C functions callable from Python. The reverse is also useful: calling Python functions from C. This is especially the case for libraries that support so-called "callback" functions. If a C interface makes use of callbacks, the equivalent Python often needs to provide a callback mechanism to the Python programmer; the implementation will require calling the Python callback functions from a C callback. Other uses are also imaginable.

Fortunately, the Python interpreter is easily called recursively, and there is a standard interface to call a Python function. (I won't dwell on how to call the Python parser with a particular string as input — if you're interested, have a look at the implementation of the '`-c`' command line option in 'Python/pythonmain.c' from the Python source code.)

Calling a Python function is easy. First, the Python program must somehow pass you the Python function object. You should provide a function (or some other interface) to do this. When this function is called, save a pointer to the Python function object (be careful to `Py_INCREF()` it!) in a global variable — or wherever you see fit. For example, the following function might be part of a module definition:

```
static PyObject *my_callback = NULL;

static PyObject *
my_set_callback(dummy, args)
    PyObject *dummy, *args;
{
    PyObject *result = NULL;
    PyObject *temp;

    if (PyArg_ParseTuple(args, "O:set_callback", &temp)) {
        if (!PyCallable_Check(temp)) {
            PyErr_SetString(PyExc_TypeError, "parameter must be callable");
            return NULL;
        }
        Py_XINCREF(temp);         /* Add a reference to new callback */
        Py_XDECREF(my_callback);  /* Dispose of previous callback */
        my_callback = temp;       /* Remember new callback */
        /* Boilerplate to return "None" */
        Py_INCREF(Py_None);
        result = Py_None;
    }
    return result;
}
```

This function must be registered with the interpreter using the `METH_VARARGS` flag; this is described in section 1.4, "The Module's Method Table and Initialization Function." The `PyArg_ParseTuple()` function and its arguments are documented in section 1.7, "Format Strings for `PyArg_ParseTuple()`."

The macros `Py_XINCREF()` and `Py_XDECREF()` increment/decrement the reference count of an object and are safe in the presence of `NULL` pointers (but note that *temp* will not be `NULL` in this context). More info on them in section 1.10, "Reference Counts."

Later, when it is time to call the function, you call the C function `PyEval_CallObject()`. This function has two arguments, both pointers to arbitrary Python objects: the Python function, and the argument list. The argument list must always be a tuple object, whose length is the number of arguments. To call the Python function with no arguments, pass an empty tuple; to call it with one argument, pass a singleton tuple. `Py_BuildValue()` returns a tuple when its format string consists of zero or more format codes between parentheses. For example:

```
    int arg;
    PyObject *arglist;
    PyObject *result;
    ...
    arg = 123;
    ...
    /* Time to call the callback */
    arglist = Py_BuildValue("(i)", arg);
    result = PyEval_CallObject(my_callback, arglist);
    Py_DECREF(arglist);
```

`PyEval_CallObject()` returns a Python object pointer: this is the return value of the Python function. `PyEval_CallObject()` is "reference-count-neutral" with respect to its arguments. In the example a new tuple was created to serve as the argument list, which is `Py_DECREF()`-ed immediately after the call.

The return value of `PyEval_CallObject()` is "new": either it is a brand new object, or it is an existing object whose reference count has been incremented. So, unless you want to save it in a global variable, you should somehow `Py_DECREF()` the result, even (especially!) if you are not interested in its value.

Before you do this, however, it is important to check that the return value isn't `NULL`. If it is, the Python function terminated by raising an exception. If the C code that called `PyEval_CallObject()` is called from Python, it should now return an error indication to its Python caller, so the interpreter can print a stack trace, or the calling Python code can handle the exception. If this is not possible or desirable, the exception should be cleared by calling `PyErr_Clear()`. For example:

```
if (result == NULL)
    return NULL; /* Pass error back */
...use result...
Py_DECREF(result);
```

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to `PyEval_CallObject()`. In some cases the argument list is also provided by the Python program, through the same interface that specified the callback function. It can then be saved and used in the same manner as the function object. In other cases, you may have to construct a new tuple to pass as the argument list. The simplest way to do this is to call `Py_BuildValue()`. For example, if you want to pass an integral event code, you might use the following code:

```
PyObject *arglist;
...
arglist = Py_BuildValue("(l)", eventcode);
result = PyEval_CallObject(my_callback, arglist);
Py_DECREF(arglist);
if (result == NULL)
    return NULL; /* Pass error back */
/* Here maybe use the result */
Py_DECREF(result);
```

Note the placement of ‘`Py_DECREF(arglist)`’ immediately after the call, before the error check! Also note that strictly spoken this code is not complete: `Py_BuildValue()` may run out of memory, and this should be checked.

1.7 Format Strings for `PyArg_ParseTuple()`

The `PyArg_ParseTuple()` function is declared as follows:

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

The *arg* argument must be a tuple object containing an argument list passed from Python to a C function. The *format* argument must be a format string, whose syntax is explained below. The remaining arguments must be addresses of variables whose type is determined by the format string. For the conversion to succeed, the *arg* object must match the format and the format must be exhausted.

Note that while `PyArg_ParseTuple()` checks that the Python arguments have the required types, it cannot check the validity of the addresses of C variables passed to the call: if you make mistakes there, your code will probably crash or at least overwrite random bits in memory. So be careful!

A format string consists of zero or more “format units”. A format unit describes one Python object; it is usually a single character or a parenthesized sequence of format units. With a few exceptions, a format unit that is not a parenthesized sequence normally corresponds to a single address argument to `PyArg_ParseTuple()`. In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that matches the format unit; and the entry in [square] brackets is the type of the C variable(s) whose address should be passed. (Use the ‘`&`’ operator to pass a variable’s address.)

Note that any Python object references which are provided to the caller are *borrowed* references; do not decrement their reference count!

‘s’ (string) [char *] Convert a Python string to a C pointer to a character string. You must not provide storage for the string itself; a pointer to an existing string is stored into the character pointer variable whose address you pass. The C string is null-terminated. The Python string must not contain embedded null bytes; if it does, a `TypeError` exception is raised.

‘`s#`’ (string) [char *, int] This variant on ‘`s`’ stores into two C variables, the first one a pointer to a character string, the second one its length. In this case the Python string may contain embedded null bytes.

‘`z`’ (string or `None`) [char *] Like ‘`s`’, but the Python object may also be `None`, in which case the C pointer is set to `NULL`.

‘`z#`’ (string or `None`) [char *, int] This is to ‘`s#`’ as ‘`z`’ is to ‘`s`’.

‘`b`’ (integer) [char] Convert a Python integer to a tiny int, stored in a C `char`.

‘`h`’ (integer) [short int] Convert a Python integer to a C `short int`.

‘`i`’ (integer) [int] Convert a Python integer to a plain C `int`.

‘`l`’ (integer) [long int] Convert a Python integer to a C `long int`.

‘`c`’ (string of length 1) [char] Convert a Python character, represented as a string of length 1, to a C `char`.

‘`f`’ (float) [float] Convert a Python floating point number to a C `float`.

‘`d`’ (float) [double] Convert a Python floating point number to a C `double`.

‘`D`’ (complex) [Py_complex] Convert a Python complex number to a C `Py_complex` structure.

‘`O`’ (object) [PyObject *] Store a Python object (without any conversion) in a C object pointer. The C program thus receives the actual object that was passed. The object’s reference count is not increased. The pointer stored is not `NULL`.

‘`O!`’ (object) [*typeobject*, PyObject *] Store a Python object in a C object pointer. This is similar to ‘`O`’, but takes two C arguments: the first is the address of a Python type object, the second is the address of the C variable (of type `PyObject *`) into which the object pointer is stored. If the Python object does not have the required type, `TypeError` is raised.

‘`O&`’ (object) [*converter*, *anything*] Convert a Python object to a C variable through a *converter* function. This takes two arguments: the first is a function, the second is the address of a C variable (of arbitrary type), converted to `void *`. The *converter* function in turn is called as follows:

status `=` *converter*`(`*object*, *address*`);`

where *object* is the Python object to be converted and *address* is the `void *` argument that was passed to `PyArg_ConvertTuple()`. The returned *status* should be `1` for a successful conversion and `0` if the conversion has failed. When the conversion fails, the *converter* function should raise an exception.

‘`S`’ (string) [PyStringObject *] Like ‘`O`’ but requires that the Python object is a string object. Raises `TypeError` if the object is not a string object. The C variable may also be declared as `PyObject *`.

‘`(`*items*`)`’ (tuple) [*matching-items*] The object must be a Python sequence whose length is the number of format units in *items*. The C arguments must correspond to the individual format units in *items*. Format units for sequences may be nested.

Note: Prior to Python version 1.5.2, this format specifier only accepted a tuple containing the individual parameters, not an arbitrary sequence. Code which previously caused `TypeError` to be raised here may now proceed without an exception. This is not expected to be a problem for existing code.

It is possible to pass Python long integers where integers are requested; however no proper range checking is done — the most significant bits are silently truncated when the receiving field is too small to receive the value (actually, the semantics are inherited from downcasts in C — your mileage may vary).

A few other characters have a meaning in a format string. These may not occur inside nested parentheses. They are:

‘`|`’ Indicates that the remaining arguments in the Python argument list are optional. The C variables corresponding to optional arguments should be initialized to their default value — when an optional argument is not specified, `PyArg_ParseTuple()` does not touch the contents of the corresponding C variable(s).

‘`:`’ The list of format units ends here; the string after the colon is used as the function name in error messages (the “associated value” of the exception that `PyArg_ParseTuple()` raises).

‘`;`’ The list of format units ends here; the string after the colon is used as the error message *instead* of the default error message. Clearly, ‘`:`’ and ‘`;`’ mutually exclude each other.

Some example calls:

```
int ok;
int i, j;
long k, l;
char *s;
int size;

ok = PyArg_ParseTuple(args, ""); /* No arguments */
    /* Python call: f() */


ok = PyArg_ParseTuple(args, "s", &s); /* A string */
    /* Possible Python call: f('whoops!') */


ok = PyArg_ParseTuple(args, "lls", &k, &l, &s); /* Two longs and a string */
    /* Possible Python call: f(1, 2, 'three') */


ok = PyArg_ParseTuple(args, "(ii)s#", &i, &j, &s, &size);
    /* A pair of ints and a string, whose size is also returned */
    /* Possible Python call: f((1, 2), 'three') */


{
    char *file;
    char *mode = "r";
    int bufsize = 0;
    ok = PyArg_ParseTuple(args, "s|si", &file, &mode, &bufsize);
    /* A string, and optionally another string and an integer */
    /* Possible Python calls:
       f('spam')
       f('spam', 'w')
       f('spam', 'wb', 100000) */
}
```

```
{
    int left, top, right, bottom, h, v;
    ok = PyArg_ParseTuple(args, "((ii)(ii))(ii)",
             &left, &top, &right, &bottom, &h, &v);
    /* A rectangle and a point */
    /* Possible Python call:
       f(((0, 0), (400, 300)), (10, 10)) */
}
```

```
{
    Py_complex c;
    ok = PyArg_ParseTuple(args, "D:myfunction", &c);
    /* a complex, also providing a function name for errors */
    /* Possible Python call: myfunction(1+2j) */
}
```

1.8 Keyword Parsing with `PyArg_ParseTupleAndKeywords()`

The `PyArg_ParseTupleAndKeywords()` function is declared as follows:

```
int PyArg_ParseTupleAndKeywords(PyObject *arg, PyObject *kwdict,
                                char *format, char **kwlist, ...);
```

The *arg* and *format* parameters are identical to those of the `PyArg_ParseTuple()` function. The *kwdict* parameter is the dictionary of keywords received as the third parameter from the Python runtime. The *kwlist* parameter is a `NULL`-terminated list of strings which identify the parameters; the names are matched with the type information from *format* from left to right.

Note: Nested tuples cannot be parsed when using keyword arguments! Keyword parameters passed in which are not present in the *kwlist* will cause `TypeError` to be raised.

Here is an example module which uses keywords, based on an example by Geoff Philbrick (philbrick@hks.com):

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

static PyObject *
keywdarg_parrot(self, args, keywds)
    PyObject *self;
    PyObject *args;
    PyObject *keywds;
{
    int voltage;
    char *state = "a stiff";
    char *action = "voom";
    char *type = "Norwegian Blue";

    static char *kwlist[] = {"voltage", "state", "action", "type", NULL};

    if (!PyArg_ParseTupleAndKeywords(args, keywds, "i|sss", kwlist,
                                     &voltage, &state, &action, &type))
        return NULL;

    printf("-- This parrot wouldn't %s if you put %i Volts through it.\n",
           action, voltage);
    printf("-- Lovely plumage, the %s -- It's %s!\n", type, state);

    Py_INCREF(Py_None);

    return Py_None;
}

static PyMethodDef keywdarg_methods[] = {
    /* The cast of the function is necessary since PyCFunction values
     * only take two PyObject* parameters, and keywdarg_parrot() takes
     * three.
     */
    {"parrot", (PyCFunction)keywdarg_parrot, METH_VARARGS|METH_KEYWORDS},
    {NULL,  NULL}   /* sentinel */
};

void
initkeywdarg()
{
  /* Create the module and add the functions */
  Py_InitModule("keywdarg", keywdarg_methods);
}
```

1.9 The `Py_BuildValue()` Function

This function is the counterpart to `PyArg_ParseTuple()`. It is declared as follows:

```
PyObject *Py_BuildValue(char *format, ...);
```

It recognizes a set of format units similar to the ones recognized by `PyArg_ParseTuple()`, but the arguments (which are input to the function, not output) must not be pointers, just values. It returns a new Python object, suitable for returning from a C function called from Python.

One difference with `PyArg_ParseTuple()`: while the latter requires its first argument to be a tuple (since Python argument lists are always represented as tuples internally), `Py_BuildValue()` does not always build a tuple. It builds a tuple only if its format string contains two or more format units. If the format string is empty, it returns `None`; if it contains exactly one format unit, it returns whatever object is described by that format unit. To force it to return a tuple of size 0 or one, parenthesize the format string.

In the following description, the quoted form is the format unit; the entry in (round) parentheses is the Python object type that the format unit will return; and the entry in [square] brackets is the type of the C value(s) to be passed.

The characters space, tab, colon and comma are ignored in format strings (but not within format units such as '`s#`'). This can be used to make long format strings a tad more readable.

'`s`' (string) [char *] Convert a null-terminated C string to a Python object. If the C string pointer is `NULL`, `None` is returned.

'`s#`' (string) [char *, int] Convert a C string and its length to a Python object. If the C string pointer is `NULL`, the length is ignored and `None` is returned.

'`z`' (string or `None`) [char *] Same as '`s`'.

'`z#`' (string or `None`) [char *, int] Same as '`s#`'.

'`i`' (integer) [int] Convert a plain C `int` to a Python integer object.

'`b`' (integer) [char] Same as '`i`'.

'`h`' (integer) [short int] Same as '`i`'.

'`l`' (integer) [long int] Convert a C `long int` to a Python integer object.

'`c`' (string of length 1) [char] Convert a C `int` representing a character to a Python string of length 1.

'`d`' (float) [double] Convert a C `double` to a Python floating point number.

'`f`' (float) [float] Same as '`d`'.

'`O`' (object) [PyObject *] Pass a Python object untouched (except for its reference count, which is incremented by one). If the object passed in is a `NULL` pointer, it is assumed that this was caused because the call producing the argument found an error and set an exception. Therefore, `Py_BuildValue()` will return `NULL` but won't raise an exception. If no exception has been raised yet, `PyExc_SystemError` is set.

'`S`' (object) [PyObject *] Same as '`O`'.

'`N`' (object) [PyObject *] Same as '`O`', except it doesn't increment the reference count on the object. Useful when the object is created by a call to an object constructor in the argument list.

'`O&`' (object) [*converter*, *anything*] Convert *anything* to a Python object through a *converter* function. The function is called with *anything* (which should be compatible with `void *`) as its argument and should return a "new" Python object, or `NULL` if an error occurred.

'`(`*items*`)`' (tuple) [*matching-items*] Convert a sequence of C values to a Python tuple with the same number of items.

'`[`*items*`]`' (list) [*matching-items*] Convert a sequence of C values to a Python list with the same number of items.

'`{`*items*`}`' (dictionary) [*matching-items*] Convert a sequence of C values to a Python dictionary. Each pair of consecutive C values adds one item to the dictionary, serving as key and value, respectively.

If there is an error in the format string, the `PyExc_SystemError` exception is raised and `NULL` returned.

Examples (to the left the call, to the right the resulting Python value):

```
Py_BuildValue("")                        None
Py_BuildValue("i", 123)                  123
Py_BuildValue("iii", 123, 456, 789)      (123, 456, 789)
Py_BuildValue("s", "hello")              'hello'
Py_BuildValue("ss", "hello", "world")    ('hello', 'world')
Py_BuildValue("s#", "hello", 4)          'hell'
Py_BuildValue("()")                      ()
Py_BuildValue("(i)", 123)                (123,)
Py_BuildValue("(ii)", 123, 456)          (123, 456)
Py_BuildValue("(i,i)", 123, 456)         (123, 456)
Py_BuildValue("[i,i]", 123, 456)         [123, 456]
Py_BuildValue("{s:i,s:i}",
              "abc", 123, "def", 456)    {'abc': 123, 'def': 456}
Py_BuildValue("((ii)(ii)) (ii)",
              1, 2, 3, 4, 5, 6)          (((1, 2), (3, 4)), (5, 6))
```

1.10 Reference Counts

In languages like C or C++, the programmer is responsible for dynamic allocation and deallocation of memory on the heap. In C, this is done using the functions `malloc()` and `free()`. In C++, the operators `new` and `delete` are used with essentially the same meaning; they are actually implemented using `malloc()` and `free()`, so we'll restrict the following discussion to the latter.

Every block of memory allocated with `malloc()` should eventually be returned to the pool of available memory by exactly one call to `free()`. It is important to call `free()` at the right time. If a block's address is forgotten but `free()` is not called for it, the memory it occupies cannot be reused until the program terminates. This is called a *memory leak*. On the other hand, if a program calls `free()` for a block and then continues to use the block, it creates a conflict with re-use of the block through another `malloc()` call. This is called *using freed memory*. It has the same bad consequences as referencing uninitialized data — core dumps, wrong results, mysterious crashes.

Common causes of memory leaks are unusual paths through the code. For instance, a function may allocate a block of memory, do some calculation, and then free the block again. Now a change in the requirements for the function may add a test to the calculation that detects an error condition and can return prematurely from the function. It's easy to forget to free the allocated memory block when taking this premature exit, especially when it is added later to the code. Such leaks, once introduced, often go undetected for a long time: the error exit is taken only in a small fraction of all calls, and most modern machines have plenty of virtual memory, so the leak only becomes apparent in a long-running process that uses the leaking function frequently. Therefore, it's important to prevent leaks from happening by having a coding convention or strategy that minimizes this kind of errors.

Since Python makes heavy use of `malloc()` and `free()`, it needs a strategy to avoid memory leaks as well as the use of freed memory. The chosen method is called *reference counting*. The principle is simple: every object contains a counter, which is incremented when a reference to the object is stored somewhere, and which is decremented when a reference to it is deleted. When the counter reaches zero, the last reference to the object has been deleted and the object is freed.

An alternative strategy is called *automatic garbage collection*. (Sometimes, reference counting is also referred to as a garbage collection strategy, hence my use of "automatic" to distinguish the two.) The big advantage of automatic garbage collection is that the user doesn't need to call `free()` explicitly. (Another claimed advantage is an improvement in speed or memory usage — this is no hard fact however.) The disadvantage is that for C, there is no truly portable automatic garbage collector, while reference counting can be implemented portably (as long as the functions `malloc()` and `free()` are available — which the C Standard guarantees). Maybe some day a sufficiently portable automatic garbage collector will be available for C. Until then, we'll have to live with reference counts.

1.10.1 Reference Counting in Python

There are two macros, `Py_INCREF(x)` and `Py_DECREF(x)`, which handle the incrementing and decrementing of the reference count. `Py_DECREF()` also frees the object when the count reaches zero. For flexibility, it doesn't call `free()` directly — rather, it makes a call through a function pointer in the object's *type object*. For this purpose (and others), every object also contains a pointer to its type object.

The big question now remains: when to use `Py_INCREF(x)` and `Py_DECREF(x)`? Let's first introduce some terms. Nobody "owns" an object; however, you can *own a reference* to an object. An object's reference count is now defined as the number of owned references to it. The owner of a reference is responsible for calling `Py_DECREF()` when the reference is no longer needed. Ownership of a reference can be transferred. There are three ways to dispose of an owned reference: pass it on, store it, or call `Py_DECREF()`. Forgetting to dispose of an owned reference creates a memory leak.

It is also possible to *borrow*[2] a reference to an object. The borrower of a reference should not call `Py_DECREF()`. The borrower must not hold on to the object longer than the owner from which it was borrowed. Using a borrowed reference after the owner has disposed of it risks using freed memory and should be avoided completely.[3]

The advantage of borrowing over owning a reference is that you don't need to take care of disposing of the reference on all possible paths through the code — in other words, with a borrowed reference you don't run the risk of leaking when a premature exit is taken. The disadvantage of borrowing over leaking is that there are some subtle situations where in seemingly correct code a borrowed reference can be used after the owner from which it was borrowed has in fact disposed of it.

A borrowed reference can be changed into an owned reference by calling `Py_INCREF()`. This does not affect the status of the owner from which the reference was borrowed — it creates a new owned reference, and gives full owner responsibilities (i.e., the new owner must dispose of the reference properly, as well as the previous owner).

1.10.2 Ownership Rules

Whenever an object reference is passed into or out of a function, it is part of the function's interface specification whether ownership is transferred with the reference or not.

Most functions that return a reference to an object pass on ownership with the reference. In particular, all functions whose function it is to create a new object, e.g. `PyInt_FromLong()` and `Py_BuildValue()`, pass ownership to the receiver. Even if in fact, in some cases, you don't receive a reference to a brand new object, you still receive ownership of the reference. For instance, `PyInt_FromLong()` maintains a cache of popular values and can return a reference to a cached item.

Many functions that extract objects from other objects also transfer ownership with the reference, for instance `PyObject_GetAttrString()`. The picture is less clear, here, however, since a few common routines are exceptions: `PyTuple_GetItem()`, `PyList_GetItem()`, `PyDict_GetItem()`, and `PyDict_GetItemString()` all return references that you borrow from the tuple, list or dictionary.

The function `PyImport_AddModule()` also returns a borrowed reference, even though it may actually create the object it returns: this is possible because an owned reference to the object is stored in `sys.modules`.

When you pass an object reference into another function, in general, the function borrows the reference from you — if it needs to store it, it will use `Py_INCREF()` to become an independent owner. There are exactly two important exceptions to this rule: `PyTuple_SetItem()` and `PyList_SetItem()`. These functions take over ownership of the item passed to them — even if they fail! (Note that `PyDict_SetItem()` and friends don't take over ownership — they are "normal.")

When a C function is called from Python, it borrows references to its arguments from the caller. The caller owns a reference to the object, so the borrowed reference's lifetime is guaranteed until the function returns. Only when such a borrowed reference must be stored or passed on, it must be turned into an

---

[2] The metaphor of "borrowing" a reference is not completely correct: the owner still has a copy of the reference.

[3] Checking that the reference count is at least 1 **does not work** — the reference count itself could be in freed memory and may thus be reused for another object!

owned reference by calling `Py_INCREF()`.

The object reference returned from a C function that is called from Python must be an owned reference — ownership is tranferred from the function to its caller.

1.10.3 Thin Ice

There are a few situations where seemingly harmless use of a borrowed reference can lead to problems. These all have to do with implicit invocations of the interpreter, which can cause the owner of a reference to dispose of it.

The first and most important case to know about is using `Py_DECREF()` on an unrelated object while borrowing a reference to a list item. For instance:

```
bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);

    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

This function first borrows a reference to `list[0]`, then replaces `list[1]` with the value `0`, and finally prints the borrowed reference. Looks harmless, right? But it's not!

Let's follow the control flow into `PyList_SetItem()`. The list owns references to all its items, so when item 1 is replaced, it has to dispose of the original item 1. Now let's suppose the original item 1 was an instance of a user-defined class, and let's further suppose that the class defined a `__del__()` method. If this class instance has a reference count of 1, disposing of it will call its `__del__()` method.

Since it is written in Python, the `__del__()` method can execute arbitrary Python code. Could it perhaps do something to invalidate the reference to `item` in `bug()`? You bet! Assuming that the list passed into `bug()` is accessible to the `__del__()` method, it could execute a statement to the effect of '`del list[0]`', and assuming this was the last reference to that object, it would free the memory associated with it, thereby invalidating `item`.

The solution, once you know the source of the problem, is easy: temporarily increment the reference count. The correct version of the function reads:

```
no_bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);

    Py_INCREF(item);
    PyList_SetItem(list, 1, PyInt_FromLong(0L));
    PyObject_Print(item, stdout, 0);
    Py_DECREF(item);
}
```

This is a true story. An older version of Python contained variants of this bug and someone spent a considerable amount of time in a C debugger to figure out why his `__del__()` methods would fail...

The second case of problems with a borrowed reference is a variant involving threads. Normally, multiple threads in the Python interpreter can't get in each other's way, because there is a global lock protecting Python's entire object space. However, it is possible to temporarily release this lock using the macro `Py_BEGIN_ALLOW_THREADS`, and to re-acquire it using `Py_END_ALLOW_THREADS`. This is common around blocking I/O calls, to let other threads use the CPU while waiting for the I/O to complete. Obviously, the following function has the same problem as the previous one:

```
bug(PyObject *list) {
    PyObject *item = PyList_GetItem(list, 0);
    Py_BEGIN_ALLOW_THREADS
    ...some blocking I/O call...
    Py_END_ALLOW_THREADS
    PyObject_Print(item, stdout, 0); /* BUG! */
}
```

1.10.4 NULL Pointers

In general, functions that take object references as arguments do not expect you to pass them `NULL` pointers, and will dump core (or cause later core dumps) if you do so. Functions that return object references generally return `NULL` only to indicate that an exception occurred. The reason for not testing for `NULL` arguments is that functions often pass the objects they receive on to other function — if each function were to test for `NULL`, there would be a lot of redundant tests and the code would run slower.

It is better to test for `NULL` only at the "source", i.e. when a pointer that may be `NULL` is received, e.g. from `malloc()` or from a function that may raise an exception.

The macros `Py_INCREF()` and `Py_DECREF()` do not check for `NULL` pointers — however, their variants `Py_XINCREF()` and `Py_XDECREF()` do.

The macros for checking for a particular object type (`Py`*type*`_Check()`) don't check for `NULL` pointers — again, there is much code that calls several of these in a row to test an object against various different expected types, and this would generate redundant tests. There are no variants with `NULL` checking.

The C function calling mechanism guarantees that the argument list passed to C functions (`args` in the examples) is never `NULL` — in fact it guarantees that it is always a tuple.[4]

It is a severe error to ever let a `NULL` pointer "escape" to the Python user.

1.11 Writing Extensions in C++

It is possible to write extension modules in C++. Some restrictions apply. If the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. This is not a problem if the main program is linked by the C++ compiler. Functions that will be called by the Python interpreter (in particular, module initalization functions) have to be declared using `extern "C"`. It is unnecessary to enclose the Python header files in `extern "C" {...}` — they use this form already if the symbol '`__cplusplus`' is defined (all recent C++ compilers define this symbol).

1.12 Providing a C API for an Extension Module

Many extension modules just provide new functions and types to be used from Python, but sometimes the code in an extension module can be useful for other extension modules. For example, an extension module could implement a type "collection" which works like lists without order. Just like the standard Python list type has a C API which permits extension modules to create and manipulate lists, this new collection type should have a set of C functions for direct manipulation from other extension modules.

At first sight this seems easy: just write the functions (without declaring them `static`, of course), provide an appropriate header file, and document the C API. And in fact this would work if all extension modules were always linked statically with the Python interpreter. When modules are used as shared libraries, however, the symbols defined in one module may not be visible to another module. The details

---

[4] These guarantees don't hold when you use the "old" style calling convention — this is still found in much existing code.

of visibility depend on the operating system; some systems use one global namespace for the Python interpreter and all extension modules (e.g. Windows), whereas others require an explicit list of imported symbols at module link time (e.g. AIX), or offer a choice of different strategies (most Unices). And even if symbols are globally visible, the module whose functions one wishes to call might not have been loaded yet!

Portability therefore requires not to make any assumptions about symbol visibility. This means that all symbols in extension modules should be declared `static`, except for the module's initialization function, in order to avoid name clashes with other extension modules (as discussed in section 1.4). And it means that symbols that *should* be accessible from other extension modules must be exported in a different way.

Python provides a special mechanism to pass C-level information (i.e. pointers) from one extension module to another one: CObjects. A CObject is a Python data type which stores a pointer (`void *`). CObjects can only be created and accessed via their C API, but they can be passed around like any other Python object. In particular, they can be assigned to a name in an extension module's namespace. Other extension modules can then import this module, retrieve the value of this name, and then retrieve the pointer from the CObject.

There are many ways in which CObjects can be used to export the C API of an extension module. Each name could get its own CObject, or all C API pointers could be stored in an array whose address is published in a CObject. And the various tasks of storing and retrieving the pointers can be distributed in different ways between the module providing the code and the client modules.

The following example demonstrates an approach that puts most of the burden on the writer of the exporting module, which is appropriate for commonly used library modules. It stores all C API pointers (just one in the example!) in an array of `void` pointers which becomes the value of a CObject. The header file corresponding to the module provides a macro that takes care of importing the module and retrieving its C API pointers; client modules only have to call this macro before accessing the C API.

The exporting module is a modification of the `spam` module from section 1.1. The function `spam.system()` does not call the C library function `system()` directly, but a function `PySpam_System()`, which would of course do something more complicated in reality (such as adding "spam" to every command). This function `PySpam_System()` is also exported to other extension modules.

The function `PySpam_System()` is a plain C function, declared `static` like everything else:

```
static int
PySpam_System(command)
    char *command;
{
    return system(command);
}
```

The function `spam_system()` is modified in a trivial way:

```
static PyObject *
spam_system(self, args)
    PyObject *self;
    PyObject *args;
{
    char *command;
    int sts;

    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = PySpam_System(command);
    return Py_BuildValue("i", sts);
}
```

In the beginning of the module, right after the line

```
#include "Python.h"
```

two more lines must be added:

```
#define SPAM_MODULE
#include "spammodule.h"
```

The `#define` is used to tell the header file that it is being included in the exporting module, not a client module. Finally, the module's initialization function must take care of initializing the C API pointer array:

```
void
initspam()
{
    PyObject *m, *d;
    static void *PySpam_API[PySpam_API_pointers];
    PyObject *c_api_object;
    m = Py_InitModule("spam", SpamMethods);

    /* Initialize the C API pointer array */
    PySpam_API[PySpam_System_NUM] = (void *)PySpam_System;

    /* Create a CObject containing the API pointer array's address */
    c_api_object = PyCObject_FromVoidPtr((void *)PySpam_API, NULL);

    /* Create a name for this object in the module's namespace */
    d = PyModule_GetDict(m);
    PyDict_SetItemString(d, "_C_API", c_api_object);
}
```

Note that `PySpam_API` is declared `static`; otherwise the pointer array would disappear when `initspam` terminates!

The bulk of the work is in the header file 'spammodule.h', which looks like this:

```
#ifndef Py_SPAMMODULE_H
#define Py_SPAMMODULE_H
#ifdef __cplusplus
extern "C" {
#endif

/* Header file for spammodule */

/* C API functions */
#define PySpam_System_NUM 0
#define PySpam_System_RETURN int
#define PySpam_System_PROTO Py_PROTO((char *command))

/* Total number of C API pointers */
#define PySpam_API_pointers 1


#ifdef SPAM_MODULE
/* This section is used when compiling spammodule.c */

static PySpam_System_RETURN PySpam_System PySpam_System_PROTO;

#else
/* This section is used in modules that use spammodule's API */

static void **PySpam_API;

#define PySpam_System \
 (*(PySpam_System_RETURN (*)PySpam_System_PROTO) PySpam_API[PySpam_System_NUM])

#define import_spam() \
{ \
  PyObject *module = PyImport_ImportModule("spam"); \
  if (module != NULL) { \
    PyObject *module_dict = PyModule_GetDict(module); \
    PyObject *c_api_object = PyDict_GetItemString(module_dict, "_C_API"); \
    if (PyCObject_Check(c_api_object)) { \
      PySpam_API = (void **)PyCObject_AsVoidPtr(c_api_object); \
    } \
  } \
}

#endif

#ifdef __cplusplus
}
#endif

#endif /* !defined(Py_SPAMMODULE_H */
```

All that a client module must do in order to have access to the function `PySpam_System()` is to call the function (or rather macro) `import_spam()` in its initialization function:

```
void
initclient()
{
    PyObject *m;

    Py_InitModule("client", ClientMethods);
    import_spam();
}
```

The main disadvantage of this approach is that the file 'spammodule.h' is rather complicated. However, the basic structure is the same for each function that is exported, so it has to be learned only once.

Finally it should be mentioned that CObjects offer additional functionality, which is especially useful for memory allocation and deallocation of the pointer stored in a CObject. The details are described in the *Python/C API Reference Manual* in the section "CObjects" and in the implementation of CObjects (files 'Include/cobject.h' and 'Objects/cobject.c' in the Python source code distribution).

CHAPTER

TWO

Building C and C++ Extensions on UNIX

Starting in Python 1.4, Python provides a special make file for building make files for building dynamically-linked extensions and custom interpreters. The make file make file builds a make file that reflects various system variables determined by configure when the Python interpreter was built, so people building module's don't have to resupply these settings. This vastly simplifies the process of building extensions and custom interpreters on Unix systems.

The make file make file is distributed as the file 'Misc/Makefile.pre.in' in the Python source distribution. The first step in building extensions or custom interpreters is to copy this make file to a development directory containing extension module source.

The make file make file, 'Makefile.pre.in' uses metadata provided in a file named 'Setup'. The format of the 'Setup' file is the same as the 'Setup' (or 'Setup.in') file provided in the 'Modules/' directory of the Python source distribution. The 'Setup' file contains variable definitions:

```
EC=/projects/ExtensionClass
```

and module description lines. It can also contain blank lines and comment lines that start with '#'.

A module description line includes a module name, source files, options, variable references, and other input files, such as libraries or object files. Consider a simple example:

```
ExtensionClass ExtensionClass.c
```

This is the simplest form of a module definition line. It defines a module, `ExtensionClass`, which has a single source file, 'ExtensionClass.c'.

This slightly more complex example uses an **-I** option to specify an include directory:

```
EC=/projects/ExtensionClass
cPersistence cPersistence.c -I$(EC)
```

This example also illustrates the format for variable references.

For systems that support dynamic linking, the 'Setup' file should begin:

```
*shared*
```

to indicate that the modules defined in 'Setup' are to be built as dynamically linked modules. A line containing only '`*static*`' can be used to indicate the subsequently listed modules should be statically linked.

Here is a complete 'Setup' file for building a `cPersistent` module:

```
# Set-up file to build the cPersistence module.
# Note that the text should begin in the first column.
*shared*

# We need the path to the directory containing the ExtensionClass
# include file.
EC=/projects/ExtensionClass
cPersistence cPersistence.c -I$(EC)
```

After the '`Setup`' file has been created, '`Makefile.pre.in`' is run with the '`boot`' target to create a make file:

```
make -f Makefile.pre.in boot
```

This creates the file, Makefile. To build the extensions, simply run the created make file:

```
make
```

It's not necessary to re-run '`Makefile.pre.in`' if the '`Setup`' file is changed. The make file automatically rebuilds itself if the '`Setup`' file changes.

2.1 Building Custom Interpreters

The make file built by '`Makefile.pre.in`' can be run with the '`static`' target to build an interpreter:

```
make static
```

Any modules defined in the Setup file before the '`*shared*`' line will be statically linked into the interpreter. Typically, a '`*shared*`' line is omitted from the Setup file when a custom interpreter is desired.

2.2 Module Definition Options

Several compiler options are supported:

Option	**Meaning**
-C	Tell the C pre-processor not to discard comments
-D*name*=*value*	Define a macro
-I*dir*	Specify an include directory, *dir*
-L*dir*	Specify a link-time library directory, *dir*
-R*dir*	Specify a run-time library directory, *dir*
-l*lib*	Link a library, *lib*
-U*name*	Undefine a macro

Other compiler options can be included (snuck in) by putting them in variables.

Source files can include files with '`.c`', '`.C`', '`.cc`', '`.cpp`', '`.cxx`', and '`.c++`' extensions.

Other input files include files with '`.a`', '`.o`', '`.sl`', and '`.so`' extensions.

2.3 Example

Here is a more complicated example from 'Modules/Setup.in':

```
GMP=/ufs/guido/src/gmp
mpz mpzmodule.c -I$(GMP) $(GMP)/libgmp.a
```

which could also be written as:

```
mpz mpzmodule.c -I$(GMP) -L$(GMP) -lgmp
```

2.4 Distributing your extension modules

When distributing your extension modules in source form, make sure to include a 'Setup' file. The 'Setup' file should be named 'Setup.in' in the distribution. The make file make file, 'Makefile.pre.in', will copy 'Setup.in' to 'Setup'. Distributing a 'Setup.in' file makes it easy for people to customize the 'Setup' file while keeping the original in 'Setup.in'.

It is a good idea to include a copy of 'Makefile.pre.in' for people who do not have a source distribution of Python.

Do not distribute a make file. People building your modules should use 'Makefile.pre.in' to build their own make file. A 'README' file included in the package should provide simple instructions to perform the build.

Work is being done to make building and installing Python extensions easier for all platforms; this work in likely to supplant the current approach at some point in the future. For more information or to participate in the effort, refer to http://www.python.org/sigs/distutils-sig/ on the Python Web site.

CHAPTER

THREE

Building C and C++ Extensions on Windows

This chapter briefly explains how to create a Windows extension module for Python using Microsoft Visual C++, and follows with more detailed background information on how it works. The explanatory material is useful for both the Windows programmer learning to build Python extensions and the UNIX programmer interested in producing software which can be successfully built on both UNIX and Windows.

3.1 A Cookbook Approach

This section provides a recipe for building a Python extension on Windows.

Grab the binary installer from http://www.python.org/ and install Python. The binary installer has all of the required header files except for 'config.h'.

Get the source distribution and extract it into a convenient location. Copy the 'config.h' from the 'PC/' directory into the 'include/' directory created by the installer.

Create a 'Setup' file for your extension module, as described in chapter 2.

Get David Ascher's 'compile.py' script from http://starship.python.net/crew/da/compile/. Run the script to create Microsoft Visual C++ project files.

Open the DSW file in Visual C++ and select **Build**.

If your module creates a new type, you may have trouble with this line:

```
PyObject_HEAD_INIT(&PyType_Type)
```

Change it to:

```
PyObject_HEAD_INIT(NULL)
```

and add the following to the module initialization function:

```
MyObject_Type.ob_type = &PyType_Type;
```

Refer to section 3 of the Python FAQ (http://www.python.org/doc/FAQ.html) for details on why you must do this.

3.2 Differences Between UNIX and Windows

UNIX and Windows use completely different paradigms for run-time loading of code. Before you try to build a module that can be dynamically loaded, be aware of how your system works.

In UNIX, a shared object ('.so') file contains code to be used by the program, and also the names of functions and data that it expects to find in the program. When the file is joined to the program, all references to those functions and data in the file's code are changed to point to the actual locations in the program where the functions and data are placed in memory. This is basically a link operation.

In Windows, a dynamic-link library ('.dll') file has no dangling references. Instead, an access to functions or data goes through a lookup table. So the DLL code does not have to be fixed up at runtime to refer to the program's memory; instead, the code already uses the DLL's lookup table, and the lookup table is modified at runtime to point to the functions and data.

In UNIX, there is only one type of library file ('.a') which contains code from several object files ('.o'). During the link step to create a shared object file ('.so'), the linker may find that it doesn't know where an identifier is defined. The linker will look for it in the object files in the libraries; if it finds it, it will include all the code from that object file.

In Windows, there are two types of library, a static library and an import library (both called '.lib'). A static library is like a UNIX '.a' file; it contains code to be included as necessary. An import library is basically used only to reassure the linker that a certain identifier is legal, and will be present in the program when the DLL is loaded. So the linker uses the information from the import library to build the lookup table for using identifiers that are not included in the DLL. When an application or a DLL is linked, an import library may be generated, which will need to be used for all future DLLs that depend on the symbols in the application or DLL.

Suppose you are building two dynamic-load modules, B and C, which should share another block of code A. On UNIX, you would *not* pass 'A.a' to the linker for 'B.so' and 'C.so'; that would cause it to be included twice, so that B and C would each have their own copy. In Windows, building 'A.dll' will also build 'A.lib'. You *do* pass 'A.lib' to the linker for B and C. 'A.lib' does not contain code; it just contains information which will be used at runtime to access A's code.

In Windows, using an import library is sort of like using '`import spam`'; it gives you access to spam's names, but does not create a separate copy. On UNIX, linking with a library is more like '`from spam import *`'; it does create a separate copy.

3.3 Using DLLs in Practice

Windows Python is built in Microsoft Visual C++; using other compilers may or may not work (though Borland seems to). The rest of this section is MSVC++ specific.

When creating DLLs in Windows, you must pass 'python15.lib' to the linker. To build two DLLs, spam and ni (which uses C functions found in spam), you could use these commands:

```
cl /LD /I/python/include spam.c ../libs/python15.lib
cl /LD /I/python/include ni.c spam.lib ../libs/python15.lib
```

The first command created three files: 'spam.obj', 'spam.dll' and 'spam.lib'. 'Spam.dll' does not contain any Python functions (such as `PyArg_ParseTuple()`), but it does know how to find the Python code thanks to 'python15.lib'.

The second command created 'ni.dll' (and '.obj' and '.lib'), which knows how to find the necessary functions from spam, and also from the Python executable.

Not every identifier is exported to the lookup table. If you want any other modules (including Python) to be able to see your identifiers, you have to say '`_declspec(dllexport)`', as in '`void _declspec(dllexport) initspam(void)`' or '`PyObject _declspec(dllexport) *NiGetSpamData(void)`'.

Developer Studio will throw in a lot of import libraries that you do not really need, adding about 100K to your executable. To get rid of them, use the Project Settings dialog, Link tab, to specify *ignore default libraries*. Add the correct 'msvcrt*xx*.lib' to the list of libraries.

CHAPTER

FOUR

Embedding Python in Another Application

Embedding Python is similar to extending it, but not quite. The difference is that when you extend Python, the main program of the application is still the Python interpreter, while if you embed Python, the main program may have nothing to do with Python — instead, some parts of the application occasionally call the Python interpreter to run some Python code.

So if you are embedding Python, you are providing your own main program. One of the things this main program has to do is initialize the Python interpreter. At the very least, you have to call the function `Py_Initialize()` (on MacOS, call `PyMac_Initialize()` instead). There are optional calls to pass command line arguments to Python. Then later you can call the interpreter from any part of the application.

There are several different ways to call the interpreter: you can pass a string containing Python statements to `PyRun_SimpleString()`, or you can pass a stdio file pointer and a file name (for identification in error messages only) to `PyRun_SimpleFile()`. You can also call the lower-level operations described in the previous chapters to construct and use Python objects.

A simple demo of embedding Python can be found in the directory 'Demo/embed/' of the source distribution.

4.1 Embedding Python in C++

It is also possible to embed Python in a C++ program; precisely how this is done will depend on the details of the C++ system used; in general you will need to write the main program in C++, and use the C++ compiler to compile and link your program. There is no need to recompile Python itself using C++.