Extending and Embedding the Python Interpreter

Release 2.2.2

Guido van Rossum
Fred L. Drake, Jr., editor

October 14, 2002

PythonLabs
Email: python-docs@python.org

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

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). Since Python may define some pre-processor definitions which affect the standard headers on some systems, you must include 'Python.h' before any standard headers are included.

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 (for example, 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 (usually 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 (possibly by trying something else, or pretending nothing went wrong).

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, such as 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:

```
static PyObject *SpamError;
```

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

```
void
initspam(void)
{
    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 a class with the base class being Exception (unless another class is passed in instead of NULL), described in the *Python Library Reference* under "Built-in Exceptions."

Note also that the SpamError variable retains a reference to the newly created exception class; this is intentional! Since the exception could be removed from the module by external code, an owned reference to the class is needed to ensure that it will not be discarded, causing SpamError to become a dangling pointer. Should it become a dangling pointer, C code which raises the exception could cause a core dump or other unintended side effects.

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,
     "Execute a shell command."},
    ...
    {NULL, NULL, 0, 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. The initialization function must be named `init`*name*`()`, where *name* is the name of the module, and should be the only non-`static` item defined in the module file:

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

Note that for C++, this method must be declared `extern "C"`.

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();
```

An 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. Note also that the `reload()` function can be used with extension modules, and will call the module initialization function (`initspam()` in the example), but will not load the module again if it was loaded from a dynamically loadable object file ('.so' on UNIX, '.dll' on Windows).

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 may depend on the style of dynamic loading your system uses; see the chapters about building extension modules (chapter 3) and additional information that pertains only to building on Windows (chapter 4) 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 on UNIX: 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, "Extracting Parameters in Extension Functions."

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 Extracting Parameters in Extension Functions

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. On success, `PyArg_ParseTuple()` returns true, otherwise it returns false and raises an appropriate exception.

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 or Unicode object) [char *] Convert a Python string or Unicode object 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. Unicode objects are converted to C strings using the default encoding. If this conversion fails, an `UnicodeError` is raised.

'`s#`' (string, Unicode or any read buffer compatible object) [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. Unicode objects pass back a pointer to the default encoded string version of the object if such a conversion is possible. All other read buffer compatible objects pass back a reference to the raw internal data representation.

'`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` or any read buffer compatible object) [char *, int] This is to '`s#`' as '`z`' is to '`s`'.

'`u`' (Unicode object) [Py_UNICODE *] Convert a Python Unicode object to a C pointer to a null-terminated buffer of 16-bit Unicode (UTF-16) data. As with '`s`', there is no need to provide storage for the Unicode data buffer; a pointer to the existing Unicode data is stored into the Py_UNICODE pointer variable whose address you pass.

'`u#`' (Unicode object) [Py_UNICODE *, int] This variant on '`u`' stores into two C variables, the first one a pointer to a Unicode data buffer, the second one its length.

'`es`' (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer] This variant on '`s`' is used for encoding Unicode and objects convertible to Unicode into a character buffer. It only works for encoded data without embedded `NULL` bytes.

The variant reads one C variable and stores into two C variables, the first one a pointer to an encoding name string (*encoding*), and the second a pointer to a pointer to a character buffer (***buffer*, the buffer used for storing the encoded data).

The encoding name must map to a registered codec. If set to `NULL`, the default encoding is used.

`PyArg_ParseTuple()` will allocate a buffer of the needed size using `PyMem_NEW()`, copy the encoded data into this buffer and adjust **buffer* to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after usage.

'`et`' (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer]
Same as '`es`' except that string objects are passed through without recoding them. Instead, the implementation assumes that the string object uses the encoding passed in as parameter.

'`es#`' (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer, int *buffer_length]
This variant on '`s#`' is used for encoding Unicode and objects convertible to Unicode into a character buffer. It reads one C variable and stores into three C variables, the first one a pointer to an encoding name string (*encoding*), the second a pointer to a pointer to a character buffer (***buffer*, the buffer used for storing the encoded data) and the third one a pointer to an integer (**buffer_length*, the buffer length).

The encoding name must map to a registered codec. If set to `NULL`, the default encoding is used.

There are two modes of operation:

If **buffer* points a `NULL` pointer, `PyArg_ParseTuple()` will allocate a buffer of the needed size using `PyMem_NEW()`, copy the encoded data into this buffer and adjust **buffer* to reference the newly allocated storage. The caller is responsible for calling `PyMem_Free()` to free the allocated buffer after usage.

If **buffer* points to a non-`NULL` pointer (an already allocated buffer), `PyArg_ParseTuple()` will use this location as buffer and interpret **buffer_length* as buffer size. It will then copy the encoded data into the buffer and 0-terminate it. Buffer overflow is signalled with an exception.

In both cases, **buffer_length* is set to the length of the encoded data without the trailing 0-byte.

'`et#`' (string, Unicode object or character buffer compatible object) [const char *encoding, char **buffer]
Same as '`es#`' except that string objects are passed through without recoding them. Instead, the implementation assumes that the string object uses the encoding passed in as parameter.

'`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`.

'`L`' (integer) [LONG_LONG] Convert a Python integer to a C `long long`. This format is only available on platforms that support `long long` (or `_int64` on Windows).

'`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_ParseTuple()`. 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 *`.

'U' (Unicode string) [PyUnicodeObject *] Like '`O`' but requires that the Python object is a Unicode object. Raises `TypeError` if the object is not a Unicode object. The C variable may also be declared as `PyObject *`.

't#' (read-only character buffer) [char *, int] Like '`s#`', but accepts any object which implements the read-only buffer interface. The `char *` variable is set to point to the first byte of the buffer, and the `int` is set to the length of the buffer. Only single-segment buffer objects are accepted; `TypeError` is raised for all others.

'w' (read-write character buffer) [char *] Similar to '`s`', but accepts any object which implements the read-write buffer interface. The caller must determine the length of the buffer by other means, or use '`w#`' instead. Only single-segment buffer objects are accepted; `TypeError` is raised for all others.

'w#' (read-write character buffer) [char *, int] Like '`s#`', but accepts any object which implements the read-write buffer interface. The `char *` variable is set to point to the first byte of the buffer, and the `int` is set to the length of the buffer. Only single-segment buffer objects are accepted; `TypeError` is raised for all others.

'(*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 semicolon 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 Parameters for Extension Functions

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. On success, `PyArg_ParseTupleAndKeywords()` returns true, otherwise it returns false and raises an appropriate exception.

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 "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,
```

```
        "Print a lovely skit to standard output."},
       {NULL, NULL, 0, NULL}   /* sentinel */
};

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

1.9 Building Arbitrary Values

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.

When memory buffers are passed as parameters to supply data to build objects, as for the '`s`' and '`s#`' formats, the required data is copied. Buffers provided by the caller are never referenced by the objects created by `Py_BuildValue()`. In other words, if your code invokes `malloc()` and passes the allocated memory to `Py_BuildValue()`, your code is responsible for calling `free()` for that memory once `Py_BuildValue()` returns.

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 used.

'`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#`'.

'`u`' (Unicode string) [Py_UNICODE *] Convert a null-terminated buffer of Unicode (UCS-2) data to a Python Unicode object. If the Unicode buffer pointer is `NULL`, `None` is returned.

'`u#`' (Unicode string) [Py_UNICODE *, int] Convert a Unicode (UCS-2) data buffer and its length to a Python Unicode object. If the Unicode buffer pointer is `NULL`, the length is ignored and `None` is returned.

'`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`'.

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

'`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`'.

'`U`' (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.

While Python uses the traditional reference counting implementation, it also offers a cycle detector that works to detect reference cycles. This allows applications to not worry about creating direct or indirect circular references; these are the weakness of garbage collection implemented using only reference counting. Reference cycles consist of objects which contain (possibly indirect) references to themselves, so that each object in the cycle has a reference count which is non-zero. Typical reference counting implementations are not able to reclaim the memory belonging to any objects in a reference cycle, or referenced from the objects in the cycle, even though there are no further references to the cycle itself.

The cycle detector is able to detect garbage cycles and can reclaim them so long as there are no finalizers implemented in Python (`__del__()` methods). When there are such finalizers, the detector exposes the cycles through the `gc` module (specifically, the `garbage` variable in that module). The `gc` module also exposes a way to run the detector (the `collect()` function), as well as configuration interfaces and the ability to disable the detector at runtime. The cycle detector is considered an optional component; though it is included by default, it can be disabled at build time using the **--without-cycle-gc** option to the **configure** script on UNIX platforms (including Mac OS X) or by removing the definition of `WITH_CYCLE_GC` in the 'pyconfig.h' header on other platforms. If the cycle detector is disabled in this way, the `gc` module will not be available.

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 (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, such as `PyInt_FromLong()` and `Py_BuildValue()`, pass ownership to the receiver. Even if the object is not actually new, you still receive ownership of a new reference to that object. 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 owned reference by calling `Py_INCREF()`.

[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!

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 processor 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 more slowly.

It is better to test for `NULL` only at the "source:" when a pointer that may be `NULL` is received, for example, 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 of visibility depend on the operating system; some systems use one global namespace for the Python interpreter and all extension modules (Windows, for example), whereas others require an explicit list of imported symbols at module link time (AIX is one example), 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!

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

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 (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(void)
{
    PyObject *m;
    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);

    if (c_api_object != NULL) {
        /* Create a name for this object in the module's namespace */
        PyObject *d = PyModule_GetDict(m);

        PyDict_SetItemString(d, "_C_API", c_api_object);
        Py_DECREF(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 (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(void)
{
    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

Defining New Types

As mentioned in the last chapter, Python allows the writer of an extension module to define new types that can be manipulated from Python code, much like strings and lists in core Python.

This is not hard; the code for all extension types follows a pattern, but there are some details that you need to understand before you can get started.

2.1 The Basics

The Python runtime sees all Python objects as variables of type `PyObject*`. A `PyObject` is not a very magnificent object - it just contains the refcount and a pointer to the object's "type object". This is where the action is; the type object determines which (C) functions get called when, for instance, an attribute gets looked up on an object or it is multiplied by another object. I call these C functions "type methods" to distinguish them from things like `[].append` (which I will call "object methods" when I get around to them).

So, if you want to define a new object type, you need to create a new type object.

This sort of thing can only be explained by example, so here's a minimal, but complete, module that defines a new type:

```
#include <Python.h>

staticforward PyTypeObject noddy_NoddyType;

typedef struct {
    PyObject_HEAD
} noddy_NoddyObject;

static PyObject*
noddy_new_noddy(PyObject* self, PyObject* args)
{
    noddy_NoddyObject* noddy;

    if (!PyArg_ParseTuple(args,":new_noddy"))
        return NULL;

    noddy = PyObject_New(noddy_NoddyObject, &noddy_NoddyType);

    return (PyObject*)noddy;
}

static void
```

```
noddy_noddy_dealloc(PyObject* self)
{
    PyObject_Del(self);
}

static PyTypeObject noddy_NoddyType = {
    PyObject_HEAD_INIT(NULL)
    0,
    "Noddy",
    sizeof(noddy_NoddyObject),
    0,
    noddy_noddy_dealloc, /*tp_dealloc*/
    0,          /*tp_print*/
    0,          /*tp_getattr*/
    0,          /*tp_setattr*/
    0,          /*tp_compare*/
    0,          /*tp_repr*/
    0,          /*tp_as_number*/
    0,          /*tp_as_sequence*/
    0,          /*tp_as_mapping*/
    0,          /*tp_hash */
};

static PyMethodDef noddy_methods[] = {
    {"new_noddy", noddy_new_noddy, METH_VARARGS,
     "Create a new Noddy object."},
    {NULL, NULL, 0, NULL}
};

DL_EXPORT(void)
initnoddy(void)
{
    noddy_NoddyType.ob_type = &PyType_Type;

    Py_InitModule("noddy", noddy_methods);
}
```

Now that's quite a bit to take in at once, but hopefully bits will seem familiar from the last chapter.

The first bit that will be new is:

```
staticforward PyTypeObject noddy_NoddyType;
```

This names the type object that will be defining further down in the file. It can't be defined here because its definition has to refer to functions that have no yet been defined, but we need to be able to refer to it, hence the declaration.

The `staticforward` is required to placate various brain dead compilers.

```
typedef struct {
    PyObject_HEAD
} noddy_NoddyObject;
```

This is what a Noddy object will contain. In this case nothing more than every Python object contains - a refcount and a pointer to a type object. These are the fields the `PyObject_HEAD` macro brings in. The reason for the macro is to standardize the layout and to enable special debugging fields to be brought in debug builds.

For contrast

```
typedef struct {
    PyObject_HEAD
    long ob_ival;
} PyIntObject;
```

is the corresponding definition for standard Python integers.

Next up is:

```
static PyObject*
noddy_new_noddy(PyObject* self, PyObject* args)
{
    noddy_NoddyObject* noddy;

    if (!PyArg_ParseTuple(args,":new_noddy"))
        return NULL;

    noddy = PyObject_New(noddy_NoddyObject, &noddy_NoddyType);

    return (PyObject*)noddy;
}
```

This is in fact just a regular module function, as described in the last chapter. The reason it gets special mention is that this is where we create our Noddy object. Defining PyTypeObject structures is all very well, but if there's no way to actually *create* one of the wretched things it is not going to do anyone much good.

Almost always, you create objects with a call of the form:

```
PyObject_New(<type>, &<type object>);
```

This allocates the memory and then initializes the object (sets the reference count to one, makes the `ob_type` pointer point at the right place and maybe some other stuff, depending on build options). You *can* do these steps separately if you have some reason to — but at this level we don't bother.

We cast the return value to a `PyObject*` because that's what the Python runtime expects. This is safe because of guarantees about the layout of structures in the C standard, and is a fairly common C programming trick. One could declare `noddy_new_noddy` to return a `noddy_NoddyObject*` and then put a cast in the definition of `noddy_methods` further down the file — it doesn't make much difference.

Now a Noddy object doesn't do very much and so doesn't need to implement many type methods. One you can't avoid is handling deallocation, so we find

```
static void
noddy_noddy_dealloc(PyObject* self)
{
    PyObject_Del(self);
}
```

This is so short as to be self explanatory. This function will be called when the reference count on a Noddy object reaches `0` (or it is found as part of an unreachable cycle by the cyclic garbage collector). `PyObject_Del()` is what you call when you want an object to go away. If a Noddy object held references to other Python objects, one would decref them here.

Moving on, we come to the crunch — the type object.

```
static PyTypeObject noddy_NoddyType = {
    PyObject_HEAD_INIT(NULL)
```

```
        0,
        "Noddy",
        sizeof(noddy_NoddyObject),
        0,
        noddy_noddy_dealloc, /*tp_dealloc*/
        0,                   /*tp_print*/
        0,                   /*tp_getattr*/
        0,                   /*tp_setattr*/
        0,                   /*tp_compare*/
        0,                   /*tp_repr*/
        0,                   /*tp_as_number*/
        0,                   /*tp_as_sequence*/
        0,                   /*tp_as_mapping*/
        0,                   /*tp_hash */
    };
```

Now if you go and look up the definition of `PyTypeObject` in 'object.h' you'll see that it has many, many more fields that the definition above. The remaining fields will be filled with zeros by the C compiler, and it's common practice to not specify them explicitly unless you need them.

This is so important that I'm going to pick the top of it apart still further:

```
        PyObject_HEAD_INIT(NULL)
```

This line is a bit of a wart; what we'd like to write is:

```
        PyObject_HEAD_INIT(&PyType_Type)
```

as the type of a type object is "type", but this isn't strictly conforming C and some compilers complain. So instead we fill in the `ob_type` field of `noddy_NoddyType` at the earliest oppourtunity — in `initnoddy()`.

```
        0,
```

XXX why does the type info struct start PyObject_*VAR*_HEAD??

```
        "Noddy",
```

The name of our type. This will appear in the default textual representation of our objects and in some error messages, for example:

```
    >>> "" + noddy.new_noddy()
    Traceback (most recent call last):
      File "<stdin>", line 1, in ?
    TypeError: cannot add type "Noddy" to string
```

```
        sizeof(noddy_NoddyObject),
```

This is so that Python knows how much memory to allocate when you call `PyObject_New`.

```
        0,
```

This has to do with variable length objects like lists and strings. Ignore for now...

Now we get into the type methods, the things that make your objects different from the others. Of course, the Noddy object doesn't implement many of these, but as mentioned above you have to implement the deallocation function.

```
        noddy_noddy_dealloc, /*tp_dealloc*/
```

From here, all the type methods are nil so I won't go over them yet - that's for the next section!

Everything else in the file should be familiar, except for this line in `initnoddy`:

```
noddy_NoddyType.ob_type = &PyType_Type;
```

This was alluded to above — the `noddy_NoddyType` object should have type "type", but `&PyType_Type` is not constant and so can't be used in its initializer. To work around this, we patch it up in the module initialization.

That's it! All that remains is to build it; put the above code in a file called 'noddymodule.c' and

```
from distutils.core import setup, Extension
setup(name = "noddy", version = "1.0",
    ext_modules = [Extension("noddy", ["noddymodule.c"])])
```

in a file called 'setup.py'; then typing

```
$ python setup.py build%$
```

at a shell should produce a file 'noddy.so' in a subdirectory; move to that directory and fire up Python — you should be able to `import noddy` and play around with Noddy objects.

That wasn't so hard, was it?

2.2 Type Methods

This section aims to give a quick fly-by on the various type methods you can implement and what they do.

Here is the definition of `PyTypeObject`, with some fields only used in debug builds omitted:

```
typedef struct _typeobject {
    PyObject_VAR_HEAD
    char *tp_name; /* For printing */
    int tp_basicsize, tp_itemsize; /* For allocation */

    /* Methods to implement standard operations */

    destructor tp_dealloc;
    printfunc tp_print;
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
    cmpfunc tp_compare;
    reprfunc tp_repr;

    /* Method suites for standard classes */

    PyNumberMethods *tp_as_number;
    PySequenceMethods *tp_as_sequence;
    PyMappingMethods *tp_as_mapping;

    /* More standard operations (here for binary compatibility) */

    hashfunc tp_hash;
    ternaryfunc tp_call;
    reprfunc tp_str;
    getattrofunc tp_getattro;
    setattrofunc tp_setattro;

    /* Functions to access object as input/output buffer */
    PyBufferProcs *tp_as_buffer;
```

```
    /* Flags to define presence of optional/expanded features */
    long tp_flags;

    char *tp_doc; /* Documentation string */

    /* Assigned meaning in release 2.0 */
    /* call function for all accessible objects */
    traverseproc tp_traverse;

    /* delete references to contained objects */
    inquiry tp_clear;

    /* Assigned meaning in release 2.1 */
    /* rich comparisons */
    richcmpfunc tp_richcompare;

    /* weak reference enabler */
    long tp_weaklistoffset;

    /* Added in release 2.2 */
    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;

    /* Attribute descriptor and subclassing stuff */
    struct PyMethodDef *tp_methods;
    struct memberlist *tp_members;
    struct getsetlist *tp_getset;
    struct _typeobject *tp_base;
    PyObject *tp_dict;
    descrgetfunc tp_descr_get;
    descrsetfunc tp_descr_set;
    long tp_dictoffset;
    initproc tp_init;
    allocfunc tp_alloc;
    newfunc tp_new;
    destructor tp_free; /* Low-level free-memory routine */
    PyObject *tp_bases;
    PyObject *tp_mro; /* method resolution order */
    PyObject *tp_defined;

} PyTypeObject;
```

Now that's a *lot* of methods. Don't worry too much though - if you have a type you want to define, the chances are very good that you will only implement a handful of these.

As you probably expect by now, we're going to go over this and give more information about the various handlers. We won't go in the order they are defined in the structure, because there is a lot of historical baggage that impacts the ordering of the fields; be sure your type initializaion keeps the fields in the right order! It's often easiest to find an example that includes all the fields you need (even if they're initialized to `0`) and then change the values to suit your new type.

```
    char *tp_name; /* For printing */
```

The name of the type - as mentioned in the last section, this will appear in various places, almost entirely for diagnostic purposes. Try to choose something that will be helpful in such a situation!

```
int tp_basicsize, tp_itemsize; /* For allocation */
```

These fields tell the runtime how much memory to allocate when new objects of this typed are created. Python has some builtin support for variable length structures (think: strings, lists) which is where the `tp_itemsize` field comes in. This will be dealt with later.

```
char *tp_doc;
```

Here you can put a string (or its address) that you want returned when the Python script references `obj.__doc__` to retrieve the docstring.

Now we come to the basic type methods—the ones most extension types will implement.

2.2.1 Finalization and De-allocation

```
destructor tp_dealloc;
```

This function is called when the reference count of the instance of your type is reduced to zero and the Python interpreter wants to reclaim it. If your type has memory to free or other clean-up to perform, put it here. The object itself needs to be freed here as well. Here is an example of this function:

```
static void
newdatatype_dealloc(newdatatypeobject * obj)
{
    free(obj->obj_UnderlyingDatatypePtr);
    PyObject_DEL(obj);
}
```

One important requirement of the deallocator function is that it leaves any pending exceptions alone. This is important since deallocators are frequently called as the interpreter unwinds the Python stack; when the stack is unwound due to an exception (rather than normal returns), nothing is done to protect the deallocators from seeing that an exception has already been set. Any actions which a deallocator performs which may cause additional Python code to be executed may detect that an exception has been set. This can lead to misleading errors from the interpreter. The proper way to protect against this is to save a pending exception before performing the unsafe action, and restoring it when done. This can be done using the `PyErr_Fetch()` and `PyErr_Restore()` functions:

```
static void
my_dealloc(PyObject *obj)
{
    MyObject *self = (MyObject *) obj;
    PyObject *cbresult;

    if (self->my_callback != NULL) {
        PyObject *err_type, *err_value, *err_traceback;
        int have_error = PyErr_Occurred() ? 1 : 0;

        if (have_error)
            PyErr_Fetch(&err_type, &err_value, &err_traceback);

        cbresult = PyObject_CallObject(self->my_callback, NULL);
        if (cbresult == NULL)
            PyErr_WriteUnraisable();
        else
            Py_DECREF(cbresult);
```

```
        if (have_error)
            PyErr_Restore(err_type, err_value, err_traceback);

        Py_DECREF(self->my_callback);
    }
    PyObject_DEL(obj);
}
```

2.2.2 Object Representation

In Python, there are three ways to generate a textual representation of an object: the `repr()` function (or equivalent backtick syntax), the `str()` function, and the `print` statement. For most objects, the `print` statement is equivalent to the `str()` function, but it is possible to special-case printing to a `FILE*` if necessary; this should only be done if efficiency is identified as a problem and profiling suggests that creating a temporary string object to be written to a file is too expensive.

These handlers are all optional, and most types at most need to implement the `tp_str` and `tp_repr` handlers.

```
    reprfunc tp_repr;
    reprfunc tp_str;
    printfunc tp_print;
```

The `tp_repr` handler should return a string object containing a representation of the instance for which it is called. Here is a simple example:

```
static PyObject *
newdatatype_repr(newdatatypeobject * obj)
{
    return PyString_FromFormat("Repr-ified_newdatatype{{size:\%d}}",
            obj->obj_UnderlyingDatatypePtr->size);
}
```

If no `tp_repr` handler is specified, the interpreter will supply a representation that uses the type's `tp_name` and a uniquely-identifying value for the object.

The `tp_str` handler is to `str()` what the `tp_repr` handler described above is to `repr()`; that is, it is called when Python code calls `str()` on an instance of your object. It's implementation is very similar to the `tp_repr` function, but the resulting string is intended for human consumption. It `tp_str` is not specified, the `tp_repr` handler is used instead.

Here is a simple example:

```
static PyObject *
newdatatype_str(newdatatypeobject * obj)
{
    return PyString_FromFormat("Stringified_newdatatype{{size:\%d}}",
        obj->obj_UnderlyingDatatypePtr->size
        );
}
```

The print function will be called whenever Python needs to "print" an instance of the type. For example, if 'node' is an instance of type TreeNode, then the print function is called when Python code calls:

```
print node
```

There is a flags argument and one flag, `Py_PRINT_RAW`, and it suggests that you print without string quotes and possibly without interpreting escape sequences.

The print function receives a file object as an argument. You will likely want to write to that file object.

Here is a sampe print function:

```
static int
newdatatype_print(newdatatypeobject *obj, FILE *fp, int flags)
{
    if (flags & Py_PRINT_RAW) {
        fprintf(fp, "<{newdatatype object--size: %d}>",
                obj->obj_UnderlyingDatatypePtr->size);
    }
    else {
        fprintf(fp, "\"<{newdatatype object--size: %d}>\"",
                obj->obj_UnderlyingDatatypePtr->size);
    }
    return 0;
}
```

2.2.3 Attribute Management Functions

```
    getattrfunc tp_getattr;
    setattrfunc tp_setattr;
```

The `tp_getattr` handle is called when the object requires an attribute look-up. It is called in the same situations where the `__getattr__()` method of a class would be called.

A likely way to handle this is (1) to implement a set of functions (such as `newdatatype_getSize()` and `newdatatype_setSize()` in the example below), (2) provide a method table listing these functions, and (3) provide a getattr function that returns the result of a lookup in that table.

Here is an example:

```
static PyMethodDef newdatatype_methods[] = {
    {"getSize", (PyCFunction)newdatatype_getSize, METH_VARARGS,
     "Return the current size."},
    {"setSize", (PyCFunction)newdatatype_setSize, METH_VARARGS,
     "Set the size."},
    {NULL, NULL, 0, NULL}           /* sentinel */
};

static PyObject *
newdatatype_getattr(newdatatypeobject *obj, char *name)
{
    return Py_FindMethod(newdatatype_methods, (PyObject *)obj, name);
}
```

The `tp_setattr` handler is called when the `__setattr__()` or `__delattr__()` method of a class instance would be called. When an attribute should be deleted, the third parameter will be `NULL`. Here is an example that simply raises an exception; if this were really all you wanted, the `tp_setattr` handler should be set to `NULL`.

```
static int
newdatatype_setattr(newdatatypeobject *obj, char *name, PyObject *v)
{
    (void)PyErr_Format(PyExc_RuntimeError, "Read-only attribute: \%s", name);
    return -1;
}
```

2.2.4 Object Comparison

```
cmpfunc tp_compare;
```

The `tp_compare` handler is called when comparisons are needed are the object does not implement the specific rich comparison method which matches the requested comparison. (It is always used if defined and the `PyObject_Compare()` or `PyObject_Cmp()` functions are used, or if `cmp()` is used from Python.) It is analogous to the `__cmp__()` method. This function should return `-1` if *obj1* is less than *obj2*, `0` if they are equal, and `1` if *obj1* is greater than *obj2*. (It was previously allowed to return arbitrary negative or positive integers for less than and greater than, respectively; as of Python 2.2, this is no longer allowed. In the future, other return values may be assigned a different meaning.)

A `tp_compare` handler may raise an exception. In this case it should return a negative value. The caller has to test for the exception using `PyErr_Occurred()`.

Here is a sample implementation:

```
static int
newdatatype_compare(newdatatypeobject * obj1, newdatatypeobject * obj2)
{
    long result;

    if (obj1->obj_UnderlyingDatatypePtr->size <
        obj2->obj_UnderlyingDatatypePtr->size) {
        result = -1;
    }
    else if (obj1->obj_UnderlyingDatatypePtr->size >
             obj2->obj_UnderlyingDatatypePtr->size) {
        result = 1;
    }
    else {
        result = 0;
    }
    return result;
}
```

2.2.5 Abstract Protocol Support

Python supports a variety of *abstract* 'protocols;' the specific interfaces provided to use these interfaces are documented in the *Python/C API Reference Manual* in the chapter "Abstract Objects Layer."

A number of these abstract interfaces were defined early in the development of the Python implementation. In particular, the number, mapping, and sequence protocols have been part of Python since the beginning. Other protocols have been added over time. For protocols which depend on several handler routines from the type implementation, the older protocols have been defined as optional blocks of handlers referenced by the type object, while newer protocols have been added using additional slots in the main type object, with a flag bit being set to indicate that the slots are present. (The flag bit does not indicate that the slot values are non-`NULL`.)

```
PyNumberMethods   tp_as_number;
PySequenceMethods tp_as_sequence;
PyMappingMethods  tp_as_mapping;
```

If you wish your object to be able to act like a number, a sequence, or a mapping object, then you place the address of a structure that implements the C type `PyNumberMethods`, `PySequenceMethods`, or `PyMappingMethods`, respectively. It is up to you to fill in this structure with appropriate values. You can find examples of the use of each of these in the 'Objects' directory of the Python source distribution.

```
hashfunc tp_hash;
```

This function, if you choose to provide it, should return a hash number for an instance of your datatype. Here is a moderately pointless example:

```
static long
newdatatype_hash(newdatatypeobject *obj)
{
    long result;
    result = obj->obj_UnderlyingDatatypePtr->size;
    result = result * 3;
    return result;
}
```

```
    ternaryfunc tp_call;
```

This function is called when an instance of your datatype is "called", for example, if `obj1` is an instance of your datatype and the Python script contains `obj1('hello')`, the `tp_call` handler is invoked.

This function takes three arguments:

1. *arg1* is the instance of the datatype which is the subject of the call. If the call is `obj1('hello')`, then *arg1* is `obj1`.
2. *arg2* is a tuple containing the arguments to the call. You can use `PyArg_ParseTuple()` to extract the arguments.
3. *arg3* is a dictionary of keyword arguments that were passed. If this is non-`NULL` and you support keyword arguments, use `PyArg_ParseTupleAndKeywords()` to extract the arguments. If you do not want to support keyword arguments and this is non-`NULL`, raise a `TypeError` with a message saying that keyword arguments are not supported.

Here is a desultory example of the implementation of the call function.

```
/* Implement the call function.
 *    obj1 is the instance receiving the call.
 *    obj2 is a tuple containing the arguments to the call, in this
 *         case 3 strings.
 */
static PyObject *
newdatatype_call(newdatatypeobject *obj, PyObject *args, PyObject *other)
{
    PyObject *result;
    char *arg1;
    char *arg2;
    char *arg3;

    if (!PyArg_ParseTuple(args, "sss:call", &arg1, &arg2, &arg3)) {
        return NULL;
    }
    result = PyString_FromFormat(
        "Returning -- value: [\%d] arg1: [\%s] arg2: [\%s] arg3: [\%s]\n",
        obj->obj_UnderlyingDatatypePtr->size,
        arg1, arg2, arg3);
    printf("\%s", PyString_AS_STRING(result));
    return result;
}
```

XXX some fields need to be added here...

```
    /* Added in release 2.2 */
    /* Iterators */
    getiterfunc tp_iter;
    iternextfunc tp_iternext;
```

These functions provide support for the iterator protocol. Any object which wishes to support iteration over it's contents (which may be generated during iteration) must implement the `tp_iter` handler. Objects which are returned by a `tp_iter` handler must implement both the `tp_iter` and `tp_iternext` handlers. Both handlers take exactly one parameter, the instance for which they are being called, and return a new reference. In the case of an error, they should set an exception and return `NULL`.

For an object which represents an iterable collection, the `tp_iter` handler must return an iterator object. The iterator object is responsible for maintaining the state of the iteration. For collections which can support multiple iterators which do not interfere with each other (as lists and tuples do), a new iterator should be created and returned. Objects which can only be iterated over once (usually due to side effects of iteration) should implement this handler by returning a new reference to themselves, and should also implement the `tp_iternext` handler. File objects are an example of such an iterator.

Iterator objects should implement both handlers. The `tp_iter` handler should return a new reference to the iterator (this is the same as the `tp_iter` handler for objects which can only be iterated over destructively). The `tp_iternext` handler should return a new reference to the next object in the iteration if there is one. If the iteration has reached the end, it may return `NULL` without setting an exception or it may set `StopIteration`; avoiding the exception can yield slightly better performance. If an actual error occurs, it should set an exception and return `NULL`.

2.2.6 More Suggestions

Remember that you can omit most of these functions, in which case you provide `0` as a value.

In the 'Objects' directory of the Python source distribution, there is a file 'xxobject.c', which is intended to be used as a template for the implementation of new types. One useful strategy for implementing a new type is to copy and rename this file, then read the instructions at the top of it.

There are type definitions for each of the functions you must provide. They are in 'object.h' in the Python include directory that comes with the source distribution of Python.

In order to learn how to implement any specific method for your new datatype, do the following: Download and unpack the Python source distribution. Go the the 'Objects' directory, then search the C source files for `tp_` plus the function you want (for example, `tp_print` or `tp_compare`). You will find examples of the function you want to implement.

When you need to verify that the type of an object is indeed the object you are implementing and if you use xxobject.c as an starting template for your implementation, then there is a macro defined for this purpose. The macro definition will look something like this:

```
#define is_newdatatypeobject(v)  ((v)->ob_type == &Newdatatypetype)
```

And, a sample of its use might be something like the following:

```
    if (!is_newdatatypeobject(objp1) {
        PyErr_SetString(PyExc_TypeError, "arg #1 not a newdatatype");
        return NULL;
    }
```

CHAPTER

THREE

Building C and C++ Extensions with distutils

Starting in Python 1.4, Python provides, on UNIX, a special make file for building make files for building dynamically-linked extensions and custom interpreters. Starting with Python 2.0, this mechanism (known as related to Makefile.pre.in, and Setup files) is no longer supported. Building custom interpreters was rarely used, and extension modules can be built using distutils.

Building an extension module using distutils requires that distutils is installed on the build machine, which is included in Python 2.x and available separately for Python 1.5. Since distutils also supports creation of binary packages, users don't necessarily need a compiler and distutils to install the extension.

A distutils package contains a driver script, 'setup.py'. This is a plain Python file, which, in the most simple case, could look like this:

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       ext_modules = [module1])
```

With this 'setup.py', and a file 'demo.c', running

```
python setup.py build
```

will compile 'demo.c', and produce an extension module named 'demo' in the 'build' directory. Depending on the system, the module file will end up in a subdirectory 'build/lib.system', and may have a name like 'demo.so' or 'demo.pyd'.

In the 'setup.py', all execution is performed by calling the 'setup' function. This takes a variable number of keyword arguments, of which the example above uses only a subset. Specifically, the example specifies meta-information to build packages, and it specifies the contents of the package. Normally, a package will contain of addition modules, like Python source modules, documentation, subpackages, etc. Please refer to the distutils documentation in *Distributing Python Modules* to learn more about the features of distutils; this section explains building extension modules only.

It is common to pre-compute arguments to `setup`, to better structure the driver script. In the example above, the'`ext_modules`' argument to `setup` is a list of extension modules, each of which is an instance of the `Extension`. In the example, the instance defines an extension named 'demo' which is build by compiling a single source file, 'demo.c'.

In many cases, building an extension is more complex, since additional preprocessor defines and libraries may be needed. This is demonstrated in the example below.

```
from distutils.core import setup, Extension

module1 = Extension('demo',
                    define_macros = [('MAJOR_VERSION', '1'),
                                     ('MINOR_VERSION', '0')],
                    include_dirs = ['/usr/local/include'],
                    libraries = ['tcl83'],
                    library_dirs = ['/usr/local/lib'],
                    sources = ['demo.c'])

setup (name = 'PackageName',
       version = '1.0',
       description = 'This is a demo package',
       author = 'Martin v. Loewis',
       author_email = 'martin@v.loewis.de',
       url = 'http://www.python.org/doc/current/ext/building.html',
       long_description = '''
This is really just a demo package.
''',
       ext_modules = [module1])
```

In this example, `setup` is called with additional meta-information, which is recommended when distribution packages have to be built. For the extension itself, it specifies preprocessor defines, include directories, library directories, and libraries. Depending on the compiler, distutils passes this information in different ways to the compiler. For example, on UNIX, this may result in the compilation commands

```
gcc -DNDEBUG -g -O3 -Wall -Wstrict-prototypes -fPIC -DMAJOR_VERSION=1 -DMINOR_VERSION=0 -I/usr/local/include -I/usr/local/include/python2.2 -c demo.c -o build/temp.linux-i686-2.2/demo.o

gcc -shared build/temp.linux-i686-2.2/demo.o -L/usr/local/lib -ltcl83 -o build/lib.linux-i686-2.2/demo.so
```

These lines are for demonstration purposes only; distutils users should trust that distutils gets the invocations right.

3.1 Distributing your extension modules

When an extension has been successfully build, there are three ways to use it.

End-users will typically want to install the module, they do so by running

```
python setup.py install
```

Module maintainers should produce source packages; to do so, they run

```
python setup.py sdist
```

In some cases, additional files need to be included in a source distribution; this is done through a 'MANIFEST.in' file; see the distutils documentation for details.

If the source distribution has been build successfully, maintainers can also create binary distributions. Depending on the platform, one of the following commands can be used to do so.

```
python setup.py bdist_wininst
python setup.py bdist_rpm
python setup.py bdist_dumb
```

CHAPTER

FOUR

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.

Module authors are encouraged to use the distutils approach for building extension modules, instead of the one described in this section. You will still need the C compiler that was used to build Python; typically Microsoft Visual C++.

Note: This chapter mentions a number of filenames that include an encoded Python version number. These filenames are represented with the version number shown as 'XY'; in practice, 'X' will be the major version number and 'Y' will be the minor version number of the Python release you're working with. For example, if you are using Python 2.2.1, 'XY' will actually be '22'.

4.1 A Cookbook Approach

There are two approaches to building extension modules on Windows, just as there are on UNIX: use the `distutils` package to control the build process, or do things manually. The distutils approach works well for most extensions; documentation on using `distutils` to build and package extension modules is available in *Distributing Python Modules*. This section describes the manual approach to building Python extensions written in C or C++.

To build extensions using these instructions, you need to have a copy of the Python sources of the same version as your installed Python. You will need Microsoft Visual C++ "Developer Studio"; project files are supplied for VC++ version 6, but you can use older versions of VC++. The example files described here are distributed with the Python sources in the 'PC\example_nt\' directory.

1. **Copy the example files**
   The 'example_nt' directory is a subdirectory of the 'PC' directory, in order to keep all the PC-specific files under the same directory in the source distribution. However, the 'example_nt' directory can't actually be used from this location. You first need to copy or move it up one level, so that 'example_nt' is a sibling of the 'PC' and 'Include' directories. Do all your work from within this new location.

2. **Open the project**
   From VC++, use the File > Open Workspace dialog (not File > Open!). Navigate to and select the file 'example.dsw', in the *copy* of the 'example_nt' directory you made above. Click Open.

3. **Build the example DLL**
   In order to check that everything is set up right, try building:

(a) Select a configuration. This step is optional. Choose Build > Select Active Configuration and select either "example - Win32 Release" or "example - Win32 Debug." If you skip this step, VC++ will use the Debug configuration by default.

(b) Build the DLL. Choose Build > Build example_d.dll in Debug mode, or Build > Build example.dll in Release mode. This creates all intermediate and result files in a subdirectory called either 'Debug' or 'Release', depending on which configuration you selected in the preceding step.

4. **Testing the debug-mode DLL**
Once the Debug build has succeeded, bring up a DOS box, and change to the 'example_nt\Debug' directory. You should now be able to repeat the following session (C> is the DOS prompt, >>> is the Python prompt; note that build information and various debug output from Python may not match this screen dump exactly):

```
C>..\..\PCbuild\python_d
Adding parser accelerators ...
Done.
Python 2.2 (#28, Dec 19 2001, 23:26:37) [MSC 32 bit (Intel)] on win32
Type "copyright", "credits" or "license" for more information.
>>> import example
[4897 refs]
>>> example.foo()
Hello, world
[4903 refs]
>>>
```

Congratulations! You've successfully built your first Python extension module.

5. **Creating your own project**
Choose a name and create a directory for it. Copy your C sources into it. Note that the module source file name does not necessarily have to match the module name, but the name of the initialization function should match the module name — you can only import a module `spam` if its initialization function is called `initspam()`, and it should call `Py_InitModule()` with the string `"spam"` as its first argument (use the minimal 'example.c' in this directory as a guide). By convention, it lives in a file called 'spam.c' or 'spammodule.c'. The output file should be called 'spam.dll' or 'spam.pyd' (the latter is supported to avoid confusion with a system library 'spam.dll' to which your module could be a Python interface) in Release mode, or 'spam_d.dll' or 'spam_d.pyd' in Debug mode.

Now your options are:

(a) Copy 'example.dsw' and 'example.dsp', rename them to 'spam.*', and edit them by hand, or

(b) Create a brand new project; instructions are below.

In either case, copy 'example_nt\example.def' to 'spam\spam.def', and edit the new 'spam.def' so its second line contains the string '`initspam`'. If you created a new project yourself, add the file 'spam.def' to the project now. (This is an annoying little file with only two lines. An alternative approach is to forget about the '.def' file, and add the option **/export:initspam** somewhere to the Link settings, by manually editing the setting in Project Options dialog).

6. **Creating a brand new project**
Use the File > New > Projects dialog to create a new Project Workspace. Select "Win32 Dynamic-Link Library," enter the name ('`spam`'), and make sure the Location is set to the 'spam' directory you have created (which should be a direct subdirectory of the Python build tree, a sibling of 'Include' and 'PC'). Select Win32 as the platform (in my version, this is the only choice). Make sure the Create new workspace radio button is selected. Click OK.

Now open the Project > Settings dialog. You only need to change a few settings. Make sure All Configurations is selected from the Settings for: dropdown list. Select the C/C++ tab. Choose the Preprocessor category in the popup menu at the top. Type the following text in the entry box labeled Addditional include directories:

```
..\Include,..\PC
```

Then, choose the Input category in the Link tab, and enter

```
..\PCbuild
```

in the text box labelled "Additional library path."

Now you need to add some mode-specific settings:

Select "Win32 Release" in the "Settings for" dropdown list. Click the Link tab, choose the Input Category, and append `pythonXY.lib` to the list in the "Object/library modules" box.

Select "Win32 Debug" in the "Settings for" dropdown list, and append `pythonXY_d.lib` to the list in the "Object/library modules" box. Then click the C/C++ tab, select "Code Generation" from the Category dropdown list, and select "Debug Multithreaded DLL" from the "Use run-time library" dropdown list.

Select "Win32 Release" again from the "Settings for" dropdown list. Select "Multithreaded DLL" from the "Use run-time library:" dropdown list.

You should now create the file spam.def as instructed in the previous section. Then chose the Insert > Files into Project dialog. Set the pattern to `*.*` and select both 'spam.c' and 'spam.def' and click OK. (Inserting them one by one is fine too.)

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* for details on why you must do this.

4.2 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.

4.3 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 'pythonXY.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/pythonXY.lib
cl /LD /I/python/include ni.c spam.lib ../libs/pythonXY.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 'pythonXY.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

FIVE

Embedding Python in Another Application

The previous chapters discussed how to extend Python, that is, how to extend the functionality of Python by attaching a library of C functions to it. It is also possible to do it the other way around: enrich your C/C++ application by embedding Python in it. Embedding provides your application with the ability to implement some of the functionality of your application in Python rather than C or C++. This can be used for many purposes; one example would be to allow users to tailor the application to their needs by writing some scripts in Python. You can also use it yourself if some of the functionality can be written in Python more easily.

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 Mac OS, 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.

See Also:

Python/C API Reference Manual
(../api/api.html)
The details of Python's C interface are given in this manual. A great deal of necessary information can be found here.

5.1 Very High Level Embedding

The simplest form of embedding Python is the use of the very high level interface. This interface is intended to execute a Python script without needing to interact with the application directly. This can for example be used to perform some operation on a file.

```
#include <Python.h>

int
main(int argc, char *argv[])
{
```

```
    Py_Initialize();
    PyRun_SimpleString("from time import time,ctime\n"
                       "print 'Today is',ctime(time())\n");
    Py_Finalize();
    return 0;
  }
```

The above code first initializes the Python interpreter with `Py_Initialize()`, followed by the execution of a hard-coded Python script that print the date and time. Afterwards, the `Py_Finalize()` call shuts the interpreter down, followed by the end of the program. In a real program, you may want to get the Python script from another source, perhaps a text-editor routine, a file, or a database. Getting the Python code from a file can better be done by using the `PyRun_SimpleFile()` function, which saves you the trouble of allocating memory space and loading the file contents.

5.2 Beyond Very High Level Embedding: An overview

The high level interface gives you the ability to execute arbitrary pieces of Python code from your application, but exchanging data values is quite cumbersome to say the least. If you want that, you should use lower level calls. At the cost of having to write more C code, you can achieve almost anything.

It should be noted that extending Python and embedding Python is quite the same activity, despite the different intent. Most topics discussed in the previous chapters are still valid. To show this, consider what the extension code from Python to C really does:

1. Convert data values from Python to C,
2. Perform a function call to a C routine using the converted values, and
3. Convert the data values from the call from C to Python.

When embedding Python, the interface code does:

1. Convert data values from C to Python,
2. Perform a function call to a Python interface routine using the converted values, and
3. Convert the data values from the call from Python to C.

As you can see, the data conversion steps are simply swapped to accomodate the different direction of the cross-language transfer. The only difference is the routine that you call between both data conversions. When extending, you call a C routine, when embedding, you call a Python routine.

This chapter will not discuss how to convert data from Python to C and vice versa. Also, proper use of references and dealing with errors is assumed to be understood. Since these aspects do not differ from extending the interpreter, you can refer to earlier chapters for the required information.

5.3 Pure Embedding

The first program aims to execute a function in a Python script. Like in the section about the very high level interface, the Python interpreter does not directly interact with the application (but that will change in th next section).

The code to run a function defined in a Python script is:

```
#include <Python.h>

int
main(int argc, char *argv[])
{
    PyObject *pName, *pModule, *pDict, *pFunc;
    PyObject *pArgs, *pValue;
    int i, result;

    if (argc < 3) {
        fprintf(stderr,"Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    Py_Initialize();
    pName = PyString_FromString(argv[1]);
    /* Error checking of pName left out */

    pModule = PyImport_Import(pName);
    if (pModule != NULL) {
        pDict = PyModule_GetDict(pModule);
        /* pDict is a borrowed reference */

        pFunc = PyDict_GetItemString(pDict, argv[2]);
        /* pFun: Borrowed reference */

        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyInt_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                /* pValue reference stolen here: */
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            if (pValue != NULL) {
                printf("Result of call: %ld\n", PyInt_AsLong(pValue));
                Py_DECREF(pValue);
            }
            else {
                PyErr_Print();
                fprintf(stderr,"Call failed\n");
                return 1;
            }
            Py_DECREF(pArgs);
            /* pDict and pFunc are borrowed and must not be Py_DECREF-ed */
        }
        else {
            PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
        }
        Py_DECREF(pModule);
    }
    else {
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
```

```
        return 1;
    }
    Py_DECREF(pName);
    Py_Finalize();
    return 0;
}
```

This code loads a Python script using `argv[1]`, and calls the function named in `argv[2]`. Its integer arguments are the other values of the `argv` array. If you compile and link this program (let's call the finished executable **call**), and use it to execute a Python script, such as:

```
def multiply(a,b):
    print "Thy shall add", a, "times", b
    c = 0
    for i in range(0, a):
        c = c + b
    return c
```

then the result should be:

```
$ call multiply 3 2
Thy shall add 3 times 2
Result of call: 6
```

Although the program is quite large for its functionality, most of the code is for data conversion between Python and C, and for error reporting. The interesting part with respect to embedding Python starts with

```
    Py_Initialize();
    pName = PyString_FromString(argv[1]);
    /* Error checking of pName left out */
    pModule = PyImport_Import(pName);
```

After initializing the interpreter, the script is loaded using `PyImport_Import()`. This routine needs a Python string as its argument, which is constructed using the `PyString_FromString()` data conversion routine.

```
    pDict = PyModule_GetDict(pModule);
    /* pDict is a borrowed reference */

    pFunc = PyDict_GetItemString(pDict, argv[2]);
    /* pFun is a borrowed reference */

    if (pFunc && PyCallable_Check(pFunc)) {
        ...
    }
```

Once the script is loaded, its dictionary is retrieved with `PyModule_GetDict()`. The dictionary is then searched using the normal dictionary access routines for the function name. If the name exists, and the object returned is callable, you can safely assume that it is a function. The program then proceeds by constructing a tuple of arguments as normal. The call to the python function is then made with:

```
    pValue = PyObject_CallObject(pFunc, pArgs);
```

Upon return of the function, `pValue` is either `NULL` or it contains a reference to the return value of the function. Be sure to release the reference after examining the value.

5.4 Extending Embedded Python

Until now, the embedded Python interpreter had no access to functionality from the application itself. The Python API allows this by extending the embedded interpreter. That is, the embedded interpreter gets extended with routines provided by the application. While it sounds complex, it is not so bad. Simply forget for a while that the application starts the Python interpreter. Instead, consider the application to be a set of subroutines, and write some glue code that gives Python access to those routines, just like you would write a normal Python extension. For example:

```
static int numargs=0;

/* Return the number of arguments of the application command line */
static PyObject*
emb_numargs(PyObject *self, PyObject *args)
{
    if(!PyArg_ParseTuple(args, ":numargs"))
        return NULL;
    return Py_BuildValue("i", numargs);
}

static PyMethodDef EmbMethods[] = {
    {"numargs", emb_numargs, METH_VARARGS,
     "Return the number of arguments received by the process."},
    {NULL, NULL, 0, NULL}
};
```

Insert the above code just above the `main()` function. Also, insert the following two statements directly after `Py_Initialize()`:

```
    numargs = argc;
    Py_InitModule("emb", EmbMethods);
```

These two lines initialize the `numargs` variable, and make the `emb.numargs()` function accessible to the embedded Python interpreter. With these extensions, the Python script can do things like

```
import emb
print "Number of arguments", emb.numargs()
```

In a real application, the methods will expose an API of the application to Python.

5.5 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++.

5.6 Linking Requirements

While the **configure** script shipped with the Python sources will correctly build Python to export the symbols needed by dynamically linked extensions, this is not automatically inherited by applications which embed the Python library statically, at least on UNIX. This is an issue when the application is linked to the static runtime library ('libpython.a') and needs to load dynamic extensions (implemented as '.so' files).

The problem is that some entry points are defined by the Python runtime solely for extension modules to use. If the embedding application does not use any of these entry points, some linkers will not include those entries in the symbol table of the finished executable. Some additional options are needed to inform the linker not to remove these symbols.

Determining the right options to use for any given platform can be quite difficult, but fortunately the Python configuration already has those values. To retrieve them from an installed Python interpreter, start an interactive interpreter and have a short session like this:

```
>>> import distutils.sysconfig
>>> distutils.sysconfig.get_config_var('LINKFORSHARED')
'-Xlinker -export-dynamic'
```

The contents of the string presented will be the options that should be used. If the string is empty, there's no need to add any additional options. The `LINKFORSHARED` definition corresponds to the variable of the same name in Python's top-level 'Makefile'.

APPENDIX

A

Reporting Bugs

Python is a mature programming language which has established a reputation for stability. In order to maintain this reputation, the developers would like to know of any deficiencies you find in Python or its documentation.

Before submitting a report, you will be required to log into SourceForge; this will make it possible for the developers to contact you for additional information if needed. It is not possible to submit a bug report anonymously.

All bug reports should be submitted via the Python Bug Tracker on SourceForge (http://sourceforge.net/bugs/?group_id=5470). The bug tracker offers a Web form which allows pertinent information to be entered and submitted to the developers.

The first step in filing a report is to determine whether the problem has already been reported. The advantage in doing so, aside from saving the developers time, is that you learn what has been done to fix it; it may be that the problem has already been fixed for the next release, or additional information is needed (in which case you are welcome to provide it if you can!). To do this, search the bug database using the search box near the bottom of the page.

If the problem you're reporting is not already in the bug tracker, go back to the Python Bug Tracker (http://sourceforge.net/bugs/?group_id=5470). Select the "Submit a Bug" link at the top of the page to open the bug reporting form.

The submission form has a number of fields. The only fields that are required are the "Summary" and "Details" fields. For the summary, enter a *very* short description of the problem; less than ten words is good. In the Details field, describe the problem in detail, including what you expected to happen and what did happen. Be sure to include the version of Python you used, whether any extension modules were involved, and what hardware and software platform you were using (including version information as appropriate).

The only other field that you may want to set is the "Category" field, which allows you to place the bug report into a broad category (such as "Documentation" or "Library").

Each bug report will be assigned to a developer who will determine what needs to be done to correct the problem. You will receive an update each time action is taken on the bug.

See Also:

How to Report Bugs Effectively
(http://www-mice.cs.ucl.ac.uk/multimedia/software/documentation/ReportingBugs.html)
: Article which goes into some detail about how to create a useful bug report. This describes what kind of information is useful and why it is useful.

Bug Writing Guidelines
(http://www.mozilla.org/quality/bug-writing-guidelines.html)
: Information about writing a good bug report. Some of this is specific to the Mozilla project, but describes general good practices.

APPENDIX

B

History and License

B.1 History of the software

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see http://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see http://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Zope Corporation (then Digital Creations; see http://www.zope.com/). In 2001, the Python Software Foundation (PSF, see http://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see http://www.opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

Release	Derived from	Year	Owner	GPL compatible?
0.9.0 thru 1.2	n/a	1991-1995	CWI	yes
1.3 thru 1.5.2	1.2	1995-1999	CNRI	yes
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	yes
2.1.1	2.1+2.0.1	2001	PSF	yes
2.2	2.1.1	2001	PSF	yes
2.1.2	2.1.1	2002	PSF	yes
2.1.3	2.1.2	2002	PSF	yes
2.2.1	2.2	2002	PSF	yes

Note: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Thanks to the many outside volunteers who have worked under Guido's direction to make these releases possible.

B.2 Terms and conditions for accessing or otherwise using Python

PSF LICENSE AGREEMENT FOR PYTHON 2.2

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.2.2 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.2.2 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001, 2002 Python Software Foundation; All Rights Reserved" are retained in Python 2.2.2 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.2.2 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.2.2.

4. PSF is making Python 2.2.2 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.2.2 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.2.2 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.2.2, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.2.2, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

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 name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM 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.