Extending and Embedding the Python Interpreter

Guido van Rossum
Dept. CST, CWI, P.O. Box 94079
1090 GB Amsterdam, The Netherlands
E-mail: guido@cwi.nl

26 Januari 1994
Release 1.0.0

Abstract

This document describes how to write modules in C or C++ to extend the Python interpreter. It also describes how to use Python as an 'embedded' language, and how extension modules can be loaded dynamically (at run time) into the interpreter, if the operating system supports this feature.

Contents

Chapter 1

Extending Python with C or C++ code

1.1 Introduction

It is quite easy to add non-standard built-in modules to Python, if you know how to program in C. A built-in module known to the Python programmer as `foo` is generally implemented by a file called '`foomodule.c`'. All but the two most essential standard built-in modules also adhere to this convention, and in fact some of them form excellent examples of how to create an extension.

Extension modules can do two things that can't be done directly in Python: they can implement new data types (which are different from classes by the way), and they can make system calls or call C library functions. Since the latter is usually the most important reason for adding an extension, I'll concentrate on adding 'wrappers' around C library functions; the concrete example uses the wrapper for `system()` in module `posix`, found in (of course) the file '`Modules/posixmodule.c`'.

Note: unless otherwise mentioned, all file references in this document are relative to the toplevel directory of the Python distribution — i.e. the directory that contains the '`configure`' script.

The compilation of an extension module depends on your system setup and the intended use of the module; details are given in a later section.

1.2 A first look at the code

It is important not to be impressed by the size and complexity of the average extension module; much of this is straightforward 'boilerplate' code (starting right with the copyright notice)!

Let's skip the boilerplate and have a look at an interesting function in '`posixmodule.c`' first:

```
static object *
```

```
posix_system(self, args)
    object *self;
    object *args;
{
    char *command;
    int sts;
    if (!getargs(args, "s", &command))
        return NULL;
    sts = system(command);
    return mkvalue("i", sts);
}
```

This is the prototypical top-level function in an extension module. It will be called (we'll see later how) when the Python program executes statements like

```
>>> import posix
>>> sts = posix.system('ls -l')
```

There is a straightforward translation from the arguments to the call in Python (here the single expression `'ls -l'`) to the arguments that are passed to the C function. The C function always has two parameters, conventionally named *self* and *args*. The *self* argument is used when the C function implements a builtin method — this is advanced material and not covered in this document. 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* parameter will be a pointer to a Python object, or `NULL` if the Python function/method was called without arguments. It is necessary to do full argument type checking on each call, since otherwise the Python user would be able to cause the Python interpreter to 'dump core' by passing invalid arguments to a function in an extension module. Because argument checking and converting arguments to C are such common tasks, there's a general function in the Python interpreter that combines them: `getargs()`. It uses a template string to determine both the types of the Python argument and the types of the C variables into which it should store the converted values.[1] (More about this later.)

If `getargs()` returns nonzero, the argument list has the right type and its components have been stored in the variables whose addresses are passed. If it returns zero, an error has occurred. In the latter case it has already raised an appropriate exception by so the calling function should return `NULL` immediately — see the next section.

1.3 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 (often a `NULL` pointer).

[1] There are convenience macros `getnoarg()`, `getstrarg()`, `getintarg()`, etc., for many common forms of `getargs()` templates. These are relics from the past; the recommended practice is to call `getargs()` directly.

Exceptions are stored in a static global variable in 'Python/errors.c'; if this variable is NULL no exception has occurred. A second static global variable stores the 'associated value' of the exception — the second argument to raise.

The file 'errors.h' declares a host of functions to set various types of exceptions. The most common one is err_setstr() — its arguments are an exception object (e.g. RuntimeError — actually it can be any string object) and a C string indicating the cause of the error (this is converted to a string object and stored as the 'associated value' of the exception). Another useful function is err_errno(), which only takes an exception argument and constructs the associated value by inspection of the (UNIX) global variable errno. The most general function is err_set(), which takes two object arguments, the exception and its associated value. You don't need to INCREF() the objects passed to any of these functions.

You can test non-destructively whether an exception has been set with err_occurred(). However, most code never calls err_occurred() to see whether an error occurred or not, but relies on error return values from the functions it calls instead.

When a function that calls another function detects that the called function fails, it should return an error value (e.g. NULL or -1) but not call one of the err_* functions — one has already been called. The caller is then supposed to also return an error indication to *its* caller, again *without* calling err_*(), and so on — the most detailed cause of the error was already reported by the function that first detected it. Once the error has reached Python's interpreter 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 err_* 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 err_clear(). The only time C code should call err_clear() is if it doesn't want to pass the error on to the interpreter but wants to handle it completely by itself (e.g. by trying something else or pretending nothing happened).

Finally, the function err_get() gives you both error variables *and clears them*. Note that even if an error occurred the second one may be NULL. You have to XDECREF() both when you are finished with them. I doubt you will need to use this function.

Note that a failing malloc() call must also be turned into an exception — the direct caller of malloc() (or realloc()) must call err_nomem() and return a failure indicator itself. All the object-creating functions (newintobject() etc.) already do this, so only if you call malloc() directly this note is of importance.

Also note that, with the important exception of getargs(), functions that return an integer status usually return 0 or a positive value for success and -1 for failure.

Finally, be careful about cleaning up garbage (making XDECREF() or DECREF() calls for objects you have already created) when you return an error!

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

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

```
static object *FooError;
```

and initialize it in your module's initialization function (`initfoo()`) with a string object, e.g. (leaving out the error checking for simplicity):

```
void
initfoo()
{
    object *m, *d;
    m = initmodule("foo", foo_methods);
    d = getmoduledict(m);
    FooError = newstringobject("foo.error");
    dictinsert(d, "error", FooError);
}
```

1.4 Back to the example

Going back to `posix_system()`, you should now be able to understand this bit:

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

It returns `NULL` (the error indicator for functions of this kind) if an error is detected in the argument list, relying on the exception set by `getargs()`. Otherwise the string value of the argument has been copied to the local variable `command` — this is in fact just a pointer assignment and you are not supposed to modify the string to which it points.

If a function is called with multiple arguments, the argument list (the argument `args`) is turned into a tuple. If it is called without arguments, `args` is `NULL`. `getargs()` knows about this; see later.

The next statement in `posix_system()` is a call to the C library function `system()`, passing it the string we just got from `getargs()`:

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

Finally, `posix.system()` must return a value: the integer status returned by the C library `system()` function. This is done using the function `mkvalue()`, which is something like the inverse of `getargs()`: it takes a format string and a variable number of C values and returns a new Python object.

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

In this case, it returns an integer object (yes, even integers are objects on the heap in Python!). More info on `mkvalue()` is given later.

If you had a function that returned no useful argument (a.k.a. a procedure), you would need this idiom:

```
        INCREF(None);
        return None;
```

`None` is a unique Python object representing 'no value'. It differs from `NULL`, which means 'error' in most contexts.

1.5 The module's function table

I promised to show how I made the function `posix_system()` callable from Python programs. This is shown later in '`Modules/posixmodule.c`':

```
    static struct methodlist posix_methods[] = {
        ...
        {"system",  posix_system},
        ...
        {NULL,      NULL}        /* Sentinel */
    };

    void
    initposix()
    {
        (void) initmodule("posix", posix_methods);
    }
```

(The actual `initposix()` is somewhat more complicated, but many extension modules can be as simple as shown here.) When the Python program first imports module `posix`, `initposix()` is called, which calls `initmodule()` with specific parameters. This creates a 'module object' (which is inserted in the table `sys.modules` under the key `'posix'`), and adds built-in-function objects to the newly created module based upon the table (of type struct methodlist) that was passed as its second parameter. The function `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 you don't need to check for errors.

1.6 Compilation and linkage

There are two more things to do before you can use your new extension module: compiling and linking it with the Python system. If you use dynamic loading, the details depend on the style of dynamic loading your system uses; see the chapter on Dynamic Loading for more info 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, in the 1.0 release this is very simple: just place your file (named '`foomodule.c`' for example) in the '`Modules`' directory, add a line to the file '`Modules/Setup`' describing your file:

```
foo foomodule.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 rebuilt the '`Makefile`' there by running `make Makefile`. (This is necessary each time you change the '`Setup`' file.)

1.7 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`'.)

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 `INCREF()` it!) in a global variable — or whereever you see fit. For example, the following function might be part of a module definition:

```
static object *my_callback = NULL;

static object *
my_set_callback(dummy, arg)
    object *dummy, *arg;
{
    XDECREF(my_callback); /* Dispose of previous callback */
    my_callback = arg;
```

```
        XINCREF(my_callback); /* Remember new callback */
        /* Boilerplate for "void" return */
        INCREF(None);
        return None;
    }
```

This particular function doesn't do any typechecking on its argument — that will be done by `call_object()`, which is a bit late but at least protects the Python interpreter from shooting itself in its foot. (The problem with typechecking functions is that there are at least five different Python object types that can be called, so the test would be somewhat cumbersome.)

The macros `XINCREF()` and `XDECREF()` increment/decrement the reference count of an object and are safe in the presence of `NULL` pointers. More info on them in the section on Reference Counts below.

Later, when it is time to call the function, you call the C function `call_object()`. 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, you must pass an empty tuple. For example:

```
    object *arglist;
    object *result;
    ...
    /* Time to call the callback */
    arglist = mktuple(0);
    result = call_object(my_callback, arglist);
    DECREF(arglist);
```

`call_object()` returns a Python object pointer: this is the return value of the Python function. `call_object()` 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 `DECREF()`-ed immediately after the call.

The return value of `call_object()` 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 `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 `call_object()` 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 `err_clear()`. For example:

```
    if (result == NULL)
```

```
        return NULL; /* Pass error back */
    /* Here maybe use the result */
    DECREF(result);
```

Depending on the desired interface to the Python callback function, you may also have to provide an argument list to `call_object()`. 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 `mkvalue()`. For example, if you want to pass an integral event code, you might use the following code:

```
    object *arglist;
    ...
    arglist = mkvalue("(l)", eventcode);
    result = call_object(my_callback, arglist);
    DECREF(arglist);
    if (result == NULL)
        return NULL; /* Pass error back */
    /* Here maybe use the result */
    DECREF(result);
```

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

1.8 Format strings for `getargs()`

The `getargs()` function is declared in '`modsupport.h`' as follows:

```
    int getargs(object *arg, char *format, ...);
```

The remaining arguments must be addresses of variables whose type is determined by the format string. For the conversion to succeed, the *arg* object must match the format and the format must be exhausted. Note that while `getargs()` checks that the Python object really is of the specified type, it cannot check the validity of the addresses of C variables provided in the call: if you make mistakes there, your code will probably dump core.

A non-empty format string consists of a single 'format unit'. A format unit describes one Python object; it is usually a single character or a parenthesized sequence of format units. The type of a format units is determined from its first character, the 'format letter':

's' (string) The Python object must be a string object. The C argument must be a `(char**)` (i.e. the address of a character pointer), and a pointer to the C string contained in the Python object is stored into it. You must not provide storage to store the string; a pointer to an existing string is stored into the character pointer variable whose address you pass. If the next character in the format string is '`#`', another C argument of type `(int*)` must be present, and the length of the Python string (not counting the trailing zero byte) is stored into it.

'`z`' **(string or zero, i.e. `NULL`)** Like '`s`', but the object may also be None. In this case the string pointer is set to `NULL` and if a '`#`' is present the size is set to 0.

'`b`' **(byte, i.e. char interpreted as tiny int)** The object must be a Python integer. The C argument must be a `(char*)`.

'`h`' **(half, i.e. short)** The object must be a Python integer. The C argument must be a `(short*)`.

'`i`' **(int)** The object must be a Python integer. The C argument must be an `(int*)`.

'`l`' **(long)** The object must be a (plain!) Python integer. The C argument must be a `(long*)`.

'`c`' **(char)** The Python object must be a string of length 1. The C argument must be a `(char*)`. (Don't pass an `(int*)`!)

'`f`' **(float)** The object must be a Python int or float. The C argument must be a `(float*)`.

'`d`' **(double)** The object must be a Python int or float. The C argument must be a `(double*)`.

'`S`' **(string object)** The object must be a Python string. The C argument must be an `(object**)` (i.e. the address of an object pointer). The C program thus gets back the actual string object that was passed, not just a pointer to its array of characters and its size as for format character '`s`'. The reference count of the object has not been increased.

'`O`' **(object)** The object can be any Python object, including None, but not `NULL`. The C argument must be an `(object**)`. This can be used if an argument list must contain objects of a type for which no format letter exist: the caller must then check that it has the right type. The reference count of the object has not been increased.

'`(`' **(tuple)** The object must be a Python tuple. Following the '`(`' character in the format string must come a number of format units describing the elements of the tuple, followed by a '`)`' character. Tuple format units may be nested. (There are no exceptions for empty and singleton tuples; '`()`' specifies an empty tuple and '`(i)`' a singleton of one integer. Normally you don't want to use the latter, since it is hard for the Python user to specify.

More format characters will probably be added as the need arises. It should (but currently isn't) be allowed to use Python long integers whereever integers are expected, and perform a range check. (A range check is in fact always necessary for the '`b`', '`h`' and '`i`' format letters, but this is currently not implemented.)

Some example calls:

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

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

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

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

ok = getargs(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') */

{
    int left, top, right, bottom, h, v;
    ok = getargs(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)) */
}
```

Note that the 'top level' of a non-empty format string must consist of a single unit; strings like '`is`' and '`(ii)s#`' are not valid format strings. (But '`s#`' is.) If you have multiple arguments, the format must therefore always be enclosed in parentheses, as in the examples '`((ii)s#)`' and '`(((ii)(ii))(ii)`'. (The current implementation does not complain when more than one unparenthesized format unit is given. Sorry.)

The `getargs()` function does not support variable-length argument lists. In simple cases you can fake these by trying several calls to `getargs()` until one succeeds, but you must take care to call `err_clear()` before each retry. For example:

```
static object *my_method(self, args) object *self, *args; {
    int i, j, k;

    if (getargs(args, "(ii)", &i, &j)) {
        k = 0; /* Use default third argument */
    }
    else {
        err_clear();
        if (!getargs(args, "(iii)", &i, &j, &k))
```

```
            return NULL;
        }
        /* ... use i, j and k here ... */
        INCREF(None);
        return None;
    }
```

(It is possible to think of an extension to the definition of format strings to accommodate this directly, e.g. placing a '|' in a tuple might specify that the remaining arguments are optional. `getargs()` should then return one more than the number of variables stored into.)

Advanced users note: If you set the 'varargs' flag in the method list for a function, the argument will always be a tuple (the 'raw argument list'). In this case you must enclose single and empty argument lists in parentheses, e.g. '`(s)`' and '`()`'.

1.9 The `mkvalue()` function

This function is the counterpart to `getargs()`. It is declared in '`Include/modsupport.h`' as follows:

```
    object *mkvalue(char *format, ...);
```

It supports exactly the same format letters as `getargs()`, but the arguments (which are input to the function, not output) must not be pointers, just values. If a byte, short or float is passed to a varargs function, it is widened by the compiler to int or double, so '`b`' and '`h`' are treated as '`i`' and '`f`' is treated as '`d`'. '`S`' is treated as '`O`', '`s`' is treated as '`z`'. '`z#`' and '`s#`' are supported: a second argument specifies the length of the data (negative means use `strlen()`). '`S`' and '`O`' add a reference to their argument (so you should `DECREF()` it if you've just created it and aren't going to use it again).

If the argument for '`O`' or '`S`' 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, `mkvalue()` will return `NULL` but won't set an exception if one is already set. If no exception is set, `SystemError` is set.

If there is an error in the format string, the `SystemError` exception is set, since it is the calling C code's fault, not that of the Python user who sees the exception.

Example:

```
    return mkvalue("(ii)", 0, 0);
```

returns a tuple containing two zeros. (Outer parentheses in the format string are actually superfluous, but you can use them for compatibility with `getargs()`, which requires them if more than one argument is expected.)

1.10 Reference counts

Here's a useful explanation of `INCREF()` and `DECREF()` (after an original by Sjoerd Mullender).

Use `XINCREF()` or `XDECREF()` instead of `INCREF()` or `DECREF()` when the argument may be `NULL` — the versions without '`X`' are faster but wull dump core when they encounter a `NULL` pointer.

The basic idea is, if you create an extra reference to an object, you must `INCREF()` it, if you throw away a reference to an object, you must `DECREF()` it. Functions such as `newstringobject()`, `newsizedstringobject()`, `newintobject()`, etc. create a reference to an object. If you want to throw away the object thus created, you must use `DECREF()`.

If you put an object into a tuple or list using `settupleitem()` or `setlistitem()`, the idea is that you usually don't want to keep a reference of your own around, so Python does not `INCREF()` the elements. It does `DECREF()` the old value. This means that if you put something into such an object using the functions Python provides for this, you must `INCREF()` the object if you also want to keep a separate reference to the object around. Also, if you replace an element, you should `INCREF()` the old element first if you want to keep it. If you didn't `INCREF()` it before you replaced it, you are not allowed to look at it anymore, since it may have been freed.

Returning an object to Python (i.e. when your C function returns) creates a reference to an object, but it does not change the reference count. When your code does not keep another reference to the object, you should not `INCREF()` or `DECREF()` it (assuming it is a newly created object). When you do keep a reference around, you should `INCREF()` the object. Also, when you return a global object such as `None`, you should `INCREF()` it.

If you want to return a tuple, you should consider using `mkvalue()`. This function creates a new tuple with a reference count of 1 which you can return. If any of the elements you put into the tuple are objects (format codes '`O`' or '`S`'), they are `INCREF()`'ed by `mkvalue()`. If you don't want to keep references to those elements around, you should `DECREF()` them after having called `mkvalue()`.

Usually you don't have to worry about arguments. They are `INCREF()`'ed before your function is called and `DECREF()`'ed after your function returns. When you keep a reference to an argument, you should `INCREF()` it and `DECREF()` when you throw it away. Also, when you return an argument, you should `INCREF()` it, because returning the argument creates an extra reference to it.

If you use `getargs()` to parse the arguments, you can get a reference to an object (by using '`O`' in the format string). This object was not `INCREF()`'ed, so you should not `DECREF()` it. If you want to keep the object, you must `INCREF()` it yourself.

If you create your own type of objects, you should use `NEWOBJ()` to create the object. This sets the reference count to 1. If you want to throw away the object, you should use `DECREF()`. When the reference count reaches zero, your type's `dealloc()` function is called. In it, you should `DECREF()` all object to which you keep references in your object, but you should not use `DECREF()` on your object. You should use `DEL()` instead.

1.11 Writing extensions in C++

It is possible to write extension modules in C++. Some restrictions apply: since the main program (the Python interpreter) is compiled and linked by the C compiler, global or static objects with constructors cannot be used. All functions that will be called directly or indirectly (i.e. via function pointers) by the Python interpreter will have to be declared using `extern "C"`; this applies to all 'methods' as well as to the module's initialization function. It is unnecessary to enclose the Python header files in `extern "C" {...}` — they do this already.

Chapter 2

Embedding Python in another application

Embedding Python is similar to extending it, but not quite. The difference is that when you extend Python, the main program of the application is still the Python interpreter, while of 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 `initall()`. 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 `run_command()`, or you can pass a stdio file pointer and a file name (for identification in error messages only) to `run_script()`. 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`'.

2.1 Embedding Python in C++

It is also possible to embed Python in a C++ program; how this is done exactly 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 with C++.

Chapter 3

Dynamic Loading

On most modern systems it is possible to configure Python to support dynamic loading of extension modules implemented in C. When shared libraries are used dynamic loading is configured automatically; otherwise you have to select it as a build option (see below). Once configured, dynamic loading is trivial to use: when a Python program executes `import foo`, the search for modules tries to find a file '`foomodule.o`' ('`foomodule.so`' when using shared libraries) in the module search path, and if one is found, it is loaded into the executing binary and executed. Once loaded, the module acts just like a built-in extension module.

The advantages of dynamic loading are twofold: the 'core' Python binary gets smaller, and users can extend Python with their own modules implemented in C without having to build and maintain their own copy of the Python interpreter. There are also disadvantages: dynamic loading isn't available on all systems (this just means that on some systems you have to use static loading), and dynamically loading a module that was compiled for a different version of Python (e.g. with a different representation of objects) may dump core.

3.1 Configuring and building the interpreter for dynamic loading

There are three styles of dynamic loading: one using shared libraries, one using SGI IRIX 4 dynamic loading, and one using GNU dynamic loading.

3.1.1 Shared libraries

The following systems supports dynamic loading using shared libraries: SunOS 4; Solaris 2; SGI IRIX 5 (but not SGI IRIX 4!); and probably all systems derived from SVR4, or at least those SVR4 derivatives that support shared libraries (are there any that don't?).

You don't need to do anything to configure dynamic loading on these systems — the '`configure`' detects the presence of the '`<dlfcn.h>`' header file and automatically configures dynamic loading.

3.1.2 SGI dynamic loading

Only SGI IRIX 4 supports dynamic loading of modules using SGI dynamic loading. (SGI IRIX 5 might also support it but it is inferior to using shared libraries so there is no reason to; a small test didn't work right away so I gave up trying to support it.)

Before you build Python, you first need to fetch and build the `dl` package written by Jack Jansen. This is available by anonymous ftp from host '`ftp.cwi.nl`', directory '`pub/dynload`', file '`dl-1.6.tar.Z`'. (The version number may change.) Follow the instructions in the package's '`README`' file to build it.

Once you have built `dl`, you can configure Python to use it. To this end, you run the '`configure`' script with the option `--with-dl=`*directory* where *directory* is the absolute pathname of the `dl` directory.

Now build and install Python as you normally would (see the '`README`' file in the toplevel Python directory.)

3.1.3 GNU dynamic loading

GNU dynamic loading supports (according to its '`README`' file) the following hardware and software combinations: VAX (Ultrix), Sun 3 (SunOS 3.4 and 4.0), Sparc (SunOS 4.0), Sequent Symmetry (Dynix), and Atari ST. There is no reason to use it on a Sparc; I haven't seen a Sun 3 for years so I don't know if these have shared libraries or not.

You need to fetch and build two packages. One is GNU DLD 3.2.3, available by anonymous ftp from host '`ftp.cwi.nl`', directory '`pub/dynload`', file '`dld-3.2.3.tar.Z`'. (As far as I know, no further development on GNU DLD is being done.) The other is an emulation of Jack Jansen's `dl` package that I wrote on top of GNU DLD 3.2.3. This is available from the same host and directory, file dl-dld-1.1.tar.Z. (The version number may change — but I doubt it will.) Follow the instructions in each package's '`README`' file to configure build them.

Now configure Python. Run the '`configure`' script with the option `--with-dl-dld=`*dl-directory*`,`*dld-directory* where *dl-directory* is the absolute pathname of the directory where you have built the '`dl-dld`' package, and *dld-directory* is that of the GNU DLD package. The Python interpreter you build hereafter will support GNU dynamic loading.

3.2 Building a dynamically loadable module

Since there are three styles of dynamic loading, there are also three groups of instructions for building a dynamically loadable module. Instructions common for all three styles are given first. Assuming your module is called `foo`, the source filename must be '`foomodule.c`', so the object name is '`foomodule.o`'. The module must be written as a normal Python extension module (as described earlier).

Note that in all cases you will have to create your own Makefile that compiles your module file(s). This Makefile will have to pass two '`-I`' arguments to the C compiler which will make it find the Python header files. If the Make variable *PYTHONTOP* points to the toplevel Python directory, your *CFLAGS* Make variable should contain the options '`-I$(PYTHONTOP) -I$(PYTHONTOP)/Include`'. (Most header files are in the '`Include`' subdirectory, but the '`config.h`' header lives in the toplevel directory.) You must also add '`-DHAVE_CONFIG_H`' to the definition of *CFLAGS* to direct the Python headers to include '`config.h`'.

3.2.1 Shared libraries

You must link the '`.o`' file to produce a shared library. This is done using a special invocation of the UNIX loader/linker, *ld*(1). Unfortunately the invocation differs slightly per system.

On SunOS 4, use

```
ld foomodule.o -o foomodule.so
```

On Solaris 2, use

```
ld -G foomodule.o -o foomodule.so
```

On SGI IRIX 5, use

```
ld -shared foomodule.o -o foomodule.so
```

On other systems, consult the manual page for *ld*(1) to find what flags, if any, must be used.

If your extension module uses system libraries that haven't already been linked with Python (e.g. a windowing system), these must be passed to the *ld* command as '`-l`' options after the '`.o`' file.

The resulting file '`foomodule.so`' must be copied into a directory along the Python module search path.

3.2.2 SGI dynamic loading

bf IMPORTANT: You must compile your extension module with the additional C flag '`-G0`' (or '`-G 0`'). This instruct the assembler to generate position-independent code.

You don't need to link the resulting '`foomodule.o`' file; just copy it into a directory along the Python module search path.

The first time your extension is loaded, it takes some extra time and a few messages may be printed. This creates a file '`foomodule.ld`' which is an image that can be loaded quickly into the Python interpreter process. When a new Python interpreter is installed, the `dl` package detects this and rebuilds '`foomodule.ld`'. The file '`foomodule.ld`' is placed in the directory where '`foomodule.o`' was found, unless this directory is unwritable; in that case it is placed in a temporary directory.[1]

[1]Check the manual page of the `dl` package for details.

If your extension modules uses additional system libraries, you must create a file '`foomodule.libs`' in the same directory as the '`foomodule.o`'. This file should contain one or more lines with whitespace-separated options that will be passed to the linker — normally only '`-l`' options or absolute pathnames of libraries ('`.a`' files) should be used.

3.2.3 GNU dynamic loading

Just copy '`foomodule.o`' into a directory along the Python module search path.

If your extension modules uses additional system libraries, you must create a file '`foomodule.libs`' in the same directory as the '`foomodule.o`'. This file should contain one or more lines with whitespace-separated absolute pathnames of libraries ('`.a`' files). No '`-l`' options can be used.