Python Extension Programming with C

Previous Page Next Page

Any code that you write using any compiled language like C, C++, or Java can be integrated or imported into another Python script. This code is considered as an "extension."

A Python extension module is nothing more than a normal C library. On Unix machines, these libraries usually end in **.so** (for shared object). On Windows machines, you typically see **.dll** (for dynamically linked library).

Pre-Requisites for Writing Extensions

To start writing your extension, you are going to need the Python header files.

On Unix machines, this usually requires installing a developer-specific package such as python2.5-dev .

Windows users get these headers as part of the package when they use the binary Python installer.

Additionally, it is assumed that you have good knowledge of C or C++ to write any Python Extension using C programming.

First look at a Python Extension

For your first look at a Python extension module, you need to group your code into four part −

The header file *Python.h*.

The C functions you want to expose as the interface from your module.

A table mapping the names of your functions as Python developers see them to C functions inside the extension module.

An initialization function.

The Header File *Python.h*

You need include *Python.h* header file in your C source file, which gives you access to the internal Python API used to hook your module into the interpreter.

Make sure to include Python.h before any other headers you might need. You need to follow the includes with the functions you want to call from Python.

The C Functions

The signatures of the C implementation of your functions always takes one of the following three forms −

```
static PyObject *MyFunction( PyObject *self, PyObject *args );

static PyObject *MyFunctionWithKeywords(PyObject *self,
                                PyObject *args,
                                PyObject *kw);

static PyObject *MyFunctionWithNoArgs( PyObject *self );
```

Each one of the preceding declarations returns a Python object. There is no such thing as a *void* function in Python as there is in C. If you do not want your functions to return a value, return the C equivalent of Python's **None** value. The Python headers define a macro, Py_RETURN_NONE, that does this for us.

The names of your C functions can be whatever you like as they are never seen outside of the extension module. They are defined as *static* function.

Your C functions usually are named by combining the Python module and function names together, as shown here −

```
static PyObject *module_func(PyObject *self, PyObject *args) {
   /* Do your stuff here. */
   Py_RETURN_NONE;
}
```

This is a Python function called *func* inside of the module *module*. You will be putting pointers to your C functions into the method table for the module that usually comes next in your source code.

The Method Mapping Table

This method table is a simple array of PyMethodDef structures. That structure looks something like this −

```
struct PyMethodDef {
   char *ml_name;
   PyCFunction ml_meth;
   int ml_flags;
   char *ml_doc;
};
```

Here is the description of the members of this structure:

ml_name: This is the name of the function as the Python interpreter presents when it is used in Python programs.

ml_meth: This must be the address to a function that has any one of the signatures described in previous seection.

ml_flags: This tells the interpreter which of the three signatures ml_meth is using.

This flag usually has a value of METH_VARARGS.

This flag can be bitwise OR'ed with METH_KEYWORDS if you want to allow keyword arguments into your function.

This can also have a value of METH_NOARGS that indicates you do not want to accept any arguments.

ml_doc: This is the docstring for the function, which could be NULL if you do not feel like writing one.

This table needs to be terminated with a sentinel that consists of NULL and 0 values for the appropriate members.

Example

For the above-defined function, we have following method mapping table −

```
static PyMethodDef module_methods[] = {
   { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
   { NULL, NULL, 0, NULL }
};
```

The Initialization Function

The last part of your extension module is the initialization function. This function is called by the Python interpreter when the module is loaded. It is required that the function be named **init*Module***, where *Module* is the name of the module.

The initialization function needs to be exported from the library you will be building. The Python headers define PyMODINIT_FUNC to include the appropriate incantations for that to happen for the particular environment in which we're compiling. All you have to do is use it when defining the function.

Your C initialization function generally has the following overall structure −

```
PyMODINIT_FUNC initModule() {
   Py_InitModule3(func, module_methods, "docstring...");
}
```

Here is the description of *Py_InitModule3* function −

func: This is the function to be exported.

***module*_methods:** This is the mapping table name defined above.

docstring: This is the comment you want to give in your extension.

Putting this all together looks like the following −

```
#include <Python.h>

static PyObject *module_func(PyObject *self, PyObject *args) {
   /* Do your stuff here. */
   Py_RETURN_NONE;
}

static PyMethodDef module_methods[] = {
   { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
   { NULL, NULL, 0, NULL }
};

PyMODINIT_FUNC initModule() {
   Py_InitModule3(func, module_methods, "docstring...");
}
```

Example

A simple example that makes use of all the above concepts −

```
#include <Python.h>

static PyObject* helloworld(PyObject* self)
{
    return Py_BuildValue("s", "Hello, Python extensions!!");
}

static char helloworld_docs[] =
    "helloworld( ): Any message you want to put here!!\n";

static PyMethodDef helloworld_funcs[] = {
    {"helloworld", (PyCFunction)helloworld,
     METH_NOARGS, helloworld_docs},
    {NULL}
};

void inithelloworld(void)
{
    Py_InitModule3("helloworld", helloworld_funcs,
                   "Extension module example!");
}
```

Here the *Py_BuildValue* function is used to build a Python value. Save above code in hello.c file. We would see how to compile and install this module to be called from Python script.

Building and Installing Extensions:

The *distutils* package makes it very easy to distribute Python modules, both pure Python and extension modules, in a standard way. Modules are distributed in source form and built and installed via a setup script usually called *setup.py* as follows.

For the above module, you need to prepare following setup.py script −

```
from distutils.core import setup, Extension
setup(name='helloworld', version='1.0',  \
      ext_modules=[Extension('helloworld', ['hello.c'])])
```

Now, use the following command, which would perform all needed compilation and linking steps, with the right compiler and linker commands and flags, and copies the resulting dynamic library into an appropriate directory −

```
$ python setup.py install
```

On Unix-based systems, you'll most likely need to run this command as root in order to have permissions to write to the site-packages directory. This usually is not a problem on Windows.

Importing Extensions

Once you installed your extension, you would be able to import and call that extension in your Python script as follows −

```
#!/usr/bin/python
import helloworld

print helloworld.helloworld()
```

This would produce the following result −

```
Hello, Python extensions!!
```

Passing Function Parameters

As you will most likely want to define functions that accept arguments, you can use one of the other signatures for your C functions. For example, following function, that accepts some number of parameters, would be defined like this −

```
static PyObject *module_func(PyObject *self, PyObject *args) {
   /* Parse args and do something interesting here. */
   Py_RETURN_NONE;
}
```

The method table containing an entry for the new function would look like this −

```
static PyMethodDef module_methods[] = {
   { "func", (PyCFunction)module_func, METH_NOARGS, NULL },
   { "func", module_func, METH_VARARGS, NULL },
```

```
   { NULL, NULL, 0, NULL }
};
```

You can use API *PyArg_ParseTuple* function to extract the arguments from the one PyObject pointer passed into your C function.

The first argument to PyArg_ParseTuple is the args argument. This is the object you will be *parsing*. The second argument is a format string describing the arguments as you expect them to appear. Each argument is represented by one or more characters in the format string as follows.

```
static PyObject *module_func(PyObject *self, PyObject *args) {
   int i;
   double d;
   char *s;

   if (!PyArg_ParseTuple(args, "ids", &i, &d, &s)) {
      return NULL;
   }

   /* Do something interesting here. */
   Py_RETURN_NONE;
}
```

Compiling the new version of your module and importing it enables you to invoke the new function with any number of arguments of any type −

```
module.func(1, s="three", d=2.0)
module.func(i=1, d=2.0, s="three")
module.func(s="three", d=2.0, i=1)
```

You can probably come up with even more variations.

The *PyArg_ParseTuple* Function

Here is the standard signature for **PyArg_ParseTuple** function −

```
int PyArg_ParseTuple(PyObject* tuple,char* format,...)
```

This function returns 0 for errors, and a value not equal to 0 for success. tuple is the PyObject* that was the C function's second argument. Here *format* is a C string that describes mandatory and optional arguments.

Here is a list of format codes for **PyArg_ParseTuple** function −

Code	C type	Meaning
c	char	A Python string of length 1 becomes a C char.
d	double	A Python float becomes a C double.
f	float	A Python float becomes a C float.
i	int	A Python int becomes a C int.
l	long	A Python int becomes a C long.
L	long long	A Python int becomes a C long long
O	PyObject*	Gets non-NULL borrowed reference to Python argument.
s	char*	Python string without embedded nulls to C char*.
s#	char*+int	Any Python string to C address and length.
t#	char*+int	Read-only single-segment buffer to C address and length.
u	Py_UNICODE*	Python Unicode without embedded nulls to C.
u#	Py_UNICODE*+int	Any Python Unicode C address and length.
w#	char*+int	Read/write single-segment buffer to C address and length.
z	char*	Like s, also accepts None (sets C char* to NULL).
z#	char*+int	Like s#, also accepts None (sets C char* to NULL).
(...)	as per ...	A Python sequence is treated as one argument per item.
|		The following arguments are optional.
:		Format end, followed by function name for error messages.

;		Format end, followed by entire error message text.

Returning Values

Py_BuildValue takes in a format string much like *PyArg_ParseTuple* does. Instead of passing in the addresses of the values you are building, you pass in the actual values. Here's an example showing how to implement an add function −

```
static PyObject *foo_add(PyObject *self, PyObject *args) {
   int a;
   int b;

   if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
      return NULL;
   }
   return Py_BuildValue("i", a + b);
}
```

This is what it would look like if implemented in Python −

```
def add(a, b):
   return (a + b)
```

You can return two values from your function as follows, this would be cauptured using a list in Python.

```
static PyObject *foo_add_subtract(PyObject *self, PyObject *args) {
   int a;
   int b;

   if (!PyArg_ParseTuple(args, "ii", &a, &b)) {
      return NULL;
   }
   return Py_BuildValue("ii", a + b, a - b);
}
```

This is what it would look like if implemented in Python −

```
def add_subtract(a, b):
   return (a + b, a - b)
```

The *Py_BuildValue* Function:

Here is the standard signature for **Py_BuildValue** function −

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

Here *format* is a C string that describes the Python object to build. The following arguments of *Py_BuildValue* are C values from which the result is built. The *PyObject** result is a new reference.

Following table lists the commonly used code strings, of which zero or more are joined into string format.

Code	C type	Meaning
c	char	A C char becomes a Python string of length 1.
d	double	A C double becomes a Python float.
f	float	A C float becomes a Python float.
i	int	A C int becomes a Python int.
l	long	A C long becomes a Python int.
N	PyObject*	Passes a Python object and steals a reference.
O	PyObject*	Passes a Python object and INCREFs it as normal.
O&	convert+void*	Arbitrary conversion
s	char*	C 0-terminated char* to Python string, or NULL to None.
s#	char*+int	C char* and length to Python string, or NULL to None.
u	Py_UNICODE*	C-wide, null-terminated string to Python Unicode, or NULL to None.
u#	Py_UNICODE*+int	C-wide string and length to Python Unicode, or NULL to None.
w#	char*+int	Read/write single-segment buffer to C address and length.

z	char*	Like s, also accepts None (sets C char* to NULL).
z#	char*+int	Like s#, also accepts None (sets C char* to NULL).
(...)	as per ...	Builds Python tuple from C values.
[...]	as per ...	Builds Python list from C values.
{...}	as per ...	Builds Python dictionary from C values, alternating keys and values.

Code {...} builds dictionaries from an even number of C values, alternately keys and values. For example, Py_BuildValue("{issi}",23,"zig","zag",42) returns a dictionary like Python's {23:'zig','zag':42}.

Previous Page Next Page