Running several threads is similar to running several different programs concurrently, but with the following benefits −

* Multiple threads within a process share the same data space with the main thread and can therefore share information or communicate with each other more easily than if they were separate processes.
* Threads sometimes called light-weight processes and they do not require much memory overhead; they care cheaper than processes.

A thread has a beginning, an execution sequence, and a conclusion. It has an instruction pointer that keeps track of where within its context it is currently running.

* It can be pre-empted (interrupted)
* It can temporarily be put on hold (also known as sleeping) while other threads are running - this is called yielding.

Starting a New Thread

To spawn another thread, you need to call following method available in *thread* module:

thread.start\_new\_thread ( function, args[, kwargs] )

This method call enables a fast and efficient way to create new threads in both Linux and Windows.

The method call returns immediately and the child thread starts and calls function with the passed list of *agrs*. When function returns, the thread terminates.

Here, *args* is a tuple of arguments; use an empty tuple to call function without passing any arguments. *kwargs* is an optional dictionary of keyword arguments.

Example

#!/usr/bin/python

import thread

import time

# Define a function for the thread

def print\_time( threadName, delay):

count = 0

while count < 5:

time.sleep(delay)

count += 1

print "%s: %s" % ( threadName, time.ctime(time.time()) )

# Create two threads as follows

try:

thread.start\_new\_thread( print\_time, ("Thread-1", 2, ) )

thread.start\_new\_thread( print\_time, ("Thread-2", 4, ) )

except:

print "Error: unable to start thread"

while 1:

pass

When the above code is executed, it produces the following result −

Thread-1: Thu Jan 22 15:42:17 2009

Thread-1: Thu Jan 22 15:42:19 2009

Thread-2: Thu Jan 22 15:42:19 2009

Thread-1: Thu Jan 22 15:42:21 2009

Thread-2: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:23 2009

Thread-1: Thu Jan 22 15:42:25 2009

Thread-2: Thu Jan 22 15:42:27 2009

Thread-2: Thu Jan 22 15:42:31 2009

Thread-2: Thu Jan 22 15:42:35 2009

Although it is very effective for low-level threading, but the *thread* module is very limited compared to the newer threading module.

The *Threading* Module:

The newer threading module included with Python 2.4 provides much more powerful, high-level support for threads than the thread module discussed in the previous section.

The *threading* module exposes all the methods of the *thread* module and provides some additional methods:

* **threading.activeCount():** Returns the number of thread objects that are active.
* **threading.currentThread():** Returns the number of thread objects in the caller's thread control.
* **threading.enumerate():** Returns a list of all thread objects that are currently active.

In addition to the methods, the threading module has the *Thread* class that implements threading. The methods provided by the *Thread* class are as follows:

* **run():** The run() method is the entry point for a thread.
* **start():** The start() method starts a thread by calling the run method.
* **join([time]):** The join() waits for threads to terminate.
* **isAlive():** The isAlive() method checks whether a thread is still executing.
* **getName():** The getName() method returns the name of a thread.
* **setName():** The setName() method sets the name of a thread.

Creating Thread Using *Threading* Module

To implement a new thread using the threading module, you have to do the following −

* Define a new subclass of the *Thread* class.
* Override the *\_\_init\_\_(self [,args])* method to add additional arguments.
* Then, override the run(self [,args]) method to implement what the thread should do when started.

Once you have created the new *Thread* subclass, you can create an instance of it and then start a new thread by invoking the *start()*, which in turn calls *run()* method.

Example

#!/usr/bin/python

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print "Starting " + self.name

print\_time(self.name, self.counter, 5)

print "Exiting " + self.name

def print\_time(threadName, delay, counter):

while counter:

if exitFlag:

thread.exit()

time.sleep(delay)

print "%s: %s" % (threadName, time.ctime(time.time()))

counter -= 1

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

print "Exiting Main Thread"

When the above code is executed, it produces the following result −

Starting Thread-1

Starting Thread-2

Exiting Main Thread

Thread-1: Thu Mar 21 09:10:03 2013

Thread-1: Thu Mar 21 09:10:04 2013

Thread-2: Thu Mar 21 09:10:04 2013

Thread-1: Thu Mar 21 09:10:05 2013

Thread-1: Thu Mar 21 09:10:06 2013

Thread-2: Thu Mar 21 09:10:06 2013

Thread-1: Thu Mar 21 09:10:07 2013

Exiting Thread-1

Thread-2: Thu Mar 21 09:10:08 2013

Thread-2: Thu Mar 21 09:10:10 2013

Thread-2: Thu Mar 21 09:10:12 2013

Exiting Thread-2

Synchronizing Threads

The threading module provided with Python includes a simple-to-implement locking mechanism that allows you to synchronize threads. A new lock is created by calling the *Lock()* method, which returns the new lock.

The *acquire(blocking)* method of the new lock object is used to force threads to run synchronously. The optional *blocking* parameter enables you to control whether the thread waits to acquire the lock.

If *blocking* is set to 0, the thread returns immediately with a 0 value if the lock cannot be acquired and with a 1 if the lock was acquired. If blocking is set to 1, the thread blocks and wait for the lock to be released.

The *release()* method of the new lock object is used to release the lock when it is no longer required.

Example

#!/usr/bin/python

import threading

import time

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, counter):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.counter = counter

def run(self):

print "Starting " + self.name

# Get lock to synchronize threads

threadLock.acquire()

print\_time(self.name, self.counter, 3)

# Free lock to release next thread

threadLock.release()

def print\_time(threadName, delay, counter):

while counter:

time.sleep(delay)

print "%s: %s" % (threadName, time.ctime(time.time()))

counter -= 1

threadLock = threading.Lock()

threads = []

# Create new threads

thread1 = myThread(1, "Thread-1", 1)

thread2 = myThread(2, "Thread-2", 2)

# Start new Threads

thread1.start()

thread2.start()

# Add threads to thread list

threads.append(thread1)

threads.append(thread2)

# Wait for all threads to complete

for t in threads:

t.join()

print "Exiting Main Thread"

When the above code is executed, it produces the following result −

Starting Thread-1

Starting Thread-2

Thread-1: Thu Mar 21 09:11:28 2013

Thread-1: Thu Mar 21 09:11:29 2013

Thread-1: Thu Mar 21 09:11:30 2013

Thread-2: Thu Mar 21 09:11:32 2013

Thread-2: Thu Mar 21 09:11:34 2013

Thread-2: Thu Mar 21 09:11:36 2013

Exiting Main Thread

Multithreaded Priority Queue

The *Queue* module allows you to create a new queue object that can hold a specific number of items. There are following methods to control the Queue −

* **get():** The get() removes and returns an item from the queue.
* **put():** The put adds item to a queue.
* **qsize() :** The qsize() returns the number of items that are currently in the queue.
* **empty():** The empty( ) returns True if queue is empty; otherwise, False.
* **full():** the full() returns True if queue is full; otherwise, False.

Example

#!/usr/bin/python

import Queue

import threading

import time

exitFlag = 0

class myThread (threading.Thread):

def \_\_init\_\_(self, threadID, name, q):

threading.Thread.\_\_init\_\_(self)

self.threadID = threadID

self.name = name

self.q = q

def run(self):

print "Starting " + self.name

process\_data(self.name, self.q)

print "Exiting " + self.name

def process\_data(threadName, q):

while not exitFlag:

queueLock.acquire()

if not workQueue.empty():

data = q.get()

queueLock.release()

print "%s processing %s" % (threadName, data)

else:

queueLock.release()

time.sleep(1)

threadList = ["Thread-1", "Thread-2", "Thread-3"]

nameList = ["One", "Two", "Three", "Four", "Five"]

queueLock = threading.Lock()

workQueue = Queue.Queue(10)

threads = []

threadID = 1

# Create new threads

for tName in threadList:

thread = myThread(threadID, tName, workQueue)

thread.start()

threads.append(thread)

threadID += 1

# Fill the queue

queueLock.acquire()

for word in nameList:

workQueue.put(word)

queueLock.release()

# Wait for queue to empty

while not workQueue.empty():

pass

# Notify threads it's time to exit

exitFlag = 1

# Wait for all threads to complete

for t in threads:

t.join()

print "Exiting Main Thread"

When the above code is executed, it produces the following result −

Starting Thread-1

Starting Thread-2

Starting Thread-3

Thread-1 processing One

Thread-2 processing Two

Thread-3 processing Three

Thread-1 processing Four

Thread-2 processing Five

Exiting Thread-3

Exiting Thread-1

Exiting Thread-2

Exiting Main Thread

Python provides two levels of access to network services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both connection-oriented and connectionless protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, and so on.

This chapter gives you understanding on most famous concept in Networking - Socket Programming.

What is Sockets?

Sockets are the endpoints of a bidirectional communications channel. Sockets may communicate within a process, between processes on the same machine, or between processes on different continents.

Sockets may be implemented over a number of different channel types: Unix domain sockets, TCP, UDP, and so on. The *socket* library provides specific classes for handling the common transports as well as a generic interface for handling the rest.

Sockets have their own vocabulary:

|  |  |
| --- | --- |
| **Term** | **Description** |
| domain | The family of protocols that is used as the transport mechanism. These values are constants such as AF\_INET, PF\_INET, PF\_UNIX, PF\_X25, and so on. |
| type | The type of communications between the two endpoints, typically SOCK\_STREAM for connection-oriented protocols and SOCK\_DGRAM for connectionless protocols. |
| protocol | Typically zero, this may be used to identify a variant of a protocol within a domain and type. |
| hostname | The identifier of a network interface:   * A string, which can be a host name, a dotted-quad address, or an IPV6 address in colon (and possibly dot) notation * A string "<broadcast>", which specifies an INADDR\_BROADCAST address. * A zero-length string, which specifies INADDR\_ANY, or * An Integer, interpreted as a binary address in host byte order. |
| port | Each server listens for clients calling on one or more ports. A port may be a Fixnum port number, a string containing a port number, or the name of a service. |

The *socket* Module

To create a socket, you must use the *socket.socket()* function available in *socket* module, which has the general syntax −

s = socket.socket (socket\_family, socket\_type, protocol=0)

Here is the description of the parameters −

* **socket\_family:** This is either AF\_UNIX or AF\_INET, as explained earlier.
* **socket\_type:** This is either SOCK\_STREAM or SOCK\_DGRAM.
* **protocol:** This is usually left out, defaulting to 0.

Once you have *socket* object, then you can use required functions to create your client or server program. Following is the list of functions required −

Server Socket Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.bind() | This method binds address (hostname, port number pair) to socket. |
| s.listen() | This method sets up and start TCP listener. |
| s.accept() | This passively accept TCP client connection, waiting until connection arrives (blocking). |

Client Socket Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.connect() | This method actively initiates TCP server connection. |

General Socket Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| s.recv() | This method receives TCP message |
| s.send() | This method transmits TCP message |
| s.recvfrom() | This method receives UDP message |
| s.sendto() | This method transmits UDP message |
| s.close() | This method closes socket |
| socket.gethostname() | Returns the hostname. |

A Simple Server

To write Internet servers, we use the **socket** function available in socket module to create a socket object. A socket object is then used to call other functions to setup a socket server.

Now call **bind(hostname, port)** function to specify a *port* for your service on the given host.

Next, call the *accept* method of the returned object. This method waits until a client connects to the port you specified, and then returns a *connection* object that represents the connection to that client.

#!/usr/bin/python # This is server.py file

import socket # Import socket module

s = socket.socket() # Create a socket object

host = socket.gethostname() # Get local machine name

port = 12345 # Reserve a port for your service.

s.bind((host, port)) # Bind to the port

s.listen(5) # Now wait for client connection.

while True:

c, addr = s.accept() # Establish connection with client.

print 'Got connection from', addr

c.send('Thank you for connecting')

c.close() # Close the connection

A Simple Client

Let us write a very simple client program which opens a connection to a given port 12345 and given host. This is very simple to create a socket client using Python's *socket* module function.

The **socket.connect(hosname, port )** opens a TCP connection to *hostname* on the *port*. Once you have a socket open, you can read from it like any IO object. When done, remember to close it, as you would close a file.

The following code is a very simple client that connects to a given host and port, reads any available data from the socket, and then exits −

#!/usr/bin/python # This is client.py file

import socket # Import socket module

s = socket.socket() # Create a socket object

host = socket.gethostname() # Get local machine name

port = 12345 # Reserve a port for your service.

s.connect((host, port))

print s.recv(1024)

s.close # Close the socket when done

Now run this server.py in background and then run above client.py to see the result.

# Following would start a server in background.

$ python server.py &

# Once server is started run client as follows:

$ python client.py

This would produce following result −

Got connection from ('127.0.0.1', 48437)

Thank you for connecting

Python Internet modules

A list of some important modules in Python Network/Internet programming.

|  |  |  |  |
| --- | --- | --- | --- |
| **Protocol** | **Common function** | **Port No** | **Python module** |
| HTTP | Web pages | 80 | httplib, urllib, xmlrpclib |
| NNTP | Usenet news | 119 | nntplib |
| FTP | File transfers | 20 | ftplib, urllib |
| SMTP | Sending email | 25 | smtplib |
| POP3 | Fetching email | 110 | poplib |
| IMAP4 | Fetching email | 143 | imaplib |
| Telnet | Command lines | 23 | telnetlib |
| Gopher | Document transfers | 70 | gopherlib, urllib |

Please check all the libraries mentioned above to work with FTP, SMTP, POP, and IMAP protocols.

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](http://packages.debian.org/etch-m68k/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 init*Module*() {

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 init*Module*() {

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}.