Extending/Embedding FAQ

Contents

Can I create my own functions in C?

Yes, you can create built-in modules containing functions, variables, exceptions and even new types in C. This is explained in the document Extending and Embedding the Python Interpreter.

Most intermediate or advanced Python books will also cover this topic.

Can I create my own functions in C++?

Yes, using the C compatibility features found in C++. Place `extern "C" { ... }` around the Python include files and put `extern "C"` before each function that is going to be called by the Python interpreter. Global or static C++ objects with constructors are probably not a good idea.

Writing C is hard; are there any alternatives?

There are a number of alternatives to writing your own C extensions, depending on what you're trying to do.

Cython and its relative Pyrex are compilers that accept a slightly modified form of Python and generate the corresponding C code. Cython and Pyrex make it possible to write an extension without having to learn Python's C API.

If you need to interface to some C or C++ library for which no Python extension currently exists, you can try wrapping the library's data types and functions with a tool such as SWIG. SIP, CXX Boost, or Weave are also alternatives for wrapping C++ libraries.

How can I execute arbitrary Python statements from C?

The highest-level function to do this is `PyRun_SimpleString()` which takes a single string argument to be executed in the context of the module `__main__` and returns `0` for success and `-1` when an exception occurred (including `SyntaxError`). If you want more control, use `PyRun_String()`; see the source for `PyRun_SimpleString()` in `Python/pythonrun.c`.

How can I evaluate an arbitrary Python expression from C?

Call the function `PyRun_String()` from the previous question with the start symbol `Py_eval_input`; it parses an expression, evaluates it and returns its value.

How do I extract C values from a Python object?

That depends on the object's type. If it's a tuple, `PyTuple_Size()` returns its length and `PyTuple_GetItem()` returns the item at a specified index. Lists have similar functions, `PyListSize()` and `PyList_GetItem()`.

For bytes, `PyBytes_Size()` returns its length and `PyBytes_AsStringAndSize()` provides a pointer to its value and its length. Note that Python bytes objects may contain null bytes so C's `strlen()` should not be used.

To test the type of an object, first make sure it isn't *NULL*, and then use `PyBytes_Check()`, `PyTuple_Check()`, `PyList_Check()`, etc.

There is also a high-level API to Python objects which is provided by the so-called 'abstract' interface – read `Include/abstract.h` for further details. It allows interfacing with any kind of Python sequence using calls like `PySequence_Length()`, `PySequence_GetItem()`, etc. as well as many other useful protocols such as numbers (`PyNumber_Index()` et al.) and mappings in the PyMapping APIs.

How do I use Py_BuildValue() to create a tuple of arbitrary length?

You can't. Use `PyTuple_Pack()` instead.

How do I call an object's method from C?

The `PyObject_CallMethod()` function can be used to call an arbitrary method of an object. The parameters are the object, the name of the method to call, a format string like that used with `Py_BuildValue()`, and the argument values:

```
PyObject *
PyObject_CallMethod(PyObject *object, const char *method_name,
                    const char *arg_format, ...);
```

This works for any object that has methods – whether built-in or user-defined. You are responsible for eventually `Py_DECREF()`'ing the return value.

To call, e.g., a file object's "seek" method with arguments 10, 0 (assuming the file object pointer is "f"):

```
res = PyObject_CallMethod(f, "seek", "(ii)", 10, 0);
if (res == NULL) {
        ... an exception occurred ...
}
else {
        Py_DECREF(res);
}
```

Note that since `PyObject_CallObject()` *always* wants a tuple for the argument list, to call a function without arguments, pass "()" for the format, and to call a function with one argument, surround the argument in parentheses, e.g. "(i)".

How do I catch the output from PyErr_Print() (or anything that prints to stdout/stderr)?

In Python code, define an object that supports the `write()` method. Assign this object to `sys.stdout` and `sys.stderr`. Call print_error, or just allow the standard traceback mechanism to work. Then, the output will go wherever your `write()` method sends it.

The easiest way to do this is to use the `io.StringIO` class:

```
>>> import io, sys
>>> sys.stdout = io.StringIO()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(sys.stdout.getvalue())
foo
hello world!
```

A custom object to do the same would look like this:

```
>>> import io, sys
>>> class StdoutCatcher(io.TextIOBase):
...     def __init__(self):
...         self.data = []
...     def write(self, stuff):
...         self.data.append(stuff)
...
>>> import sys
>>> sys.stdout = StdoutCatcher()
>>> print('foo')
>>> print('hello world!')
>>> sys.stderr.write(''.join(sys.stdout.data))
foo
hello world!
```

How do I access a module written in Python from C?

You can get a pointer to the module object as follows:

```
module = PyImport_ImportModule("<modulename>");
```

If the module hasn't been imported yet (i.e. it is not yet present in `sys.modules`), this initializes the module; otherwise it simply returns the value of `sys.modules`

`["<modulename>"]`. Note that it doesn't enter the module into any namespace – it only ensures it has been initialized and is stored in `sys.modules`.

You can then access the module's attributes (i.e. any name defined in the module) as follows:

```
attr = PyObject_GetAttrString(module, "<attrname>");
```

Calling `PyObject_SetAttrString()` to assign to variables in the module also works.

How do I interface to C++ objects from Python?

Depending on your requirements, there are many approaches. To do this manually, begin by reading the "Extending and Embedding" document. Realize that for the Python run-time system, there isn't a whole lot of difference between C and C++ – so the strategy of building a new Python type around a C structure (pointer) type will also work for C++ objects.

For C++ libraries, see Writing C is hard; are there any alternatives?.

I added a module using the Setup file and the make fails; why?

Setup must end in a newline, if there is no newline there, the build process fails. (Fixing this requires some ugly shell script hackery, and this bug is so minor that it doesn't seem worth the effort.)

How do I debug an extension?

When using GDB with dynamically loaded extensions, you can't set a breakpoint in your extension until your extension is loaded.

In your `.gdbinit` file (or interactively), add the command:

```
br _PyImport_LoadDynamicModule
```

Then, when you run GDB:

```
$ gdb /local/bin/python
gdb) run myscript.py
gdb) continue # repeat until your extension is loaded
gdb) finish   # so that your extension is loaded
```

```
gdb) br myfunction.c:50
gdb) continue
```

I want to compile a Python module on my Linux system, but some files are missing. Why?

Most packaged versions of Python don't include the `/usr/lib/python2.x/config/` directory, which contains various files required for compiling Python extensions.

For Red Hat, install the python-devel RPM to get the necessary files.

For Debian, run `apt-get install python-dev`.

How do I tell "incomplete input" from "invalid input"?

Sometimes you want to emulate the Python interactive interpreter's behavior, where it gives you a continuation prompt when the input is incomplete (e.g. you typed the start of an "if" statement or you didn't close your parentheses or triple string quotes), but it gives you a syntax error message immediately when the input is invalid.

In Python you can use the `codeop` module, which approximates the parser's behavior sufficiently. IDLE uses this, for example.

The easiest way to do it in C is to call `PyRun_InteractiveLoop()` (perhaps in a separate thread) and let the Python interpreter handle the input for you. You can also set the `PyOS_ReadlineFunctionPointer()` to point at your custom input function. See `Modules/readline.c` and `Parser/myreadline.c` for more hints.

However sometimes you have to run the embedded Python interpreter in the same thread as your rest application and you can't allow the `PyRun_InteractiveLoop()` to stop while waiting for user input. The one solution then is to call `PyParser_ParseString()` and test for `e.error` equal to `E_EOF`, which means the input is incomplete). Here's a sample code fragment, untested, inspired by code from Alex Farber:

```
#include <Python.h>
#include <node.h>
#include <errcode.h>
#include <grammar.h>
#include <parsetok.h>
#include <compile.h>

int testcomplete(char *code)
  /* code should end in \n */
  /* return -1 for error, 0 for incomplete, 1 for complete */
{
  node *n;
  perrdetail e;

  n = PyParser_ParseString(code, &_PyParser_Grammar,
                           Py_file_input, &e);
  if (n == NULL) {
    if (e.error == E_EOF)
      return 0;
    return -1;
  }

  PyNode_Free(n);
  return 1;
}
```

Another solution is trying to compile the received string with `Py_CompileString()`. If it compiles without errors, try to execute the returned code object by calling `PyEval_EvalCode()`. Otherwise save the input for later. If the compilation fails, find out if it’s an error or just more input is required - by extracting the message string from the exception tuple and comparing it to the string “unexpected EOF while parsing”. Here is a complete example using the GNU readline library (you may want to ignore **SIGINT** while calling readline()):

```
#include <stdio.h>
#include <readline.h>

#include <Python.h>
#include <object.h>
#include <compile.h>
#include <eval.h>

int main (int argc, char* argv[])
{
  int i, j, done = 0;                          /* lengths of line, coc
  char ps1[] = ">>> ";
  char ps2[] = "... ";
  char *prompt = ps1;
  char *msg, *line, *code = NULL;
  PyObject *src, *glb, *loc;
  PyObject *exc, *val, *trb, *obj, *dum;

  Py_Initialize ();
  loc = PyDict_New ();
  glb = PyDict_New ();
  PyDict_SetItemString (glb, "__builtins__", PyEval_GetBuiltins ());

  while (!done)
  {
    line = readline (prompt);

    if (NULL == line)                          /* Ctrl-D pressed */
    {
      done = 1;
    }
    else
    {
      i = strlen (line);

      if (i > 0)
        add_history (line);                    /* save non-empty lines

      if (NULL == code)                        /* nothing in code yet
        j = 0;
      else
        j = strlen (code);

      code = realloc (code, i + j + 2);
      if (NULL == code)                        /* out of memory */
        exit (1);

      if (0 == j)                              /* code was empty, so *
        code[0] = '\0';                        /* keep strncat happy *

      strncat (code, line, i);                 /* append line to code
      code[i + j] = '\n';                      /* append '\n' to code
      code[i + j + 1] = '\0';

      src = Py_CompileString (code, "<stdin>", Py_single_input);

      if (NULL != src)                         /* compiled just fine -
      {
        if (ps1  == prompt ||                  /* ">>> " or */
            '\n' == code[i + j - 1])           /* "... " and double '\
        {                                               /* so execute
          dum = PyEval_EvalCode (src, glb, loc);
          Py_XDECREF (dum);
          Py_XDECREF (src);
          free (code);
          code = NULL;
          if (PyErr_Occurred ())
            PyErr_Print ();
          prompt = ps1;
        }
      }                                        /* syntax error or E_EO
      else if (PyErr_ExceptionMatches (PyExc_SyntaxError))
      {
        PyErr_Fetch (&exc, &val, &trb);        /* clears exception! */

        if (PyArg_ParseTuple (val, "sO", &msg, &obj) &&
            !strcmp (msg, "unexpected EOF while parsing")) /* E_EOF */
        {
          Py_XDECREF (exc);
          Py_XDECREF (val);
          Py_XDECREF (trb);
          prompt = ps2;
        }
        else                                   /* some other syntax er
        {
          PyErr_Restore (exc, val, trb);
          PyErr_Print ();
          free (code);
          code = NULL;
          prompt = ps1;
        }
      }
      else                                     /* some non-syntax erro
      {
        PyErr_Print ();
        free (code);
        code = NULL;
        prompt = ps1;
      }

      free (line);
    }
  }

  Py_XDECREF(glb);
  Py_XDECREF(loc);
  Py_Finalize();
  exit(0);
}
```

How do I find undefined g++ symbols __builtin_new or __pure_virtual?

To dynamically load g++ extension modules, you must recompile Python, relink it using g++ (change LINKCC in the Python Modules Makefile), and link your extension module using g++ (e.g., `g++ -shared -o mymodule.so mymodule.o`).

Can I create an object class with some methods implemented in C and others in Python (e.g. through inheritance)?

Yes, you can inherit from built-in classes such as `int`, `list`, `dict`, etc.

The Boost Python Library (BPL, http://www.boost.org/libs/python/doc/index.html) provides a way of doing this from C++ (i.e. you can inherit from an extension class written in C++ using the BPL).