Extending Python

Paul F. Dubois[1] and T. -Y. Yang[2]

Programmable Applications in Python

Python is a small and easy to learn language with some very surprising capabilities. It is a free, portable, interpreted, object-oriented language with a full range of sophisticated features such as first-class functions, garbage collection, and exception handling. Python has a set of properties that make it especially appealing for scientific programming: In the last issue of Scientific Programming, we introduced Python and its new high-performance numerical extension.

What made that numerical extension possible was the great ease with which Python can be extended to include functions and even new object types that are implemented in C, C++, or Fortran. In this issue, we give an introduction to the process of extending Python in this way. Once you can connect your compiled sources to the Python interpreter, your application can use the Python interpreter as its user interface, becoming a powerful programmable application as discussed in the article, “Making Applications Programmable” (CIP 7 #1, Jan/Feb 1994).

It is possible to embed Python in your application, too. The distinction between embedding and extending is really minor, a question of whether you add something to Python or add Python to something. For extensions written in C++, embedding is preferred due to C++’s “static constructor” problem, but for purposes of this paper we will confine ourselves to C and Fortran extensions to Python. On many plat-

1. Paul F. Dubois (dubois1@llnl.gov) is a mathematician at Lawrence Livermore National Laboratory, Livermore, CA.
2. T.-Y. Yang (yang4@llnl.gov) is a physicist at Lawrence Livermore National Laboratory, Livermore, CA

forms this means that the extensions can be dynamically loaded into the existing Python program.

The Documentation

Before we begin, a word about the documentation. The documentation for Python is extensive but fragmented at this point. A book in the O'Reilly series should be available by the time this article is published, which should definitely improve things. The documentation written by Guido von Rossum, Python's author, and available at http://www.python.org, is organized as follows:

- Python Tutorial
- Python Reference Manual
- Extending and Embedding the Python Interpreter
- Python Library Reference

to which we add the definitive reference, The Source. (You know, as in "Use the Source, Luke!".

In understanding the source it helps to realize that early versions of Python used a logical but potentially troublesome set of names for the C routines which make up the Python interpreter and run-time package. Later, a renaming occurred, in which a very consistent and logical set of names was developed. Newer modules have been written using the new names, but since there was a lot of older source around in user-developed extensions, a period of graceful transition was sought by using a file of macros to map the old names to new ones.

The file that does this renaming is in the source tree under Include/rename2.h. For the novice extender of Python, this file makes a handy "cheat-sheet" as to the available calls. Also worth consulting is a new "abstract" interface Include/abstract.h, which makes it easier and more consistent to access such things as sequence and mapping types without being precise as to which exact type they are.

Tools

There are three tools that can help you extend Python. In this article we briefly discuss Modulator and FIDL, which help write C and Fortran extensions, respectively. A more ambitious project, BGEN, attempts to make Python/C interfaces automatically from the C source, but requires more effort to learn.

As we were finishing this article, David Beazley (dmb@viking.lanl.gov) announced a new tool, Simple Wrapper Interface Generator, at http://www.cs.utah.edu/~beazley/SWIG. SWIG is designed to generate interfaces to several different languages, including TCL and Python.

A Simple Functional Extension in C

Let's build a simple extension in C. You'll notice as we do so that we do not need to modify the C source to which we are connecting in any way. We are simply going to build some "glue" between Python and our existing C sources. Suppose we have a C source, is_zero.c, with its header file is_zero.h. This header file declares:

```
extern double tolerance;
extern int is_zero (double x, double y);
```

Here, *is_zero* is a function returning 1 or 0 depending on whether or not *(x, y)* is within a distance *tolerance* of the origin. We wish to be able to query and set the value of *tolerance* and call *is_equal* from Python. We will create a module named *cip1* which will supply three functions to Python:

1. *tolerance ()* will return the current value of the C variable *tolerance*.
2. *set_tolerance (new_tolerance)* will assign a new value to *tolerance*.
3. *is_equal (x, y)* will call the C routine *is_equal* and return its result as a Python integer. Our interface will check for the correct number of arguments and coerce any actual arguments to C type double.

The simplest way to get this job done is to use Modulator. Modulator comes with the Python source distribution, and is easiest to use if you install Tk/Tcl and build Python so that the Tkinter extension is included. Then, Modulator will display a window in which you simply enter the module name you want (cip1), and the names of the functions that are to be in the module (tolerance, set_tolerance, and is_equal). Then you press the button "Generate Code", and when prompted give it the file name "cip1module.c". That file is created and your extension is essentially complete. In fact, it will already compile and load. All we need to do is fill in correct behaviors for these three functions. The example below has been produced in this way.

The file cip1module.c

First we include the Python definitions file “Python.h” and the header file for our C routine, “is_zero.h”, and set up a macro for error checking:

```
#include "Python.h"
#include "is_zero.h"

/* Define a macro for error handling:
     Py_Try(BOOLEAN)
        If BOOLEAN is FALSE, assume the error object has
            been set and return NULL
*/

#define Py_Try(BOOLEAN) {if (!(BOOLEAN)) return NULL;}
```

Next we have the actual definitions of our three functions. We give them names that begin with the module name as a matter of convention; in fact, any names can be used. For each of the three functions we include a string variable which will hold the documentation which will be available to the user. First, is_zero:

```
static char cip1_is_zero__doc__[] =
"is_zero (x, y) tests whether (x, y) almost equals (0.0, 0.0)"
;
static PyObject *
cip1_is_zero(PyObject *self, PyObject *args)
{
  double x, y;
  int result;

  /* Get the two arguments into x and y */
  Py_Try (PyArg_ParseTuple (args, "dd", &x, &y));

  /* call is_equal and return the answer as a Python integer */
  return Py_BuildValue ("i", is_zero (x, y));
}
```

The first argument to cip1_is_zero, self, is not actually used. When we define our own objects we will use this argument to represent the instance to which the function is applied. For uniformity, it was decided to have all extension functions have the same form, with self as the first argument.

The function PyArg_ParseTuple does the work of getting the arguments from Python, making sure there are exactly two of them, and converting them to C doubles, storing their values in our local variables x and y. The argument “dd” is what indicates that two doubles are expected. The complete documentation for PyArg_ParseTuple is in the “Extending Python” manual.

Of course, things might not be as expected. If anything goes wrong, such as a call with three arguments, or a string argument where we expected a double, PyArg_ParseTuple will set the Python error status and return NULL. The Py_Try macro simply returns at this point in this event. If any of our extension functions return NULL, Python knows an error has occurred.

Having gotten our arguments x and y, we can call the real C is_equal, and finally return the answer as a Python integer. That is done with Py_BuildValue, which does the opposite job from Py_ParseTuple, producing a Python result given a C argument. Again, full documentation for Py_BuildValue is in the “Extending Python” manual.

```
static char cip1_tolerance__doc__[] =
“The tolerance used by is_zero”

static PyObject *
cip1_tolerance(PyObject *self, PyObject *args)
{
  /* Make sure there were no arguments */
  Py_Try(PyArg_ParseTuple(args, ““));
  /* Return the tolerance as a Python float */
  return Py_BuildValue (“d”, tolerance);
}
```

Nothing new here except the use of an argument "" to Py_ArgParseTuple to indicate no arguments.

```
static char cip1_set_tolerance__doc__[] =
“set_tolerance(d) sets the tolerance used by is_zero to ‘d’”
;

static PyObject *
cip1_set_tolerance(PyObject *self, PyObject *args)
{
  Py_Try(PyArg_ParseTuple(args, “d”, &tolerance));
  Py_INCREF (Py_None);
  return Py_None;
}
```

The mysterious

```
Py_INCREF (Py_None);
 return Py_None;
```

is the idiom for a routine that returns nothing. (Nothing is an object in Python, called None at the Python level and Py_None at the C level.) Python is garbage collected via reference counts on each object, so here we are returning an object, Py_None, and thus creating another reference to it in the system, so we have to increment its reference count first. Py_BuildValue has been taking care of the reference count on the returned value in our other functions. Be sure to read the section on reference counting, and the abstract.h interface, before you do anything fancier than our example.

Making the methods table

Next comes the crux of the matter. First is a table which connects together the names of the functions in Python, the actual routine in this module, and the documentation strings:

```
static struct PyMethodDef cip1_methods[] = {
  {"is_zero",cip1_is_zero,1,cip1_is_zero__doc__},
  {"tolerance",cip1_tolerance,1,cip1_tolerance__doc__},
  {"set_tolerance",cip1_set_tolerance, 1,cip1_set_tolerance__doc__},

  {NULL, NULL}/* sentinel */
};
```

Initializing the module

Finally, the piece that connects all this to Python. Its name is the name expected by Python for a module named cip1:

```
/* Initialization function for the module (*must* be called initcip1) */

static char cip1_module_documentation[] =
"Supplies function for determining if a point is nearly (0.0, 0.0)"
;
void
initcip1()
{
  PyObject *m;

  /* Create the module and add the functions */
  m = Py_InitModule4("cip1", cip1_methods,
                     cip1_module_documentation,
                     (PyObject*)NULL,
                     PYTHON_API_VERSION);

  if (PyErr_Occurred())
  {
    Py_FatalError ("can't initialize module cip1");
  }
}
```

Note how the call to Py_InitModule4 connects together the module name, its documentation, and its methods table.

Linking and Executing

How do we attach our extension to Python? On most platforms, this can be done dynamically. For example, here is a Makefile for a Solaris machine:

```
CFLAGS= -g -I. -I$(PYTHON)/include/Py

cip1module.so: cip1module.o is_zero.o
	ld -G cip1module.o is_zero.o -o $@

cip1module.o: cip1module.c

is_zero.o: is_zero.c is_zero.h
```

Now the resulting file cip1module.so is dynamically loaded into Python when the module is imported, perhaps by:

```
>>> import cip1
>>> print cip1.is_zero(1.0e-7, 2.0e-7)
1
>>> print cip1.is_zero.__doc__
is_zero (x, y) tests whether (x, y) almost equals (0.0, 0.0)
```

Adding Your Own Object Types in C

Making Cookies

It is possible to add new objects to the Python type system by using its object-oriented features. However, it is sometimes useful to make such extensions in C. This is usually done because of either a need to implement some operations most efficiently, as in the Numerical extension we discussed last issue, or to connect Python to some external resource, such as a database file or variables in some C sources.

Suppose that in our first example the file `is_equal.h` now contains a declaration for a certain kind of C struct, say:

```
struct cookie {
        char *flavor;
        double weight;
};
```

and a certain instance of it, say:

```
extern struct cookie w;
```

Again using modulator, we create an extension cip2module.c which will define a new kind of python object called a “cookie”. The module will contain two methods,

- cookie("flavor", weight) to create a new cookie object, and
- w() to return a cookie whose flavor and weight equal those of the w declared in is_zero.h.

Further, cookie objects will have two attributes, flavor and weight, which can be queries and set using the usual dot notation, and cookies will know how to represent themselves as strings, so that we can do things like:

```
>>> from cip2 import cookie, w
>>> c=cookie("vanilla", 12.0)
>>> print c
cookie ("vanilla", 1.200000e+01)
>>> print c.weight, c.flavor
12.0 vanilla
>>> c.weight=12.0
>>> c.flavor="butterscotch"
>>> print c
cookie ("butterscotch", 1.200000e+01)
>>> print w()
cookie ("chocolate", 2.000000e+00)
```

Note that this particular extension gives read-only access to the variable w. It is equally easy to create objects which are direct proxy objects for code variables.

Since more can go wrong in this example, we set up an error object so that there can be module-specific error objects returned in case of trouble.

The cip2 module

```
#include "Python.h"
#include "string.h"
#include "is_zero.h"
static PyObject *ErrorObject;

/* Define 2 macros for error handling:
     Py_Try(BOOLEAN)
         If BOOLEAN is FALSE, assume the error object has
              been set and return NULL

     Py_Assert(BOOLEAN,ERROBJ,MESS)
         If BOOLEAN is FALSE set the error object to
              ERROBJ, and the message to MESS

*/

#define Py_Try(BOOLEAN) {if (!(BOOLEAN)) return NULL;}
#define Py_Assert(BOOLEAN,ERROBJ,MESS) {
            if (!(BOOLEAN)) PyErr_SetString((ERROBJ), (MESS)); \
                                                  return NULL;} \
                                          }
```

Now we come to the fun part: declaring a new kind of object. This is done by using a simple recipe. We declare a simple struct that begins with the PyObject_HEAD macro:

```
/* Declarations for objects of type cookie */

typedef struct {
  PyObject_HEAD
    double weight;
    char *flavor;
} cookieobject;

staticforward PyTypeObject Cookietype;
```

Next comes a method table. Our particular cookie objects have no methods, but most objects would have some. Constructing them is the same as module methods except that now you use the self argument as a pointer to the current instance of the object.

```
static struct PyMethodDef cookie_methods[] = {
   {NULL, NULL}/* sentinel */
};
```

Creating new cookies

Next it is usual to write at least one function which will create new instances of our object from within C. Later we write a Python interface to this function.

```
static cookieobject *
newcookieobject (char *f, double w)
{
  /* Create a new cookie of weight w and flavor f */
   cookieobject *self;

   Py_Assert (strcmp (f, "") != 0, ErrorObject, "Unacceptable cookie flavor");
   Py_Try((self = PyObject_NEW(cookieobject, &Cookietype)));

   self->weight = w;
   self->flavor = (char *) malloc (strlen(f) + 1);
   Py_Assert (self->flavor != (char *) NULL, ErrorObject, "out of memory");
   strcpy (self->flavor, f);
   return self;
}
```

Next comes a series of routines that implement operations on cookies. These are for operations that are triggered by the Python interpreter. For example, we must supply a “dealloc” routine to clean up after an object when the object is dying:

```
static void
cookie_dealloc(cookieobject *self)
{
  /* release memory and die */
   free(self->flavor);
   PyMem_DEL(self);
}
```

The remainder of these routines can either be implemented or not. If you don't implement a certain behavior, say subscripting, then an attempt to do that operation will result in an error which informs you that cookies don't do that:

```
>>> print c
cookie ("chocolate", 2.000000e+00)
>>> print c[0]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: unsubscriptable object
```

By implementing the correct routines, you can choose to make objects can be added, multiplied, subscripted, sliced, mapped, etc. In particular you can supply up to three routines that are invoked in different contexts to produce a printed representation of an object. For cookies, for example, we might do this, using the function PyString_FromString which makes a Python string object which is a copy of a C string:

```
static PyObject *
cookie_repr (cookieobject *self)
{
   PyObject *s;
   char temp[500];  /* this isn't totally safe, it is an example */
   sprintf (temp, "cookie (\"%s\", %e)", self->flavor, self->weight);
   s = PyString_FromString (temp);
   return s;
}
```

Implementing structure-like access

Two routines "getattr" (get attribute) and "setattr" (set an attribute to a value) are invoked when the dot notation is used to reference attributes of an object, as in:

```
a_cookie.weight = 3.0
print a_cookie.flavor
```

If there are only a few attributes one can do this by simple brute force as below.

```
static PyObject *
cookie_getattr(cookieobject *self, char *name)
{
 /*  a_cookie.flavor, a_cookie.weight */
  if (strcmp (name, "flavor") == 0)
    {
      return Py_BuildValue ("s", self->flavor);
    }
  if (strcmp (name, "weight") == 0)
    {
      return Py_BuildValue ("d", self->weight);
    }
  return Py_FindMethod(cookie_methods, (PyObject *)self, name);
}

static int
cookie_setattr(cookieobject *self, char *name, PyObject *v)
{
  /* a_cookie.weight = w or a_cookie.flavor = "flavor" */
  char *f;
  double w;
  if (strcmp (name, "flavor") == 0)
    {
      Py_Try(PyArg_Parse (v, "s", &f));
      free (self->flavor);
      self->flavor = (char *) malloc (strlen(f) + 1);
      strcpy (self->flavor, f);
      return 0;
    }
  if (strcmp (name, "weight") == 0)
    {
      Py_Try(PyArg_Parse (v, "d", &w));
      self->weight = w;
      return 0;
    }
  return -1;
}
```

The connection between the Python interpreter’s grammar operations and our “cookie” object is made with a table. This table can contain a pointer to subsidiary tables which implement the number, sequence, or mapping behaviors, but in our case these are not present.

```
static char Cookietype__doc__[] =
"Cookies have a flavor and weight"
;

static PyTypeObject Cookietype = {
  PyObject_HEAD_INIT(&PyType_Type)
  0,                            /*ob_size*/
  "cookie",                     /*tp_name*/
  sizeof(cookieobject),         /*tp_basicsize*/
  0,                            /*tp_itemsize*/
  /* methods */
  (destructor)cookie_dealloc,   /*tp_dealloc*/
  (printfunc)cookie_print,      /*tp_print*/
  (getattrfunc)cookie_getattr,  /*tp_getattr*/
  (setattrfunc)cookie_setattr,  /*tp_setattr*/
  (cmpfunc)0,                   /*tp_compare*/
  (reprfunc)cookie_repr,   /*tp_repr*/
  0,                       /*tp_as_number*/
  0,                       /*tp_as_sequence*/
  0,                       /*tp_as_mapping*/
  (hashfunc)0,             /*tp_hash*/
  (ternaryfunc)0,          /*tp_call*/
  (reprfunc)cookie_str,    /*tp_str*/

  /* Space for future expansion */
  0L,0L,0L,0L,
  Cookietype__doc__        /* Documentation string */
};
```

This completes our definition of a cookie object. We could at this point declare more kinds of objects in the same way.

Module methods

Finally, we come to the methods of the cip2 module itself, in this case a routine cookie to create cookies that will be called from Python, and a function w() which will return the value of the C object w which is of type struct cookie.

```
static char cip2_cookie__doc__[] =
"cookie(flavor, weight) = new cookie"
;

static PyObject *
cip2_cookie(PyObject *self, PyObject *args)
{
  /* self is not used */
  PyObject *result;
  char *flavor;
  double weight;

  Py_Try(PyArg_ParseTuple(args, "sd", &flavor, &weight));
  result = (PyObject *) newcookieobject (flavor, weight);
  return result;
}

static char cip2_w__doc__[] =
"w() = value of the struct cookie w in is_equal.c"
;

static PyObject *
cip2_w(PyObject *self, PyObject *args)
{
  /* self is not used */
  PyObject *result;
  char *flavor;
  double weight;

  Py_Try(PyArg_ParseTuple(args, ""));
  result = (PyObject *) newcookieobject (w.flavor, w.weight);
  return result;
}

static struct PyMethodDef cip2_methods[] = {
  {"cookie",cip2_cookie,1,cip2_cookie__doc__},
  {"w",  cip2_w,  1,           cip2_w__doc__},

  {NULL, NULL}/* sentinel */
};
```

```
static char cip2_module_documentation[] =
"cookie (flavor, weight) creates a new cookie"
;

void
initcip2()
{
  PyObject *m, *d;

  /* Create the module and add the functions */
  m = Py_InitModule4("cip2", cip2_methods,
                     cip2_module_documentation,
                     (PyObject*)NULL,
                     PYTHON_API_VERSION);

  /* Add some symbolic constants to the module */
  d = PyModule_GetDict(m);
  ErrorObject = PyString_FromString("cip2.error");
  PyDict_SetItemString(d, "error", ErrorObject);

  /* Check for errors */
  if (PyErr_Occurred()) {
    Py_FatalError("can't initialize module cip2");
  }

}
```

Extending Python with C++

Using C++ to write extensions instead of C is of course easy using the C external linkage facilities of C++. However, in general C++ likes to be the “main” code, because of the static constructor problem (C++ requires that certain actions take place both before and after the main program executes, and if the main program isn’t written in C++, or if the module is loaded dynamically, this doesn’t happen.) Therefore, unless you are sure that no global or static objects with constructors will be used, and that no global or static variables will be initialized by function calls (e.g., new and malloc), our best advice is to embed Python into your C++ program rather than extend Python. The effect is the same, for after you have initialized you call the Python interpreter and let it take over. One simple way to do this is to compile the Python main() function in C++, add into the config.c file the module you have written, and link with the Python libraries libModules.a, libPython.a, libObjects.a, and libParser.a. All these libraries and config.c are generated during the installation of the Python interpreter.

A more interesting challenge is setting up parallel class structures between C++ and Python. Various approaches are possible, depending on the relations between the Python classes (or object types) and the C++ classes.

- James C. Ahlstrom presented a paper (http://www.python.org/workshops/1995-12/papers/ahistrom1.html) at the third Python Workshop, describing a way to derive Python classes from an existing C++ class library. The Python classes inherit C++ methods from parent classes and Python can call these methods. Python classes can also have their own methods, which override C++ virtual functions and can be called by Python or by C++.
- David Redish posted in the Python newsgroup (news:comp.lang.python, March 1996) C++ lessons that he had learned in writing a large scale neural simulation in Python / C++. In his simulation, new Python object types are implemented in C++ classes instead of C structures as described earlier.
- At LLNL, we use Python to control and monitor, in a non-intrusive way, a physics application written in C++. A new Python object type was implemented as a shadow class over the C++ classes. Through this shadow class, the Python interpreter can gain access to the public members of the C++ classes. The C++ classes are not aware of the existence of the shadow classes, so that our users can also run the C++ application in a stand-alone fashion.

This is definitely an active research area so be sure to check the newsgroup and website for the latest ideas.

Extending Python with Fortran

Since you can connect C to Fortran, one approach to connecting Python and Fortran is to simply use your favorite techniques for connecting the C extension you write to the Fortran to which you wish to connect. However, one can do more because Fortran has such a limited range of data types, so that it should be possible to automate this process more or less.

James Fulton has produced a tool using one approach. Called FIDL (Fortran Interface Definition Language), the tool processes a simple descriptive language which describes Fortran interfaces into the actual C extension that provides the Python-to-Fortran connection.

A similar approach that might help is the new tool SWIG discussed above. It might need to be combined with your favorite C / Fortran interface method.

Conclusions

It should be emphasized that most of the coding presented in our examples was automatically generated. In practice, we always either use an automatic generator or begin with an existing extension when we wish to write a new one.

We're excited by the many extensions to Python that have already been created. These include facilities for graphics, databases, Web programming, audio and image processing, and numerics.