Building a Programmable Interface for Physics Codes Using Numeric Python

T.-Y. B. Yang
P. F. Dubois
Z. C. Motteler

This paper was prepared for submittal to the
Fourth International Python Workshop
Livermore, CA
June 3-6, 1996

April 16, 1996

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.
DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.
DISCLAIMER

Portions of this document may be illegible in electronic image products. Images are produced from the best available original document.
Building a Programmable Interface for Physics Codes Using Numeric Python

T.-Y. B. Yang, P. F. Dubois, and Z. C. Motteler

Lawrence Livermore National Laboratory

1.0 Introduction

Our goal is to create a “plug and play” programmable interface that gives the users flexibility to run the applications in the way appropriate for their physics problems, and also allows the code developers to query and to change, from the Python interpreter, variables buried in the physics modules, which for speed reasons are implemented in C, C++, and Fortran. Some of the philosophy behind such programmable applications was presented elsewhere. The programmable applications with Python interface, from the bottom up as shown in Fig. 1, consists of the following four levels:

1. Compiled packages: These are source codes and libraries that already exist or are to be written in compiled languages because of efficiency reasons, for example physics and mathematics packages, data storage packages, graphics packages and so on.

2. Python extensions in compiled languages: These are Python extensions written mainly in C and C++ possibly with some auxiliary routines written in other compiled languages. These basically serve as “glue” between Python and compiled packages.

3. Program Author Scripts: These are Python modules (*.py) containing wrapper classes and functions which are the main user interfaces that hide the implementation details and give the users an physically intuitive and universal view of the applications. As an example, let’s suppose there are several hydrodynamics packages, and each of which has its own data structure for a physical quantity, say density. The wrapper classes and functions allow the users to query and set the density in an intuitive way that does not depend on which package is chosen.

4. User scripts: These are Python scripts that users write, using the user interfaces provided in the program author scripts. The three lower levels are organized into modules. There may be more than one modules that serve the same function but with different underlying implementations, e.g., two hydrodynamics modules with different algorithms. Well implemented wrapper classes and functions in the program author scripts allow users to do “plug and play” with different choices of underlying modules without major changes in the user scripts.
In the following sections, we will discuss two applications, one for a physics module and the other for a data-storage module, that are organized in the way described above. A graphics module that provides interface with Narcisse, a graphics package developed in France, will be discussed in Z. C. Motteler’s paper presented in this meeting.

### 2.0 A Physics Module

We have implemented a Python interface for a C++ application. Users can run the application interactively at the Python prompt as follows:

```python
>>> from hydro_class import HYDRO
>>> hyd = HYDRO()
>>> hyd.initialize()
>>> hyd.cycle()
>>> hyd.density()
>>> hyd.ncells = 20
```

Here, the module hydro_class.py is a program author script which imports a Python extension module (hydromodule.so) and defines a Python class called HYDRO. The following is an excerpt from hydro_class.py:

```python
from Numeric import *
import hydro

cell_list = hydro.phy_var.cell_list
```

---

April 16, 1996
class HYDRO:
    __states__ = hydro.phy_var
    def __getattr__(self, name):
        try:
            return getattr(self.__states__, name)
        except AttributeError:
            return getattr(hydro, name)
    def __setattr__(self, name, value):
        setattr(self.__states__, name, value)
    density(self):
        fnc = lambda i: cell_list[i].av0()[6]
        return array(map(fnc, range(self.ncells)), 'd')

Notice that the object hydro.phy_var, to which the attribute __states__ is assigned, is an instance of an object type cplus_var defined in hydromodule.so. The purpose of this object is to allow queries and changes of the C++ global variables to be done from the Python interpreter. The attributes of the object hydro.phy_var (e.g., 'cell_list' and 'ncells') all correspond to some global variables in the underlying C++ code.

The C++ extension module hydromodule.so is the "glue" between Python and the compiled C++ source code. For each C++ function that is to be called from the Python interpreter, a glue-method is defined and inserted into the method table of hydromodule.so.

Query and changes of the global variables in the C++ source are handle by the object hydro.phy_var as mentioned earlier. The getattr and setattr methods of the cplus_var object type, of which hydro.phy_var is one and the only one instance, look up a Python dictionary varlist to gain access of the C++ global variables. When the module hydro is imported, the dictionary varlist is created and filled with Python objects each contains informations (e.g., name and address) of a particular C++ global variable.

The Python objects that are stored in the dictionary varlist are of the `derived types` of a particular type called var_item. The following is how the var_item type is defined:

typedef struct {
    PyObject *(*getatrr) (PyObject *self);
    PyObject *setattr(PyObject *self, PyObject *v);
} Var_att_type;

#define VAR_HEAD
    PyObject_HEAD
    void *address:
    Var_att_type *var_type:
    char *comments:

typedef struct {

April 16, 1996
We have taken the liberty to use the term ‘derived types’ in the same sense as calling all the Python object types ‘derived types’ of the PyObject type, since pointers to other object types can always be casted into (*PyObject) and be treated as such. Notice that each ‘derived type’ of var_item type has a unique var_type member. This allow the getattr and setattr methods of the cplus_var object type to dispatch their jobs without knowing the exact type of the object they retrieve from the dictionary varlist.

An example of ‘derived types’ of var_item type is array_item defined as:

typedef struct {
    VAR_HEAD
    char array_type;
} array_itemobject;

When creating a new array_item object for a C++ global array, the address member of the array_item object points to a new PyArray object. Created by calling the function PyArray_FromDimsAndData of the Array module, the data member of the new PyArray object points to the address of the global C++ array that we desire to query and change from the Python interpreter. For the getattr method of the cplus_var object type to correctly dispatch its job, var_type.getattr member of a array_item object points to a function which returns the PyArray object pointed to by the array_item object’s address member.

Another example of ‘derived types’ of var_item type is scalar_item whose address member stores the address of a scalar global variable of type int, double or float. The var_typesetattr member of a scalar_item object points to a function which change the value of the global variable pointed to by the scalar_item object’s address member, so that

>>> hyd.ncells = 20

changes the value of the global int variable ‘ncells’ to 20, instead of re-pointing the ‘ncells’ attribute of ‘hyd’ to a Python integer of value 20.

We have also implemented a shadow object type for C++ classes in order to gain access to the public member functions of C++ classes. There are also ‘derived types’ of var_item type for global C++ pointer variables pointing to instances and arrays of instances of C++ classes. When the name of a global pointer variable associated with a C++ object is evoked from the Python interpreter, a shadow object corresponding to the C++ object is returned. The attribute cell_list of hydro.phy_var is an instance of such types, corresponding to an array of C++ objects. The implementation of the method density, shown on page 3, makes use of cell_list.

All the above are done in a non-intrusive way, that is the C++ source code is not aware of the existences of all the levels above it. This allows our users to run the application either in the Python mode or in a stand-alone mode.
3.0 A Data-Storage Module

PDB is a portable scientific-data-management system created as part of the PACT project by Stewart Brown and his group at LLNL. A Python extension has been written and linked with the PACT libraries to store and retrieve data in the PDB format from the Python interpreter. In terms of the levels shown in Fig. 1, the compiled-package level consists of the PACT libraries, the Python-extension level is a C extension, pypdbmodule.so, and the program-author-script level consists of two Python modules, PR.py and PW.py, for reading and writing, respectively, PDB files.

The C extension pypdbmodule.so is mainly a "glue" between Python and the PACT libraries. The module defines two new object types PDBfile and pseudostruct and has methods to open a PDB file and create a PDBfile object associated with the file. The PDBfile type has attributes typically expected of a file object such as read, write, and close. In addition, it has methods for the directory structure in the PDB file.

The pseudostruct type is implemented so that when a C-struct stored in a PDB file is retrieved its members are accessible to the Python interpreter. When an application is fully integrated into PACT, an arbitrary C-struct can be stored with pointer-following done by the library. This, however, requires that the retrieving packages have intimate informations about the C-struct, for the members of the C-struct can be pointers pointing to other C-structs. For the purposes of general usage, such capability is not fully implemented in pypdbmodule.so.

The PR.py module defines a Python class PR which is a wrapper class for PDBfile objects created for data retrieval. When an instance X of PR class is created in association with a readable PDB file, the variables stored in the file can be regarded as attributes of the instance X. When a variable, say Y, is first referred to by the Python command ‘X.Y’, the variable Y is retrieved from the PDB file as a Python object and is stored in X’s dictionary besides being returned to the Python interpreter. Subsequent occurrences of the command ‘X.Y’ return the variable from X’s dictionary instead of the PDB file.

The PW.py module defines a Python class PW which is a wrapper class for PDBfile objects created for writing data to a PDB file. For an instance X of PW class associated with a PDB file opened for writing, the first appearance of a Python assignment to an attribute of X, say Y in X.Y=2, causes an integer variable named Y with value 2 to be stored into the PDB file. Subsequent assignments to X.Y in the same directory have no effect other than causing a warning. Assignments to other types which are acceptable to the PDB file work in the same way.

4.0 Summary

With its portability, ease to add built-in functions and objects in C, and fast array facility among many other features, Python, as our experiences showed, proved to be an excellent language for creating programmable scientific applications. In addition to the two modules presented here, there are also other progresses at LLNL in using Python. For example,
Python interfaces are being developed for at least three graphics packages, and Python interpreter and applications have been built on distributed platforms such as meiko and Cray T3D. Much more works still need to be done, and we will report our further progress in the future workshops.

Acknowledgments

The work described here was performed under the auspices of the U.S. Department of Energy by the Lawrence Livermore National Laboratory under Contract W-7405-ENG-48.

References

