Astronomical Data Analysis Software and Systems IX
ASP Conference Series, Vol. 216, 2000
N. Manset, C. Veillet, and D. Crabtree, eds.

Scientific Computing with Python

D. M. Beazley

Department of Computer Science, University of Chicago, Chicago, IL 60637

Abstract. Scripting languages have become a powerful tool for the construction of flexible scientific software because they provide scientists with an interpreted programming environment, can be easily interfaced with existing software written in C, C++, and Fortran, and can serve as a framework for modular software construction. In this paper, I describe the process of adding a scripting language to a scientific computing project by focusing on the use of Python with a large-scale molecular dynamics code developed for materials science research at Los Alamos National Laboratory. Although this application is not related to astronomical data analysis, the problems, solutions, and lessons learned may be of interest to researchers who are considering the use of scripting languages with their own projects.

1. Introduction

As a developer of scientific software, one of the biggest obstacles is the problem of how to make scientific software flexible, easy to use, simple to maintain, and easy to adapt to the increased size and complexity of new problems. Of course, there are many reasons for wanting these qualities–not the least of which is that they allow scientists to spend less time fighting software and more time doing science.

Unfortunately, the current state of scientific software is less than ideal. First, increased computing power has allowed scientists to tackle problems of an unprecedented scale and complexity. However, as a side-effect, the software needed to perform such tasks has also become large and complex. Second, it is commonplace for scientists to work with a wide variety of systems ranging from simulation codes, data analysis packages, databases, visualization tools, and home-grown software–each of which presents the user with a different set of interfaces and file formats. As a result, a scientist may spend a considerable amount of time simply trying to get all of these components to work together in some manner (e.g., massaging data with awk scripts, transferring data between machines, etc.). Finally, efforts to rewrite scientific software using modern software engineering principles have only yielded mixed results at best. For one, scientists are generally reluctant to abandon existing software. Second, such efforts are notoriously vulnerable to second-system effects in which the designers try to construct a system that is so general purpose and broad in scope that it is either impossible to realize an implementation or the system is too general to address the idiosyncratic demands of a specific problem.

Despite these difficulties, one of the more promising technologies to emerge in recent years is the use of common scripting languages such as Python, Perl, and Tcl as a tool for creating better scientific software (Python, 1999; Perl, 1999; Tcl, 1999). In this paper, my intent is to shed light on the scripting approach by focusing on the use of the Python programming language with a specific scientific application over a four-year period. In particular, I hope to illustrate how scripting languages are well-suited to scientific software projects and how they provide evolutionary development path for addressing important software engineering problems such as reliability, components, reuse, and project management. Although the discussion will focus on Python, I wish to emphasize that the principles apply to scripting languages in general.

2. The Nature of Scientific Software Development

Consider the current spectrum of scientific software. On one extreme, there are shrink-wrap packages such as IDL and MATLAB. On the other, there are home-grown programs written to address a specific class of problem (often created by the researcher). Even though the shrink-wrap packages are more polished, it is more likely for the interesting scientific content to be contained within the home-grown software. For instance, a special purpose program might be written to solve a specific class of partial differential equation whereas a package like IDL would only be used to make plots of the results.

Most new scientific software is created when a small group of researchers decide to address a new computational problem. Initially, the goal is to solve a very specific problem defined by a particular set of equations, initial condition, boundary condition, and numerical technique. In many cases, the problem can be captured by writing a small program in C or Fortran with some assistance from Numerical Recipes and some programming libraries.

Initially, programs start small. However, if they are successful, they may be expanded to solve a variety of closely related problems. To handle these variations, additional flexibility is typically added to the system in the form of a minimal user interface that either queries the user for various problem parameters or requires the user to supply a series of command line options. Unfortunately, as more features are added in this manner, the scientist may become overwhelmed by a proliferation of command line options and cryptic input scripts.

When a program reaches this stage, the developers may decide to overhaul its structure and user interface. More often than not, this tends to result in a home-grown command interpreter that allows users to set parameters and execute simple commands. Of course, this is not a surprise. For one, implementing a simple command interpreter is not a very difficult task for a reasonably competent programmer. Second, such interfaces closely emulate those of more polished scientific software.

Unfortunately, this usually represents the final stage of development for many projects. For one, going beyond a simple interpreter involves a set of programming skills not widely held by scientific programmers. For instance, turning a command interpreter into a full programming language requires some knowledge of parser construction and language design techniques. Similarly, building a graphical user interface requires an equally complex set of programming skills. A more likely scenario however, is that the scientists do not see the payoff of adding these features being worth the investment in development time that would be required. As a result, it is no surprise that a scientist's toolbox is filled with a variety of programs, each with their own quirky command language.

With this said however, bad user interfaces are only part of the problem. Instead, the key point is that most useful scientific software starts small and is grown in a relatively piecemeal fashion over a long period of time. In general, there is no formal software design other than a general statement about the type of problem that is to be solved. Furthermore, such projects are rarely started with the goal of creating a general purpose software package–although a successful program may become more general purpose over time.

Because of these issues, efforts to improve scientific software face a number of significant challenges. First, scientists are rarely willing to abandon existing software–especially if they are familiar with its operation and are confident with the accuracy of its results. Second, efforts to impose formal software design on an inherently unstructured process are likely to fail (even if a design could be devised, it is unlikely that the designers would be able to conceive of every possible scientific problem to arise in the future). Finally, solutions need to emphasize simplicity and continue to encourage the kind of experimentation and development in which scientific software is created in the first place.

3. Python

Python is an interpreted object-oriented programming language that is starting to receive considerable attention in scientific applications (Python, 1999). This is because Python, and scripting languages in general, represent a next logical step for many scientific projects (Dubois 1994). First, Python provides an interpreted programming language that can be viewed as an extension of the simple command languages already used by scientific programs. Second, Python is easily integrated with software written in other languages. As a result, it can serve as both a control language for driving existing programs as well as a glue language for combining different systems together. Finally, Python provides a large collection of third party modules, an established user base, and a variety of documentation in the form of books and online references. For this reason, one might view it as a highly polished and expanded version of what scientists often try to accomplish when writing their own command interpreters.

4. A Molecular Dynamics Application

To illustrate the use of Python, the remainder of this paper concentrates on experience gained by adding Python to the SPaSM short-range molecular dynamics code developed at Los Alamos National Laboratory (Beazley & Lomdahl 1994). This application was originally developed in 1992 for performing large-scale molecular dynamics simulations of materials in three-dimensions. The primary problems of interest included crack propagation, dislocation dynamics, friction, and ion-implantation. Initially, SPaSM was written for the 1024 processor Connection Machine-5 massively parallel supercomputer. Later versions were also adapted to run on the Cray T3D, multiprocessor Sun servers, and single processor workstations. Although considerable success was achieved in areas of performance tuning and scalability, the system was too difficult to use in practice. Specifically, the following problems became obstacles to achieving the scientific goals of the project:

1. None of our data analysis and visualization tools were capable of processing the resulting data (consisting of the trajectories of up to 100 million interacting atoms);
2. The user interface, composed of command line options and a simple command processor, was too inflexible and difficult to use. For example, setting up a large problem often involved a time-consuming process of modifying batch processing scripts, submitting a few smaller tests jobs, transferring datafiles to a barely function data analysis tool, and taking an educated guess at the appropriate parameters for a larger run;
3. The system had grown to be a monolithic package of approximately 25000 lines of ANSI C code. As a result, new users were reluctant to make modifications and even when changes were made, it was common for them to be made to a local copy. As a result, the incorporation of bug-fixes and new features became a nightmare of tracking down all of the conflicting versions of the software and performing a merge.

The decision to add Python was driven entirely by these concerns. First, a high-level interpreted language would provide an excellent mechanism for problem specification and control–possibly eliminating much of the need to make modifications to the C source. Second, to fix data analysis problems, we planned to integrate a data analysis/visualization capability directly into the simulation code (thereby eliminating the problem of offloading data to underpowered third-party tools). However, in order to make this work, a very powerful and simple user interface would be required. Therefore, we viewed Python as a way to build a system that, on the surface, worked a lot like IDL or MATLAB. Finally, Python was seen as a reasonable choice in terms of maintainability and performance. For instance, it was portable to a wide variety of systems, didn't introduce a significant amount of conceptual overhead to the users, and could be added to our existing code.

5. Building a Python Interface

Before describing the details of adding Python, it is important to understand that most programs can be broken into two components; a collection of functions that perform the actual work of the system, and a main program that is responsible for reading input parameters and providing some kind of high-level control or user interface. As a result, there are two ways one can introduce Python to the system. First, Python can be used to provide a wrapper around the original program interface (in this case, Python behaves much like a shell-script language). Alternatively, the Python interpreter can be interfaced directly with the underlying functions of the application in which case the functions become a collection of new Python "commands." In our application, we adopted the latter approach.

Preparing an application for a Python interface requires two steps: Removing the main program and optionally the old user-interface code, and repacking all of the underlying functions into a programming library. In many cases these steps require nothing more than a few changes to a few makefiles. Furthermore, removal of the main program is often enough to eliminate the old user-interface even if portions of this interface exist in the library.

Once in library form, the next step is to write a collection of wrappers that provide the glue between the Python interpreter and the underlying functions. If written by hand, a typical wrapper might look similar to the following:

```
PyObject *wrap_foo(PyObject *self, PyObject *args) {
    int    arg1;
    double arg2;
    double result;
    if (!PyArg_ParseTuple(args,"id", &arg1, &arg2)) return NULL;
    result = foo(arg1,arg2);   /* Call the real function */
    return Py_BuildValue("d",result);
}
```

In principle, a separate wrapper must be written for each function in the library–a task that is extremely time-consuming and prone to error for a large library. Fortunately, this task can be performed automatically using SWIG, an automatic wrapper generation tool we have developed (Beazley 1998; SWIG, 1999). Using SWIG, the contents of the library are described using familiar ANSI C syntax (as found in a header file). For example:

```
%module spasm
%{
#include "spasm.h"
%}
void ic_shock(int nx, int ny, int nz, double vel, double width,
           double gap, double temp, double r0, double cutoff);
int  timesteps(int nsteps, int en, int on, int cn);
void set_boundary_periodic();
void init_lj(double epsilon, double sigma, double cutoff);
void set_path(char *);
...
```

Once specified in this manner, the interface is compiled by SWIG into a collection of C wrapper functions. These wrapper functions are then compiled and linked against the original C library into a shared library (or dynamic link library). Now, the resulting shared library can be loaded into the Python interpreter. For example:

```
% python
Python 1.5.2 (#1, Sep 19 1999, 16:29:25)  [GCC 2.7.2.3] on linux2
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>> import spasm
>>> spasm.init_lj(1.0, 1.0, 2.5)
>>> spasm.set_path("/sda/sda1/beazley/MD")
>>> spasm.ic_shock(15,15,50,8.5,0.3333,0.10,0.01,1.09,2.5)
...
```

The key observation is that the original program is generally not modified during the construction of the Python interface—rather it is only repackaged. In addition, by using automated tools such as SWIG, the process of building the interface can be extremely rapid–often only requiring less than 30 minutes of work. Finally, the resulting Python interface does not change the way in which the program operates. Rather, it merely exposes the underlying functionality to an interpreter. Thus, instead of calling individual functions from C, they can now be invoked interactively from the Python interpreter much as one might execute a program from within a debugger.

6. Dead Code Elimination

Once the Python interface has been constructed, an application may still include a substantial amount of code related to its previous user interface. Since this code is no longer executed, it is often possible to either eliminate it entirely or to move it out of the library into a separate set of files. This not only reduces the size of the original application, it separates the real functionality of the application from its user interface. As a result, the application appears much more like a programming library (or software component) than a monolithic application. In the case of SPaSM, several thousand lines of code were eliminated over a period of a few months.

7. Improving Reliability

When interfaced with Python, users gain the ability to interactively issue commands in any order and at any time. Unfortunately, this can also cause a number of run-time problems including program crashes and incorrect results. Most of these problems are due to the fact that the old program was never written to be used in such a highly flexible manner. For example, the main program might have been written to perform a precise sequence of initialization steps or to guide the computational process in a tightly controlled manner. The most common sources of these problems are as follows:

1. Execution order dependencies. A function may only operate correctly if another function has executed first (common in program initialization);
2. Re-entrant functions. Certain functions such as memory management or initialization of I/O may only be callable once;
3. Argument value checking. Some functions may impose implicit constraints such as only expecting positive or negative argument values.

To eliminate these problems, it is necessary to modify either the Python interface or the original application by introducing additional checking code. In almost all cases, this is easily accomplished by simply introducing additional status variables. For example:

```
void init_memory() {
   static int initialized = 0;
   if (initialized) error("Already initialized memory!");
   initialized = 1;
   ...
}
```

In our case, these types of errors were eliminated over a period of several months. In many cases, status variables and simple checks were added as shown. In other cases, functions were actually modified to execute properly in a more flexible environment (by allowing reentrancy for instance).

8. Exception Handling

Closely related to the problem of improving reliability is the problem of proper error handling. More often than not, scientific software has a weak notion of error handling–doing nothing more than printing a message and calling the system `exit()` function to abort its operation. Python, on the other hand, provides full support for exception handling. Thus, it is often desirable to modify the application so that it propagates errors back to the interpreter instead of simply terminating execution. In the case of SPaSM, error handling was added to the system in stages. First, functions were modified to print a message and return immediately in the event of an error. Later, an enhanced exception mechanism was introduced in which C functions could effectively “throw” an error that would be propagated back to the interpreter in the form of a Python exception.

9. Improving Modularity

Since Python loads extensions in the form of shared libraries, the Python interpreter is completely decoupled from the underlying application (in the sense that they are not linked together as a single executable). Furthermore, the dynamic loading mechanism allows any number of extension modules to be loaded simultaneously. As a result, it is possible to break a monolithic system into a collection of smaller subsystems–each compiled into its own library module.

In the case of SPaSM, the system was gradually split into a collection of smaller modules over a period of two years. In the current system, a small library provides the core of the application while problem-specific modules provide the implementation of specific experiments. This scheme has a number of advantages. First, instead of working with the entire application, users can focus on a specific module consisting of only be a few thousand lines of code. Second, by breaking the system up into small modules, compilation time is greatly reduced–making it easier for users to incorporate changes. For example, recompiling a module might only take ten seconds whereas recompiling the entire monolithic application might require ten minutes. Finally, by setting up a repository of modules, it is easier to maintain and manage the software. For instance, a bug-fix to one of the modules can be automatically propagated to all of the users by simply copying it to the repository.

10. Retrofitted Object Orientation

Object-oriented programming techniques are becoming increasingly common in scientific projects because they provide programming abstractions that make it easier for developers to think about organization of data and the manner in which different subsystems interface with each other. Where this pays off is in describing the high-level functionality of various pieces of a system. For example, a package to solve partial differential equations might define a C++ class that describes the set of operations a "solver" is supposed to implement. Specific solvers are then derived from this class to implement specific solvers. Ideally, this scheme allows any type of solver to simply be plugged into the system without making changes to other parts of the code.

Unfortunately, most existing applications are unlikely to be written in such an object-oriented manner or to use an OO language for that matter. However, by interfacing these applications with Python, it is possible to use the object-oriented features of Python to retrofit a high-level object-oriented interface onto an application. Tools such as SWIG are able to repackage C structures and functions into a class-like interface. Furthermore, by writing additional Python code, one can provide many of the benefits of object-orientation without the nightmare of rewriting everything in C++.

Many aspects of our system were modified to appear "object-oriented" from the Python interface. However, only minor changes to the underlying C implementation were required.

11. Systems Integration

Perhaps the most useful part of adding Python to an application is the opportunity it provides for integrating legacy code with new systems. One of our initial goals was to add a remote data analysis and visualization capability. To do this, we took a small parallel graphics library we had developed and combined it with some simple visualization functions that knew how to extract data directly from the simulation code. In addition, a small amount of network code was added to send images back to the user's workstation. All of this code was then compiled into a separate Python module that could be loaded whenever a user wanted to look at their data (Beazley & Lomdahl 1997).

Although simple, this integration revolutionized the use of our application. First, by moving analysis tasks to the machines upon which simulations were performed, tasks that used to take 3 or 4 hours on a high-end SGI graphics workstation could now be performed in only 10 seconds. Second, this eliminated almost all of the problems associated with dumping data to files, offloading it to another machine, massaging the data format, and so forth. Finally, this integration allowed the code to be used in an entirely different manner than the traditional batch-job. For example, users could use the system to interactively set up large problems, step through parts of a simulation, look at intermediate results, and use the analysis system to perform diagnostics and verification of physical models, boundary conditions, and initial conditions.

The other important aspect of systems integration is that Python provides a wide range of built-in and third party modules. Most of these can be used in addition to the scientific modules created for our application. For example, one feature we added later was a simple web-server that allowed users to remotely connect to a running simulation from a browser and query its current status. The web-server used a variety of existing Python modules for socket programming and additionally interfaced with the visualization module we had already written. In total, the web-server was written entirely in Python, involved only 150 lines of code, and was written in a single afternoon by one person. Yet, this simple addition is enough to allow users to remotely monitor running simulations from any machine on the Internet–a feature not previously envisioned by the users.

12. Performance Issues

To many, the thought of adding an interpreted language to their application raises serious performance concerns–both in terms of resource utilization and runtime performance. In our case, the addition of the Python interpreter had only a marginal impact on the size of the program. In addition, all of the serious number crunching in our application was still implemented in C and consumed most of the CPU cycles. Therefore, the fact that high-level control was being performed by an interpreter had little if any impact (timing measurements reveal a statistically insignificant 0.1% increase in execution time).

As for the performance impact in general, much depends on the degree of granularity provided in the Python interface. Clearly, writing the inner loops of computationally intensive operations in Python would be a bad idea as would writing the entire application in Python. However, given the ease by which Python can interface with C, computationally expensive operations can always be implemented in a compiled language when necessary.

13. Limitations

Although the use of Python has worked well for our project, there are a number of limitations to keep in mind. First, changing a system to rely upon shared libraries, dynamic loading, and an interpreter requires a set of skills that is likely to confuse users of more traditional software. For example, the process of creating shared libraries is still highly non-portable and a bit of an art-form. Second, users may find it difficult to understand how the code is structured and where things are defined. Finally, some applications are particularly hard to script due to unusual argument usage or reliance upon advanced language features such as C++ templates and overloaded functions. Although it is possible to build a Python interface to such applications, it may require substantially more work than has been described here.

14. Conclusions

Python has proven to be an invaluable tool for our application because it has greatly improved the way in which we develop and use scientific software. Furthermore, these improvements were made in an incremental fashion from an existing base of code. In fact, at no time was the code ever unusable during this transition process. Finally, as a result of these efforts, users now typically perform more simulations in a month than in the first three years of our project combined (and this is clearly the most important performance metric).

It is also worth noting a number other Python related scientific computing projects. The numeric Python extension adds fast array and matrix manipulation to Python (Dubois 1996), MMTK is Python-based toolkit for molecular modeling (Hinsen 1999), the Biopython project is developing Python-based tools for life-science research (Biopython 1999), and the Visualization Toolkit (VTK) is an advanced visualization package with Python bindings (VTK, 1999). In addition, ongoing projects in the Python community are developing extensions for image processing and plotting. Finally, work presented in (Greenfield, 2000) describes the use of Python in projects at the STScI.

15. Acknowledgments

This work was performed in the Condensed Matter and Statistical Physics group (T-11) in the Theoretical Division at Los Alamos National Laboratory. I'd like to thank my collaborators Peter Lomdahl, Tim Germann, Brad Holian and, Shujia Zhou.

References

Beazley, D. & Lomdahl, P. 1994, Parallel Computing, 20, 173-195

Beazley, D. & Lomdahl, P. 1997, Computers in Physics, 11(3), 230-238

Beazley, D. 1998, Dr. Dobb's Journal, 282, 30-36

Dubois, P. 1994, Computers in Physics, 8(1), 70-73

Dubois, P. 1996, Computers in Physics, 10(3), 262-267

Greenfield, P. 2000, this volume, 79

Hinsen, K. 1999, `http://starship.python.net/crew/hinsen/MMTK`

Biopython 1999, `http://www.biopython.org`

Perl 1999, `http://www.perl.com`

Python 1999, `http://www.python.org`

SWIG 1999, `http://www.swig.org`

Tcl 1999, `http://www.scriptics.com`

VTK 1999, `http://www.kitware.com`