Using Python to Modernize Astronomical Software

Richard L. White and Perry Greenfield

Space Telescope Science Institute
Baltimore MD 21218
http://www.stsci.edu
rlw@stsci.edu
perry@stsci.edu

Abstract

We have developed a Python replacement for the command language used in IRAF, the most widely used data analysis software system in astronomy. The new system allows access to the hundreds of data analysis tasks in the IRAF system, retains the package and parameter structures currently used in IRAF, handles the graphics output and image display interactions, and is capable of completely emulating the old scripting language by translating it into Python. We expect the new language to become widely used in astronomy because it combines access to the familiar suite of IRAF tasks with a more powerful programming language that is also capable of manipulating data directly through the NumPy module. This paper describes the problems we faced, the solutions we have adopted, and our future plans. The entire project has been much easier than we anticipated, due largely to the excellent facilities provided by the Python language and user community.

1 Introduction

The most widely used data analysis software in the astronomical community is a system called the Image Reduction and Analysis Facility (IRAF) [Tody84]. It is estimated that approximately half of astronomers use IRAF regularly. The Space Telescope Science Institute (STScI) has used IRAF as the basis of all the software it has developed for calibration and analysis of Hubble Space Telescope data. While IRAF has proved a durable environment for software development and has a number of positive aspects that are particularly useful for astronomical data analysis software, it is becoming increasingly outmoded.

One approach to updating our software environment would be to find or develop a completely new software system. Unfortunately this would require rewriting a great number of applications already developed under IRAF—some million lines of code in which hundreds of man-years are invested. Starting from scratch is considered untenable due to both the effort required to replace the existing applications and the time required to produce a critical mass of applications sufficient to attract astronomers to the new system. Any effort to transform our analysis environment must be evolutionary in nature and must retain the ability to run the vast majority of applications developed to date. This paper describes how we are using Python to do just that.

2 IRAF and Data Analysis in the Astronomical Community

Understanding the nature of the problem and the solution we have chosen requires some explanation of the IRAF system and the nature of data reduction and analysis in astronomy. One of the fundamental goals of the IRAF system was to allow astronomical applications to be developed in a portable way that would make it easy to distribute the software to astronomers worldwide. Typically astronomers take data at telescope facilities far from their home institutions. Understandably, most astronomers would like to reduce and analyze their data at their home institution rather than remaining at the observatory. Since there are many aspects of calibration, data reduction and analysis common to most astronomical data, it makes sense to have widely distributed applications rather than to expect astronomers to code all their data analysis applications (which was the common approach before IRAF was developed.)

Distributing software to a diverse set of computing platforms is still a thorny problem; it was a thornier problem at the time the IRAF system was designed (1979–1984). The IRAF design was primarily centered around allowing easy portability for applications while retaining the ability to write computationally efficient programs. IRAF solved this problem quite well. Programs written in the IRAF system port fantastically easily. Applications virtually never require platform specific code; there are rarely any problems building or running such applications on supported platforms. As long as the IRAF system has been ported to your computing platform you can have great confidence that the applications software written for IRAF will work fine on it (well, aside from the bugs that plague software in general—these are platform-independent bugs!)

This portability comes at a cost, however. Two major choices were made to ensure portability. One was that IRAF would provide a virtual operating system interface to applications. All applications are required to use only the virtual operating system interface. This interface provides access to all the usual system services such as file and terminal I/O, process control, and signal handling. The second choice was to require IRAF programs to be written in a special language. It was judged at the time that no existing language was sufficiently portable, had the necessary system features (such as memory allocation), and was efficient enough for numerical computation. Consequently, the designer of IRAF chose to implement a new language (called SPP) that the IRAF system ultimately translates into FORTRAN 66. This language, which combines elements of FORTRAN and C, was until recently the only language having access to the full functionality of the system libraries for applications. Virtually all of the applications developed for IRAF were, until recently, written in SPP.

While these choices did meet the goal of portability, they entail drawbacks that are becoming more painful as time goes on. The decision to provide a virtual OS interface commits the maintainer of the system to the heavy burden of maintaining many aspects of an OS with far fewer resources than are usually available to those that are responsible for such systems. Adding the new capabilities that are found in other current systems is even more difficult. The net result has been that IRAF has lagged behind the features and capabilities commonly expected in new OS's and applications. Worse, the design makes it very difficult to integrate outside libraries and applications because of the requirement that all code use the provided virtual OS interface. It is an extremely closed system.

Finally, the use of a custom language that lacks many of the features found in more modern languages—particularly object-oriented features—has obvious problems. Programmers using it often feel that they are not learning marketable skills, and they do not get the productivity benefits seen in newer, standard languages. Most troubling of all is the large code base that cannot be easily moved into any other environment without a complete rewrite.

3 Desired Changes

Many areas of the IRAF system could either be improved or replaced to make it more open to use with other software. One step in this direction at STScI was the generation of bindings that allow writing IRAF programs in C. This at least allows us to program in a standard language, although the dependence on the virtual OS interface still presents many barriers to integrating IRAF software with other software.

We decided that the next most important improvement would be to develop an alternate "shell" for the IRAF system. The IRAF system has its own "Command Language" (CL) which is effectively its virtual OS shell. From this CL, one can run IRAF tasks, perform file operations, get directory listings, print files, etc. Scripts can be written in the CL, and as such it is also a scripting language. But it has many limitations, not least of which is that it has no error or exception handling. This makes writing robust, complex scripts difficult, if not impossible.

If we could replace the existing IRAF CL with a more capable shell environment, we would reap some important benefits. We could write significantly more powerful and robust scripts, so many tasks that now must be developed as C programs could be instead written in the scripting language. We could develop GUI interfaces much more easily. (Writing GUI tasks in IRAF is now possible, but they are both tedious to develop and difficult to maintain.) Perhaps most importantly, we could integrate non-IRAF tasks and libraries with IRAF far more easily, allowing the addition of new capabilities to the system.

Writing a good scripting language is a lot of work and is hard to justify if a good, general purpose, extensible language already exists. Thus our interest in Python.

Given the limitations of IRAF just described, how would it be possible to do what we have outlined? After all, if it is difficult to link in outside programs because of the virtual OS restrictions, why would we be able to do it with Python? The reason is that the CL environment need not be directly linked with the IRAF virtual OS. Almost all IRAF applications run as sub-processes of the CL and communicate with the CL through stdin and stdout pipes. This is not to say that handling the tasks is simple, but the fact that the CL communicates with tasks through pipes instead of through an OS-based link makes the problem tractable. The problem reduces to creating a scripting environment that emulates the task communication protocol, so that the applications believe they are running in the old CL environment.

For a new CL environment to successfully run the existing applications in a convenient way, the following problems must be solved:

- The environment must start up the application executables as sub-processes and communicate with them through pipes following a special protocol. This protocol multiplexes several I/O streams from the application (stdout, stderr, graphics, etc.) onto one set of input/output pipes. The new environment must de-multiplex the various I/O streams and must properly manage the sub-process (including sending appropriate environment information, initiating the execution of one of several tasks in an executable, responding to task requests for parameters, handling interrupts and errors, etc.) The entire process is rather complicated.

- Applications (tasks) in IRAF are organized in a hierarchical set of packages, each of which may contain any number of tasks or packages (similar to a directory tree.) CL scripts are used to define the hierarchical structure of these packages, with a package script indicating where the executable for each task can be found (an executable may contain multiple task entry points) and where the next level of package scripts can be found. A new environment must be able to traverse the package tree from the existing CL scripts, because replicating the dynamic package structure information would be error-prone and time-consuming.

- The CL environment manages the parameters controlling the tasks. Each task usually has an associated parameter file which defines the names of parameters, their types, default values, enumerated values or ranges for each parameter, and prompt strings. Parameter values are persistent between task invocations (and, in fact, between IRAF sessions); this turns out to be very convenient for typical data analysis sessions. The CL itself reads the parameter files and interacts with the user to change current values. A new CL environment must be able to find the parameter files, read and interpret them properly, apply the appropriate type and range constraints, and respond to the task's requests for parameter values.

- The CL also manages the graphics that any task produces. All graphics produced by tasks result in a metacode stream of simple graphics instructions to the CL. The CL must determine which device the graphics are destined for and translate the metacode into device-specific graphics instructions.

- The CL manages the interactions between the task and image display programs. A number of tasks request the position of an image cursor along with a keystroke character value to determine what action the task should take (i.e., interactive responses). The CL must be able to communicate with standard astronomical image display programs (e.g., ximtool and saoimage) using their specialized protocol.

- A few tasks may request arbitrary CL commands to be executed. Replicating this behavior essentially requires emulating the old CL completely.

- Many 'tasks' are actually implemented as CL scripts. Retaining access to their functionality requires either rewriting them in the new scripting language or, again, emulating the old CL.

In short, to be able to create a new CL, one must be able to perform the above functions or sacrifice some capabilities. Whichever tool is used to implement a new CL must be capable of sub-process creation, communication, and management; it must be powerful enough to parse and emulate the old CL syntax and semantics. It must be able to interpret a graphics metacode stream and render graphics; it must provide all the necessary capabilities to communicate with image display programs and handle user interactions; and it must be able to do so quickly and without great implementation effort.

As if this weren't enough to ask, we required even more. The new system must interface easily with existing C code, and it must have a usable interactive environment for astronomers, many of prefer to interact with tasks by using a simple shell-like syntax (e.g., space-delimited argument lists with quotes optional on strings). Another design consideration is portability. Although IRAF has been ported to a number of platforms, all the supported platforms are Unix variants (with the exception of VMS, which will soon be unsupported). Aspects of the new CL system are necessarily Unix-specific, since without a Windows or Mac port of IRAF, we do not know how the specific mechanisms will work on those operating systems. For example, one needs to know how IRAF will implement process communications and fork functionality, which could use several different approaches on Windows. Even so, we desire that the non-IRAF enhancements to the new CL be portable to other platforms even if IRAF is not yet implemented on them.

Finally, and perhaps most important, a key element in making a new CL environment more powerful is providing access to powerful and efficient numerical array operations. While IRAF is the most widely used astronomical data reduction and analysis facility, a substantial fraction of astronomers use the array-based language IDL to write reduction and analysis tasks. IDL (Interactive Data Language, http://www.rsinc.com/) has proven very useful in the astronomical context, and many astronomers swear by it. But it is expensive and lacking as a general programming language. Currently, the IRAF CL has essentially none of the data manipulation facilities that make IDL so attractive. On the other hand, it is difficult to write integrated, high-level applications in IDL that are competitive with those in IRAF, and running IRAF tasks from IDL is next to impossible. Neither is easy to integrate with the other.

We would like the new CL that we are developing to integrate the powerful, interactive, array and graphics capabilities that make prototyping new scientific analysis algorithms so easy in IDL with the existing suite of fully developed IRAF applications that can handle many common (but difficult) data analysis problems.

4 Using Python as the New Environment for IRAF

4.1 Requirements for underlying scripting language

When searching for the appropriate tool to implement a new CL, Python looked like a likely candidate. The existing set of libraries and code, along with the ability to easily interface with C programs, led us to believe that it might be possible to implement much of the CL in Python. NumPy looked like a very promising basis on which to provide array-based analysis features. Still, Python is not unique in this regard. We also required that the language: (1) had a wide enough user base to expect that it would be around for a while, (2) provided good support for object-oriented programming, and, most important, (3) was readable enough for many astronomers to feel that it was a programming language that they could use. In this regard, Python stood out as a clear choice. Nevertheless, it still was not clear that replacing the IRAF CL with Python was feasible.

4.2 Python made it much easier than expected

As it turned out, we needn't have worried. We have been able to implement a new CL with far less effort than we expected using Python. Over the span of one year, we have implemented all the major functionality required, including nearly complete emulation of the old CL. Virtually all our development has been done in the Python language itself without any serious performance problems. The basic system was developed with approximately a total combined effort of 6 man-months. The following will broadly indicate the tools we used to implement the new CL.

4.3 Sub-process control & communication

We use the subproc module (available from ftp://ftp.python.org/pub/python/contrib) as a basis for the control of the IRAF applications running as subprocesses. While some modifications were necessary, we have been able to use it to reliably control the IRAF executables. We have used NumPy in a few areas to convert between IRAF 16-bit characters and ASCII, and to handle IRAF 16-bit data transfers (e.g., graphics metacode).

4.4 Graphics

PyOpenGL was used to implement a graphics kernel that renders the IRAF graphics metacode in Tkinter (Togl) widgets. We are currently exploring the possibility of using wxPython as an alternative to Tkinter.

4.5 Task objects

IRAF tasks are represented as Python objects. A task object embodies the specific information needed to actually execute the task. Such objects are created when IRAF packages are loaded in the Python environment (with lazy instantiation used for some attributes to avoid excessive initialization costs.) Packages are also represented as task objects and, when executed, load the tasks and packages contained within the package by creating new task objects.

The task object's `__call__` method has been defined to allow execution of the task when the task object is called as a function. Both positional and keyword arguments are supported; in keeping with the IRAF CL style, parameters defined as "hidden" are accessible only as keyword arguments. Unambiguous abbreviations are also allowed for keyword names.

Currently we have a few different alternatives for the namespaces in which the task objects appear. In one case they appear in the main namespace, in another they appear in the `iraf` module namespace. We expect to settle on a standard approach to namespaces after some experience using the system.

4.6 Task parameters

IRAF tasks usually have parameter files that specify the names of parameters for the tasks along with their types, allowable values, default or current values, and prompt strings. We decided to map these parameter names to task attributes, with a few twists. Assigning to a parameter attribute changes the parameter value, but it is important to prevent a typo in the parameter name from creating a new attribute rather than changing the value of the intended parameter. We use `__setattr__` to prevent such errors. We also allow name completion on parameter attributes (so a parameter name can be abbreviated to an unambiguous shorter string) using suitable modifications of `__setattr__` and `__getattr__`. Finally, when a parameter attribute is assigned a value, the type and value are checked to insure they conform to the parameter definition.

A GUI parameter editor (using file browsers, choice lists for parameters with enumerated values, integrated help, etc.) has been written using Tkinter.

When a task is executed, defaults for all omitted parameters come from the persistent values read from the parameter file or set by the user before running the task. Python has the flexibility to allow parameter-setting mechanisms that are very similar to those provided by the IRAF CL, making use of the new task and parameter interface easy for current IRAF users.

4.7 Emulation of the IRAF CL

The old CL was emulated by using Aycock's "little languages" framework [Aycock98] to translate CL code to Python. The Python code can be either saved as source code or compiled and executed in Python. Indeed, we now have a system that will accept CL commands in exactly the same syntax as the original as well in Python syntax. While this particular parsing module may not be the fastest available (though it is quite elegant and powerful, and we have made a few improvements to make it faster), its speed is not viewed as a serious problem. The one-time cost of translating previously untranslated IRAF CL scripts to Python is acceptable because the resulting Python code can easily be saved for future use via a combination of pickle and shelve.

4.8 Front-end interpreter

We have developed a front end for the Python interpreter to permit use of a simple alternate syntax as well as provide many of the conveniences expected by IRAF users (described below). The alternate syntax is blended with normal Python by keying off a task dictionary. If the user types a line beginning with an identifier followed by a space and more identifiers or expressions (the first cannot start with parentheses, to make distinguishing Python function calls easier), then the initial identifier is looked up in the task dictionary (using the minimum matching capability mentioned later). If it is found, then the line is interpreted as being in the old CL syntax. Otherwise, it is treated as Python syntax (subject to the modifications mentioned below).

In this way, if the user types something like

```
imcopy infile oufile
```

then the task `imcopy` is found in the dictionary of currently loaded tasks and the command is translated into

```
iraf.imcopy(“infile”,”outfile”)
```

and fed to the Python interpreter. Of course, this mechanism prevents use of a whole class of legal Python statements, but we judged that there were simple ways of duplicating the "shadowed" Python syntax. For example, if one defines a variable

```
imcopy = “just a simple string”
```

then one cannot type interactively

```
imcopy
```

and expect to see the string printed. But the workaround is simple: `print imcopy`.

The tasks that can be invoked using the command-style syntax are not limited to IRAF applications. We have generalized the task interface to cover Python functions as well. One can register a Python function as a task to be included in the task dictionary, and then it can be invoked in much the same way with a similar parameter interface. Such registration requires indicating the parameter types expected by the Python function. Then each argument is checked to see if it conforms to the task parameter type specifications and an exception is raised if it cannot be coerced properly. This alleviates the need for type and value checking in the Python functions.

Other interactive conveniences provided by the front-end interpreter include:

- A convenient mechanism for shell escapes: `!ls *.*`

- Command logging. While an alternate syntax is allowed—the "command" style—the statements logged to the file are the translated representation, so the file can be used as a Python script.

- Basic file and directory utilities, i.e., directory listings, file renaming, copying, deletion, and printing; creating and removing directories; searching for strings in files, etc. These suffice for the most common file manipulations.

- I/O redirection and piping *a la* the Unix shell.

- Task name abbreviations. Any unambiguous abbreviation can be used in place of the full names.

- Use of '{' and '}' as an alternate statement blocking mechanism (GASP!). This is available in interactive mode only (what is logged is the equivalent indented source). Its provides a convenient way of typing a multiple-statement block on a single line, making for easy command line recall.

4.9 Non-Python code

The only C code required fell into two categories. A small number of specialized Xlib utilities were needed to handle window functions not supplied with Tk (such as focus setting to the original terminal window and cursor warping). A wrapper was created for an existing C library that provides functions to communicate with image display programs used in astronomy [Fitzpatrick97].

4.10 Summary

The net result of our efforts is a new environment that allows one to run nearly all IRAF tasks from Python, including setting up an environment for IRAF tasks and locating packages, executables and parameter files from the existing IRAF installation.

The new CL, which we have tentatively dubbed *Pyraf*, is now available to all users at STScI and is being beta tested at a few outside sites. We plan to release it to the general community in summer of 2000. Like all our previous software, it will be freely available including the source code. While most of the code is only of interest to IRAF users, some of it (the parameter interface, improvements to subproc and Aycock's framework, and the front-end interpreter) may be useful for other Python projects.

5 Future Work

While we have made great progress, this is only the first in a series of steps to provide astronomers with greater data analysis and reduction capabilities. The existing functionality basically only replicates the existing IRAF functionality and, by itself, will not spur many users to switch unless they have a critical need for the enhanced programming environment that Python provides. A number of future developments are also essential for widespread acceptance of the new Python CL environment by the astronomical community.

Develop new Pyraf applications at STScI that provide functionality not available in the current IRAF CL. We have already started two such projects.

Demonstrate the ability to write useful GUI applications far more easily than is possible to do in IRAF alone. We have already started such a project.

Provide enhancements to the interactive environment that make using IRAF tasks easier than the current IRAF CL. We have already included a few minor improvements: for example, it is now possible to have multiple plotting windows and to recall old plots, and the GUI parameter editor is easier to use than the existing IRAF editor.

Add IDL-like data manipulation functionality to the Pyraf environment. To a certain extent, this already exists in NumPy, but both generic and IRAF-specific enhancements are needed. Major items include:

- ❑ The means to read and write data in common astronomical disk formats to NumPy arrays. This is under development as part of the PyFITS module [Barrett98]. Even more useful would be a mechanism to pass NumPy arrays directly to IRAF tasks, which would probably require changes to the IRAF system.

- ❑ Plotting and image display facilities at the level provided by IDL and other similar software. At this writing (November 1999), Python is not quite there. There are a number of scientific plotting packages available in Python but none satisfies the requirement that it be freely available, portable, and sufficiently powerful. (As an aside, astronomical data analysis generally has had little demand for advanced 3-D visualization tools; the great majority of astronomers are quite happy with 2-D graphics.) Recent progress with Piddle and Graphite [Strout99] gives us some optimism that these will form the basis for a good astronomical plotting suite.

- Enhance NumPy's basic functionality to make it equivalent to (or better than!) IDL, Matlab, and other array languages. Astronomers deal with large arrays (newer instruments are generating 8K x 8K pixel images). Efficient use of memory and efficient standard functions are musts when handling such large images. Currently it is difficult to write NumPy programs that retain data in 16-bit or single precision floating point format without automatic up-casting to larger data types. Lack of gather/scatter capability and efficient intrinsic functions is another problem (there is no efficient histogram function for example). We know that others are working to remedy these problems, and we are also planning to contribute effort to improving NumPy.

Make the software easy to distribute and install. The problem is not so much with the programs we write; the Python code we develop will be trivial to install, and there is not much C code so it also should not present a serious problem. The difficulty is in having users install all the extensions that are necessary to run our software. Users must install Python itself (with the necessary modules enabled such as fcntl). They also need to install PyOpenGL and related extensions, which is more troublesome. We're aware that much work is going on in the distutils SIG to unify the installation process for extension modules, and we eagerly look forward to the fruits of that work. But at this time, we consider it needlessly painful to install the required software.

References

[Aycock98] Aycock, John (1998) "Compiling Little Languages in Python", Proceedings of the Seventh International Python Conference, p. 100 (http://www.foretec.com/python/workshops/1998-11/proceedings/papers/aycock-little/aycock-little.html)

[Barrett98] Barrett, P. & Bridgman, W. (1998) "Py-FITS, a FITS Module for Python", Astronomical Data Analysis Software and Systems VIII, Astronomical Society of the Pacific Conference Series, 171, 483 (http://monet.astro.uiuc.edu/adass98/Proceedings/-barrettpa/)

[Fitzpatrick97] Fitzpatrick, Michael (1997) "The IRAF Client Display Library", Astronomical Data Analysis Software and Systems VII, Astronomical Society of the Pacific Conference Series, 145, 200 (http://www.stsci.edu/stsci/meetings/adassVII/-fitzpatrickm.html)

[Strout99] Strout, J. (1999) (see http://www.strout.net/python/ for details of status of Piddle and Graphite development, the former of which involves several people)

[Tody84] Tody, D. (1984) "The IRAF Data Reduction and Analysis System", Proc. SPIE Instrumentation in Astronomy VI, ed. D. L. Crawford, 627, 733 (see also http://iraf.noao.edu/)