The PyRAF Tutorial

Richard L. White & Perry Greenfield

May 2, 2002

Space Telescope Science Institute
Email: help@stsci.edu

Abstract

PyRAF, based on the Python scripting language, is a command language for IRAF that can be used in place of the existing IRAF CL. This document provides a tutorial introduction to the PyRAF system, showing how it can be used to run IRAF tasks. It describes how to run IRAF tasks using either the familiar IRAF syntax or Python syntax. It also describes PyRAF's graphical parameter editor and the graphics system.

Contents

1 Introduction

1.1 Why a New IRAF CL?

The current IRAF CL has some shortcomings as a scripting language that make it difficult to use in writing scripts, particularly complex scripts. The major problem is that the IRAF CL has no error or exception handling. If an error occurs, the script halts immediately with an error message that has little or no information about where the error occurred. This makes it difficult to debug scripts, impossible to program around errors that can be anticipated, and difficult to provide useful error messages to the user.

But there are other important reasons for wanting to replace the CL. We want to develop a command language that is a stronger environment for general programming and that provides more tools than the existing IRAF CL. Python (http://www.python.org), like Perl and Tcl, is a free, popular scripting language that is useful for a very wide range of applications (e.g., writing CGI scripts, text processing, file manipulation, graphical programming, etc.). Python is also an open system that can be easily extended by writing new modules in Python, C, or Fortran. By developing a Python interface to IRAF tasks, we open up the IRAF system and make it possible to write scripts that combine IRAF tasks with the numerous other capabilities of Python.

Ultimately, we plan for PyRAF to have command-line data access and manipulation facilities comparable to those of IDL (the Interactive Data Language, http://www.rsinc.com, which has extensive array-processing and image manipulation capabilities). It should soon be possible to write applications in PyRAF that can manipulate the data directly in memory and display it much as IDL does. The data access and computation abilities are already present, and we are currently working on a new scientific graphics modules for Python.

We are hoping that Python and PyRAF will become the programming language of first resort, so that programmers and astronomers will only infrequently need to write programs in C or Fortran. Moreover, when it is necessary to use compiled languages, programs written in C, C++, and Fortran can be easily linked with and integrated into PyRAF.

1.2 Why Should I Use PyRAF?

PyRAF can run the vast majority of IRAF tasks without any problems. You can display graphics and use interactive graphics and image display tasks as well. You can even run more than 90% of IRAF CL scripts! While many things work the same way as they do in the IRAF CL, there are some differences in how things are done, usually for the better.

The most important difference between PyRAF and the old IRAF CL is that PyRAF allows you to write scripts using a much more powerful language that has capabilities far beyond simply running IRAF tasks. Some of the things that now become possible are:

- Easier script debugging.
- Ability to trap errors and exceptions.
- Access to all the features of Python including:
  - built-in list (array) and dictionary (hash table) data structures
  - extensive string handling libraries
  - GUI libraries that make it straightforward to wrap IRAF tasks with GUI interfaces
  - web and networking libraries making it easy to generate html, send email, handle cgi requests, etc.
- Simple access to other software packages written in C or Fortran.
- Ability to read and manipulate FITS files and tables using pyfits and numarray.

As far as the interactive environment goes, the new system has most of the current features and conveniences of the IRAF CL as well as a number of new features. It is possible to use exactly the same familiar IRAF CL syntax to run IRAF tasks (e.g., `imhead dev$pix long+`). There are some additional features such as command-history recall and editing using the arrow keys, tab-completion of commands and filenames, a GUI equivalent to the epar parameter editor, the ability to display help in a separate window, and a variety of tools for manipulation of graphics. We believe the new environment is already in many ways more convenient for interactive use than the IRAF CL, and further enhancements are planned.

1.3 Python Prerequisites

This guide will not attempt to serve as a complete Python introduction, tutorial or guide. There are a number of other documents that serve that role well. At some point we recommend spending a few hours familiarizing yourself with Python basics. Which Python documentation or books to start with depends on your programming background and preferences. We have written *A Quick Tour of Python* that may be helpful for IRAF users. Some people will find quick introductions like the Python tutorial (see our list of available Python documentation for sources for this and other books and documentation) or the *Essential Python Reference* all they need. Others may prefer the slower-paced introductions found in the *Learning Python* or *Quick Python* books.

In any case, do not spend too much time reading before actually starting to run PyRAF and use Python. The Python environment is an interactive one and one can learn quite a bit by just trying different things after a short bit of reading. In fact, you may prefer to try many of the examples given in this guide before reading even a single Python document. We will give enough information for you to get started in trying the examples. Our approach will be largely based on examples.

1.4 IRAF Prerequisites

This guide also is not intended to be an introduction to IRAF. We assume that current PyRAF users are relatively experienced with IRAF. We plan to prepare documentation (and user tools) intended for first-time IRAF users, but this beta release targets those who are already using IRAF and are looking for a better command language.

2 Starting and Using PyRAF Interactively

If PyRAF is installed (and your environment is properly configured), typing `pyraf` will start up PyRAF using the front-end interpreter. (There are a few command line options, described in §6, that can be listed using `pyraf --help`.) Here is an illustration of starting PyRAF and running some IRAF tasks using the CL emulation syntax.

```
% pyraf
setting terminal type to xterm...

    NOAO Sun/IRAF Revision 2.11.2 Wed Aug 11 13:24:18 MST 1999
    This is the EXPORT version of Sun/IRAF V2.11 for SunOS 4 and Solaris 2.7

    Welcome to IRAF.  To list the available commands, type ? or ??.  To get
    detailed information about a command, type 'help command'.   To  run  a
    command  or  load  a  package,  type  its name.   Type  'bye' to exit a
    package, or 'logout' to get out of the CL.   Type 'news'  to  find  out
    what is new in the version of the system you are using.   The following
    commands or packages are currently defined:
.
.
.
--> imheader dev$pix long+
dev$pix[512,512][short]: m51  B  600s
No bad pixels, min=-1., max=19936.
Line storage mode, physdim [512,512], length of user area 1621 s.u.
Created Mon 23:54:13 31-Mar-1997, Last modified Mon 23:54:14 31-Mar-1997
Pixel file "HDR$pix.pix" [ok]
'KPNO-IRAF'           /
'31-03-97'            /
IRAF-MAX=           1.993600E4  /  DATA MAX
IRAF-MIN=          -1.000000E0  /  DATA MIN
IRAF-BPX=                   16  /  DATA BITS/PIXEL
IRAFTYPE= 'SHORT   '            /  PIXEL TYPE
CCDPICNO=                   53  /  ORIGINAL CCD PICTURE NUMBER
.
.
.
HISTORY '24-04-87'
HISTORY 'KPNO-IRAF'           /
HISTORY '08-04-92'            /
--> imstat dev$pix
#               IMAGE      NPIX      MEAN    STDDEV       MIN       MAX
              dev$pix    261626     105.1     75.86       -1.     1000.
--> imcopy dev$pix mycopy.fits
dev$pix -> mycopy.fits
--> stsdas


      +------------------------------------------------------------+
      |       Space Telescope Science Data Analysis System         |
      |          STSDAS Version 2.1, September 29, 1999            |
      |                                                            |
      |   Space Telescope Science Institute, Baltimore, Maryland   |
      |          For help, send e-mail to help@stsci.edu           |
      +------------------------------------------------------------+
```

```
stsdas/:
 analysis/      examples        hst_calib/      sobsolete/
 contrib/       fitsio/         playpen/        toolbox/
 describe       graphics/       problems
--> fitsio
--> catfits mycopy.fits
EXT#  FITSNAME      FILENAME              EXTVE DIMENS     BITPI OBJECT

0     mycopy.fits                               512x512    16    m51  B  600s
--> imhead dev$pix long+ > devpix.header
```

You may notice a great similarity between the PyRAF login banner and the IRAF login banner. That's because PyRAF reads your normal 'login.cl' file and goes through exactly the same startup steps as IRAF when a session begins. If you have customized your 'login.cl' or 'loginuser.cl' files to load certain packages, define tasks, etc., then those customizations will also take effect in your PyRAF environment.

You can start up PyRAF from any directory; unlike the IRAF CL, you are not required to change to your IRAF home directory. PyRAF determines the location of your IRAF home directory by looking for your 'login.cl' file, first in your current working directory and then in a directory named 'iraf' in your home directory. So as long as your IRAF home directory is '~/iraf', you can start up PyRAF from any working directory. (You can start from other directories as well, but without access to 'login.cl' your IRAF environment will be only partly initialized. We expect to add a startup configuration file, '.pyrafrc', that allows you customize your initial PyRAF configuration including your IRAF home directory.)

The first time you run PyRAF, it creates a 'pyraf' directory in your IRAF home directory. At the moment all it stores there is a directory named 'clcache', which is used to save translated versions of your own custom CL scripts. (The files in that directory have cryptic names that actually encode the contents of the corresponding CL scripts, thus allowing the translation to be used regardless of the CL script name.)

Note that the task syntax shown above is identical to that of the IRAF CL. But there is no escaping that you are really running in a Python environment. Should you make a mistake typing a task name, for example,

```
--> imstart dev$pix
  File "<console>", line 1
    imstart dev$pix
              ^
SyntaxError: invalid syntax
```

or should you use other CL-style commands,

```
--> =cl.menus
  File "<console>", line 1
    =cl.menus
    ^
SyntaxError: invalid syntax
```

then you'll see a Python error message. At this stage, this is the most likely error you will see aside from IRAF-related ones. (We plan to improve some of these messages in the future to make them more self-explanatory for new PyRAF users.)

Aside from some noticeable delays (on startup, loading graphics modules, or in translating CL scripts not previously encountered), there should be little difference between running IRAF tasks in CL emulation mode and running them in the IRAF CL itself.

2.1 New Capabilities in PyRAF

Several capabilities in the PyRAF interpreter make it very convenient for interactive use. The up-arrow key can be used to recall previous commands (no need to type `ehis!`), and once recalled the left and right arrow keys can be used to edit it. The control-R key does pattern-matching on the history. Just type part of the command (not necessarily at the beginning of the line) and you'll see the matched command echoed on the command line. Type ˆR again to see other matches. Hit return to re-execute a command, or other line-editing keys (left/right arrow, ˆE, ˆA, etc.) to edit the recalled command. There are many other ways to search and manipulate the history – see the gnu readline documentation for more information.

The tab key can be used to complete commands, in a way familiar to users of tcsh and similar shells. At the start of the command line, type `imhe<tab>` and PyRAF fills in `imheader`. Then type part of a filename `<tab>` and PyRAF fills in the rest of the name (or fills in the unambiguous parts and prints a list of alternatives). This can be a great timesaver for those long HST filenames! You can also use tab to complete IRAF task keyword names (e.g., `imheader lon<tab>` fills in `longheader`, to which you can add `=yes` or something similar). And when using Python syntax (see below), tab can be used to complete Python variable names, object attributes, etc.

The function

```
saveToFile filename
```

saves the current state of your PyRAF session to a file (including package, task, and IRAF environment variable definitions and the current values of all task parameters.) The function

```
restoreFromFile filename
```

restores the state of your session from its previously saved state. A save filename can also be given as a Unix command line argument when starting up PyRAF, in which case PyRAF is initialized to the state given in that file. This can be a very useful way both to start up in just the state you want and to reduce the startup time.

2.2 Differences from the CL and Unimplemented CL Features

Some differences in behavior between PyRAF and the CL are worth noting. PyRAF uses its own interactive graphics kernel when the CL stdgraph variable is set to a device handled by the CL itself (e.g., xgterm). If stdgraph is set to other values (e.g. stdplot or the imd devices), the appropriate CL task is called to create non-interactive plots. Currently only the default PyRAF graphics window supports interactive graphics (so you can't do interactive graphics on image display plots, for example.) Graphics output redirection is not implemented at the moment. Both full interactive support of additional graphics devices and graphics redirection are being considered for future development.

Some IRAF CL commands have the same names as Python commands; when you use them in PyRAF, you get the Python version. The ones most likely to be encountered by users are `print` and `del`. If you want to use the IRAF print command (which should rarely be needed), use `clPrint` instead. If you want the IRAF delete command, just type more of the command (either `dele` or `delete` will work).

Another similar conflict is that when an IRAF task name is identical to a reserved keyword in Python (to see a list, do `import keyword; print keyword.kwlist`), then it is necessary to prepend a 'PY' (yes, in capital letters) to the IRAF task name. Such conflicts should be relatively rare, but note that 'lambda' and 'in' are both Python keywords.

The PyRAF help command is a little different than the IRAF version. If given a string argument, it looks up the CL help and uses it if available. For other Python argument types, `help` gives information on the variable. E.g., `help(module)` gives information on the contents of a module. There are some optional arguments that are useful in Python programs (type `help(help)` for more information). We plan further enhancements of the help system in the near future.

If you need to access the standard IRAF help command without the additional PyRAF features, use `system.help taskname options`.

Note that the IRAF help pages are taken directly from IRAF and do not reflect the special characteristics of PyRAF. For example, if you say `help while`, you get help on the CL while loop rather than the Python while statement. The login message on startup also comes directly from IRAF and may mention features not available (or superseded) in PyRAF. We will eventually remove some of these inconsistencies, but for the moment users will occasionally encounter such conflicts.

There are a few features of the CL environment and CL scripts that are not yet implemented:

Packages cannot be unloaded. Currently there is no way to unload a loaded IRAF package. The bye command exists but does not do anything; the keep command also does nothing (effectively all modifications to loaded tasks and IRAF environment variables are kept). This will be added in a future version, although we are also considering alternative ways to organize the namespace of tasks and packages that may be more "Pythonic".

No GOTO statements in CL scripts. Python does not have a goto statement, so converting CL scripts that use goto's to Python is difficult. At the moment we have made no effort to do such a conversion, so CL scripts with GOTO's raise exceptions. GOTOs may not ever get implemented.

Background execution is not available. Background execution in CL scripts is ignored. This will get added if there is sufficient demand.

Error tracebacks in CL scripts do not print CL line numbers. When errors occur in CL scripts, the error message and traceback refer to the line number in the Python translation of the CL code, not to the original CL code. We plan to add the CL code itself to the error messages eventually. (If you want to see the Python equivalent to a CL task, use the getCode method – e.g. `print iraf.spy.getCode()` to see the code for the `spy` task)..

3 The EPAR Parameter Editor

For PyRAF we have written a task parameter editor that is similar to the IRAF epar function but that uses a graphical user interface (GUI) rather than a terminal-based interface. PyRAF's EPAR has some features not available in the IRAF CL, including a file browser for selecting filename parameters and a pop-up window with help on the task. Upon being invoked in the usual manner in IRAF CL emulation mode, an EPAR window for the named task appears on the screen:

```
--> epar imcalc
```

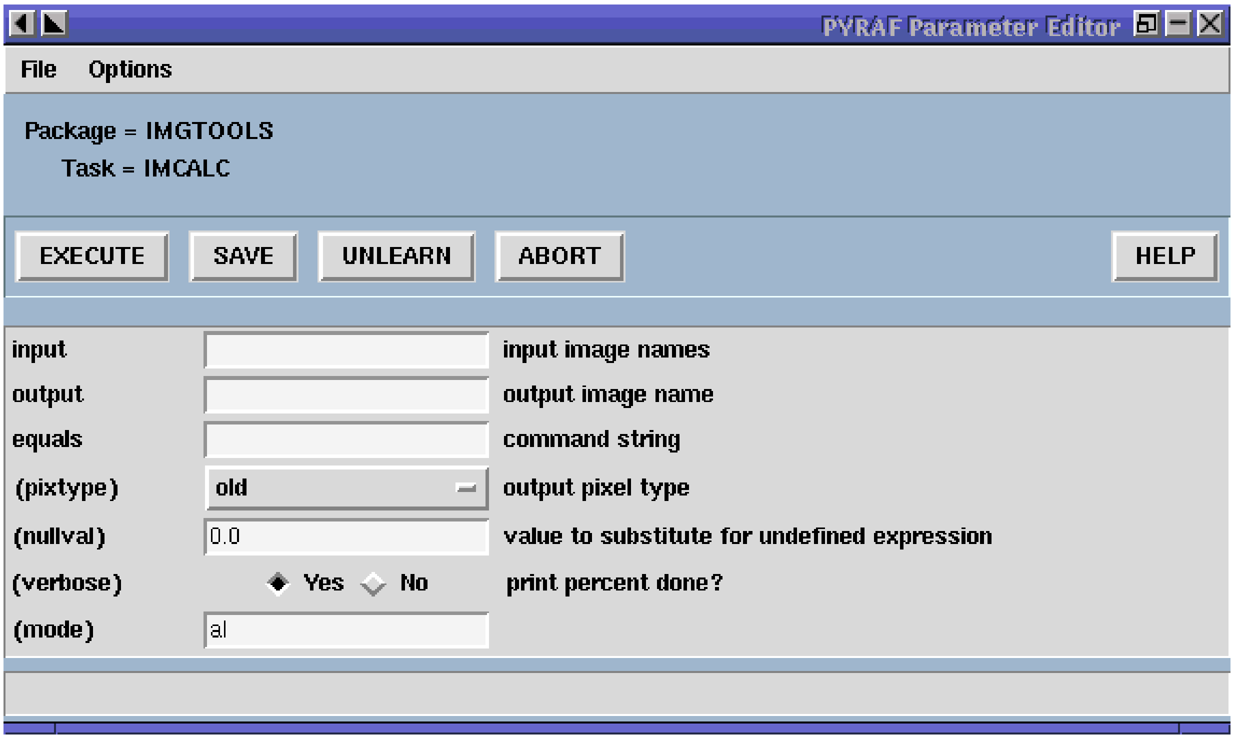
An EPAR window consists of a menu bar, current package and task information, action buttons, the parameter editing panel, and a status box. If there are more parameters than can fit in the displayed window, they will appear in a scrolling region.

3.1 Action Buttons

The EPAR action buttons are:

Execute Execute the task with parameter values currently displayed in the EPAR windows. (Several windows may be open at once if the task has PSET parameters – see below.) If parameter values were changed and not saved (via the Save button), these new values are automatically verified and saved before the execution of the task. The EPAR window (and any child windows) is closed, and the EPAR session ends.

Save This button saves the parameter values associated with the current EPAR window. If the window is a child, the child EPAR window closes. If the window is the parent, the window closes and the EPAR session ends. The task is not executed.



Unlearn This button resets all parameters in the current EPAR window to their system default values.

Cancel This button exits the current EPAR session without saving any modified parameter values. Parameters revert to the values they had before EPAR was started; the exception is that PSET changes are retained if PSETs were editted and explicitly saved.

Task Help This button displays the IRAF help information for a task.

3.2 Menu Bar

The EPAR menu bar consists of **File**, **Options**, and **Help** menus. All of the **File** menu choices map directly to the action button functionality. The **Options** menu allows the user to choose the way help pages are displayed; the information can be directed to the user's web browser or to a pop-up window (the default). The **Help** menu gives access to both the IRAF task help and information on the operation of EPAR itself.

3.3 Parameter Editing Panel

Different means are used to set different parameter types. Numeric and string parameters use ordinary entry boxes. Parameters with an enumerated list of allowed values use choice lists. Booleans are selected using radio buttons. PSETs are represented by a button that when clicked brings up a new EPAR window. PSET windows and the parent parameter windows can be edited concurrently (you do not have to close the child window to make further changes in the parent window).

Parameters may be editted using the usual mouse operations (select choices from pop-up menus, click to type in entry boxes, and so on.) It is also possible to edit parameter without the mouse at all, using only the keyboard. When the editor starts, the first parameter is selected. To select another parameter, use the Tab or Return key (Shift-Tab or Shift-Return to go backwards) to move the focus from item to item. The Up and Down arrow keys also move between fields. Use the space bar to "push" buttons, activate pop-up menus, and toggle boolean values.

The toolbar buttons are also accessible from the keyboard using the Tab and Shift-Tab keys. They are located in sequence before the first parameter and after the last parameter (since the item order wraps around at the end.) If the first parameter is selected, Shift-Tab backs up to the “Task Help” button, and if the last parameter is selected then Tab wraps around and selects the “Execute” button. See the **Help-¿Epar Help** menu item for more information on keyboard shortcuts.

Parameters entered using entry boxes (strings and numbers) are checked for correctness when the focus shifts to another parameter (either via the Tab key or the mouse.) The parameter values are also checked when either the Save or Execute button is clicked. Any resulting errors are either displayed in the status area at the bottom (upon validation after return or tab) or in a pop-up window (for Save/Execute validation).

For parameters other than PSETs, the user can click the right-most mouse button within the entry box or choice list to generate a pop-up menu. The menu includes options to invoke a file browser, clear the entry box, or unlearn the specific parameter value. “Clear” removes the current value in the entry, making the parameter undefined. “Unlearn” restores the system default value for this specific parameter only. The file browser pops up an independent window that allows the user to examine the directory structure and to choose a filename for the entry. Some items on the right-click pop-up menu may be disabled depending on the parameter type (e.g., the file browser cannot be used for numeric parameters.)

3.4 Status Line

Finally, the bottom portion of the EPAR GUI is a status line that displays help information for the action buttons and error messages generated when the parameter values are checked for validity.

4 PyRAF Graphics and Image Display

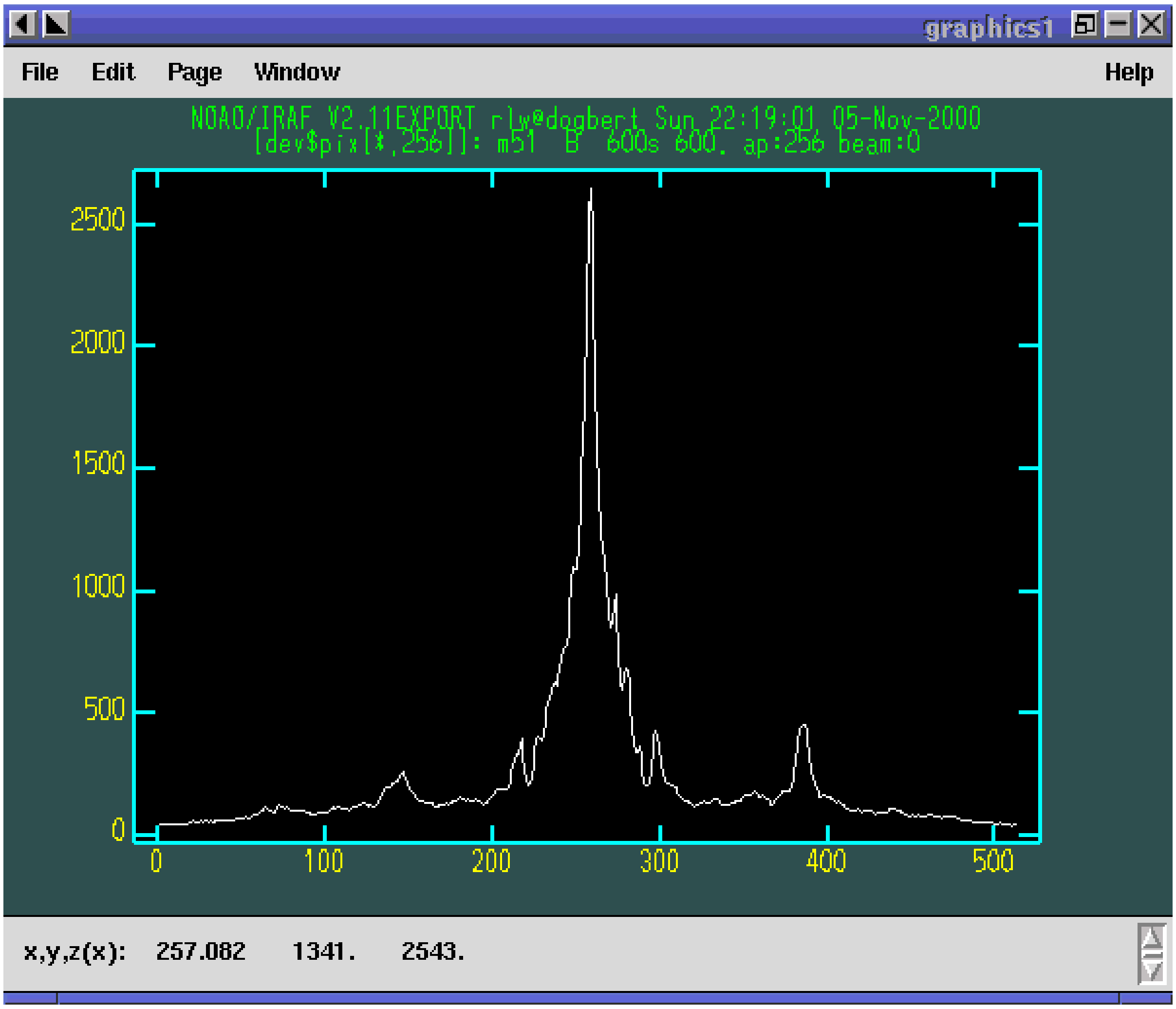
PyRAF has its own built-in graphics kernel to handle interactive IRAF graphics. Graphics tasks can be run from any terminal window — there is no need to use the IRAF xgterm. If the value for stdgraph set in your login.cl would have the IRAF CL use its built-in graphics kernel, in PyRAF it will use PyRAF’s built-in kernel. The PyRAF kernel is not identical to IRAF’s but offers much the same functionality — it lacks some features but adds others. If you specify a device that uses other IRAF graphics kernels (e.g., for printers or image display plots), PyRAF will use the IRAF graphics kernel to render those plots. There are currently some limitations when using IRAF kernels. For example, it is not possible to use interactive graphics tasks with those kernels. But otherwise, most of their functionality is available.

Currently the PyRAF built-in graphics kernel is based on OpenGL and Tkinter (other alternatives will probably be added in the future). Graphics windows are created from PyRAF directly. One can run a graphics task like any other. For example, typing

```
--> prow dev$pix 256
```

will (after some delay in loading the graphics modules) create a graphics window and render a plot in it. The graphics window is responsive at all times, not just while an IRAF task is in interactive graphics mode. If the window is resized, the plot is redrawn to fit the new window. There is a menu bar with commands allowing previous plots to be recalled, printed, saved, etc. The **Edit-¿Undo** menu command can remove certain graphics elements (e.g., text annotations and cursor marks.) It is possible to create multiple graphics windows and switch between them using the **Window** menu. See the **Help** menu for more information on the capabilities of the PyRAF graphics window. At present, few of the options (such as the default colors) are easily configurable, but future versions will improve on this.

Interactive graphics capability is also available. For example, typing `implot dev$pix` will put the user into interactive graphics mode. The usual graphics keystroke (gcur) commands recognized by the task will work (e.g., lowercase letter commands such as `c`) and colon commands will work as they do in IRAF. Most CL-level (capital letter) keystroke commands have not yet been implemented; the following CL level commands are currently available:



- The arrow keys move the interactive cursor one pixel. Shift combined with the arrow keys moves the cursor 5 pixels.
- **C** prints the current cursor position on the status line.
- **I** immediately interrupts the task (this is the gcur equivalent to control-C).
- **R** redraws the plot with annotations removed (also available through the **Edit-¿Undo All** menu item.)
- **T** annotates the plot at the current cursor position, using a dialog box to enter the text.
- **U** undoes the last "edit" to a plot (annotations or cursor markers). This can be repeated until only the original plot remains. (Also available using **Edit-¿Undo**.)
- A colon (**:**) prompts on the status line for the rest of the colon command. Other input from interactive graphics tasks may also be done from the status line.
- The **:.markcur** directive is recognized. It toggles the cursor marking mode (**:.markcur+** enables it, **:.markcur-** disables it). Currently this directive cannot be abbreviated.

Help for interactive IRAF tasks can usually be invoked by typing **?**; the output appears in the terminal window. Output produced while in cursor-mode (e.g., readouts of the cursor position) appear on the status line at the bottom of the graphics window. Note that the status line has scrollbars allowing previous output to be recalled.

It is likely much of the additional functionality of CL level gcur commands (zooming, etc.) will also find its way into menus or other GUI elements of the PyRAF graphics window. Where possible we also will try to make these available through the keystroke commands familiar to expert IRAF users.

PyRAF attempts to manipulate the window focus and the cursor location in a sensible way. For example, if you start an interactive graphics task, the mouse position and focus are automatically transferred to the graphics window. If the task does not appear to be responding to your keyboard input check to see that the window focus is on the window expecting input.

4.1 Printing Graphics Hardcopy

It is possible to generate hard copy of the plotted display by using the **File-¿Print** menu item or, in gcur mode, the equal-sign (`=`) key. PyRAF will use the current value of stdplot as the device to plot to for hardcopy. Inside scripts, a hardcopy can be printed by

```
--> from pyraf.gki import printPlot # only need this once per session
--> printPlot()
```

This could be used in a Python script that generates graphics using IRAF tasks. It is also possible to do other graphics manipulations in a script, e.g., changing the display page.

4.2 Multiple Graphics Windows

It is possible to display several graphics windows simultaneously. The **Window-¿New** menu item can create windows, and the **Window** menu can also be used to select an existing window to be the active graphics window. Windows can be destroyed using the **File-¿Quit Window** menu item or directly using the facilities of the desktop window manager (close boxes, frame menus, etc.)

It is also possible to create new windows from inside scripts. If you type:

```
--> from pyraf import gwm # only need this once per session
--> gwm.window("My Special Graphic")
```

you will create a new graphics window which becomes the current plotting window for PyRAF graphics. The `gwm.window("GraphicsWindowName")` function makes the named window the active graphics window. If a graphics window with that name does not yet exist, a new one is created. Windows can be deleted by closing them directly or using `gwm.delete("GraphicsWindowName")`. Using these commands, one can write a script to display several plots simultaneously on your workstation.

4.3 Other Graphics Devices

To plot to standard IRAF graphics devices such as xterm or xgterm one can

```
--> set stdgraph = stgkern
--> iraf.stdgraph.device = "xgterm"
```

or whatever device you wish to use. [Note the Python version of the set statement is `iraf.set(stdgraph="stgkern")`.] In this way it is possible to generate plots from a remote graphics terminal without an Xwindows display. The drawback is that is is not possible to run interactive graphics tasks (e.g., `implot` or `splot`) using this approach. It may be necessary to call `iraf.gflush()` to get the plot to appear.

One can generate plots to other devices simply by setting `stdgraph` to the appropriate device name (e.g., `imdr` or `stdplot`). Only special IRAF-handled devices such as `xgterm` and `xterm` need to use the “magic” value `stgkern` for `stdgraph`.

IRAF tasks such as `tv.display` that use the standard image display servers (**ximtool**, **saoimage**, **saotng**) should work fine. Interactive image display tasks such as `imexamine` work as well (as long as you are using the PyRAF graphics window for plotting.) Graphics output to the image display (allowing plots to overlay the image) is currently supported only through the IRAF kernel, but a PyRAF built-in kernel is under development.

5 Running Tasks in Python Mode

If that’s all there was to it, using PyRAF would be very simple. But we would be missing much of the point of using it, because from CL emulation mode we can’t access many of the powerful programming features of the Python language. CL emulation mode may be comfortable for IRAF users, but when the time comes to write new scripts you should learn how to run IRAF tasks using native Python syntax.

Currently there are a number of ways of running tasks and setting parameters. The system is still under development, and depending on user feedback, we may decide to eliminate some of them. Below we identify our preferred methods, which we do not intend to eliminate and which we recommend for writing scripts.

5.1 The PyRAF Interpreter Environment

When the PyRAF system is started using the **pyraf** command as described previously, the user’s commands are actually being passed to an enhanced interpreter environment that allows use of IRAF CL emulation and provides other capabilities beyond those provided by the standard Python interpreter. In fact, when **pyraf** is typed, a special interpreter is run which is a front end to the Python interpreter. This front-end interpreter handles the translation of CL syntax to Python, command logging, filename completion, shell escapes and the like which are not available in the default Python interpreter.

It is also possible to use PyRAF from a standard Python session, which is typically started by simply typing **python** at the Unix shell prompt. In that case the simple CL syntax for calling tasks is not available and tab-completion, logging, etc., are not active. For interactive use, the conveniences that come with the PyRAF interpreter are valuable and we expect that most users will use PyRAF in this mode.

One important thing to understand is that the alternate syntax supported by the PyRAF front end interpreter is provided purely for interactive convenience. When such input is logged, it is logged in its translated, Python form. Scripts should always use the normal Python form of the syntax. The advantage of this requirement is that such scripts need no preprocessing to be executed by Python, and so they can be freely mixed with any other Python programs. In summary, if one runs PyRAF in its default mode, the short-cut syntax can be used; but when PyRAF is being used from scripts or from the standard Python interpreter, one must use standard Python syntax (not CL-like syntax) to run IRAF tasks.

Even in Python mode, task and parameter names can be abbreviated and, for the most part, the minimum matching used by IRAF still applies. As described above, when an IRAF task name is identical to a reserved keyword in Python, it is necessary to prepend a ’PY’ to the IRAF task name (i.e., use `iraf.PYlambda`, not `iraf.lambda`). In Python mode, when task parameters conflict with keywords, they must be similarly modified. The statement `iraf.imcalc(in="filename")` will generate a syntax error and must be changed either to `iraf.imcalc(PYin="filename")` or to `iraf.imcalc(input="filename")`. This keyword/parameter conflict is handled automatically in CL emulation mode.

Some of the differences between the PyRAF interpreter and the regular Python interpreter besides the availability of CL emulation mode:

	PyRAF interpreter	Python default interpreter
prompt	`-->`	`>>>`
print interpreter help	`.help`	n/a
exit interpreter	`.exit`	EOF (control-D) or `sys.exit()`
start logging input	`.logfile` *filename*	n/a
append to log file	`.logfile` *filename* `append`	n/a
stop logging input	`.logfile`	n/a
run system command	`!` *command*	`os.system('`*command*`')`
start a subshell	`!!`	`os.system('/bin/sh')`

5.2 Example with Standard Python Syntax

This example mirrors the sequence for the example given above in the discussion of CL emulation (§2). In the discussion that follows we explain and illustrate some variants.

```
% pyraf
--> iraf.imheader("dev$pix", long=yes)
dev$pix[512,512][short]: m51  B  600s
No bad pixels, min=-1., max=19936.
Line storage mode, physdim [512,512], length of user area 1621 s.u.
Created Mon 23:54:13 31-Mar-1997, Last modified Mon 23:54:14 31-Mar-1997
Pixel file "HDR$pix.pix" [ok]
'KPNO-IRAF'           /
'31-03-97'            /
IRAF-MAX=           1.993600E4  /  DATA MAX
IRAF-MIN=          -1.000000E0  /  DATA MIN
IRAF-BPX=                   16  /  DATA BITS/PIXEL
IRAFTYPE= 'SHORT   '            /  PIXEL TYPE
CCDPICNO=                   53  /  ORIGINAL CCD PICTURE NUMBER
.
.
.
HISTORY '24-04-87'
HISTORY 'KPNO-IRAF'           /
HISTORY '08-04-92'            /
--> iraf.imstat('dev$pix')
#            IMAGE      NPIX      MEAN    STDDEV       MIN       MAX
           dev$pix    261626     105.1     75.86        -1.     1000.
--> iraf.imcopy("dev$pix","mycopy.fits")
--> iraf.stsdas()


      +-------------------------------------------------------------+
      |       Space Telescope Science Data Analysis System          |
      |          STSDAS Version 2.1, September 29, 1999             |
      |                                                             |
      |   Space Telescope Science Institute, Baltimore, Maryland    |
      |         For help, send e-mail to help@stsci.edu             |
      +-------------------------------------------------------------+


stsdas/:
 analysis/      examples        hst_calib/      sobsolete/
 contrib/       fitsio/         playpen/        toolbox/
 describe       graphics/       problems
```

```
--> iraf.fitsio()
--> iraf.catfits("mycopy.fits")
EXT#  FITSNAME      FILENAME              EXTVE DIMENS     BITPI OBJECT

0     mycopy.fits                               512x512    16    m51  B  600s
--> iraf.imhead('dev$pix',long=yes,Stdout='devpix.header')
```

The mapping of IRAF CL syntax to Python syntax is generally quite straightforward. The most notable requirements are:

- The task or package name must be prefixed with `iraf.` because they come from the iraf module. In scripts, use

  ```
  from pyraf import iraf
  ```

  to load the iraf module. Note that the first time PyRAF is imported, the normal IRAF startup process is executed (which can take a while). We are working on techniques to do fast, lightweight startups for stand-alone Python scripts that use PyRAF. At the moment the only such approach is to startup in a directory with a custom version of login.cl that defines a minimal IRAF environment.

  It is also possible to import tasks and packages directly using

  ```
  from pyraf.iraf import stsdas, imcalc
  ```

  With this approach, packages are automatically loaded if necessary and tasks can be used without the `iraf.` prefix. Like the IRAF CL, packages must be loaded for tasks to be accessible.

- The task must be invoked as a function in Python syntax (i.e., parentheses are needed around the argument list). Note that parentheses are required even if the task has no arguments – e.g., use `iraf.fitsio()`, not just `iraf.fitsio`.

- String arguments such as filenames must be quoted.

Another change is that boolean keywords cannot be set using appended + or − symbols. Instead, it is necessary to use the more verbose `keyword=value` form (e.g., `long=yes` in the example above). We have defined Python variables `yes` and `no` for convenience, but you can also simply say `long=1` to set the (abbreviated) `longheader` keyword to true.

Emulating pipes in Python mode is also relatively simple. If a parameter `Stdout=1` is passed to a task, the task output is captured and returned as a list of Python strings, with one string for each line of output. This list can then be processed using Python's sophisticated string utilities, or it can be passed to another task using the Stdin parameter:

```
--> s = iraf.imhead("dev$pix", long=yes, Stdout=1)
--> print s[0]
dev$pix[512,512][short]: m51  B  600s
--> iraf.head(nl=3, Stdin=s)
dev$pix[512,512][short]: m51  B  600s
No bad pixels, min=-1., max=19936.
Line storage mode, physdim [512,512], length of user area 1621 s.u.
```

`Stdin` and `Stdout` can also be set to a filename or a Python filehandle object to redirect output to or from a file. `Stderr` is also available for redirection. Note the capital 'S' in these names – it is used to eliminate possible conflicts with task parameter names, but in the future we may decide to simplify these names by eliminating the uppercase 'S'.

5.3 Setting IRAF Task or Package Parameters

Currently there are multiple ways of setting parameters. The most familiar is simply to provide parameters as positional arguments to the task function. For example

```
--> iraf.imcopy("dev$pix","mycopy.fits")
```

Alternatively, one can set the same parameters using keyword syntax:

```
--> iraf.imcopy(input="dev$pix",output="mycopy.fits")
```

Hidden parameters can only be set in the argument list this way (analogous to IRAF). As in the IRAF CL, the parameter values are learned for non-hidden parameters (depending on the mode parameter settings) but are not learned (i.e., are not persistent) for hidden parameters.

But parameters can also be set by setting task attributes. For example:

```
--> iraf.imcopy.input = "dev$pix"
--> iraf.imcopy.output = "mycopy.fits"
--> iraf.imcopy()  # run the task with the new values
```

These attribute names can be abbreviated (don't expect this behavior for most Python objects, it is special for IRAF task objects):

```
--> iraf.imcopy.inp = "dev$pix"
--> iraf.imcopy.o = "mycopy.fits"
```

PyRAF is flexible about the types used to specify the parameter so long as the conversion is sensible. For example, one can specify a floating point parameter in any of the following ways:

```
--> iraf.imstat.binwidth = "33.0"
--> iraf.imstat.binwidth = "33"
--> iraf.imstat.bin      = 33.0
--> iraf.imstat.bin      = 33
```

but if the following is typed:

```
--> iraf.imstat.bin = "cow"
Traceback (innermost last):
  File "<console>", line 1, in ?
ValueError: Illegal float value 'cow' for parameter binwidth
```

An error traceback results. When running in the PyRAF interpreter, a simplified version of the traceback is shown that omits functions that are part of the pyraf package. The `.fulltraceback` command (which can be abbreviated as can all the executive commands) will print the entire detailed traceback; it will probably only be needed for PyRAF system developers. Python tracebacks can initially appear confusing, but they are very informative once you learn to read them. The entire stack of function calls is shown from top to bottom, with the most recently called function (where the error occurred) listed last. The line numbers and lines of Python code that generated the error are also given.

One can list the parameters for a task using one of the following commands (in addition to the usual IRAF `lpar imcopy`):

```
--> iraf.imcopy.lParam()
--> iraf.lpar(iraf.imcopy)      # Note there are no quotation marks
--> iraf.lpar('imcopy')
```

For those who have encountered object-oriented programming, `iraf.imcopy` is an 'IRAF task object' that has a method named `lParam` that lists task parameters. On the other hand, `iraf.lpar` is a function (in the iraf module) that takes either an IRAF task object or a string name of a task as a parameter. It finds the task object and invokes the `.lParam()` method.

One can start the EPAR utility for the task using a parallel set of commands:

```
--> iraf.imcopy.eParam()
--> iraf.epar(iraf.imcopy)
--> iraf.epar('imcopy')
```

Tasks appear as attributes of packages, with nested packages also found. For example, if you load the `stsdas` package and the `restore` subpackage, then the `mem` task can be accessed through several different means: `iraf.mem`, `iraf.stsdas.mem`, or `iraf.restore.mem` will all work. Ordinarily the simple `iraf.mem` is used, but if tasks with the same name appear in different packages, it may be necessary to add a package name to ensure the proper version of the task is found.

5.4 Other Ways of Running IRAF Tasks

One way of reducing the typing burden (interactively or in scripts, though perhaps it isn't such a good idea for scripts) is to define an alias for the iraf module after it is loaded. One can simply type:

```
--> i = iraf
--> i.imcopy('dev$pix','mycopy.fits')
--> i.imstat('mycopy.fits')
```

But don't use `i` for a counter variable and then try doing the same! E.g.,

```
--> i = 1
--> i.imcopy('dev$pix','mycopy.fits')
```

will give you the following error message

```
AttributeError: 'int' object has no attribute 'imcopy'
```

since the integer '1' has no imcopy attribute.

6 Command Line Options

There are a few command-line options available for PyRAF:

-s Silent initialization (does not print startup messages)

-n No splash screen during startup

-v Set verbosity (repeated 'v' increases the level; mainly useful for system debugging)

-m Run the PyRAF command line interpreter to provide extra capabilities (default)

-i Do not start the special PyRAF interpreter; just run a standard Python interactive session

-h List the available options. There are long versions of all options (e.g., `--help` instead of `-h`) which are also described.

A save filename (see §2.1) can be given as a command line argument when starting up PyRAF, in which case PyRAF is initialized to the state given in that file. This allows you to start up in a particular state (preserving the packages, tasks, and variables that have been defined) and also reduces the startup time.