PyRAF
PyRAF FAQ

Last edited: 6 Jan 2017.

0. Support Status

Due to reduced budgets for HST, we can only provide minimal pyraf support. If you have a problem that cannot be solved quickly by our first tier support, we may be unable to help. This is unfortunate, but it is the best we can do with the resources available.

1. Installation

2. Startup

3. Tasks

4. Parameters

5. Graphics

6. General information

1. Installation

1.1. How do I download and install PyRAF?

Instructions for downloading and installing the latest version of PyRAF

There are two ways to download and install PyRAF: either as part of the larger "stsci_python" package, or as the stand-alone version. Installation instructions for both cases are provided.

1.2. I am on a Mac. Can I use PyRAF with/without running X11?

Yes - either is possible. PyRAF v1.8 included a stable, Mac-native version of PyRAF - a way to run PyRAF natively on MacOS, via Aqua libraries, without calling or linking with or requiring X11 in any way. At the time, this was enabled via the use of the environment variable named PYRAF_WUTIL_USING_AQUA (no longer needed). This provides standard PyRAF capabilities using native Aqua graphics, as well as the automatic focus changing and mouse moving, like PyRAF does on Linux

This should work out of the box, but for some users/installations you will need to:

install "PyObjC" (pythonhosted.org/pyobjc/install.html) under the non-X linked Python you are using
note: PyObjC is easy to install via both "pip" and "easy_install"

If your Mac does not have PyObjC installed, you will get a message saying "Could not import aqutil", but you will still be able to use the basic PyRAF functionality as described above.

UREKA USERS: as of PyRAF v1.11, this became the standard version delivered for MacOS, and it does not require anything different during installation. The Python installed via Ureka for MacOS is, like the native Frameworks version (e.g. /usr/bin/python), natively linked (Aqua), not X11-linked.

In addition, if you want to see smooth AGG graphics and scalable fonts, see the matplotlib question in the Graphics section.

1.2.1. But wait, I actually want an X11 version of PyRAF for the Mac.

If you prefer the X11 version (e.g. say you are remotely logging into a powerful Mac server and want to display its X11 graphics back to your local node), and you don't mind the requirements to have X11 installed and an X server running in the background, you can still build an X11-friendly PyRAF from the same PyRAF source code. Simply download the stand-alone tar-ball of PyRAF, install it under an X11-linked version of Python, and set the environment variable:

```
setenv PYRAF_WUTIL_USING_X 1
```

The most important step here is to get an X11-linked version of Python itself.

Your X11-linked version of Python will **need to have been built with X11-linked Tcl/Tk libraries**, not the native versions (and maybe libpng as well). Tcl/Tk source code can be found at opensource.apple.com/tarballs, and when you build Python itself, you may need to edit its setup.py file (for building extensions) to disable detect_tkinter() and detect_tkinter_darwin(), so that it does not seek/find the Framework (Aqua) version of Tcl/Tk when building.

To verify which way your Python installation is linked, run the visual test described in this graphics FAQ.

Then, to build PyRAF, use the full pathname to that particular version of Python in the build command. E.g., let's say you have two versions of Python on your machine (one X11-linked in /bobs/ur/uncle/bin/python, and one natively-linked in /usr/bin/python). And let's say you prove to yourself that /bobs/ur/uncle/bin/python is the X11 version. Then, all you do is build PyRAF with it via:

```
/bobs/ur/uncle/bin/python setup.py install
```

Please let us know how it goes.

1.2.2. OK, got it, but I am still having graphics trouble which may be X11-related...

If you are able to run "pyraf" but have trouble when displaying graphics (or see graphics errors upon startup) please follow the instructions in FAQ 5.13 and send us the results.

1.3. Can I run PyRAF without IRAF?

Yes. As of v1.9, PyRAF may be run without an IRAF installation. In this mode, the PyRAF Python shell will run, and - although no IRAF tasks will be found (imheader, implot, etc) - the basic capabilities of the PyRAF command-line are still supported. Some examples of working features are:

parameter handling (e.g. epar/tpar/lpar/dpar)
CL script conversion to Python (except of course any IRAF tasks called)
both Python and CL syntax on the command-line
minimum matching
tab-completion
aliases
command history (up-arrow), which is persistent after exit
access to shell variables and commands
spawning native executables
behind-the-scenes conversion to parenthesis (e.g. "sleep 3" -> "sleep(3)")
the full span of Python's capabilities and Tkinter graphics

Probably the most important missing features in such a setup would be:

the actual data-manipulating IRAF tasks (consider Python versions! tried numpy?)
native IRAF graphics/plotting (consider matplotlib)
some IRAF system commands

Let us know what your favorite/needed IRAF commands are - we may consider implementing some of them directly in Python.

1.3.1. What do I have to do?

On platforms where IRAF is not available (e.g. Windows without Cygwin), the "No-IRAF" mode will be entered automatically upon starting PyRAF. If you wish to try this on another platform (where IRAF is available but is not installed), simply set the environment variable "PYRAF_NO_IRAF". Otherwise, PyRAF will run as usual looking for an installation of IRAF.

1.3.2. What if I have IRAF, but want to try this anyway?

We're not sure there will be a lot of need for this particular case, other than pure curiosity. Nonetheless, if you have IRAF installed and want to run PyRAF without it, perform the following steps before starting pyraf:

setenv PYRAF_NO_IRAF
setenv iraf /some/invalid/path/name
remove your login.cl file (back it up first!)

1.3.3. What are the requirements?

On platforms where IRAF is not supported, all that is required to run PyRAF is Python (including readline), and the stand-alone PyRAF download.

Please let us know how it goes, and tell us about your use cases for this.

1.4. What are the errors about Python.h / numpy.h ?

Some linux distributions separate their Python (and Numpy) packages into a "user" package and a "developer" package. If you get an error about missing include files, your system is probably missing the "developer" package. You need to find and install it.

The developer package is usually named the same as the user package, with "-dev" or "-devel" in the name. The name varies among distributions, but look for "python-dev", "python-devel", "numpy-dev", "numpy-devel", or something similar.

1.5. Can I run PyRAF with the 64-bit version of IRAF (v2.15)?

64-bit

Yes. PyRAF now supports 64-bit IRAF 2.15.1a, after some initial difficulty with base v2.15. Definitely use 2.15.1a instead of v2.15. Be aware that some users have reported problems (not with PyRAF but with 64-bit IRAF tasks themselves), but most are having success with v2.15.1a. Note that all known PyRAF graphics-interface problems (due to changes in v2.15.*) have been resolved in PyRAF v1.11 (see ticket #156). Please also read this statement concerning STSDAS.

32-bit

Yes. The 32-bit IRAF 2.15.1a is fully supported by PyRAF.

PyRAF v1.11, and the nightly PyRAF tar-ball, work correctly with IRAF 2.15.1a. The latest version of PyRAF can be found here.

As far as IRAF 2.15.* goes, testing with PyRAF is ongoing. We have so far found and resolved a few graphics problems (WCS errors, see Trac ticket #156) generating errors like:

```
ERROR (851, "Cannot restore graphics world coordinate systems")
```

or

```
IrafError: IRAF graphics WCS is singular!
```

as well as one minor IRAF bug (topic 90578; likely corrected in IRAF 2.15.1a) causing a stack trace upon PyRAF startup which includes the following line:

```
pytools.irafglobals.IrafError: Attempt to set unknown lone parameter dp
```

This is due to a typo in iraf/unix/hlib/extpkg.cl. To fix it, you will need to change line 23 in that file (within your 2.15 IRAF installation) from:

```
dp = mktemp ("tmp$dpkg")
```

to this:

```
dpkg = mktemp ("tmp$dpkg")
```

Then PyRAF should start up as usual.

1.6. Does PyRAF run on OSX Mountain Lion (10.8)?

Yes. Some users have noticed problems after the upgrade, but they are not problems with PyRAF. One example is where "python" points to a different python in their PATH after the upgrade, than the python installation where PyRAF was installed. Another has to do with a missing GCC compiler, but this is a problem with installing PyRAF, not running it. This issue is fixed by installing the latest Xcode or another GCC compiler.

Another issue is that some users are seeing PyObjC warnings such as

```
2013-05-14 10:52:58.724 python[1384:303] PyObjCPointer created: at 0x7fff7945fa40 of type {__CFBoolean=}
```

We believe this to be an annoying though harmless debugging message that the author of PyObjC accidentally left in that code. This only occurs on Mountain Lion, and PyRAF is not adversely affected.

1.7. How do I run PyRAF on Windows?

First, make sure you are aware of what you are asking. PyRAF functionality on Windows is severely limited to the few available tasks which run on Windows currently - Python tasks (e.g. see drizzlepac), because IRAF itself does not run on Windows. Please read FAQ 1.3.

Next, install the Windows version. This will be available on the stsci_python download page, but it is also available as a windows version of the nightly PyRAF tar-ball on the PyRAF download page.

If you install from the nightly tar-ball (see the instructions), you will see a PyRAF icon on your desktop. Simply double-click this. If you do not have that icon, you may type "pyraf" in a DOS window for the same result.

NOTE: PyRAF on Windows requires that the Python-installation "Scripts" directory be added to your PATH. This should have occurred when you installed Python, or at least you should have been reminded or instructed to do this.

If you see the following:

```
'runpyraf.py' is not recognized as an internal or external command, operable program or batch file.
```

it means that the Python "Scripts" directory is not on the user's PATH.

1.8. Does PyRAF work under Python 3?

Yes! In May of 2012, a beta version was made available for download on the stand-alone download page. Be aware that the full stsci_python software suite is not yet ported. The code in the download for Python 3 is the same exact code base as the Python 2 version, except that it will be run through the "2to3" tool automatically for you during installation.

Installation instructions are exactly the same for both versions (Python 2.x or 3.x).

As of this writing, there are no known bugs specific to the Python 3 version of PyRAF (i.e. aside from any issues already known to exist under Python 2.x), however the following limitations and caveats currently exist. Testing is ongoing and user feedback will be very helpful.

Under Python 3:

Third party software "matplotlib" is not yet officially ported, though it's development trunk does build under Python 3 and this is expected to be official/live by Summer/Fall 2012. This affects the use of the "matplotlib" graphics kernel (see FAQ 5.8).
CL script caching to disk is not yet implemented (due only to development resource constraints). This may have a slight performance impact, or it may be unnoticed. PyRAF still translates CL script code to Python and runs it as usual, but it is not loading pre-translated code objects from disk.
Third party OSX software "PyObjC" is not yet ported to Python 3. This only affects PyRAF on OSX when running with native/Aqua Python (see FAQ 1.2). Graphics will still work, just without the automatic focus changing and mouse moving.
The stsci.tools package is only partially ported (only the parts which PyRAF requires, e.g. TEAL) for now. This work is expected to be completed soon. If you rely on scripts/modules within stsci.tools (you would know), then you may need the Python 2 compliant version obtained with all of stsci_python. If you don't know what this refers to, then don't worry about it - you are not affected by it.
The pyraf/tools directory is only partially ported. This only affects PyRAF developers as it is not installed with PyRAF.
Only Python 3.2 and 3.3.0 have been used in testing so far. PyRAF may or may not work correctly with Python 3.1. It will likely NOT work correctly with any variant of Python 3.0.
We will attempt to keep this list up to date, but sending an email to help@stsci.edu may be your best bet for the latest Python 3 support information.

1.9. What is "X11/X.h: No such file or directory"?

When you install PyRAF, it compiles a C extension to interact with X Windows (needed if on Linux, or if using X on OSX). This requires that you have installed, on your machine, the X Windows development libraries and include files. On Linux, you can install these with your package manager. The details vary in different Linux distributions, but the package you want is usually named something like "libx11-dev", "x11-dev" or "libX11-devel". Install this with your Linux package manager or whichever of yum/apt-get you have on your machine, then try installing PyRAF again.

If you wish to test your installation to see if the "xutil" C-extension built properly when you installed PyRAF, please run the following (after seeing FAQ 5.13 to help determine which **"python"** executable to use):

```
/absolute/pathname/to/python -c 'from pyraf import xutil; print(xutil); print(dir(xutil))'
```

If successful, it will list a valid path to the xutil library, and the cursor-moving function names.

If the xutil build was unsuccessful, it will say:

```
ImportError: cannot import name xutil
```

in which case the most likely problem is that your machine is missing the above-mentioned packages. To confirm this issue, manually re-run the installation of the pyraf package itself, via `"python setup.py install"`, and watch for any errors.

If you have questions, please send us the output.

1.10. Does PyRAF work within IPython Notebook?

It should, but with two caveats:

1. you will need to use the direct Python API (see below)
2. you may not be able to use IRAF graphics tasks (for now, see below)

This is a mostly untested feature. Let us know how it works for you...

Explanation

In general, there are two main ways to use PyRAF:

1. use the PyRAF command line (Python version of IRAF's CL), or
2. use the PyRAF/Python API to get to IRAF tasks from pure Python (scripting in Python)

The former is achieved by running `"pyraf"` in the terminal shell. This will NOT work as expected with the IPython Notebook.

The latter is achieved by running a python session (or ipython session) by typing `"[i]python"` in the shell, and then by typing "from pyraf import iraf" and calling IRAF tasks via lines like `"iraf.imstat('dev$pix')"`, etc. This should work in the IPython Notebook, as long as PyRAF is installed and may be imported inside of IPython. Try it first in the terminal version of IPython, and make sure that works (imports correctly), before trying it in the Notebook setting.

You will also likely want to turn off PyRAF graphics since they may not display correctly from inside the notebook web page (see FAQ 2.5 for how to turn off graphics). We have heard of some however, who have been able to get PyRAF graphics to pop up in separate Tkinter windows from the notebook via IRAF graphics tasks, and be responsive (local server, Mac OSX 10.8). This behavior may not always work. If there is enough interest, we may work on developing this capability using the matplotlib notebook version graphics kernel - please let us know.

In the meantime, you should be able to plot your data via direct matplotlib calls within IPython Notebook.

1.11. Do you support other downloads of PyRAF??

Sorry, no. We are only able to provide support for the software that we package and provide here (or from places described there such as pypi).

You may find that you are able to get PyRAF packaged from other sources (e.g. Scisoft). Of course, since we have no control over what is provided by other sources, or what versions are used, or how libraries are built or packaged, we cannot vouch for or support that software. Note that the Scisoft packagers state here that their package is not supported by STScI. This FAQ entry is provided because we have received numerous help calls about difficulties with such unsupported configurations.

How should I obtain PyRAF?

Simply download and install PyRAF (or all of stsci_python) from the supported locations described at this web site:

> www.stsci.edu/institute/software_hardware/pyraf

Also, if you are looking for an easy, one-click install-everything option, you may now try Ureka:

> ssb.stsci.edu/ureka

2. Startup

2.1. How do I execute a Python script during startup?

If you are interested in execution of code passed in on the command line, please see this FAQ, but if you are interested in augmenting your startup, read on.

You can do this, although it is a bit obscure. If the PYTHONSTARTUP environment variable is set to a filename, Python (and PyRAF) execute the Python code in that file at startup. You can do any general Python initialization that you like in that file; it is also possible to do some special initialization if the user is running PyRAF. Here is one approach:

(1) Define PYTHONSTARTUP (if you haven't already):

```
setenv PYTHONSTARTUP "/home/rlw/python/pythonstartup.py"
```

(2) Add this special code in PYTHONSTARTUP:

```
# PyRAF initialization
          import sys, os
          executable = sys.argv[0]
          while os.path.islink(executable):
                executable = os.readlink(executable)
          if os.path.split(executable)[1] == "pyraf":
                # this code executes only if this is a pyraf session
                from pyraf import iraf
                startup = iraf.osfn("home$pyraflogin.py")
                if os.path.exists(startup):
                      execfile(startup)
                del startup
          del executable   # clean up namespace
```

What that does is execute a file in your IRAF home directory named pyraflogin.py, if it exists. (This is similar to the approach the CL uses for home$loginuser.cl.) That file can do whatever initialization you want. It runs in the user's namespace, so it can define variables & functions, do imports, etc. It can also execute pyraf/iraf tasks. Here is a sample:

```
# import some Python modules
          import re, urllib
          # set an IRAF environment variable
          iraf.set(mypython="home$python/")
          # run an IRAF task
          iraf.spy()
```

We will probably eventually migrate this capability into the PyRAF startup file itself so it is easier to use.

2.2. How can I make my PyRAF session start up faster?

Use the 'saveToFile' function to save a copy of your environment to a file, and specify that file on the command line when you start up. Immediately after starting PyRAF, type the command

```
saveToFile mystartup.save
```

or in Python syntax

```
iraf.saveToFile('mystartup.save')
```

Then next time you start up PyRAF, use

```
pyraf mystartup.save
```

to run PyRAF. Depending on how many packages and tasks you define during your startup process, this will start up about twice as fast as the standard login process.

The disadvantage of this approach is that it also restores parameters for any tasks and packages that run during the startup process to their values when you did the save. But that should not be a problem as long as you don't run any complex tasks at startup time. You should repeat this process (starting up PyRAF with no initial save file) when your login.cl or the IRAF system changes.

2.3. Why do I get messages about the "CL script cache" when I start up?

You may see messages when you start up PyRAF:

```
Unable to open CL script cache file /home/rlw/iraf/pyraf/pyraf.Database
        Unable to open any CL script cache for writing
```

This is a sign that you are running a very old version of PyRAF and should upgade to the current version. The old version had some restrictions on running multiple PyRAF sessions simultaneously, but the current version has no such restrictions.

2.4. Why does PyRAF fail on startup with a message about togl.so?

If PyRAF has the following error on start-up, the problem is the directory path to the Togl shared object (togl.so) is not set.

```
File "/usr/ra/pyraf/irafimport.py", line 30, in _irafImport
          return _originalImport(name, globals, locals, fromlist)
       ImportError: ld.so.1: /usr/local/bin/python: fatal: togl.so: open
       failed: No such file or directory
```

One can resolve this issue by adding the togl.so directory path to the default search path for libraries. This can be done by adding the following line to your .setenv file:

```
% setenv LD_LIBRARY_PATH /yourToglDirectoryPath:${LD_LIBRARY_PATH}
```

Togl should be a subdirectory of the directory location where the Tcl/Tk libraries reside. The Togl subdirectory should contain togl.so.

2.5. How can I tell PyRAF that I want to run without graphics?

Say, for example, you are at a remote terminal, or a terminal without any X Windows support. You can tell PyRAF to skip all graphics initialization and run in terminal-only mode, either via:

```
% pyraf -x
```

or, via an environment variable

```
% setenv PYRAF_NO_DISPLAY 1
% pyraf
```

or, to do so programmatically,

```
% python
>>> from stsci.tools import capable
>>> capable.OF_GRAPHICS = False
>>> from pyraf import iraf
```

Of course, any tasks which attempt to display graphics will fail. See `pyraf --help` for more information on run time arguments.

2.6. Why do I see "Could not import aqutil"? What's that?

This may occur on a Mac running the native (Aqua) version of PyRAF/Python, if your installation does not yet include PyObjC. If the only warning you saw was "Limited graphics available", then your PyRAF installation is fine, but is simply missing a few extra features.

You should install PyObjC into the Python on your Mac via "pip" or your favorite method.

This and other graphics issues are described in FAQ 1.2 and FAQ 5.13.

2.7. I always use the same options at startup - is there a shortcut?

Yes. Any command-line options that you put in the environment variable PYRAF_ARGS will be automatically appended to the command-line at run-time.

For example, if PYRAF_ARGS is set to '--ipython -n', then typing:

```
% pyraf
```

would be equivalent to typing:

```
% pyraf --ipython -n
```

This only applies to the PyRAF command line, as opposed to the Python API (e.g. 'from pyraf import iraf').

2.8. Why do I get "CL syntax error at 'NEWLINE'"?

```
SyntaxError: CL syntax error at `NEWLINE' (line 28)
```

Have you recently changed your login.cl file?

You very likely have a line in your login.cl with no right-hand-side value. Look at the line number listed in the exception (line 28 in the example above).

2.9. How do I execute code from the command line (i.e. how do I run non-interactively)?

There are three ways to run non-interactively:

1. The Python API: import PyRAF into your python script

   Run a Python script which contains, for example:

   ```
   from pyraf import iraf
            iraf.images(_doprint=0)
            iraf.imutil(_doprint=0)
            iraf.plot(_doprint=0)
            iraf.imheader("dev$pix")
            iraf.implot("dev$pix")
   ```

   produces the imheader output and pops up the implot graphics window.

2. **Redirect Stdin:** "pyraf -s < faq.cl" (.cl can be a mix of CL and Python)

   Running the pyraf command above with a file (e.g. faq.cl) containing, for example:

   ```
   clear
            print "in faq_item.cl..."
            images
            imutil
            imheader "dev$pix"
            .exit
   ```

   produces this output:

   ```
         in faq_item.cl...
                  dev$pix[512,512][short]: m51  B  600s
   ```

3. The Dash-C Argument: "pyraf -s -c 'my_pyraf_cmd' "

   The value of the -c option can be CL or Python - any legal PyRAF code:

   ```
         % pyraf -s -c 'images; imutil; print "hello"; imheader dev$pix'
   ```

produces this output:

```
hello
        dev$pix[512,512][short]: m51  B  600s
```

2.10. Why does PyRAF fail on startup with "ImportError: No module named gki"?

Do you get this?

```
ImportError: No module named gki
```

There are actually a couple of errors which will cause this strange message, most of them having nothing to do with the gki module. This occurs because the package-loading error handling during startup is different (more touchy) than it is during normal operation. Also complicating matters is the fact that PyRAF necessarily overrides Python's import builtin in order to simulate IRAF package loading. Often gki is being imported (in turn, itself importing the iraf module) when an IRAF startup error occurs.

Missing or corrupt login.cl

So for example, if you see the error listed above, but inside the stack trace you ALSO see this:

```
KeyError: "Undefined environment variable `helpdb'"
```

or:

```
IrafError: "Undefined variable `uparm' in string ..."
```

or maybe this:

```
AttributeError: Undefined IRAF task `vo'
```

then your problem isn't about gki. You have a user account for which the `'mkiraf'` script has not yet been run, which creates a login.cl file for you. Please do so.

If you already have a login.cl file, check it for errors. If you happen to have a loginuser.cl file somewhere (most PyRAF users do not seem to use this IRAF feature), please also check it for errors. This can most easily be done by making a safe copy of it and then deleting it.

Out of date cache

Or for example, if you see the error listed above, but inside the stack trace you ALSO see this:

```
File "", line 38
iraf.set(as = 'host$as/')
          ^
SyntaxError: invalid syntax
```

then this isn't about gki either. This is due to an outdated cache file for zzsetenv. You simply need to delete your cache:

```
/bin/rm -r ~/iraf/pyraf/clcache*
```

and start PyRAF fresh. An IRAF startup file (zzsetenv.def) uses "as" as a variable name, which PyRAF needs to handle specially since "as" is a reserved word in Python. Old PyRAF installs (from old Python versions) did not treat "as" specially and may still be around in the form of old cache files. To see your Python version's list of reserved words, type:

```
python -c 'import keyword; print(keyword.kwlist)' # used in irafutils
```

Anything else?

If you see this kind of error on startup, and it cannot be fixed by either deleting your cache (as above) or by creating/editing your login.cl file (or removing your loginuser.cl file), then please contact us, sending your full stack trace to help@stsci.edu.

2.11. Why does PyRAF fail on startup with "PANIC in x_system.e: interrupt save stack underflow"?

```
PANIC in x_system.e: interrupt save stack underflow
```

This is very likely due to using a new version of PyRAF (e.g. v1.11) along with an old version of IRAF (e.g. v2.14). PyRAF was changed in the later versions to keep up with changes in IRAF, including revisions to the the task communication (IPC) binary layout. Using the new PyRAF with the old IRAF may cause this error (noticed on Ubuntu).

2.13. Why does PyRAF tell me on startup "ValueError: Attempt to access undefined local variable curdir"?

```
ValueError: Attempt to access undefined local variable `curdir'
```

This is due to a recent version of extpkg.cl (IRAF 2.16.1). When PyRAF converts this to Python, the resulting code is invalid. The CL has this same issue as Python, but is simply less strict about variable and type checking. This can be easily remedied by adding the following line

```
curdir = "."
```

to iraf/unix/hlib/extpkg.cl around line number 14, or right before this line where the "curdir" variable is first used:

```
printf ("!pwd\n") | cl () | scan (curdir)
```

You may also need to delete your CL cache after this step:

```
% rm -r ~/iraf/pyraf/*
```

3. Tasks

3.1. How do I define an IRAF task in a Python script?

To define an IRAF task that is defined as a CL script, use the usual 'task' command. On the PyRAF command line, the syntax is familiar from IRAF:

```
--> task mytask=home$mytask.cl
```

Inside a Python script, you must convert this to the equivalent Python syntax:

```
from pyraf import iraf
         iraf.task(mytask="home$mytask.cl")
```

For information on how to define a task that calls a Python function instead of a CL script, see the next question.

3.2. How do I create an IRAF task that calls a Python function?

It is possible to define a Python function that is works just like any other IRAF task. It has parameters that have specific types (integer, string, etc.), are learned and persist between sessions, can be edited with epar, prompt for values, etc. Here is an example.

Create two files, pytestmodule.py and pytest.par. pytest.par is a standard IRAF parameter file that contains the parameter definitions:

```
spar,s,a,"spar default",,,"string parameter"
         j,i,h,5,,,"hidden integer parameter"
         bpar,b,h,no,,,"hidden boolean parameter"
         mode,s,h,"al"
```

The comma-separated fields in this file are the parameter name, type (s=string, i=integer, etc.), mode (h=hidden), default value, minimum value (or choice list), maximum value, and prompt string. See the IRAF documentation and numerous examples of .par files in the IRAF distribution for more information.

pytestmodule.py is a Python file that contains both the Python function definition and the IRAF task definition:

```
from pyraf import iraf
    # define a function to be wrapped as an IrafTask object
    def pytest(spar, j, bpar):
       """spar is a string, j is an int, bpar is a boolean; j, bpar are hidden"""
       print "pytest: spar `%s' j `%s' bpar `%s'" % (spar,j,bpar)
```

```
            k = j+5
            if bpar:
               print 'bpar true: k=j+5=',k
            else:
               print 'bpar false: k=j+5=',k

         # create the IrafTask object wrapping the function pytest
         # call it 'pytest', with parameters from 'pytest.par'
         t = iraf.IrafTaskFactory(taskname='pytest', value='pytest.par',
    function=pytest)
```

The parameters to the function need to be in the same order as in the parameter file. They do not necessarily need to have the same names as in the .par file (though they do in this example.) When the task is called, the task wrapper takes care of applying constraints (e.g. ensuring that parameters have the right types and that they meet the min/max/choice requirements.) It will also prompt for values for non-hidden parameters. Note that (unlike in IRAF CL tasks) the prompt always happens, even if the parameter is not used in the function, and it happens only once before the function is called.

The 'function' keyword parameter to IrafTaskFactory is what gets called when the task executes. There are no quotes around it (because it is not a string, it is the function itself.) If you want, you can specify a function using the full name, e.g. mod.func for a function 'func' defined in module 'mod'. The function can even be a method of some Python object, in which case it will have access to the instance attributes of that object.

The function is called with the actual values of the parameters (in the above example, a string, an int, and a boolean yes/no value.) If you change the values of the parameter variables inside the function, it does not change the corresponding IRAF parameter values. If you need to change the saved parameter values, you can do this inside the function:

```
      spar = "new value for spar"
             iraf.pytest.spar = spar
```

Once you have created the task object, it is automatically available at the pyraf command line:

```
 --> import pytestmodule
          --> lpar pytest
                 spar = spar default    string parameter
                   (j = 5)              hidden integer parameter
                 (bpar = no)            hidden boolean parameter
                 (mode = al)
          --> epar pytest
          --> pytest j=10
          string parameter ('spar default'): abcde
          pytest: spar `abcde' j `10' bpar `no'
          bpar false: k=j+5= 15
```

The task will also show up in the current package listing you get when you type '?'. Once the task is defined as shown above, you can invoke it from CL scripts too by calling it just like any other task. As far as PyRAF is concerned it is just another IRAF task. Naturally you can't use it from the IRAF

CL though.

Currently the only parameter types supported are those used by IRAF tasks. If you pass a Python variable not supported by IRAF (for example, a file handle or module) then the task wrapper attempts to convert it to the requested type. That can lead to some surprising behavior:

```
--> iraf.pytest(sys)
      = pytest: spar '<module 'sys' (built-in)>' j  = '5' bpar  = 'no'
       bpar false: k=j+5= 10
```

Note that the sys module's descriptive string is passed to pytest() instead of the sys module itself.

3.3. How do I set an IRAF environment variable?

At the PyRAF command line, set an IRAF environment variable uses the usual syntax:

```
set myenvvar=home$test/
```

The Python equivalent to this (which you can use inside Python scripts) is:

```
from pyraf import iraf
       iraf.set(myenvvar="home$test/")
```

3.4. How do I redirect output when running tasks in a Python script?

The shell-style redirection used at the PyRAF command line does not work inside Python scripts. Instead, Python tasks have some special keyword parameters (Stdin, Stdout, Stderr) that can be used for input & output redirection. Here are a few examples.

First, here are some simple cases in the command-line syntax:

```
--> imhead dev$pix > imhead.stdout
         --> imhead dev$pix >& imhead.stderr_and_stdout
         --> imhead dev$pix >& imhead.stderr_only > imhead.stdout_only
         --> head nlines=5 < .cshrc
```

The Python syntax equivalents to these are:

```
from pyraf import iraf
         iraf.imhead("dev$pix", Stdout="imhead.stdout")
         iraf.imhead("dev$pix", Stderr="imhead.stderr_and_stdout")
         iraf.imhead("dev$pix", Stderr="imhead.stderr_only",
   Stdout="imhead.stdout_only")
```

```
        iraf.head(nlines=5, Stdin=".cshrc")
```

When a string is specified for one of these special keyword parameters, it is assumed to be a file name. You can also pass a Python filehandle (e.g., as is returned from the built-in open() function). In fact, any Python object with the necessary read() or write() method can be used. This can be used to implement the equivalent of pipes. In command-line syntax:

```
--> imhead dev$pix long+ | head nlines=5
```

A PyRAF equivalent to this is:

```
import StringIO
        fh = StringIO.StringIO()
        iraf.imhead("dev$pix", long=1, Stdout=fh)
        fh.seek(0)
        iraf.head(nlines=5, Stdin=fh)
        fh.close()
```

Another special behavior of the Stdout keyword is that if it is set to a non-zero integer value, the output of the task is returned as a list of strings, with one string for each output line. For example,

```
--> s = iraf.imhead("dev$pix", Stdout=1)
        --> print s
        ['dev$pix[512,512][short]: m51  B  600s']
```

If such a list of lines is passed as the Stdin parameter, the input is read from it. So another equivalent to the piping example given above is:

```
s = iraf.imhead("dev$pix", long=1, Stdout=1)
        iraf.head(nlines=5, Stdin=s)
```

or even just:

```
        iraf.head(nlines=5, Stdin=iraf.imhead("dev$pix",long=1,Stdout=1))
```

The graphics output of a graphics task can be redirected as well, via StdoutG or StdoutAppendG:

```
        iraf.plot('dev$pix', row=256, StdoutG='myfile.gki') # equivalent to "--> prow dev$pix 256 >G myfile.gki"
```

The output file (myfile.gki) consists of the binary GKI metacode plotting instructions, which (for the curious) may be converted to readable text via the gkidecode command.

3.5. Can I run IRAF tasks in parallel using PyRAF?

Yes. There are a few ways to do this. The usual concern with parallelizing IRAF tasks is the potential for conflicting parameter values being read from and written to the user's single ~/iraf/uparm directory. To run IRAF tasks in parallel we must get around this issue.

One way, which is unattractive, is to set up a full temporary IRAF-home area for each parallel task. That method is not described here.

Another way, which is the method we suggest, is to use the Python API to IRAF tasks provided by PyRAF for direct Python scripting. Using this API, you will want to set ALL non-default parameter values *DIRECTLY* in your script.

Example Using Multiprocessing Package

The following is a simple example using the 'imstat' task and the multiprocessing package, familiarity with both of which is assumed. The 'imstat' task below will not write any .par values to the user's uparm directory, and thus may be parallelized. Because of this and because of process-caching being disabled, it will be independent of all other concurrently running 'imstat' instances.

```
#!/usr/bin/env python
#
import multiprocessing, sys
from pyraf import iraf

# global items done in parent
iraf.prcacheOff()
iraf.set(writepars=0)
iraf.images()
iraf.imutil()
iraf.imstatistics.unlearn()

def run_a_task(fname, showformat):
    """ processing delegated to child process """
    print("Running imstat on:"+fname+", with format = "+str(showformat))
    iraf.imstat(fname, format=showformat, cache=False)

def main():
    NUM = 5
    # set up my input data, and create my processes (but don't start)
    subprocs = []
    for i in range(NUM):
       fname = 'copy_%d' % i
       iraf.imcopy('dev$pix', fname)
       showformat = i % 2 > 0  # odd numbered ones only, for fun
       p = multiprocessing.Process(target=run_a_task, args=(fname, showformat))
       subprocs.append(p)

    print("Starting all processes now")

    # now start them all
    for p in subprocs: p.start()
```

```
    # wait on them all
    for p in subprocs: p.join()

main()
```

produces the output:

```
dev$pix -> copy_0
dev$pix -> copy_1
dev$pix -> copy_2
dev$pix -> copy_3
dev$pix -> copy_4
Starting all processes now
Running imstat on:copy_0, with format = False
Running imstat on:copy_1, with format = True
Running imstat on:copy_2, with format = False
Running imstat on:copy_3, with format = True
Running imstat on:copy_4, with format = False
#               IMAGE      NPIX      MEAN    STDDEV       MIN       MAX
#               IMAGE      NPIX      MEAN    STDDEV       MIN       MAX
copy_0  262144  108.3154  131.298  -1.  19936.
               copy_1    262144     108.3     131.3       -1.    19936.
               copy_3    262144     108.3     131.3       -1.    19936.
copy_2  262144  108.3154  131.298  -1.  19936.
copy_4  262144  108.3154  131.298  -1.  19936.
```

Example Using Shell Script

The next example is a simple brute-force script using the C-Shell, but could be made more efficient if needed.

Let the following Python script (named "parallel_imstat.py ") represent a single task to be run concurrently with others like itself:

```
#!/usr/bin/env python
import sys
from pyraf import iraf
# init.: load pkg and clear out any par files
assert len(sys.argv) == 2, 'Usage: '+sys.argv[0]+' <upper-lim-value>'
iraf.set(writepars=0)
iraf.images()
iraf.imutil()
iraf.imstatistics.unlearn()
# this run
uprlimit = int(sys.argv[-1])
lwr = 10
if uprlimit % 2 == 0:
     lwr = 0
# set some optional non-default pars to prove our point
```

```
iraf.imstatistics.format = 'no'
iraf.imstatistics.lower = lwr
# run the task - give it the required par 'images'
iraf.imstatistics("dev$pix", upper=uprlimit)
# this may also be called as such:
# iraf.imstatistics(ParList="my_special_par_file.par")
```

And then invoke it a few times in parallel, putting it in the C-Shell background (via the ampersand):

```
% foreach ulim (22 33 44 55 66 77)
foreach? python parallel_imstat.py $ulim &
foreach? end
```

The result is the following output. Note that the printed output may come out of order, as it did here:

```
dev$pix  86  15.59302  3.266147  6.  22.
dev$pix  142  21.80986  7.694461  10.  33.
dev$pix  78897  50.73204  8.879236  6.  66.
dev$pix  105229  56.07858  12.1298  10.  77.
dev$pix  51662  45.30723  5.468132  10.  55.
dev$pix  25653  40.65236  2.680893  6.  44.
```

Writing *.par files

In default non-parallel use, IRAF tasks write out their parameter values to parameter files (ending in ".par") in a directory called "uparm" under the user's irafhome. If you want to be very certain that task parameter files are NOT being written out to disk (e.g. by different parallel task instances), you may turn off ALL *.par file writing by inserting the following line into your login.cl file:

```
set writepars = 0
```

or including this line in your script (as we have above):

```
iraf.set(writepars=0)
```

Without such a setting, PyRAF will as usual write task parameter values to disk automatically for you, to save/persist them between runs.

Process Cache

If your parallel tasks are going to use the same instance of PyRAF (and thus the same process cache), as in the case of running the entire parallel program inside a single Python script via, say, the multiprocessing module, you will want to turn off process caching. This turns off the ability for PyRAF to use/re-use the same continually running, connected, IRAF task subprocess to do the work of each call to the task. With no process caching allowed, each new call you make to the IRAF task will start a new, separate IRAF executable, which will live only as long as it is doing your work, which is what you want. To turn off process caching, include this line in your Python script:

```
iraf.prcacheOff()
```

if you imported iraf from pyraf as above. Do this immediately after you have imported "iraf" (e.g. "from pyraf import iraf"), whether that be in the controlling parent process, or in the child/subprocess.

PSETs

One thing to be aware of is whether your IRAF tasks use PSET parameters. Anything non-default will need to be specified in the Python script somehow, otherwise you will have multiple concurrent processes trying to access the same .par file for that PSET.

3.6. Why do I get this IrafError (Not a legal IRAF pipe record)?

I get the following error running an IRAF task:

```
IrafError: Error running IRAF task drizzle
('Not a legal IRAF pipe record', 32, 'Broken pipe')
```

This IRAF issue has been known to occur when account limits are too stringent. A quote from the IRAF web site explains:

"Specifically, pointers allocated in the normal course of a task may occasionally be at an address outside the user's per-process stack space, resulting in a 'memory has been corrupted' or 'segmentation violation' error."

While the CL startup script has some added code to increase the user process' stack space, PyRAF does not do this. The fix, if you run into this rare issue, is to increase your stacksize limit. Put one of the following lines into your shell startup script:

```
%limit stacksize unlimited # for tcsh users
%ulimit -s unlimited # for bash users
```

The good folks at Columbia describe their encounter with this issue.

3.7. Why do I get this Error (FXF: must specify which FITS extension)?

I get the following error running an IRAF image-handling task, related to FITS extensions:

```
IrafError: Error running IRAF task imcopy
IRAF task terminated abnormally
ERROR (1113, "FXF: must specify which FITS extension (...) ")
```

This is actually an IRAF error, due to changes between 2.14 and 2.16. This is not a PyRAF issue. Please see this IRAF bug report, or google "IRAF use_new_imt".

3.8. How do I double-quote string variables in a CL script?

Let us say you are writing/maintaining a CL script (which you want to run via PyRAF) which contains CL code with string variables as for example this procedure does:

```
procedure crj()
string mode = "al"
begin
      string crrejstr, crrejstr1, crrejstr2
      crrejstr1 = "!ls "
      crrejstr2 = "'abc*'"
      crrejstr = crrejstr1 // crrejstr2
      printf(">%s<\n", crrejstr)
      print(crrejstr) | cl
end
```

And let us also assume for some reason you cannot port this to direct Python (which would be your best option).

When you run this, you will find that the crrejstr2 string loses its double quotes during the string concatenation. This occurs because PyRAF sees the crrejstr2 as a string variable and does some special string-handling for you, assuming that you didn't really mean to double-quote it (as, indeed, in other cases it is a Good Thing that PyRAF does this checking on CL parameters).

The trigger for this behavior is that the two sets of quotes are right next to each other. To fix this, put spaces or text in between the outer and inner quotes, as in

```
      crrejstr2 = " 'abc*' "
```

The CL code should now act as intended when run through PyRAF.

3.9. What IRAF task should I use for my science?

This is a common help call question for those new to IRAF.

First and foremost we suggest you take one of the many available IRAF tutorials (web search: "IRAF Tutorial").

You may also find the IRAF FAQ helpful.

3.10. Why does my task say "Cannot open tmp file"?

If you see this or something like it, while using IRAF 2.16.1:

```
mscred> mscimage asd dsa
WCS reference image is asd[im4]
Resampling asd[im1] ...
ERROR: Cannot open file (tmp9374tk)
"nxblock=nxblk, nyblock=nyblk, verbose=no)"
line 202: mscsrc$mscimage.cl
called as: `mscimage (input=asd, output=dsa)'
mscred> ls tmp*
tmp9374uk.fits
```

Then, according to the IRAF developers, this "may be a known issue with the new image template code". To workaround it try:

```
reset use_new_imt = no
```

before running the task. You can set this in your login.cl or make a permanent system change in the hlib$zzsetenv.def file if it works.

See also: http://iraf.net/forum/viewtopic.php?showtopic=1468738&fromblock=yes

4. Parameters

4.1. How can I access CL parameters such as gcur, imcur, list, etc.? How can I do the equivalent of =gcur?

The cl has a number of predefined parameters, e.g. s1, s2, s3. If you assign a value to s1:

```
s1 = "test"
```

then you create a Python variable rather than using the cl variable of the same name.

To avoid this, access the cl parameters using 'iraf.cl.s1' instead. This form can be used both in assigning and retrieving values:

```
iraf.cl.s1 = "test"
print iraf.cl.s1
```

Getting a graphics cursor position using '=gcur' is just a special case (since gcur is a CL parameter). Its PyRAF equivalent is:

```
print iraf.cl.gcur
```

or, at the interactive command line, simply type:

```
iraf.cl.gcur
```

Note, do NOT accidentally type:

```
iraf.cl.gcur()
```

as the use of the parenthesis is incorrect - 'gcur' is not a function call.

4.2. How do I run epar on a remote terminal (without X Windows support)?

You can't run epar, because it requires graphics. However, you can run tpar - a text-based parameter editor added in PyRAF v1.3. This requires the 3rd party Python package named "urwid".

Note also that you can set parameters by changing the task's attributes as described in the PyRAF Tutorial.

4.3. How can I see the value of a string which is longer than the entry widget in epar?

You can use the left-most mouse button to do a slow "scroll" through the entry, or you can use the middle-mouse button to "pull" the value in the entry back and forth quickly. In either case, just click in the entry widget with the left-most or the middle mouse button and then drag the mouse to the left or the right. If there is a selection highlighted, the middle mouse button may paste it in when clicked. It may be necessary to click once with the left mouse button to undo the selection before using the middle button.

You can also use the left and right arrow keys to scroll through the selection. Control-A jumps to the beginning of the entry, and Control-E jumps to the end of the entry.

4.4. How do I get a list of the parameters for an IRAF task?

In PyRAF, IRAF task objects have a number of methods that can be useful in scripts. The getParList() method returns a list of the task parameters. Here is an example:

```
--> from pyraf import iraf
--> plist = iraf.imhead.getParList()
--> for par in plist:
...     print par
<IrafParS images s a 'dev$pix' None None "image names">
<IrafParS imlist s h '*.imh,*.fits,*.pl,*.qp,*.hhh' None None "default image names">
<IrafParB longheader b h no None None "print header in multi-line format">
<IrafParB userfields b h yes None None "print the user fields (instrument parameters)">
<IrafParS mode s h 'al' None None "">
```

```
<IrafParI $nargs i h 0 None None "">
```

The list returned by getParList() consists of parameter objects that contain the complete descriptions of each parameter. If you print them (as in the example), the string gives the parameter class, name, type, mode, value, minimum, maximum, and prompt string. You can retrieve or change the parameter values using the get() and set() methods:

```
--> par = plist[0]
--> print par
<IrafParS images s a 'dev$pix' None None "image names">
--> print par.get()
dev$pix
--> par.set('newvalue')
--> print par.get()
newvalue
```

Note that these parameters are actually shared with the original task (they are not copies), so modifying them also changes the task parameters:

```
--> lpar imhead
images= newvalue        image names
(imlist = *.imh,*.fits,*.pl,*.qp,*.hhh) default image names
(longheader = no)         print header in multi-line format
(userfields = yes)          print the user fields (instrument parameters)
(mode = al)
```

If you want a completely independent set of parameters, use the Python copy module or the optional docopy parameter to getParList:

--> plist1 = iraf.imhead.getParList(docopy=1) # this is a copy --> import copy --> plist2 = copy.deepcopy(iraf.imhead.getParList()) # so is this

Here are a few other task methods that may be handy for manipulating parameters:

```
getDefaultParList() Returns list of all parameter objects with default values.
 getParObject(name)  Returns the IrafPar object for the given parameter.
 getParDict()        Returns dictionary of all parameter objects.  Example:
       --> d = iraf.imhead.getParDict()
       --> print d['mode']
       <IrafParS mode s h 'al' None None "">
```

4.5. How do I create a copy of a task parameter list and edit it with epar?

The parameter list returned by the getParList() task method (see the previous question) cannot be editted directly using epar. However, you can convert it to an IrafParList object:

```
--> from pyraf import iraf
--> from pyraf.irafpar import IrafParList
--> plist = IrafParList('imhead', parlist=iraf.imhead.getParList(docopy=1))
```

IrafParList objects have methods to do various useful things:

```
lParam()                Lists parameters in lpar format
eParam()                Run parameter editor
dParam()                Print parameters in dpar (assignment) format
saveParList(filename)   Save parameters to file in .par format
```

Parameters that have been saved to a file can be used to create an IrafParList as well:

```
--> plist.saveParList('mylist.par')
5 parameters written to mylist.par
--> newplist = IrafParList('imhead', filename='mylist.par')
--> newplist.eParam()
```

You can get a complete list of the available IrafParList methods using help(plist).

4.6. How do I save parameters for a task in the uparm directory or in a file?

IRAF task objects have a special method, saveParList(), that saves the current list of parameters to a file in the standard IRAF .par format:

```
--> iraf.imhead.saveParList('myfile.par')
'5 parameters written to myfile.par'
```

If the filename is omitted, the parameters are saved in the uparm directory in a file with the usual name created from a combination of the task and package. You can print task.scrunchName() to see the filename.

If you just want to insure that the parameters used in the call to an IRAF task get saved in your uparm directory, include the special keyword parameter _save as a task parameter:

```
--> iraf.imhead('dev$pix', _save=1)
```

In the absence of the _save parameter, the parameters used for task execution do not get saved. Note that this is also the default when executing IRAF tasks from CL scripts, but in Python scripts you do have the option to decide that parameters should be saved.

See the answer to the previous question (on making copies of parameter lists and editing them with epar) for more information on using these .par files.

4.7. How do I run a task with a list of parameters?

The task parameter lists from the getParList() method or from .par files (see previous questions) can be used to run an IRAF task with a completely specified set of parameters. This can be a useful way to run the same task with various parameter sets for different purposes. For example, to create a file with a parameter set:

```
from pyraf import iraf
from pyraf.irafpar import IrafParList
task = iraf.imstat
plist = IrafParList(task.getName(), filename='mypars.par',
parlist=task.getParList(docopy=1))
iraf.epar(plist)
```

Now we have created the file 'mypars.par' with the desired parameters. To run the task using those parameters, set the special ParList task keyword parameter to the .par filename:

```
iraf.imstat(ParList='mypars.par')
```

or alternatively ParList can accept an IrafParList instead of a filename:

```
plist = IrafParList('', filename='mypars.par')
iraf.imstat(ParList=plist)
```

Or, you may have reached this FAQ simply intending to supply argument pairs in Python to the task in a "pythonic" way:

```
from pyraf import iraf
iraf.images()
iraf.imutil()
iraf.imstat('dev$pix', lower=50, upper=12000, format='no')
```

4.7.1 PSET Parameter Note

Note that if you are calling an IRAF task which has PSET parameters, you may specify them in the same argument list simply by using their unadorned name (unadorned by their PSET name), so long as there is no conflict with actual parameters of the main task, such as:

```
from pyraf import iraf
iraf.tables()
iraf.sgraph('dev$pix', pattern='dotted')  # the 'pattern' par is from the 'pltpar' PSET
```

but if you do this, you must use the whole name of that PSET parameter. Use of parts of the PSET name only, e.g. `(..., patt='dotted')` may or may not work and is not supported.

4.8. How do I handle when a task's parameter name conflicts with Python syntax?

You might run into a problem calling IRAF tasks that have parameters which conflict with Python reserved words. Here is an example (from ticket #57):

```
--> noao
--> onedspec
--> iraf.dispcor(input=specname, output=specname, global='NO')
  File "", line 1
    iraf.dispcor(input=specname, output=specname, global='NO')
                                                  ^
 SyntaxError: invalid syntax
```

In this case, the parameter named 'global' matches the Python reserved word with the same name.

However, the parameter's value can be manipulated this way:

```
iraf.dispcor.setParam('global','yes')
```

to set it, and then:

```
iraf.dispcor.getParam('global')
```

to find out the value of this parameter.

You can then call dispcor without specifying the value of this parameter.

5. Graphics

5.1. How do I plot graphics to the image display, xgterm, or any other graphics device?

If the stdgraph environment variable is set to a graphics device for which the CL uses its own built-in kernel (such as xgterm or xterm), PyRAF uses its own built-in graphics window. Since the PyRAF graphics window requires native graphics windowing capability (X-windows on Linux, or Aqua on OSX), you can't run graphics tasks when remotely logged in without graphics capabiliy (e.g. through a simple text terminal).

By default, PyRAF uses its own graphics winow, and for most purposes the PyRAF graphics window is preferred. Note that you can also run PyRAF in an xgterm terminal window, but the graphics will appear in the standard PyRAF plot window.

However, if you need to use other IRAF graphics devices, do this:

```
set stdgraph=stgkern
iraf.stdgraph.device="xterm"  # or whatever device
```

This forces the use of the standard IRAF graphics kernel. You cannot run interactive graphics tasks (that read the graphics cursor position, e.g. splot) using this approach, but non-interactive graphics tasks should work. You will generally need to do a gflush to get the graphics to appear.

Setting stdgraph to devices that are not built-in to IRAF (e.g., "stdplot" or "imdr") also works for non-interactive graphics. In that case you do not have to use the special "stgkern" value for stdgraph.

We do plan to add the capability of doing interactive plot overlays on the image display, and we will probably support interactive graphics on some other devices in the future.

5.2. How do I create a new graphics window?

Use the gwm.window() function to create a new graphics window:

```
from pyraf import gwm
         gwm.window('New Window Name')
```

If the new window name is omitted, an unused default name of the form 'graphics<n>' is selected. If a window with the specified name already exists, the graphics focus is switched to that window (so that the next plot will appear there), but the existing window is not modified.

5.3. How do I switch the plots so they go to a different (existing) graphics window?

To switch the graphics focus, use the gwm.window() function with the name of an existing graphics window (the name appears in the window header bar):

```
   from pyraf import gwm
          gwm.window('graphics1')
```

The 'graphics1' window is not modified, but the next plot will appear in that window. If a graphics window with the specified name does not exist, a new window is created.

5.4. How do I get a list of the graphics windows, delete windows, etc.?

There are functions in the pyraf.gwm module that can be used to manage the graphics windows:

```
gwm.window([windowName])     Create new window or switch focus to
                                  existing window
         gwm.delete([windowName])       Delete specified window (active window if
                                  name is omitted)
         gwm.raiseActiveWindow()        Raise the active window to the front on
```

```
                                  the screen
        gwm.getActiveWindowName()    Returns name of the active window
        gwm.getGraphicsWindowManager()  Returns the graphics window manager object
```

The latter function can be used to get a list of all the graphics windows:

```
--> wm = pyraf.gwm.getGraphicsWindowManager()
         --> print wm.windows.keys()
         ['graphics2', 'graphics1']
```

The windows attribute is a dictionary with entries for all existing graphics windows.

5.5. How do I print a plot?

There are several different ways to print or save a plot displayed in the graphics window.

The **File->Print** graphics menu item can be used to print the current plot on your default IRAF printer, as specified by the stdplot variable.

pyraf.gki.printPlot() can be called from a Python script (or typed at the PyRAF command line) to do the same thing.

In gcur mode -- while running an interactive IRAF graphics task so the crosshair cursor is visible in the graphics window -- typing an equal sign = or the colon command **:.snap** prints the plot.

The **File->Save** graphics menu item can also be useful for printing. It allows you to save the graphics metacode for the current plot to a file. That metacode can be later loaded back into the graphics window (using File->Load) and can also be printed using special purpose IRAF tasks. For example, the plot.stdplot task takes a metacode file as input and prints the result to the stdplot output device.

5.6. Can I automate igi graphics using a native python script?

Yes! Use the "Stdin" facility. Put the igi commands you wish to execute in a list, with each item in the list being a command line, for example:

```
from pyraf import iraf
         from iraf import stsdas, graphics, stplot, igi, gflush
         s = ['zsection "myimage[100:300,500:600]"']
         s = s + ['location 0.1 0.9 0.1 0.9']
```

and so on, and then

```
s = s + ['end']   # don't forget to terminate the igi script
         igi(Stdin = s)    # execute the commands
```

```
gflush            # flush the graphics buffer.
```

If you've set up igi for postscript output ('psi-port' or 'psi-land'), this should produce an eps file.

5.7. Why do I get an X Error (X_QueryColors, BadValue) using "epar" on Linux?

Some Linux users are seeing a color-related X Error with 2007/2008 versions of their X server. This may appear during a call to eparam or during a plot. The error looks like the following:

```
--> epar display
       X Error of failed request:  BadValue (integer parameter out of range for operation)
       Major opcode of failed request:  91 (X_QueryColors)
       Value in failed request:  0xff141312
       Serial number of failed request:  2524
       Current serial number in output stream:  2524
```

and then restart your X server (a reboot may be necessary).

As an alternative, "Nedit" users found a work-around for the same issue by setting the XLIB_SKIP_ARGB_VISUALS environment variable. In (t)csh:

```
> setenv XLIB_SKIP_ARGB_VISUALS 1
> pyraf
```

Also, if you do not yet have an Xorg configuration file (e.g. some Fedora Core 10 installs do not have one in /etc/X11/xorg), you may generate one via:

1. as su, type "yum install system-config-display"
2. type 'system-config-display' and configure proper display and video driver

Note that color depth issues in general may be debugged by running "xwininfo" and clicking on any GUI components in question.

5.8. Can I plot via matplotlib for smoother plot lines and fonts?

Yes! PyRAF is now able to plot using a matplotlib graphics kernel (since v1.6, ticket #80). The use of anti-grain geometry (AGG) has the advantage of making plot lines look smoother, and matplotlib's own builtin fonts produces plot text that is more legible at varying window sizes.

PyRAF does not make matplotlib an installation requirement, so as to leave installation as simple as possible. However, if the user has matplotlib installed on their machine (with the TkAgg back-end), they may enable the matplotlib graphics kernel by simply setting an environment variable:

```
setenv PYRAFGRAPHICS matplotlib
```

Note that PyRAF does receive plotting instructions at a very low level (GKI), so there are limits to how much matplotlib widgetry can be brought to bear inside the PyRAF graphics windows.

5.9. Why do I get an exception when using matplotlib v0.99 ?

There is a "feature" in matplotlib v0.99 which assumes that an Axes object will always be used in conjunction with a Line2D object, which is not true for PyRAF's use of matplotlib as a low-level graphics kernel. The result is this exception during a PyRAF plot:

```
AttributeError: 'NoneType' object has no attribute 'get_xbound'
```

This feature is has been fixed in the next version of matplotlib (0.99.1). PyRAF users who want to continue using the matplotlib graphics kernel can work around this by either:

upgrading to matplotlib v0.99.1 or higher, or
reverting to matplotlib v0.98.5.* (NOT RECOMMENDED), or
editing matplotlib v0.99, changing lines.py (line 499) to "if self._subslice and self.axes:"

5.10. How do I force the graphics windows to always stay on top ?

PyRAF does some automatic things for the user when s/he is moving between the terminal and graphics windows during an interactive session, such as switching the focus for you (so you don't have to click on a new window just to give it focus), and automatically moving (warping) the mouse cursor for you.

Some users (OSX mostly) have requested a feature whereby all drawn graphics windows are popped to the foreground and left there (even after finishing, interactive or not) with the focus remaining on them. This way, the focus will not be placed back onto terminal as it usually is.

Since this would be a large change from the expected default behavior, this is for now an experimental feature. It may be enabled by setting an environment variable:

```
setenv PYRAF_GRAPHICS_ALWAYS_ON_TOP
```

If this becomes popular, we may decide to support this feature more formally. Let us know what you think!

5.11. Why do I get a TclError (expected floating-point number) ?

Some Linux users have run into the following error while displaying a window of some kind (e.g. epar or a plot window).

```
TclError: expected floating-point number but got "1.0"
```

For example (from ticket #92), this may occur during a call to implot:

```
--> implot abc.fits
    Traceback (innermost last):
    File "", line 1, in ?
    iraf.implot('abc.fits', _save=1)
    File "lib-tk/Tkinter.py", line 2254, in yview_moveto
    self.tk.call(self._w, 'yview', 'moveto', fraction)
    TclError: expected floating-point number but got "1.0"
```

This error doesn't seem to make sense (as 1.0 is a float), but it stems from the fact that somehow the locale was changed into a flavor with which Tkinter has trouble. The "1.0" string is not as Tkinter expects. The simplest workaround is to change your environment:

```
setenv LC_ALL C
```

or, within your Python script which calls PyRAF:

locale.setlocale(locale.LC_ALL, 'C')

However, if you need to use a different locale, note that implot and other interactive tasks (which take specific single-character input) are going to have difficulty with non-English locales.

5.13. Help - I am having graphics problems!

If you are able to run "pyraf" but are having trouble displaying graphics windows (plots, EPAR, etc.), and you feel sure you should be able to display graphics, or if you unexpectedly see this message upon startup:

```
No graphics display available for this session.
```

or

```
No graphics/display possible for this session.
```

or

```
Limited graphics available
```

then there may be some issue with your local graphics capability. We would like to help you debug this. Please do the following and send us the results:

1) Find out which Python binary you are running when you run "pyraf":

```
% which pyraf
% head `which pyraf`
```

And notice the top line which is (hopefully) an absolute path to your Python binary. Let's say it is "/absolute/pathname/to/python".

2) Now use THAT binary to verify your login session has graphics. Use **"python"**, not "pyraf" here. (see also this FAQ)

```
/absolute/pathname/to/python -c 'import Tkinter as T; T._test()'
```

If you can display graphics, this should always work. It will pop up a little "Tcl/Tk" dialog box.

3) Now use THAT SAME binary to get some diagnostics to send us. Again, use **"python"**, not "pyraf" here.

```
/absolute/pathname/to/python -c 'from pyraf import wutil; wutil.dumpspecs()'
```

Please send us the full output.

Linux Users! If this output shows `"OF_GRAPHICS=True"`, and `"imported xutil=False"`, then please see FAQ 1.9 for a possible failure during the build of a needed C-extension library.

5.14. Help - I am having X11 graphics problems on a Mac.

If you are on a Mac, you will probably prefer to use native (non-X11) windowing and graphics. Or, you may have specific needs/requirements to continue to use X11. PyRAF works either way but depends on the Python under which it was installed (and cannot be changed on the fly as a result). PyRAF WILL however try to determine which type you are using on the fly (Aqua vs. X11) and load the appropriate libraries for you.

Mac users please read FAQ 1.2 for a discussion on this and feel free to contact us for support, but please don't forget to include the output we request in FAQ 5.13 !

5.15. How can I visually determine my Python graphics linkage on OSX?

In order to determine to which graphics your Python installation is linked (especially interesting on OSX), run the command:

```
/absolute/pathname/to/python -c 'import Tkinter; Tkinter._test()'
```

If the pop-up dialog has an "X" in the title bar, it is X11-linked (this is expected on Linux). You can also tell from the look of the buttons (e.g the "Click me" button). If you are on OSX, look at the buttons. The slightly rounded-corner ones are native Aqua, but the sharper, rectangular ones are X11.

5.16. How can I get PyRAF and "pylab" to work well together?

Some users have noted that there are problems on OSX between PyRAF and IPython's "pylab" mode, specifically in the following situation:

```
on OSX
% ipython --pylab
In [1]: from pyraf import iraf
In [2]: iraf.prow('dev$pix')

: Fatal Python error: PyEval_RestoreThread: NULL tstate
```

PyRAF ticket #108 describes the issue in great detail but the gist is that it seems to have to do with the backend chosen at load-time, and the simple fix is to force both to use the `TkAgg` backend via the matplotlib rc file. Simply add:

```
backend : TkAgg
```

to the matplotlibrc (which, on OSX is in ~/.matplotlib/matplotlibrc). For more on the matplotlibrc, see the matplotlib online docs.

6. General information

6.1. Why do I get a syntax error when what I've typed is perfectly legal IRAF CL syntax?

Most likely because you either typed the name of a package or task that has not yet been loaded or you are trying to use CL syntax that doesn't begin with a task, package, or keyword name (for example, =gcur). In the first case, if PyRAF does not see the task name in the list of currently loaded tasks, it assumes what you typed is in Python syntax and treats as such. This will generally lead to some syntax error. Likewise, if you don't start the line with a name of some sort, PyRAF assumes that you are typing a Python statement.

6.2. Why don't the IRAF 'del' or 'print' commands work?

Because these are also Python keywords. The PyRAF interpreter also allows Python statements, so we do not allow any CL mode commands to start with Python keywords. Type 'clPrint' (note the capital P) to execute the IRAF 'print' command (although the Python print statement should usually suffice). Type more of the IRAF 'delete' command, e.g. 'dele', to remove the ambiguity with the Python del statement.

6.3. Why can't I recall commands in a Solaris cmdtool window?

This is the result of a bug in cmdtool, not a problem with PyRAF. For example, if you try to run tcsh in the window, it has the same problem (it appears to ignore the arrow keys.)

The fix is to turn off the cmdtool scrollbar by right-clicking in the window and selecting 'Disable Scrolling' from the 'Scrolling' menu. Or you can use another type of terminal window (e.g., xterm) instead of cmdtool.

6.4. Why do I get an error when I try to print to a pipe (e.g, when running scan or scanf)?

This results in an error:

```
--> print ("382 4783") | scan(i,j)
    Traceback (innermost last):
       File "<console>", line 1, in ?
    NameError: scan
```

This problem is caused by mixing Python syntax with CL syntax. The print statement is interpreted in Python mode (because "print" is a Python keyword and also because it has an open parenthesis). That puts the whole line into Python mode, so scan is not found. The command will work fine inside a CL script (since it is legal CL even if it is not legal Python.) In PyRAF you can do this:

```
--> clPrint "382 4783" | scan(i,j)
    --> print iraf.cl.i, iraf.cl.j
    382 4783
```

Note that you have to use clPrint instead of just print to avoid conflicts with the Python print command, and you have to leave the parentheses off the clPrint to get into CL emulation mode so the pipe is properly translated. Also note the somewhat awkward syntax for accessing the CL parameters i, j (which are not the same as Python variables i, j).

In Python code (including at the PyRAF command line) there are better alternatives to scan. Here's one Python approach:

```
--> f = string.split("382 4783")
        --> print f
        ['382', '4783']
        --> i = int(f[0])
        --> j = int(f[1])
        --> print i,j
        382 4783
```

Or if you want a 1-line alternative:

```
--> i, j = map(int, string.split("382 4783"))
```

The combination of the Python string and re modules for string processing is generally more powerful than scan/scanf, and the results go into simple Python variables instead of CL task parameters.

6.5. Why do I get a TypeError with 1.8.1 when running an IRAF task?

There is a bug in the sleep() function in v1.8.1 (bundled with stsci_python 2.9) which may cause the following error:

```
TypeError: sleep() takes exactly 1 argument (0 given)
```

please see the Release Notes for v1.8.2. This is fixed in 1.8.2 and above.

6.6. How does PyRAF handle integer division?

There are many modes of operation in PyRAF, and it may be unclear as to how integer division is handled in each one. In IRAF (e.g. CL scripts), integer division is performed the "traditional" way (with an integer result, using truncation, as is done in Python 2.x).

For example:

```
print (9/5)
```

yields

```
1
```

in IRAF as well as in Python 2.x. However, the same command yields

```
1.8
```

in Python 3.x, where the '/' operator always performs floating point division.

In general, the STScI thinking is that when running in a CL-like context, PyRAF should behave like the CL. In all other cases, it should behave like the version of Python it is being run in. The following are different possible uses of PyRAF and how they handle integer division:

Translation of streamed/redirected CL script (e.g. '--> cl < myscript.cl'). PyRAF uses integer division like the CL.
Definition of CL script as a task (e.g. '--> task bob = myscript.cl ; bob'). PyRAF uses integer division like the CL.
Simple PyRAF interpreter use (e.g. '--> print(9/5)'). PyRAF uses integer division in Python 2.x but floating point division in Python 3.x.
Python script using API to IRAF (e.g. python -c 'from pyraf import iraf; print(iraf.deg(9/5))') PyRAF uses integer division in Python 2.x but floating point division in Python 3.x.