PyRAF Programmer's Guide

Philip E. Hodge

May 11, 2004

Version 1.0

Space Telescope Science Institute
Email: help@stsci.edu

1 General Introduction

The main goal of this guide is to describe how to write scripts that take full advantage of the features PyRAF provides. The presumption is that the scripts will be written in Python, but the use of IRAF CL scripts in PyRAF will be described as well. We will often show how an operation can be done in both Python and in the IRAF CL, primarily to help those who are familiar with the IRAF CL to find the corresponding operation in Python.

A Python script can call IRAF tasks, and a Python script can be defined as an IRAF-like task in PyRAF. These are not mutually exclusive options, but they are two distinct features of PyRAF. The first will be described in the section "Writing Python Scripts that Use IRAF/PyRAF Tasks," while the latter will be described in the section "Python Tasks." Other sections will explain how to define tasks in PyRAF based on IRAF executables, CL scripts, or host operating system commands ("foreign" tasks).

While this document is an introduction to programming with PyRAF, it will be assumed that the reader is familiar with Python and with the calling conventions for Python functions, classes and methods. Python functions and IRAF CL scripts have some features in common, such as positional as well as keyword arguments, and a variable argument list.

2 Writing Python Scripts that Use IRAF/PyRAF Tasks

Here is a very simple example to show the essential components of a Python script that calls an IRAF task, in this case `imstatistics`. The file 'imstat_example.py' contains the following:

```
#! /usr/bin/env python

import sys
from pyraf import iraf

def run_imstat(input):
    iraf.images()
    for image in input:
        iraf.imstat(image)

if __name__ == "__main__":
    run_imstat(sys.argv[1:])
```

This calls `imstatistics` on a list of images. This can be run either from the shell or from a Python or PyRAF session. To run it from the shell, type `imstat_example.py` (which must have execute permission) followed by one or more image names. To run it from Python or PyRAF, first type `import imstat_example`; then it can be run as follows (for example): `imstat_example.run_imstat(["file.fits[1]", "file.fits[2]", "file.fits[3]"])`. The argument is a list, and one or more images may be specified in the list.

The statement `from pyraf import iraf` makes IRAF tasks in general available. The statement `iraf.images()` loads the `images` package, which contains `imstatistics`. If `images` is already loaded, then calling `iraf.images()` does nothing; a package may be loaded any number of times. The `run_imstat()` function accepts a list of images and runs the `imstatistics` task (abbreviated as `imstat`) on each of them.

Like many IRAF tasks, `imstatistics` accepts a "file name template" as input, i.e. a string containing one or more image names, separated by commas, and possibly making use of wildcard characters. An alternative to running `imstat` separately on each image is to construct such a string containing all the image names, and to call `imstat` just once, with that string as input. Here is another version of `run_imstat` that concatenates the list elements to a comma-separated string. Other differences with this version are described below.

```
def run_imstat(input):
    iraf.images(_doprint=0)
    all_exist = 1               # boolean flag
    for image in input:
        if not iraf.imaccess(image):
            all_exist = 0
            print "Error:  can't open", image
    if not all_exist:
        return
    iraf.imstat(",".join(input))
```

As in the previous version, `iraf.images()` is used to load the `images` package, but the `_doprint=0` argument has been included to disable printing of the tasks in the package. The default for `_doprint` (a boolean flag) is 1; unless `_doprint=0` is explicitly specified, when a package is loaded for the first time the names of the tasks in that package will be printed to the screen. Showing the tasks is OK for interactive use, but when loading a package in a script, the user wouldn't normally want to see such output. If the package is already loaded (and images may well be), then the tasks will not be shown anyway, but in general it's preferable to specify `_doprint=0` when loading any package in a script.

A bit of error handling was also added to this version of `run_imstat`. `imstat` itself just prints a warning and continues if one or more of the input images do not exist. Much of the time, that may be all the error handling that is needed. Additional checking may be useful in cases where the functions being called take a lot of time or write intermediate files that would need to be cleaned up if one of the input files did not exist. The `imaccess()` function used in this example is actually of limited use for error checking, however. It does not catch an invalid image section, for example, and for a FITS file, the extension number is ignored. See the section on error handling for further discussion.

Frequently, one wants something even simpler than this, such as a file that just invokes one or more IRAF tasks without defining a function. For example, suppose the following is in the file 'simple.py':

```
iraf.images()
iraf.imstat("file.fits[1]")
iraf.imstat("file.fits[2]")
iraf.imstat("file.fits[3]")
```

Then it could be run by typing `execfile("simple.py")`. Commands in this file could also be executed (once) by `import simple`, but in order to do that the file would need to begin with the statement `from pyraf import iraf`. Using execfile is simpler, and it is also much easier to repeat; import can only be done once, after which you must use reload.

Here is another example, using slightly different style, and with comments that explain what is being done.

```
#! /usr/bin/env python

# This takes a range of numbers from the command line, constructs
# the filename by appending each number to the root "ca" and appending
# ".fits[0]", and splots the files one at a time.  It’s intended as a
# little pyraf demo script.

import sys

from pyraf import iraf

# This function is invoked when running from the shell.
def multiSplot():
    if len(sys.argv) < 2 or len(sys.argv) > 3:
        print >> sys.stderr, "syntax:  runSplot.py first <last>"
        print >> sys.stderr, "first..last is the range of integers (inclusive)"
        print >> sys.stderr, "to append to the root name ’ca’"
        sys.exit()
    # The command-line arguments are strings; convert to integer.
    ifirst = int(sys.argv[1])
    if len(sys.argv) > 2:
        ilast = int(sys.argv[2])
    else:
        ilast = ifirst
    for i in range(ifirst, ilast+1):
        runSplot(i)

# Use this function when running from Python or PyRAF.
def runSplot(i, root="ca", extension=0):

    # Load packages; splot is in the onedspec package, which is in noao.
    # The special keyword _doprint=0 turns off displaying the tasks
    # when loading a package.
    iraf.noao(_doprint=0)
    iraf.onedspec(_doprint=0)

    # Construct the image name.
    imname = "%s%d.fits[%d]" % (root, i, extension)
    print imname        # just to see it

    # Set/view IRAF task parameter.
    # (This is done only to show how it can be done.)
    iraf.onedspec.splot.save_file = "splot_%s.log" % (root,)

    # Call IRAF task, and specify some parameters.
    iraf.onedspec.splot(imname, function="chebyshev", order=6)

# Standard Python mechanism for handling tasks called from the command line
# (see for example Martelli, Python in a Nutshell, chapter 7, "The Main
# Program," or Beazley, Python Essential Reference, chapter 8).
if __name__ == "__main__":
    multiSplot()
```

3 Defining Tasks in PyRAF

As with the IRAF CL, one can define tasks around IRAF executables, CL scripts, or "foreign" tasks. But the big advantage of PyRAF is that it allows one to write a Python program (that need not use IRAF at all) that can be treated like an IRAF task with the familiar CL command-line syntax and parameter editing facilities. In this way it is possible to integrate IRAF and Python programs into the same user environment and take advantage of the strengths of both.

3.1 Python Tasks

This section describes how to define PyRAF tasks that are implemented as Python scripts. We have already described Python scripts that call IRAF tasks. The scripts described in this section may or may not call IRAF tasks; the relevant feature is that these are PyRAF tasks. From the user's perspective, these look the same as any other PyRAF/IRAF task, i.e. they will typically have parameters, and they can be run in so-called "command mode," without using parentheses or enclosing strings in quotes.

Note that Python tasks can only be used in PyRAF, not in the IRAF CL, because the CL cannot run Python. It is possible to write IRAF package CL scripts that define a mixture of IRAF and Python tasks that will work gracefully with both PyRAF and the IRAF CL in the sense that both types of task will work in PyRAF, and if the package is loaded in the CL a warning message will be printed that indicates that some tasks require PyRAF. If one attempts to run such a Python task from the IRAF CL, another warning message will be printed. While the task doesn't work, it does tell the user why not (i.e. it requires PyRAF).

A Simple Example

Here is a bare-bones example for creating a task 'xyz' written in Python that can be called from PyRAF just like any other IRAF task. Two files are used, 'xyz.py' and 'xyz.par'. In this example, the name "xyz" is used throughout, but this is not required. While the rootname of the parameter file does need to be the same as the task name (as in the IRAF CL), the other names may differ. There is another example below that uses different file names.

The parameter file 'xyz.par' is an ordinary IRAF par file. In this example the file contains the following:

```
input,s,a,"",,,"string to print"
mode,s,h,"al"
```

'xyz.py' contains the following. `<path>` should actually be the name of the directory that includes 'xyz.par' (see below for clarification):

```
from pyraf import iraf

def xyz(input):
    print input

parfile = iraf.osfn("<path>xyz.par")
t = iraf.IrafTaskFactory(taskname="xyz", value=parfile,
              function=xyz)
```

In PyRAF, define 'xyz' as a task by running the `pyexecute()` function:

```
--> pyexecute("<path>xyz.py")
```

At this point 'xyz' is a PyRAF task; you can 'lpar xyz', 'epar xyz', or just run it.

The `value` parameter in `IrafTaskFactory` is the complete path name of the parameter file 'xyz.par'. This could be given explicitly, but it is cleaner to use the `iraf.osfn()` function to take an IRAF "virtual file name" and return an operating-system dependent directory and file name. For example, if 'xyz.par' were in the scripts subdirectory of the user's IRAF home directory, the argument to `iraf.osfn` would be `"home$scripts/xyz.par"`. It is also possible to use the Python functions in the os and os.path modules to find files, fill in path names, etc. The rootname of a parameter file must be the same as the task name, and the filename extension must be ".par" (as in IRAF).

Note that the value of the `function` parameter in `IrafTaskFactory` is `xyz`, not `"xyz"`. It's a reference to the function, not a string containing the function name. This is the function to be executed when the task is invoked. `IrafTaskFactory` can be used to create a wide variety of tasks, such as a package, CL script, pset, or "foreign" task; the `function` parameter is only used when the task being defined is a Python function.

The argument to `pyexecute` should include the directory (using IRAF notation), unless 'xyz.py' is in the default directory when `pyexecute` is called. The task will be defined after `IrafTaskFactory` has executed. Running `pyexecute` is the recommended way to do this, but it isn't the only way. You could instead use `execfile("<path>xyz.py")`, using host syntax for the directory and file name. Or you could use `import xyz` if 'xyz.py' is in your PYTHONPATH. One advantage of `pyexecute` is that you can call it from a CL script. If the script is run from a PyRAF session, the Python/PyRAF task will be defined; if the script is run from the IRAF CL (and STSDAS has been loaded), a warning will be printed to say that PyRAF is required, but it will not raise an exception. This works because there are files 'pyexecute.cl' and 'nopyraf.cl' in the STSDAS directory, and this pyexecute is what will be run if the user is in the IRAF CL rather than in PyRAF.

Note that 'pyexecute.cl' has no intrinsic connection to STSDAS. If you wish to have the flexibility to include Python scripts in your IRAF packages and still be able to run either PyRAF or the IRAF CL, but STSDAS will not necessarily be loaded, you can simply copy 'pyexecute.cl' and 'nopyraf.cl' from STSDAS to some IRAF package that will be loaded and define these as tasks. You can install these in the IRAF tree if you have permission to do so.

The statement `import iraf` can be used in scripts that run in PyRAF. If a script might be run from Python or from the host operating system command line, use `from pyraf import iraf` instead. IRAF parameters, tasks and packages are objects, just like everything else in Python. Packages in IRAF may be loaded by executing them, which is very similar to the way they are loaded in the IRAF CL; for example: `iraf.images()`. The primary way that tasks are invoked in PyRAF is by using the `__call__()` method of the task object, i.e. it looks like any other function call. Since tasks (and parameters, etc.) are objects, they can be assigned to variables and passed to functions:

```
t = iraf.imhead
lparam(t)
```

An Example Using the _iraf Filename Convention

Here is another example, this one using different file names, partly to illustrate a convention that's used in STSDAS, to separate the PyRAF interface from the main Python script and to use a name ending in '_iraf.py' for the former. The files are assumed to be in the scripts subdirectory of IRAF "home". The task name is `ncounts`, and the files are 'ncounts.par', 'xyz_iraf.py' and 'nc.py'. Note that the task name and root of the par file name are the same; the other names may differ.

This task uses the `images.imstat` task to compute the total number of counts in an image. The standard output of `imstat` is assigned to a Python variable `text_output`, which is a list of strings that in this case contains just one string, ['npix mean'] (not this literal string, but rather the numerical values). The `split()` method splits this into two strings, one with `npix` and one with `mean`. The result is simply the product of these two, after converting from string to float. The result is assigned to the task parameter `total`, and it is also printed if `verbose=yes`.

The parameter file 'ncounts.par' contains the following:

```
image,s,a,"",,,"image name"
verbose,b,h,yes,,,"print the value?"
total,r,h,0.,,,"(task output) number of counts in the image"
mode,s,h,"al"
```

'xyz_iraf.py' contains the following:

```
from pyraf import iraf
import nc

def _abc(image, verbose, total):

    total = nc.calc_ncounts(image=image, verbose=verbose)
    if verbose:
        print "total =", total

    # Update the value in the par file.
    iraf.ncounts.total = total

parfile = iraf.osfn("home$scripts/ncounts.par")
t = iraf.IrafTaskFactory(taskname="ncounts", value=parfile,
            function=_abc)
```

'nc.py' contains the following:

```
from pyraf import iraf

def calc_ncounts(image, verbose):
    """use imstat to get the total number of counts in an image"""

    iraf.images(_doprint=0)      # load the images package
    text_output = iraf.imstatistics(image, fields="npix,mean",
            format=0, Stdout=1)

    values = text_output[0].split()

    #      number of pixels     mean value
    return float(values[0]) * float(values[1])
```

In PyRAF, define ncounts as a task:

```
--> pyexecute("home$scripts/xyz_iraf.py")
```

The statement `iraf.images(_doprint=0)` loads the `images` package (without printing the task names and subpackage names). This could be skipped if the `images` package is already loaded, e.g. by the user's 'login.cl' file, but it is generally not safe to make such an assumption, and it is harmless to load a package more than once.

The `Stdout=1` parameter in the call to `imstat` means that the standard output of the task will be returned as the `imstat` function value. In this example the output is assigned to a Python variable `text_output`, which is then processed using Python. The variable `text_output` is a list of strings, one for each line of output from the task, in this case just one line. This feature serves as a substitute for I/O redirection, but for many applications it is also much more convenient than using a temporary file. This is discussed further in the section on I/O redirection.

In the above `ncounts` example, separating the Python code into two files 'xyz_iraf.py' and 'nc.py' was not necessary, it's a convention to isolate the PyRAF-specific code. 'xyz_iraf.py' contains the part that defines the task, deals with the parameters, and calls a Python function to do the work. The latter function is in 'nc.py'. The separation in this case is a little artificial, since `calc_ncounts` in 'nc.py' still calls an IRAF task. On the other hand, 'nc.py' could be imported into Python or (with minor additions) invoked from the shell, while 'xyz_iraf.py' defines a task and requires a parameter file, which is more closely associated with the interactive aspect of PyRAF.

IRAF and Python Interfaces

Python and IRAF use different conventions for undefined values. The interface for an IRAF task should use IRAF conventions. If the script includes a Python function that could be used stand-alone, however, it would be more reasonable if that function used Python conventions. For example, a task might have `input` and `output` arguments, and it might modify the input file in-place if an output file name was not specified. In the IRAF CL a string may be left unspecified by setting it to null (`""`) or blank, and there is a special `INDEF` value for numeric variables. In Python, `None` is used for any unspecified value. One purpose for the '_iraf.py' file is to convert unspecified values from one convention to the other. Another purpose is to check that input files do exist and that all required parameters have actually been specified. It's very helpful to the user of a script to check for parameter problems at the beginning, especially if the task could run for some time.

Importing Modules

Note that 'xyz_iraf.py' uses `import nc`. In order for this to work, 'nc.py' must be in your Python path (`sys.path`) when you run `pyexecute`. For tasks in the STSDAS directory tree, this is accomplished by having a 'python' subdirectory of `stsdas`, with a link to each of the packages (in the Python sense) that may be imported; the `stsdas$python/` directory is included in `sys.path` when the `stsdas` package is loaded in PyRAF. When writing your own tasks that are not to be included in the STSDAS tree, one option is to simply put all the source files in one directory and add that to your Python path. Another option is to create a package subdirectory for each task or related set of tasks, with the root of these subdirectories in your Python path.

3.2 IRAF Executables

IRAF executables are created by compiling and linking code written typically in SPP (but Fortran and C can also be used). The `task` statement in the SPP code is converted by the SPP preprocessor into code that makes the connection with the CL. One executable may contain multiple tasks. A task in an IRAF executable can be defined in a Python script by using the `task()` function, for example: `iraf.task(xyz = "home$scripts/abc.e")`. Note that the keyword argument has the same syntax as would be used in the IRAF CL (or interactively in PyRAF) for defining a task, except that quotes are required. There must be a parameter file with the same root name as the task name and extension '.par', in the same directory as specified for the executable. Additional keyword arguments `PkgName` and `PkgBinary` may be used to specify the package name and list of directories to be searched for the executable (e.g. `PkgName="clpackage", PkgBinary=["bin$"]`).

The `IrafTaskFactory()` function may be used instead of `task()`. `IrafTaskFactory` was described earlier for defining Python scripts, but both of these functions are quite general in terms of the types of tasks that they can define.

The SPP `task` statement can define multiple tasks (these are just different subroutines) to be included in the same executable. It is common practice to use just one or perhaps a few executables for the tasks in an IRAF package. This saves disk space for the executables (since the IRAF libraries are common to all tasks), and it reduces loading time. The syntax for defining multiple tasks in the same executable is a little peculiar. Each task except the last is given in quotes as an argument, and the last task is given as a keyword argument using the syntax shown earlier. For example, `iraf.task("task_a", "task_b", task_c = "home$stuff.e")`.

One obscure feature of IRAF that you need to be aware of is that when the CL looks for the executable for a task, it looks first in the bin directories (e.g. 'iraf$bin.redhat'); it only looks in the directory specified in the task statement if the file is not found in any of the bin directories. Thus if you use an executable name that happens to be the same as one in an IRAF package (e.g. 'x_tools.e' is in STSDAS), your executable will not be found. This explains the peculiar wording in the first paragraph of this section, "in the same directory as specified for the executable." The task statement might say the executable is in 'home$scripts/', while it might actually have been installed in a bin directory; nevertheless, the parameter file must be in 'home$scripts/', not in the bin directory.

3.3 IRAF CL Scripts

CL scripts may be defined as tasks in PyRAF using the `task()` function, e.g. `iraf.task(jqz = "home$scripts/jqz.cl")`. A `PkgName` could be defined, but it wouldn't make sense to specify a `PkgBinary`. Interactively in PyRAF, the same syntax is used as in the IRAF CL, e.g. `task jqz = home$scripts/jqz.cl"`.

The `goto` statement is not yet supported (though its addition is being considered for a future version of PyRAF), so CL scripts that use `goto` statements cannot currently be defined as tasks in PyRAF without being revised to avoid the use of `goto`.

Even if you intend to use a CL script exclusively in the IRAF CL, defining it as a task in PyRAF is useful for testing and debugging, since PyRAF prints informative error messages. In PyRAF, the CL script is translated into a Python script when the task is defined, and it is the Python script that is executed when the task is run. You can obtain a copy of the Python script using `getCode()`, which is a method of the `IrafCLTask` class, e.g. `p = iraf.jqz.getCode()`. The code is in the form of a Python string, so it can be executed using the `exec` statement or the `eval()` built-in function.

3.4 Foreign Tasks

The `task()` function can be used for defining "foreign" tasks. Interactively, a task could be defined by, for example, `task $emacs = "$foreign"`. The word `$emacs` cannot be used as a keyword argument because of the dollar sign, however, so the `task()` function for this example would be as follows: `iraf.task(DOLLARemacs = "$foreign")`. To define both emacs and vim as foreign tasks in one call, use `iraf.task("$emacs", DOLLARvim = "$foreign")`.

The dollar sign before the task name (emacs or vim, in this example) means that the task has no parameter file. Arguments may be given when invoking the foreign task, however, and those arguments will be passed directly to the command (except that file names using IRAF environment variables will be converted to host file names). The dollar sign in `$foreign` indicates a foreign task, while the word "foreign" means the task name and command name are the same. Here is an example where the names are not the same: `iraf.task(DOLLARll = '$ls -lg')`; interactively, this would be `task $ll = "$ls -lg"`.

4 Dealing with I/O Redirection

IRAF supports I/O redirection using the same syntax as in Unix, e.g.
`listpix x.fits[1][371:375,290:281] wcs="physical" > x.txt`. Python does not have this option, but PyRAF has a workaround, making use of "special" task parameters `Stdin`, `Stdout` and `Stderr`. They are special in the sense that they are not in the parameter file for any task, but you can specify them when calling a task in PyRAF.

`Stdout` and `Stderr` can be set to a numerical value (0 or 1), which will be taken as a boolean flag, or the value can be a file name or Python file handle for a file that is open for writing (or appending). The example at the beginning of this section redirected the listpixels output to a text file 'x.txt'; this can be done in PyRAF as follows: `iraf.listpix ("x.fits[1][371:375,290:281]", wcs="physical", Stdout="x.txt")`. Note that Python syntax is used, rather than IRAF "command mode" syntax. If `Stdout` or `Stderr` is set to 1, the task standard output or standard error respectively will be returned as the value of the task, rather than being printed to the terminal window or written to a file. For example, `x_txt = iraf.listpix ("x.fits[1][371:375,290:281]",` wcs="physical", Stdout=1). The function value (x_txt, in this example) will be a list of strings, one string for each line of text in the output (the newlines will not be included).

If only Stderr is specified, then both Stderr and Stdout are written to the specified Stderr location. If *both* Stderr and Stdout are specified, the output for the two streams are redirected separately. It is possible to specify redirection of only Stderr, though the syntax is a bit weird:

```
task(params, Stderr=filename, Stdout="STDOUT")
```

"STDOUT" is a "magic" value that causes output to be redirected to the normal sys.stdout. Similarly, "STDERR" and "STDIN" are magic for their corresponding redirection keywords. This is included for compatibility with the IRAF CL, which does the same thing.

Here is a simple example of using this list-of-strings output from listpixels. The first two "words" in each string are the pixel coordinates in the X and Y directions, and the image pixel value is the third word. This example computes the flux-weighted sums of X and Y pixel values, then divides by the sum of the weights to get the X and Y centroids.

```
--> sum_w = 0.
--> sum_wx = 0.
--> sum_wy = 0.
--> for line in x_txt:
...   words = line.split()
...   x = float(words[0])
...   y = float(words[1])
...   w = float(words[2])
...   sum_w += w
...   sum_wx += w * x
...   sum_wy += w * y
...
--> print sum_wx / sum_w
22.6071772774
--> print sum_wy / sum_w
62.0303834708
```

The Stdin special parameter can be used to specify the standard input for a task. The value for Stdin can be a Python list of strings containing the text (i.e. the same format as the variable returned as standard output when using Stdout=1), or it can be a file name or a Python file handle for a file open for reading. A pipe may be emulated by using a Python variable to pass the standard output of one task to another, without using a temporary file.

Setting Stderr=1 (instead of Stdout=1) may be used to capture messages that were sent explicitly to standard error, but see the discussion below about errors vs. warnings. Some tasks write messages about an error condition to the standard output rather than to standard error; in that case, using Stderr=1 is not sufficient to separate the normal text output from an error message. When an error condition is encountered, PyRAF raises an exception rather than writing the error message and traceback to the standard error. This information can be captured (see the next section), but not by using Stderr=1.

5 Dealing with Errors

One of the major advantages to writing scripts in Python rather than in the IRAF CL is the ability to handle errors using try and except. This works even when calling IRAF tasks; that is, an error in a CL script or a call error in an SPP program can be caught using try in Python. For example,

```
--> try:
...   iraf.columns ("garbage", 37)
... except iraf.IrafError, e:
...   print "error was caught"
...   print e
...
Killing IRAF task 'columns'
error was caught
Error running IRAF task columns
IRAF task terminated abnormally
ERROR (741, "Cannot open file (garbage)")
```

There's a catch, however. Many IRAF tasks can operate on a list of input files, and these tasks usually convert some errors (such as file not found) into a warning, allowing the task to continue trying to process the other input files. But warnings are not caught by `try` and `except`. The `imstat` task used by 'nc.py' in the `ncounts` example is a case in point; a `try` statement was not used there because it would not trap the most common problems, such as a mistake in the name of the input image. If you want to do robust error handling, it is best to check parameter values in Python before calling the IRAF task to ensure predictable behavior when errors occur.

6 Task Parameters

One of the strengths of IRAF is the way it handles task parameters. The very useful IRAF commands `lparam, dparam, eparam, unlearn` are available in PyRAF with the same syntax. One significant difference is that `eparam` brings up a GUI parameter editor, rather than the IRAF text-based parameter editor. This section describes parameter files and in-memory parameter lists, the copy of a par file in the uparm directory, and a few useful methods for handling parameters and par files.

6.1 Parameter Files, Parameter Lists

Not every task has a parameter file, as was mentioned earlier (that was in the section on foreign tasks, but other tasks may lack parameter files as well). For a task that does have parameters, there is a template parameter file (or a CL script file) that specifies the default values, the types, and other information. This file is typically found in the package directory, has root name equal to the task name, and has extension ".par" (or ".cl"). There is another copy of that parameter file in the user's 'uparm' directory, which by default is a subdirectory of the user's IRAF home directory. This copy is what is deleted when `unlearn` is used. After running `eparam` on a task, or after successfully executing the task, the copy in the uparm directory will be updated (or created, if necessary) to reflect the values set using `eparam` or specified when the task was run (but see below for further details). The name of the file in the uparm directory is somewhat garbled ("scrunched," in PyRAF terminology). The file name begins with the first two characters of the package that contains the task, followed by the last character of the package name (or just the package name if it is less than or equal to three characters). The rest of the root name is the first five characters of the task name and the last character of the task name (or just the task name if it is less than or equal to six characters). The extension is '.par'. This scheme dates back to the 1980's, when file names on some operating systems were severely limited in length. For example, the uparm file names for `imcopy` and `imtranspose` are 'imlimcopy.par' and 'immimtrae.par' respectively (these tasks are in the imutil and imgeom subpackages of images).

PyRAF uses the same template and uparm parameter files, and there can also be in-memory copies of the parameters for a task. In the PyRAF code these parameter lists are called the _defaultParList, the _runningParList and the _currentParList. The default par list is what you have after a task is unlearned; the parameter values are copied from the template par file. The running par list is the set that is used when a task is run, i.e. it would include values assigned on the command line. If the task completes successfully, the running par list will be copied to the current par list and to the uparm copy of the parameter file (but see below). The current par list contains updates after running a task or by direct assignment to the parameters, e.g. using eparam. The current par list generally (but not always) agrees with the uparm file.

6.2 The uparm Copy of the Parameter File

It was stated above that the uparm copy of the parameter file will be updated after a task runs successfully. This is not always true. This section describes the conditions under which the uparm file will or will not be updated, and which parameter values will be modified.

If a task is executed using Python syntax, then the uparm copy of the par file will not be updated (unless the special boolean keyword `_save=1` was specified). This is the case regardless of whether the task is run interactively or in a Python script. On the other hand, if a task is run interactively using the ordinary command-line syntax that looks like IRAF's "command mode," then query/learn parameters (`mode="ql"` or `"al"`) will be updated in the uparm file; hidden parameters will not normally be updated. When using Python syntax, if `_save=1` is specified then query/learn parameters *will* be updated in the uparm file, i.e. the uparm update behavior is the same as when using command mode. Except for the _save keyword, this is intended to closely emulate the way IRAF handles updating the uparm file. The significant difference is that PyRAF either updates the uparm file or not depending on the syntax (IRAF's command mode vs. Python syntax). IRAF updates the uparm file if the task was run interactively, and it does not update the uparm file if the task was invoked from a script or was run as a background job.

Running a task in command mode normally does not result in hidden parameters (`mode="h"`) being updated in the uparm file. Parameters may be set explicitly, either by direct assignment (e.g. `iraf.imstat.fields="mean"`) or by calling the `setParam()` method. Doing so does not update the uparm file immediately; however, if the task is subsequently run in command mode, the values thus set will be updated in the uparm file, and this is so even if the parameters are hidden.

There are two simple ways to get an entire parameter list updated in the uparm copy of the par file, regardless of whether the parameters are query or hidden. One way is to run eparam and click on "Save"; in this case all the parameters in the uparm file will be set to the values shown by eparam. The other way, which can conveniently be used in a script, is to call the `saveParList()` method for the task.

6.3 Getting and Setting Parameters

In PyRAF, IRAF tasks are `IrafTask` (or related) objects. Four of the most useful methods of this class are described below.

The IRAF CL syntax for printing the value of a task parameter is, for example, `print (fxheader.fits_file)`, or `=fxheader.fits_file`. A parameter may be set using similar syntax, `fxheader.fits_file = "xyz.fits"`. A similar syntax works in Python, too: `print iraf.fxheader.fits_file`, `iraf.fxheader.fits_file = "xyz.fits"`. The PyRAF methods `getParam()` and `setParam()` serve a similar purpose, and they allow more control over prompting (for query parameters), data type of the returned value, and access to the "p_" fields (described below). The first argument to `getParam()` and `setParam()` is `qualifiedName`, which in its simplest form is a parameter name. The "qualified" aspect means that the parameter may be qualified with a package name, task name, or field, with the various parts separated by periods. The parameter name may include an array index in brackets if the parameter is an array. If a package name is specified, then a task name must be given; otherwise, the package would appear to be a task name.

You may be wondering why one would give a task name, when these methods are already associated with an IRAF task object. The answer is that you can access the parameters of any loaded task this way. For example, `print iraf.imcopy.getParam("fxheader.fits_file")` and `print iraf.fxheader.getParam("fits_file")` are equivalent. To get the value of a parameter in the task's package, precede the parameter name with the "`_.`" qualifier; for example, `print iraf.imcopy.getParam("_.version")` prints the version parameter in the imutil package par file. The fields are referred to as "parameter attributes" in the IRAF help page (type `phelp parameters`). The fields are `p_name, p_value, p_default, p_xtype, p_type, p_prompt, p_filename, p_minimum, p_maximum, p_mode`. In IRAF there is also a `p_length`, which is a maximum string length, but in PyRAF this is not needed and is not included. Note that for string and (in PyRAF) integer parameters, `p_minimum` is used for the list of allowed values; for example:

```
--> print iraf.imarith.getParam("op.p_type")
s
--> print iraf.imarith.getParam("op.p_minimum")
|+|-|*|/|min|max|
```

The calling sequence for `getParam()` is **`getParam(qualifiedName, native=0, mode=None, exact=0, prompt=1)`**. The value will be gotten from the running par list if that is defined, or the current par list if that is defined, or the default par list. The value returned is by default a string; specifying `native=1` gives the value in the native type of the parameter. Minimum matching is supported unless `exact=1` was specified. The default `prompt=1` means that the current value will be printed to the terminal window, and the user will have the option to specify a new value.

The calling sequence for `setParam()` is **`setParam(qualifiedName, newvalue, check=1, exact=0)`**. The value will be set in the running parameter list if that is defined, or the current par list if that is defined. Values set don't typically appear in the copy of the file in the uparm directory until after the task has been run successfully. The parameter fields that may be set are `p_value, p_prompt, p_filename, p_minimum, p_maximum, p_mode`. The default `exact=0` means that minimum matching is supported, but exact can be set to 1 to require an exact match. The default `check=1` means that the specified parameter value will be compared with the minimum and maximum allowed values (or the choice list, for a string), and a value that is out of range will not be accepted. However, sometimes one wants to give an out-of-range value, e.g. for testing how the task deals with it, and that can be done by specifying `check=0`.

`saveParList(filename=None)` writes the current parameter list to `filename`, or to the file in the uparm directory if `filename` was not specified. The format of such a file is a standard IRAF parameter file.

Parameters can be restored from a file using **`setParList()`**, with the file specified using the special keyword `ParList`. The keyword value can be either a file name or an `IrafParList` object. If a file name is given, the filename extension must be '.par'. An alternative is to specify the special keyword `ParList` when running the task.

```
--> # Save the parameters to the file hedit.par.
--> iraf.hedit.saveParList(filename="hedit.par")
--> # Restore the parameters from the file, and then run hedit.
--> iraf.hedit.setParList(ParList="hedit.par")
--> iraf.hedit(mode="h")
--> # Alternatively, restore parameters from saved file
--> # and run hedit in one call.
--> iraf.hedit(ParList="hedit.par")
```

7 Handy Functions

`eparam(*args, **kw)` invokes a GUI parameter editor.

`lparam()`, **`dparam()`**, **`update()`** and **`unlearn()`** work like their namesakes in the IRAF CL, except that multiple task names may be given:

```
--> # The arguments may be strings ...
--> lparam("imcopy", "hedit")
--> # ... or IrafTask objects.
--> lparam(iraf.imcopy, iraf.hedit)
```

The **`setVerbose(value=1, **kw)`** function sets the verbosity level that determines how much extra information is printed when running a task. You can use `print iraf.Verbose` to see the current verbosity level, which defaults to 0. The default `value=1` for `setVerbose` results in only slightly more output. Use `value=2` (the maximum) for considerably more output, very likely more than you would want to see on a regular basis, but possibly useful for debugging.

The **`saveToFile(savefile, **kw)`** and **`restoreFromFile(savefile, doprint=1, **kw)`** functions respectively write the IRAF environment to `savefile` and read from that file. IRAF filename syntax may be used for `savefile`, e.g. 'home$pyraf.save'. The default `doprint=1` means that the tasks and packages at the CL level will be listed; this is equivalent to typing "?" in PyRAF.

`envget(var, default=None)` is similar to the IRAF CL `envget()` function. The `show` command can also be used to print the value of an environment variable. The variable may either be one defined in IRAF or in the host operating system. If the variable is not defined, a KeyError exception will be raised, unless `default` is not None, in which case the value of `default` will be returned. These options (the ability to catch an exception and the `default` parameter) are new with the PyRAF version of `envget`.

`osfn(filename)` returns the operating-system file name equivalent of `filename`. This works the same as the IRAF function of the same name.

`mktemp(root)` appends a string based on the process ID and a letter (or two) to `root` and returns the resulting string, similar to the IRAF function of the same name. This is intended to return a unique string each time it is called.

`nint(x)` returns the nearest integer to `x`. If `x` is an odd multiple of 0.5, rounding is done away from zero (i.e. `nint(0.5)` is 1, and `nint(-0.5)` is -1). The type of the result is integer.

`frac(x)` returns the fractional part of `x`, or 0.0 if `x` is an integer. The returned value has the same sign as `x`.

`real(x)` accepts a float, integer or string and returns a float, similar to the corresponding IRAF function. If `x` is float it is returned unchanged. If it is a string the contents can be numerical, or it can be in d:m:s.s or d:m.m format.

```
--> print iraf.real("57:18:30")
57.3083333333
--> print iraf.real("57:18.5")
57.3083333333
```

`clDms(x, digits=1, seconds=1)` returns `x` as a string in degrees, minutes, seconds (d:m:s.s) format, with `digits` figures after the decimal point. If `seconds=0`, the result is degrees, minutes and decimals of minutes.

```
--> print iraf.clDms(89.3083333333)
89:18:30.0
--> print iraf.clDms(89.3083333333, digits=3, seconds=0)
89:18.500
```

An alternative is to use the '%h' or '%m' format with the printf function, which works with PyRAF and with the IRAF CL.

```
cl> printf("%12.1h\n", 89.3083333333) | scan (s1)
cl> print(s1)
89:18:30.0
cl> printf("%12.3m\n", 89.3083333333) | scan (s1)
cl> print(s1)
89:18.500
--> iraf.printf("%12.1h", 89.3083333333, Stdout=1)[0]
'  89:18:30.0'
--> iraf.printf("%12.3m", 89.3083333333, Stdout=1)[0]
'   89:18.500'
```

`radix(value, base=10, length=0)` returns a string with `value` converted to `base`. The base must be an integer between 2 and 36 inclusive. If `length` is specified, the output string will be padded on the left with zeros to a minimum length of `length`.

```
--> print iraf.radix(0x21, 2)
100001
--> print iraf.radix(31, 2, 7)
0011111
```

There is also a radix function in the IRAF CL:

```
cl> print(radix(21x, 2))
100001
cl> print(radix(31, 2))
11111
```

`pyexecute(filename)` has been described before. This uses the Python `execfile()` function on `filename`, in the namespace of the current IRAF package. IRAF environment variables and syntax can be used for `filename`. Optional arguments include `PkgName` and `PkgBinary`.

8 Locating and Examining Translated CL Scripts

When the source for a PyRAF task is an IRAF CL script, PyRAF converts that CL script into an equivalent Python script, and it is that Python script that is actually executed when the task is invoked. The translation from CL to Python is normally done just once, when the task is first referenced, for example by using `lpar` or by running the task. Clearly, then, the translated script must be saved. The translated script (the Python equivalent of the CL script) is saved in a cache directory, which by default is the 'pyraf/clcache' subdirectory of the user's IRAF home directory.

The file names in the clcache directory look peculiar (e.g. 'uJJQIVVFrKg-9jq2znC4OA=='). Each name is constructed from the `hash()` of the contents of the corresponding CL script file, so the clcache files are tied to the contents of the CL scripts rather than to their names. One advantage of this is that if there are two or more identical CL scripts, they will map to the same clcache filename; such duplication does occur within the IRAF system. Another advantage to this scheme is that a CL script can be moved from one package to another (e.g. installing a script after testing in a local directory) without requiring recompilation of the cached version. It also allows multiple versions of IRAF on the same file system.

The files in the clcache directory are in Python pickle format. The `getCode()` method of the task retrieves the Python version of the script from the clcache directory and returns it as a string. Using the 'jqz.cl' example given above, the syntax would be `x = iraf.jqz.getCode()`. The cached version of the CL script will be regenerated automatically if the CL script is edited (and then used). However, sometimes it may be necessary to force regeneration of the Python version without touching the original CL script. For example, this would be necessary if PyRAF itself were modified in a way that affected the parameter handling mechanism. The `reCompile()` method recreates the Python equivalent of the CL script and saves the result in the clcache directory; for example, `iraf.jqz.reCompile()`.

9 Debugging Tips

When an error is encountered while running a Python script, you normally get a traceback that shows the location in all the functions that were called. This can be very verbose, and it is often not very helpful to see the complete traceback, so by default PyRAF only shows the most relevant portion of the traceback. If you really do want to see the full traceback, however, use the `.fulltraceback` command directive (which can be abbreviated, e.g. `.full`), which prints the complete traceback from the last error encountered.

There is a verbosity attribute which controls how much information is printed to the terminal window when running a task. The `setVerbose(value=1)` function sets the verbosity level to the specified value. You can use `print iraf.Verbose` to see the current verbosity level, or `if iraf.Verbose:` to test it. When PyRAF is started, `setVerbose(value=0)` is run to initialize the value to 0. Setting it to 1 results in only slightly more output, but setting it to 2 prints information about starting, running and shutting down a task.

10 Function and Object Reference

Note: This section is incomplete. More information will be added later.

The base classes for PyRAF tasks and parameters are `IrafTask` and `IrafPar` respectively; `IrafParList` is the class for a list of PyRAF parameters. These are used in the Python equivalent of CL scripts, and they include methods that are generally useful as well. `getCode()`, for example, is a method of `IrafCLTask` (which inherits `IrafTask`). This section describes some of the more useful methods of these classes, as well as some utility functions. A more complete description of these and some other classes will be added to this document at a later time.

10.1 Example

As an example, it may be helpful to explain the syntax of the Python equivalent of this very simple CL script:

```
procedure ttt(input)

string input = "" {prompt = "name of input image"}
string mode = "al"

begin
        imstat(input)
end
```

This is the Python equivalent, obtained by `iraf.ttt.getCode()`:

```
from pyraf import iraf
from pyraf.irafpar import makeIrafPar, IrafParList
from pyraf.irafglobals import *
import math

def ttt(input='', mode='al', DOLLARnargs=0, taskObj=None):

    Vars = IrafParList('ttt')
    Vars.addParam(makeIrafPar(input, datatype='string', name='input', mode='a',
        prompt='input image'))
    Vars.addParam(makeIrafPar(mode, datatype='string', name='mode', mode='h'))
    Vars.addParam(makeIrafPar(DOLLARnargs, datatype='int', name='$nargs',
        mode='h'))

    iraf.imstat(Vars.input)
```

The `makeIrafPar()` function and `IrafParList` class are imported from `pyraf.irafpar` because the script will use these for defining the task parameters. The `irafglobals` module defines a variety of things that may be used in CL scripts, such as the IRAF boolean values `yes` and `no`, `EOF` (end of file), and `INDEF`, which is often used as the initial value for task parameters.

`IrafParList` returns an object (`Vars`) that will contain a list of all the parameters for the task, in the form of `IrafPar` objects. `makeIrafPar()` returns an `IrafPar` object. An alternative to `makeIrafPar()` is the `IrafParFactory()` function, but its arguments correspond to the comma-separated descriptors in IRAF par files, and this is not very intuitive unless one is rather familiar with these files. The `addParam()` method of `IrafParList` appends the `IrafPar` object to the parameter list.

The `imstatistics` task is called with the value of the `ttt.input` parameter as the image name. Note that attribute syntax is used to get the value of this parameter. An alternative would be to use the `getValue()` method of `IrafParList`, but unless `native=1` is specified, `getValue` returns the value as a string, which could conflict with the data type expected by the called task. Using attribute syntax with the parameter name, the parameter value is returned with the "native" data type.

10.2 IrafTask

`IrafTask` is the base class for several more specific classes, such as `IrafCLTask`, `IrafPkg`, `IrafPset`, `IrafPythonTask` and `IrafForeignTask`. These would not normally be instantiated directly by a user or programmer. Once a task has been defined in PyRAF, the IRAF task object is "`iraf.`" followed by the task name. The task name can be abbreviated as long as the portion that was specified is unique ("minimum match"). The code for `IrafTask` is in 'iraftask.py' in the pyraf directory. The following describes some of the methods in `IrafTask`.

Parameter values may be set or gotten using the **`setParam()`** or **`getParam()`** method respectively. In the argument list, `qualifiedName` is the parameter name; the "qualified" part means that it may be prefixed by the task or package name, or it may have an index (for an array parameter) or a field specification. The field is what is referred to as a "parameter attribute" in the IRAF help page for parameters (type `phelp parameters`). The fields that may be set are `p_value, p_prompt, p_filename, p_minimum, p_maximum, p_mode`. The fields that may be gotten are `p_name, p_value, p_default, p_xtype, p_type, p_prompt, p_filename, p_minimum, p_maximum, p_mode`. Specify `exact=1` to disable minimum matching for the parameter name and qualifiers. The default `check=1` in `setParam()` means that the new value will be checked to see that it is within the minimum and maximum, or within the choice list for a string. The default `native=0` in `getParam()` means that the returned value will be a string; specify `native=1` to ensure that the returned value be in the "native" data type for the parameter.

`getParObject(paramname, exact=0, alldict=0)` returns the `IrafPar` object for `paramname`. The default `exact=0` means that minimum matching is allowed for the parameter name. The default `alldict=0` means that the parameter will be searched for only within the parameter list(s) for the current task, i.e. the parameter list for the task itself and all its psets. Specify `alldict=1` to enable searching for the parameter in all loaded tasks and packages, including the CL. One reason why it may be useful to get the `IrafPar` object is to make use of the methods in that class (see below).

The **`lParam(verbose=0)`** and **`dParam(cl=1)`** methods are like the `lparam` and `dparam` commands, or the `iraf.lparam()` and `iraf.dparam()` functions in the `iraf` module. Specifying `verbose=1` when calling `lParam` means that the minimum and maximum values or the choice list will also be printed, if these are defined. By default, `dParam` prints parameters and values the way IRAF's `dparam` does. The default `cl=1` means that the output will be in IRAF syntax, while specifying `cl=0` results in Python syntax. One difference is that with `cl=0` each line will begin with "`iraf.`", e.g. `iraf.imstatistics.images = 'l61h.fits[1]'`. Another is that if a parameter value was not specified, `cl=0` results in `par = None`, while `cl=1` results in `par = `. Note that "not specified" in this case is not the same as `INDEF` or `""`; it means that the parameter file does not give a default value and none has been assigned.

`unlearn()` deletes the copy of the parameter file from the user's uparm directory and resets the "current" parameter list to the default values which are defined in the original (template) parameter file.

10.3 IrafPar

`IrafPar` is the base class for scalar parameters. There is one subclass for each data type (or set of closely related types) that a parameter may have. Other base classes are `IrafArrayPar` and `IrafParL`, for array parameters and list-directed parameters respectively. Multiple inheritance with a "mix-in" design is used to construct classes for different parameter types, such as strings and list-directed parameters. The code for `IrafPar` is in 'irafpar.py' in the pyraf directory. The following describes some of the methods in the `IrafPar` class.

`getWithPrompt()` prints the prompt string and current value, gives the user the option to supply a new value (using tkreadline), and it updates the value attribute in the IrafPar object with the new value. It does not return the value.

`get(field=None, index=None, lpar=0, prompt=1, native=0, mode="h")` returns the value (or other field) of the parameter, prompting the user if `mode="q"` (or if `mode` inherits "q" from the CL). The parameter value (`p_value`) is the default for `field`; the options for `field` are (these are all strings, and minimum matching may be used when specifying them): `p_name, p_type, p_xtype, p_value, p_prompt, p_minimum, p_maximum, p_filename, p_mode, p_default`. Note that `p_length` is defined for IRAF string parameters, but this is not needed in PyRAF and has not been defined.

`set(value, field=None, index=None, check=1)` sets the value (or other field) of the parameter. Specifying `check=0` allows the parameter value to be outside the range from minimum to maximum (or not in the choice list for a string or integer parameter).

`checkValue(value, strict=0)` checks whether the specified value is within the range of allowed values, i.e. between the minimum and maximum for numeric parameters, or in the choice list if the object is a string or integer parameter. The specified value does not need to be of the same type as the parameter; the value will be converted to the correct type before being compared with the range of legal values. If the value is legal, the value converted to the correct type (or None) will be returned; if not, a ValueError exception will be raised.

`dpar(cl=1)` returns a string with "name = value" for the parameter. If the parameter is a string, the value will be enclosed in quotes. `cl=1` means write the assignments in a format that would be accepted by the iraf cl; `cl=0` means that if the parameter is a string and its value is null, the value will be shown as None instead of "".

`pretty(verbose=0)` returns a string with the parameter description as it would appear if the user typed the IRAF `lparam` command. String values will not be enclosed in quotes. If `verbose=1` is specified, the string includes the choice list or allowed range. For example:

```
--> print iraf.imarith.getParObject("op").pretty(verbose=1)
           op = +                Operator
                                 |+|-|*|/|min|max|
--> print iraf.sgraph.getParObject("left").pretty(verbose=1)
        (left = 0.0)             Left edge of viewport [NDC]
                                 0.0 <= left <= 1.0
```

`save(dolist=0)` returns a string with the parameter information formatted as you would see it in a parameter file. If `dolist=1`, the returned value will instead be a list of strings, one for each of the various components.

The `makeIrafPar()` factory function is a convenient way to create an `IrafPar` object from scratch:

`makeIrafPar(init_value, datatype=None, name="<anonymous>", mode="h", array_size=None, list_flag=0, min=None, max=None, enum=None, prompt="", strict=0)`

`init_value` can be an initial value, but an alternative is for it to already be an `IrafPar` object, in which case that object will simply be returned unchanged. `datatype` must be specified; the options are (these are strings): `string, char, file, struct, int, bool, real, double, gcur, imcur, ukey, pset`. Leave `array_size=None` (not 0) for a scalar parameter. Leave `min=None` when specifying `enum` (a choice list of possible values for a string parameter). A choice list can be given for an integer parameter as well, e.g. `enum = "36|37|38"`; this is an option which is not supported in IRAF. There is also a `filename` argument, but its use is deprecated. The default `strict=0` means that the function attempts to work around errors when reasonable; use `strict=1` to raise an exception for any error.

```
--> print pyraf.irafpar.makeIrafPar(3.141592653589793, datatype="real",
... name="pi", prompt="test")
<IrafParR pi r h 3.1415926535900001 None None "test">
```

There are a number of utility functions in PyRAF that are defined in the file 'iraffunctions.py' in the pyraf directory. There are of order 100 of these functions. Some implement functions in the IRAF CL, while others are unique to PyRAF.

Here are descriptions of the more useful of these functions in 'iraffunctions.py'.

Note: More detailed descriptions for these functions will be added later. The descriptions shown are taken from the doc strings.

help(object=__main__, variables=1, functions=1, modules=1, tasks=0, packages=0, hidden=0, padchars=16, regexp=None, html=0, **kw)
(This function is actually defined in 'irafhelp.py'.)
List the type and value of all the variables in the specified object.

- help() with no arguments will list all the defined variables.
- help("taskname") or help(IrafTaskObject) displays IRAF help for the task
- help(object) where object is a module, instance, etc., will display information on the attributes and methods of the object.
- help(function) will give the calling sequence for the function.

Optional keyword arguments specify what information is to be printed. The keywords can be abbreviated:

variables=1 Print info on variables/attributes
functions=1 Print info on function/method calls
modules=1 Print info on modules
tasks=0 Print info on IrafTask objects
packages=0 Print info on IrafPkg objects
hidden=0 Print info on hidden variables/attributes (starting with '_')
html=0 Use HTML help instead of standard IRAF help for tasks

regexp=None Specify a regular expression that matches the names of variables of interest. For example, `help(sys, regexp='std')` will give help on all attributes of sys that start with std. All the re patterns can be used.

Other keywords are passed on to the IRAF help task if it is called.

defpar(paramname)
Returns true if parameter is defined.

access(filename)
Returns true if file exists.

imaccess(filename)
Returns true if image matching name exists and is readable.

defvar(varname)
Returns true if CL variable is defined.

deftask(taskname)
Returns true if CL task is defined.

defpac(pkgname)
Returns true if CL package is defined and loaded.

boolean(value)
Convert Python native types (string, int, float) to IRAF boolean.

fscan(locals, line, *namelist, **kw)
fscan function sets parameters from a string or list parameter.

Uses local dictionary (passed as first argument) to set variables specified by list of following names. (This is a bit messy, literally using call-by-name for these variables.)

Accepts an additional keyword argument strconv with names of conversion functions for each argument in namelist.

Returns number of arguments set to new values. If there are too few space-delimited arguments on the input line, it does not set all the arguments. Returns EOF on end-of-file.

`fscanf(locals, line, format, *namelist, **kw)`
fscanf function sets parameters from a string/list parameter with format.

Implementation is similar to fscan but is a bit simpler because special struct handling is not needed. Does not allow strconv keyword.

Returns number of arguments set to new values. If there are too few space-delimited arguments on the input line, it does not set all the arguments. Returns EOF on end-of-file

`scan(locals, *namelist, **kw)`
Scan function sets parameters from line read from stdin.

This can be used either as a function or as a task (it accepts redirection and the _save keyword.)

`scanf(locals, format, *namelist, **kw)`
Formatted scan function sets parameters from line read from stdin.

This can be used either as a function or as a task (it accepts redirection and the _save keyword.)

`nscan()`
Return number of items read in last scan function.

`set(*args, **kw)`
Set IRAF environment variables.

`show(*args, **kw)`
Print value of IRAF or OS environment variables.

`unset(*args, **kw)`
Unset IRAF environment variables.

This is not a standard IRAF task, but it is obviously useful. It makes the resulting variables undefined. It silently ignores variables that are not defined. It does not change the OS environment variables.

`time(**kw)`
Print current time and date.

`beep(**kw)`
Beep to terminal (even if output is redirected).

`clOscmd(s, **kw)`
Execute a system-dependent command in the shell, returning status.

`stty(terminal=None, **kw)`
IRAF stty command (mainly not implemented).

`edit(*args, **kw)`
Edit text files.

`clear(*args, **kw)`
Clear screen if output is terminal.

`flprcache(*args, **kw)`
Flush process cache. Takes optional list of tasknames.

`prcache(*args, **kw)`
Print process cache. If args are given, locks tasks into cache.

gflush(*args, **kw)
Flush any buffered graphics output.

history(n=20, *args, **kw)
Print history.

Does not replicate the IRAF behavior of changing default number of lines to print.

hidetask(*args, **kw)
Hide the CL task in package listings.

task(*args, **kw)
Define IRAF tasks.

redefine(*args, **kw)
Redefine an existing task.

package(pkgname=None, bin=None, PkgName="", PkgBinary="", **kw)
Define IRAF package, returning tuple with new package name and binary.

PkgName, PkgBinary are old default values. If `Stdout=1` is specified, returns output as string array (normal task behavior) instead of returning PkgName, PkgBinary. This inconsistency is necessary to replicate the inconsistent behavior of the package command in IRAF.

clPrint(*args, **kw)
CL print command – emulates CL spacing and uses redirection keywords.

printf(format, *args, **kw)
Formatted print function.

pwd(kw)**
Print working directory.

chdir(directory=None, **kw)
Change working directory.

cd(directory=None, **kw)
Change working directory (same as `chdir`).

back(kw)**
Go back to previous working directory.

Expand(instring, noerror=0)
Expand a string with embedded IRAF variables (IRAF virtual filename).

Allows comma-separated lists. Also uses `os.path.expanduser` to replace `'~'` symbols. Set noerror flag to silently replace undefined variables with just the variable name or null (so `Expand("abc$def") = "abcdef"` and `Expand("(abc)def") = "def")`. This is the IRAF behavior, though it is confusing and hides errors.

load(pkgname, args=(), kw=None, doprint=1, hush=0, save=1)
Load an IRAF package by name.

run(taskname, args=(), kw=None, save=1)
Run an IRAF task by name.

getAllTasks(taskname)
Returns list of names of all IRAF tasks that may match taskname.

getAllPkgs(pkgname)
Returns list of names of all IRAF packages that may match pkgname.

getTask(taskname, found=0)
Find an IRAF task by name using minimum match.

Returns an `IrafTask` object. Name may be either fully qualified (package.taskname) or just the taskname. taskname is also allowed to be an `IrafTask` object, in which case it is simply returned. Does minimum match to allow abbreviated names. If found is set, returns None when task is not found; default is to raise exception if task is not found.

`getPkg(pkgname, found=0)`
Find an IRAF package by name using minimum match.

Returns an `IrafPkg` object. pkgname is also allowed to be an `IrafPkg` object, in which case it is simply returned. If found is set, returns None when package is not found; default is to raise exception if package is not found.

`getTaskList()`
Returns list of names of all defined IRAF tasks.

`getTaskObjects()`
Returns list of all defined `IrafTask` objects.

`getPkgList()`
Returns list of names of all defined IRAF packages.

`getLoadedList()`
Returns list of names of all loaded IRAF packages.

`getVarDict()`
Returns dictionary of all IRAF variables.

`getVarList()`
Returns list of names of all IRAF variables.

`listAll(hidden=0, **kw)`
List IRAF packages, tasks, and variables.

`listPkgs(**kw)`
List IRAF packages.

`listLoaded(**kw)`
List loaded IRAF packages.

`listTasks(pkglist=None, hidden=0, **kw)`
List IRAF tasks, optionally specifying a list of packages to include.

Package(s) may be specified by name or by `IrafPkg` objects.

`listCurrent(n=1, hidden=0, **kw)`
List IRAF tasks in current package (equivalent to '?' in the cl).

If parameter n is specified, lists n most recent packages.

`listVars(prefix="", equals="\t= ", **kw)`
List IRAF variables.

`curpack()`
Returns name of current CL package.

`curPkgbinary()`
Returns name of pkgbinary directory for current CL package.

`pkgHelp(pkgname=None, **kw)`
Give help on package (equivalent to CL '? [taskname]').

`allPkgHelp(**kw)`
Give help on all packages (equivalent to CL '??').

`clCompatibilityMode(verbose=0, _save=0)`
Start up full CL-compatibility mode.

clArray(array_size, datatype, name="<anonymous>", mode="h", min=None, max=None, enum=None, prompt=None, init_value=None, strict=0)
Create an `IrafPar` object that can be used as a CL array.

clExecute(s, locals=None, mode="proc", local_vars_dict=, local_vars_list=[], verbose=0, **kw)
Execute a single cl statement.

IrafTaskFactory(prefix="", taskname=None, suffix="", value=None, pkgname=None, pkgbinary=None, redefine=0, function=None)
Returns a new or existing `IrafTask`, `IrafPset`, or `IrafPkg` object.

Type of returned object depends on value of suffix and value.

Returns a new object unless this task or package is already defined. In that case if the old task appears consistent with the new task, a reference to the old task is returned. Otherwise a warning is printed and a reference to a new task is returned.

If redefine keyword is set, the behavior is the same except a warning is printed if it does *not* exist.