IRAF CL Script Tips & Tricks

by the NOAO IRAF Team:

Mike Fitzpatrick

Rob Seaman

Frank Valdes

Nelson Zárate

R. Seaman – 22 July 2003

References

- *“An Introductory User’s Guide to IRAF Scripts” by Anderson and Seaman*
- *Document refers to v2.8, but still valid*
- *http://iraf.noao.edu/docs/prog.html*
- *Further references within User’s Guide*
- *http://iraf.noao.edu/iraf/web/irafnews*
- *Keep it simple! Let the tasks do the work.*

What’s new?

- *What’s new since the User’s Guide?*
- `scan()` *from a pipe*
- `printf()` *– more C-like than SPP*
- *The CL* `printf` *is called as a task, not as a function in expressions, but*
- *CL* `printf` *requires no* `call`*, unlike SPP*
- *CL* `printf` *supports same formats as SPP*

Procedure Scripts

- *All scripts should be* **procedure** *scripts*
- *The task name must match the file name*
- **procedure** *arguments are query parameters*
- *Two CL modes, “command” & “compute”*
- *Command mode is used interactively*
- *Compute mode requires parentheses, commas and quoted string literals*
- **procedure** *scripts require CL compute mode*

Magic Words

- *The magic word for SPP is* `flpr`
- *The magic word for CL scripts is* `unlearn`
- *Any time a parameter is changed when writing a script, unlearn the task*

Prompting Users for Input

- *Don't use* `printf` *and* `scan` *to ask questions*
- *Use query parameters instead*
- `task askit = home$askit.cl`

Prompting Users (*example*)

```
procedure askit (question)

bool question = yes {prompt="Do you want to continue?"}

begin
        bool l_question

        l_question = question

        if (l_question)
            printf ("The answer was yes\n")
        else
            printf ("The answer was no\n")
end
```

Prompting Users (*example* 2)

```
procedure askit2 (question)

bool question {prompt="must have some placeholder"}

begin
        bool l_question

        question.p_prompt = "Is this a question?"
        question.p_value = no

        l_question = question

        if (l_question)
            printf ("The answer was yes\n")
        else
            printf ("The answer was no\n")
end
```

Prompting Users (*example 3*)

```
cl> askit
Do you want to continue? (yes): <cr>
The answer was yes

cl> askit2
Is this a question? (no): yes
The answer was yes
```

List Directed Parameters

- *Use “list directed” parameters to read input from files.*
- *A list directed parameter is specified by prepending an asterisk to a parameter declaration of any type (but typically string).*
- *Open a file by assigning a value. (LHS)*
- *Each subsequent reference will return the next line in the file pointed to by the parameter. (RHS)*
- *Close a file by reading to EOF or by assigning a null string.*

List Directed Parameters (*ex.*)

```
cl> type test1
this is line 1
this is line 2
this is line 3
cl> string *ld
cl> ld = "test1"
cl> = ld
this is line 1
cl> = ld
this is line 2
cl> = ld
this is line 3
cl> = ld
EOF
```

List Directed Parameters (*ex.* 2)

```
cl> task listit = listit.cl
cl> listit listit.cl
procedure listit (input)

string input {prompt="Input file"}
string  *list

begin
        string l_input
        struct line

        l_input = input

        list = l_input
        while (fscan (list, line) != EOF) {
            printf ("%s\n", line)
        }
end
```

`string` versus `struct`

- *A* `string` *is a* `string` *is a* `string` *(or a* `char`*)*
- *A* `struct` *is identical to a* `string` *for all purposes except when scanning a value*
- *Scanning into a* `string` *terminates at any whitespace character*
- *Scanning into a* `struct` *continues to the end of the input line (up to 64 characters)*

`scan()` and `fscan()`

- *Use* `fscan()` *to read from string*
- *Use* `scan()` *to read from STDIN*
- *Each function returns the number of values successfully scanned – or returns EOF*
- *A subsequent* `nscan()` *returns the no. of values*
- `scan()` *from pipe to capture task output into a variable or several variables*
- `scan()` *from* `printf()` *is equivalent to* `sprintf()`

scan() and fscan() (*ex. 1*)

```
cl> string test = "word 17 3.14 now is the time"
cl> = fscan (test, s1, i, x, line)
4
cl> = s1
word
cl> = i
17
cl> = x
3.14
cl> = line
now is the time
cl> = nscan()
4
```

`scan()` and `fscan()` (*ex.* 2)

`STDIN` *may be used most places a filename is allowed*

`fscan (STDIN, s1)` *is equivalent to* `scan (s1)`

```
cl> = fscan (STDIN, s1)
asdf
1
cl> = s1
asdf
```

scan() and fscan() (*ex. 3*)

```
cl> = scan (s1) ^D
-2
```

```
cl> grep "EOF" /iraf/iraf/unix/hlib/iraf.h
define  EOF             -2
```

```
cl> !stty all
...
discard dsusp   eof     ...
^O      ^Y      ^D      ...
```

scan() and fscan() (*ex. 4*)

```
cl> imstat dev$pix
#               IMAGE      NPIX      MEAN    STDDEV       MIN       MAX
              dev$pix    262144     108.3     131.3       -1.    19936.


cl> imstat ("dev$pix", fields="mean,stddev", format-) | scan (x, y)
cl> = x
108.3154
cl> = y
131.298


cl> printf ("%6.2f +/- %6.2f\n", x, y) | scan (line)
cl> = line
108.32 +/- 131.30
```

Direct Command Execution

The cl can be called as a task to interpret a command as with the Unix `eval` *command:*

```
cl> printf ("imstat ('%s', fields='%s', format-)\n", s1, s2) | cl
108.3154  262144
```

```
cl> printf ("imstat ('%s', fields='%s', format-)\n", s1, s2) | cl \
>>> | scan (x, i)
cl> real total
cl> total = x * i
cl> = total
28394232.2176
```

Host Command Execution

Sometimes the best way to perform some chore is to escape from the CL to the Unix shell. (A little of this goes a long way.)

```
cl> s1 = mktemp ("tmp$tmp")
cl> imhead ("dev$pix", lo+, > s1)
cl> printf (”!grep Overscan %s\n”, osfn(s1)) | cl
BT-FLAG = 'Apr 22 14:11 Overscan correction strip is [515:544,3:510]'
```

but, be sure you really need to do so, first:

```
cl> match ("Overscan", s1)
BT-FLAG = 'Apr 22 14:11 Overscan correction strip is [515:544,3:510]'
```

Host Commands (#2)

or even:

```
cl> imhead ("dev$pix", lo+) | match ("Overscan")
BT-FLAG = 'Apr 22 14:11 Overscan correction strip is [515:544,3:510]'
```

and, if you do need to run a host level command, a foreign task is often best:

```
cl> task $grep = "$grep $1 $(2)" # parentheses substitute host path
cl> grep ("Overscan", s1)
BT-FLAG = 'Apr 22 14:11 Overscan correction strip is [515:544,3:510]'
```

Host Commands (#3)

Note that foreign tasks can be run in the IRAF background and that their input and output can be redirected to a file or pipe:

```
cl> imhead ("dev$pix", lo+) | grep ("Overscan")
BT-FLAG = 'Apr 22 14:11 Overscan correction strip is [515:544,3:510]'
```

And a reminder that IRAF networking is ubiquitous:

```
cl> !hostname
tucana
cl> !gemini!hostname
gemini
```

String and Math Functions

User's Guide mentions various references, e.g.:

```
cl> phelp language      # and individual help pages
cl> phelp strings       # strings can also be directly manipulated
cl> phelp mathfcns      # typical variety of functions
```

Strings can be compared using `==`*, or operated on directly, e.g.,* `//` *concatenation. Functions are:*

```
s1 = str (x)                      # convert x to a string
s1 = substr ("abcdefg", 2, 4)     # s1 = "bcd"
 i = stridx ("abc", " eeboq")     #  i = 4
 i = strlen ("abc")               #  i = 3
s1 = envget ("imtype")            # s1 = "fits"
```

Temporary (Scratch) Files

Use the `mktemp()` *function:*

```
cl> string tmpfile
cl> tmpfile = mktemp (”tmp$junk”)
cl> hselect (”dev$pix”, “naxis*”, yes, > tmpfile)
cl> = tmpfile
tmp$junk9674a
cl> type tmp$junk9674a
2       512     512
cl> type (tmpfile)
2       512     512
```

Image and File Templates

- *Templates include single files and comma delimited lists.*
- *Templates include "`*`" and "`?`" wildcards.*
- *Templates include "`%`" and "`//`" operators.*
- *Templates include "`@`" files.*
- *Don't interpret image or file templates directly.*
- *Don't pass explicit lists of images or files.*
- *Rather, use the* `sections` *or* `files` *tasks and allow the user to pass in any template they wish.*

Image and File Templates (*ex.*)

```
cl> sections *.fits
testim1.fits
testim2.fits
testim3.fits

cl> sections %test%out%im?.fits
outim1.fits
outim2.fits
outim3.fits

cl> sections (imlist, opt=”full”, > tmpfile)
```

Using Image Sections

Image sections are implicit in many image processing operations:

```
imtranspose test[-*,*] cw90   # rotate 90 degrees clockwise
imtranspose test[*,-*] ccw90 # rotate 90 degrees counter-clockwise
imcopy test[-*,-*] rot180     # rotate 180°
imcopy test[-*,*] vflip       # flip about the vertical (y) axis
imcopy test[*,-*] hflip       # flip about the horizontal (x) axis
```

Subsample horizontally by a factor of three and vertically by four or find the max data value in the first fifty even numbered pixels of line seven:

```
imcopy test[*:3,*:4] test
imstat test[2:100:2,7] fields=max
```

Reading Image Headers

- `imgets` *requires special handling to work correctly in the background*
- *Background tasks cannot update parameters*
- `cache` *tasks to avoid problem, but*
- `hselect` *is better solution anyway since it allows multiple keywords to be read at once*

Reading Image Headers (*ex.*)

```
cache imgets
imgets ("dev$pix", "ra")
s1 = imgets.value
x = real (s1)
imgets ("dev$pix", "dec")
s1 = imgets.value
x = real (s1)
```

versus

```
hselect ("dev$pix", "ra,dec", yes) | scan (x, y)
```

Note that sexigesimal values are recognized as real numbers by the CL.

Put it all together

Task that expands an image template, reads each header and does something:

```
cl> template *.fits
1: testim1.fits 13:29:24.00 (13.490) 47:15:34.00 (47.259)
2: testim2.fits 12:59:47.00 (12.996) 43:12:59.00 (43.216)
3: testim3.fits 09:22:13.00 ( 9.370) 11:38:13.00 (11.637)

cl> template @inlist
1: testim3.fits 09:22:13.00 ( 9.370) 11:38:13.00 (11.637)
2: testim2.fits 12:59:47.00 (12.996) 43:12:59.00 (43.216)
3: testim1.fits 13:29:24.00 (13.490) 47:15:34.00 (47.259)
```

Script for this example is on the following slide.

```
# Expand image template, read headers, do something

procedure template (images)

string images {prompt="Input images"}
string *list

begin
        string l_images, img, tmpfile, ra, dec
        int i

        l_images = images

        tmpfile = mktemp ("tmp$tmp")
        sections (l_images, opt="full", > tmpfile)

        list = tmpfile
        for (i=1; fscan (list, img) != EOF; i+=1) {
            hselect (img, "ra,dec", yes) | scan (ra, dec)
            printf ("%d: %s %s (%6.3f) %s (%6.3f)\n",
                i, img, ra, real(ra), dec, real(dec))
        }

        delete (tmpfile, ver-, >& "dev$null")
end
```

Useful Tasks for Scripts

- sections *or* files
- translit
- fields
- joinlines
- match
- mktemp
- imextensions *or* mscextensions
- imaccess *or* access
- imexpr
- wcstran
- imstat *or* mimstat
- hedit

stty Playback Scripts

CL scripts are not the only type of IRAF scripts. Playback (`stty`*) scripts are ideal for demos and regression testing (see* `help stty`*):*

```
cl> stty login=test.stty
cl> imhead *.fits
...
cl> stty reset

cl> stty play=test.stty
cl> imhead *.fits                      # these lines were
testim1.fits[10][short]: m51  B  600s  # automatically executed
testim2.fits[10][short]: m51  B  600s  # by the computer
testim3.fits[10][short]: m51  B  600s  #
cl> stty reset                         #

cl> edit test.stty    # simple format for easy revision
```

What’s Next?

Visit http://iraf.noao.edu
Send email to iraf@noao.edu