Programación de Scripts en IRAF (I)

Marzo de 2006

Ricardo Gil-Hutton

- Un script permite ejecutar una serie de comandos como un conjunto de órdenes.
- CL (*"command language"*) es un intérprete de instrucciones.
- CL se puede utilizar en dos modos diferentes: el modo tradicional denominado de **comando**, y como un lenguaje de programación en el modo de **programa**.
- ambos modos pueden mezclarse en un script.
- Los componentes y funciones del lenguaje se detallan en el paquete LANGUAGE (*help language*).
- La tarea *MKSCRIPT* es una forma simple de crear scripts, pero poco práctica.
- también se pueden agrupar comandos en órdenes compuestas (con '{}') para crear scripts.

```
cl> {
>>> imcopy ima1.fit ima2.fit
>>> imarith ima2.fit * 2. ima2.fit
>>> clear
>>> }
```

Elementos del lenguaje - Variables:

- variables pre-declaradas:
  - 3 variables de cadenas de carácteres o string (*s1, s2, s3*).
  - 3 variables lógicas (*b1, b2, b3*).
  - 3 variables enteras (*i, j, k*).
  - 3 variables reales (*x, y, z*).
  - 1 variable de lista (*list*).
- para asignar y ver el valor de una variable:

```
cl> i = 8
cl> = i
8
```

- las variables de string **siempre** se encierran en comillas:

```
cl> s1 = "Hola a todos!"
cl> = s1
Hola a todos!
cl> s1 = Hola
ERROR: parameter ‘Hola’ not found
```

Elementos del lenguaje - Componentes:

- condicional *if* - *else*:

```
cl> i = 2
cl> if (i > 0){
>>> print("grande")}
>>> else {
>>> print("chico")}
grande
```

- condicional *while*:

```
cl> i = 3
cl> while (i > 0){
>>> print(i)
>>> i += 1 }
3
2
1
```

- loop *for*:

```
cl> for (i = 0; i < 3; i += 1) print(i)
0
1
2
```

Elementos del lenguaje - Componentes:

- *break - next*:

```
cl> for (i = 0; i < 8; i += 1) {
>>> if (i == 6)
>>> break
>>> if (i == 2)
>>> next
>>> print(i) }
0
1
3
4
5
```

- cursores *gcur - imcur*:

  devuelve un string conteniendo las coordenadas del cursos gráfico o de imagen, respectivamente, y la tecla presionada. Si la tecla es ":" espera que se ingrese un string seguido de ¡enter¿:

  X, Y, sist. coord., comando (char)

  X, Y, sist. coord., ':', comando (string)

```
cl> =gcur
123.22 324.65 1 k
```

- también hay disponibles instrucciones para realizar bifurcaciones (como *switch - case - default*), saltos (*return - goto - label*) y **definir nuevas tareas** (*procedure*).

Comandos y funciones - Tareas:

- las tareas se pueden ejecutar en **dos modos**. En modo **comando**:
  tarea arg1 arg2 ... par1=val1 par2=val2 ... redir
  como, por ejemplo:

```
cl> imcopy arch1 arch2 verbose=no
```

- y en modo **programa**:
  tarea(arg1, arg2, ..., par1=val1, par2=val2, ..., redir)
  como, por ejemplo:

```
cl> imcopy("arch1","arch2",verbose=no)
```

- en los modos de comando y programa *redir* es un argumento de redirección:

>arch	salida al archivo *arch*
<arch	entrada desde el archivo *arch*
>>arch	agrega la salida al archivo *arch*
>& arch	errores y salida al archivo *arch*
>>& arch	agrega errores y salida al archivo *arch*
>G arch	redirecciona la salida gráfica al archivo *arch*
>>G arch	agrega la salida gráfica al archivo *arch*

Comandos y funciones - Parámetros:

- las tareas pueden requerir argumentos o *parámetros*:

  en modo comando:

  tarea arg1 arg2 ... par1=val1 par2=val2 ... redir

  en modo programa:

  tarea(arg1, arg2, ..., par1=val1, par2=val2, ..., redir)

  los argumentos *arg1, arg2, ...* se denominan *parámetros posicionales* de la tarea, mientras que los valores *val1, val2, ...* se asignan a los parámetros *par1, par2, ...* que se denominan *parámetros ocultos*.

- los argumentos de los *parámetros posicionales* deben aparecer al inicio de la lista de argumentos y en un orden establecido.

- los argumentos de los *parámetros posicionales* deben indicarse explicitamente, en caso contrario serán solicitados por la tarea.

- los *parámetros ocultos* no deben guardar un orden dado ni indicarse explicitamente.

- para los parámetros a los que se asigna un valor lógico (*si* o *no*) se puede utilizar una sintaxis alternativa con '+' o '-'.

  Ejemplo:

  ```
  cl> imcopy("arch1","arch2",verbose=no)
  ```

  es igual a:

  ```
  cl> imcopy("arch1","arch2",verbose-)
  ```

Comandos y funciones - Parámetros:

- los parámetros pueden ser de tipo:
  - *char*: hasta 64 caracteres. Un solo caracter es tratado como un *int* con un valor igual al código ASCII. Se separan con espacios o tabs.
  - *int*: se almacenan internamente como long int (32 bits).
  - *real*: se almacenan internamente como double, pueden tener o no punto decimal o exponente (E no D).
  - *bool*: sólo permite valores de *si* o *no*.
  - *string*: es similar al *char*.
  - *struct*: es similar al *string*, pero no realiza parsing cuando se usa con las funciones *scan* u *fscan*.
  - *file*: es un *string* que deben ser nombres de archivo válidos.
  - *gcur,imcur*: poseen un *string* asociado con un formato fijo.
- parámetros que son *punteros a listas*: se indican con un '*' antes del nombre, y se inicializan con el nombre de un archivo que contenga una lista a leer. En referencias sucesivas contiene los elementos de la lista a que apunta. Para cerrar el archivo se le asigna un string nulo.
- los parámetros poseen un *modo* que determina si el parámetro se solicita (*query*) o si se guarda para ser reutilizado (*learn*).

Comandos y funciones - Parámetros:

- el usuario puede definir *arrays* de dimensiones arbitrarias, los que se referencian mediante índices en corchetes:

```
real x[10], y[10], z[10,10], w[6,10]

y = 2. * x
y = z[1,*]
w = z[1:6,*]
```

- un parámetro es accesible implicitamente si la tarea en la que esta definido se encuentra activada.
- un parámetro es accesible explicitamente si el paquete que contiene a la tarea en la que esta definido se encuentra activado (cargado).
- CL esta siempre activo y sus parámetros son accesibles.
- los valores para los parámetros se almacenan en:
  1. si es un script, los valores iniciales están en el mismo archivo del script.
  2. si es un ejecutable, los valores iniciales están en un archivo en el mismo directorio que el ejecutable.
  3. si el flag *learn* esta seteado, las tareas "aprenden" nuevos valores para sus parámetros que se guardan en un archivo en el directorio **uparm**. Para volver a los valores iniciales se debe ejecutar UNLEARN *tarea*.

scan - fscan - scanf - fscanf:

- SCAN lee parámetros desde STDIN.
- FSCAN lee parámetros desde un archivo u otro parámetro.
- SCANF es igual a SCAN pero acepta formato.
- FSCANF es igual a FSCAN pero acepta formato.

```
scan (p1, p2, ..., pn)
fscan (param, p1, p2, ..., pn)
scanf (format, p1, p2, ..., pn)
fscanf (param, format, p1, p2, ...,pn)
```

pN son los parámetros de salida, *param* es el parámetro que se va a leer, y *format* es un formato estilo C (del tipo "% [w]C").

- en el caso de FSCAN o FSCANF, si *param* es un struct puntero a lista la función lee una línea del archivo referenciado, la que se divide en valores (separados por espacios o tabs) que se asignan a cada parámetro.
- al ejecutarse devuelven el número de parámetros leidos o EOF si encuentra el fin de archivo.

```
cl> list = coords
cl> while (fscan (list,x,y) != EOF){
>>> z = x + y }
cl> =list
EOF
```

fprint - print - printf:

- FPRINT imprime a un parámetro.
- PRINT imprime a STDOUT.
- PRINTF es igual a PRINT pero acepta formato.

```
fprint param expr [expr ...]
print expr [expr ...]
printf format [arg ...]
```

expr es una expresión o el string a imprimir, *param* es el parámetro donde se va a escribir el valor de *expr*, *format* es un formato estilo C (del tipo "% [w][.d]Cn"), y *arg* es cualquier expresión, variable, o valor a utilizar con el format.(*n* puede ser un retorno de carro, un tab, etc.).

```
cl> for (x=1.; x < 7.; x += 1.)
>>> print(x,(x+2.),>> "coors")
cl> list = coords
cl> while (fscan (list,x,y) != EOF) print (x,y)
1. 3.
2. 4.
3. 5.
4. 6.
5. 7.
6. 8.
```

Funciones para manipular strings:

```
str (arg)
substr (str, init, fin)
stridx (test, str)
strldx (test, str)
strlen (str)
strlwr (str)
strupr (str)
strstr (str1,str2)
strlstr (str1,str2)
```

- STR convierte *arg* en string.
- SUBSTR extrae una sub-string de *str*.
- STRIDX encuentra la posición de la primera ocurrencia de cualquier caracter de *test* en *str*, devolviendo cero en error.
- STRLDX encuentra la posición de la última ocurrencia de cualquier caracter de *test* en *str*, devolviendo cero en error.
- STRLEN encuentra la longitud de *str*.
- STRLWR convierte *str* a minúsculas.
- STRUPR convierte *str* a mayúsculas.
- STRSTR encuentra la posición de la primera ocurrencia de *str1* en *str2*, devolviendo cero en error.
- STRLSTR encuentra la posición de la última ocurrencia de *str1* en *str2*, devolviendo cero en error.
- para unir dos strings se utiliza '//'.

Funciones para manipular strings:

```
cl> j = 3
cl> s1 = str (j)
cl> = s1
3
cl> s1 = substr ("qwertyui",3,6)
cl> = s1
erty
cl> s1 = substr ("qwertyui",5,2)
trew
cl> j = stridx ("abcde","qwertyui")
cl> = j
3
cl> j = strldx ("/","/home/rgh/test/test.imh")
cl> = j
15
cl> j = strlen ("test.imh")
cl> = j
8
cl> j = strstr ("test","/home/rgh/test/test.imh")
cl> = j
11
cl> j = strlstr ("test","/home/rgh/test/test.imh")
cl> = j
16
```

Formatos numéricos:

- números en formato sexagesimal (*GG:MM:SS*) pueden ingresarse cuando el argumento es un real.
- los enteros en octal se indican agregando 'b' o 'B' y en hexadecimal agregando 'x' o 'X'. La función *radix* puede convertir a cualquier base:

```
cl> = radix (aX,10)
10
```

- la función *radix* puede utilizarse para obtener códigos ASCII:

```
cl> i ="Z"
cl> = radix (i,10x)
5A
```

Comparaciones lógicas:

&&	AND
||	OR
!	NOT
==	igual a
!=	no igual a
>	mayor que
>=	mayor o igual que
<	menor que
<=	menor o igual que

Funciones matemáticas:

- siguen una notación similar a FORTRAN.

```
sin (x)
cos (x)
tan (x)
atan2 (x,y)
sqrt (x)
exp (x)
log (x)
log10 (x)
frac (x)
abs (x)
min (x, y, z, ...)
max (x, y, z, ...)
real (i)
int (x)
```

Estructura de los scripts:

- La forma básica de un script es:

```
procedure xxxxx (arg1, arg2, arg3, ...)

declaracion de parametros posicionales
declaracion de parametros ocultos

begin
  declaracion de variables locales

  comandos
  ...
  ...
end
```

  donde *argN* son los parámetros posicionales requeridos.

- es usual guardar el script en un archivo con el mismo nombre y extensión '*.cl* '.

- un *puntero a lista* **DEBE** ser declarado como parámetro, pero no como argumento.

- **TODAS** las variables locales deben declararse en el script.

- los paquetes a utilizar por el script **DEBEN** estar cargados previamente.

- toda línea del script que se inicie con '#' se considera un comentario.

Estructura de los scripts:

- para ejecutar el script:

  ```
  cl> cl < xxxxx.cl
  ```

  o convertirlo previamente a una tarea y luego ejecutarlo como tal:

  ```
  cl> task xxxxx = xxxxx.cl
  cl> xxxxx
  ```

- si el script no tiene parametros se agrega un '$' delante del nombre de la tarea para indicar esta condición:

  ```
  cl> task $xxxxx = xxxxx.cl
  ```

- si el script se va a usar frecuentemente es conveniente definirlo como tarea e incluirlo en el archivo *loginuser.cl* para que sea cargado al iniciar IRAF. En este caso, se deben agregar las siguientes líneas al archivo *loginuser.cl*:

  ```
  task $xxxxx = /home/.../xxxxx.cl
  keep
  ```

- si se convierte el script a una tarea, cuando se definen los parámetros se pueden especificar valores iniciales y el *prompt*:

  ```
  procedure centro (imagen, coords)
  string imagen {prompt="Imagenes a centrar"}
  string coords {prompt="Centros aproximados"}
  int cbox {5,min=5,max=9,prompt="Caja de centrado"}
  ...
  ...
  begin
    ...
    ...
  end
  ```

Un script de ejemplo:

```
procedure ejem1 (archivo)
#
# script de ejemplo de la funcion fscan
# definiendo la variable line como STRUCT
#
#--------------------------------------
# se definen los parametros
#
string archivo  {prompt="Archivo a procesar"}
struct *flist

# inicio del script
#
begin
  struct line # linea a leer
  flist=archivo # asigna archivo a puntero

# lee hasta encontrar EOF e imprime
#
  while(fscan(flist,line) != EOF)
    print(line)
end
#
# fin script
```

Algunas funciones útiles:

- para un listado más extenso, ver el paquete LANGUAGE (*help language*):

access	comprueba que un archivo existe
clear	limpia la pantalla de la terminal
defpac	comprueba que un paquete esta definido
defpar	comprueba que un parámetro este definido
deftask	comprueba que una tarea esta definida
envget	obtiene el valor (en string) de una variable del ambiente
error	imprime el código de error, un mensaje y aborta la tarea
mktemp	crea un nombre de archivo temporario y único
reset	resetea el valor de una variable del ambiente
set	asigna un valor a una variable del ambiente
show	muestra el valor de una variable del ambiente
time	imprime la hora actual
unlearn	restablece los valores iniciales de parámetros
whereis	localiza todas las ocurrencias de una tarea en paquetes
which	localiza una tarea en paquetes

¿Cómo hacer el help de la tarea?

- en el paquete SOFTOOLS, la tarea MKMANPAGE carga en el editor un template de help y permite completarlo.
- haciendo *help lroff* se obtiene una explicación de la sintaxis.
- el help de una tarea *xxxxx* debe guardarse en un archivo *xxxxx.hlp*, y éste en un subdirectorio *doc* del directorio donde se guarda la tarea (o el script).
- help para la tarea EJEM1 (antes script!):

```
.help ejem1 Jan06 curso
.ih
NAME
ejem1 -- script de ejemplo sobre lectura de archivos
.ih
USAGE
ejem1 arch
.ih
PARAMETERS
.ih arch
Es el archivo de donde se lee.
DESCRIPTION
Este script permite estudiar como se lee un archivo
con informacion util desde IRAF. Es muy sencillo
pero permite probar diferentes tipos de variables
para comprender como se utilizan.
...
```

¿Cómo hacer un paquete con mis tareas?

1. crear un directorio para el paquete, y en éste un subdirectorio *doc*. Poner los scripts en éste directorio y los helps en el subdirectorio *doc*.

2. incluir en el directorio del paquete un archivo *curso.cl* con el siguiente formato:

```
# script de definicion del paquete CURSO
#
# cargar antes otros paquetes necesarios
# que se requieran

# directorio de la tarea

set direc   = "/home/rgh/imagenes/curso/"

# defino el paquete

package curso

task ejem1  = "direc$ejem1.cl"
task ejem1a = "direc$ejem1a.cl"
task ejem1b = "direc$ejem1b.cl"
task ejem1c = "direc$ejem1c.cl"

clbye()
```

3. incluir en el directorio del paquete un archivo *curso.hd* con el siguiente formato:

```
# archivo .hd del paquete CURSO
#

$doc        = "/home/rgh/imagenes/curso/doc/"
```

```
ejem1       hlp=doc$ejem1.hlp,  src=ejem1.cl
ejem1a      hlp=doc$ejem1a.hlp, src=ejem1a.cl
ejem1b      hlp=doc$ejem1b.hlp, src=ejem1b.cl
ejem1c      hlp=doc$ejem1c.hlp, src=ejem1c.cl
```

4. incluir en el directorio del paquete un archivo *curso.men* con el siguiente formato:

```
ejem1   - programa de ejemplo 1
ejem1a  - programa de ejemplo 1a
ejem1b  - programa de ejemplo 1b
ejem1c  - programa de ejemplo 1c
```

5. incluir en el directorio del paquete un archivo *root.hd* con el siguiente formato:

```
# directorio raiz del help del paquete CURSO
#
_curso    pkg = _curso.hd
```

   y ejecutar desde el paquete SOFTOOLS:

```
cl> mkhelpdb root.hd cursodb.mip
```

6. por último, incluir en el directorio del paquete un archivo *_curso.hd* con el siguiente formato:

```
# archivo de definiciones del paquete CURSO
#
curso     men = curso.men,
          hlp = ..,
          sys = curso.hlp,
          pkg = curso.hd,
          src = curso.cl
```

7. para definir el paquete se debe agregar en el archivo *loginuser.cl* las líneas:

```
task $curso = /home/rgh/imagenes/curso/curso.cl
reset helpdb = (envget("helpdb") //
         ",/home/rgh/imagenes/curso/cursodb.mip")

keep
```

lo que permite cargar de ahora en mas el paquete CURSO con:

```
cl> curso
   ejem1  ejem1a  ejem1b  ejem1c
```

¿Cómo preguntar si se desea hacer alguna acción?

```
procedure ejem2 (archivo)
#
# script de ejemplo de como preguntar
#--------------------------------------
string archivo  {prompt="Archivo a procesar"}
struct *flist

begin
  struct line # linea a leer
  bool resp
  int dummy

  flist=archivo # asigna archivo a puntero

# lee hasta encontrar EOF e imprime
#
  while(fscan(flist,line) != EOF){
    print(line)
    print("Desea continuar?")
    dummy=scan(resp)
    if(!resp)break
  }
end
```

Otra opción para preguntar

```
procedure ejem2a (archivo)
#
# script de ejemplo de como preguntar
# aprovechando el Query Mode
#-------------------------------------
string archivo  {prompt="Archivo a procesar"}
bool resp   {yes,prompt="Desea continuar?",mode="q"}
struct *flist

begin
  struct line # linea a leer

  flist=archivo # asigna archivo a puntero

# lee hasta encontrar EOF e imprime
#
  while(fscan(flist,line) != EOF){
    print(line)
    if(!resp)break
  }
end
```

¿Cómo leer el cursor de imagen?

```
procedure ejem3 (imagen,archivo)
#
# script de ejemplo de como leer y
# guardar el cursor de imagen
#-------------------------------------
string imagen   {prompt="Imagen a procesar"}
string archivo   {prompt="Archivo de coord."}

begin
  struct coman
  string tecla
  real x,y
  int sis

  display(imagen,1)
  while(fscan(imcur,x,y,sis,coman) != EOF){
    tecla = substr(coman,1,1)
    if(tecla == "q")break
    print(x,y,coman,>> archivo)
  }
end
```