

¿Qué es un shell script?

- Es un archivo con una secuencia de comandos de determinado shell (en nuestro caso bash)
- Un script de bash contiene
 - $_{\circ}$ Comandos
 - o Manejo de variables (no es fuertemente tipado)
 - o Comentarios: líneas que comienzan con #
 - o Estructuras de control
 - if
 - case
 - for
 - whileDeclaración de funciones
 - Puede usar código definido en otros scripts (comando source)



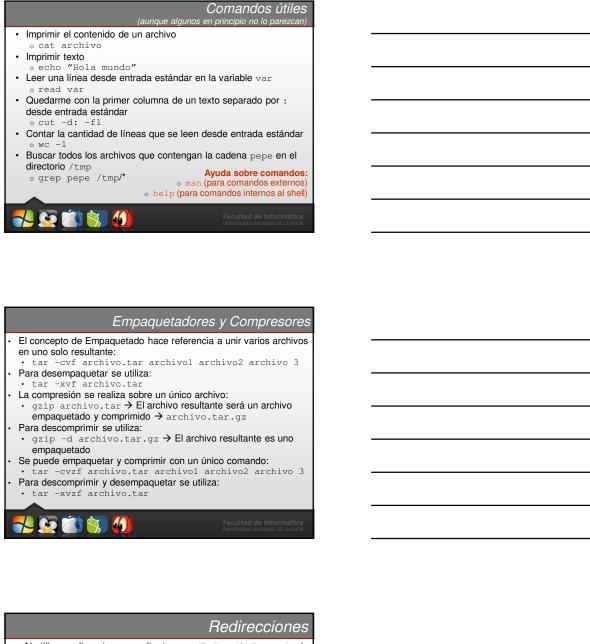
Facultad de Informática
UNIVERSIDAD NACIONAL DE LA PLATA

¿Qué es un shell script?

- En un shell script se puede hacer todo lo que se puede hacer de forma interactiva y viceversa
 - o echo, ls, cat, cd, if, while, etc...
- ¿Por qué escribir un shell script en vez de un programa en c/java/python/perl/ruby...?
 - o Es ideal para automatizar tareas que implican usar comandos
 - La ejecución de comandos es directa (con su nombre basta)
 - Es muy simple guardar el resultado de la ejecución de un comando
 - No requiere la instalación de ningún interprete o compilador (todo GNU/Linux tiene al menos un shell)
 - GNU/Linux expone gran parte del sistema como archivos, y es muy fácil manipular archivos desde un shell



Facultad de Informática



· Al utilizar redirecciones mediante > (Redirección Destructiva) o Si el archivo de destino no existe, se lo crea; o Si el archivo existe, se lo trunca; y se escribe la información en él. En cambio, al utilizar redirecciones mediante >> (Redireccion NO Destructiva) o Si el archivo de destino no existe, se lo crea; o Pero si el archivo existe, no se lo trunca; o y se agrega la información al final del mismo. o Funciona como un append. cd ls >> /tmp/lista.txt cd /tmp ls >> /tmp/lista.txt ¿Qué pasa si cambiamos >> por >?

Facultad de Informátic
UNIVERSIDAD NACIONAL DE LA PLAT

Entrada/Salida/Error estándar

- Todo programa en GNU/Linux tiene 3 archivos abiertos con un numero de descriptor
 - o entrada estándar (por defecto el teclado)
 - 1 salida estándar (por defecto la pantalla)
 - 2 error estándar (por defecto la pantalla)
- Es posible:
 - o desviar la salida de un comando a un archivo con >

ls > lista.txt

- o Volcar los errores a un archivo con 2>
 - -name foo 2> errores.txt
- o Redirigir la salida de un comando hacia otro mediante | (tuberías o pipes)

ls | grep .sh

Se pueden encadenar y combinar

ls | grep .sh > scripts.txt









Uso de Pipes

- El "|" nos permite comunicar dos procesos por medio de un pipe o tubería desde la shell.
- El pipe conecta stdout (salida estándar) del primer comando con la stdin (entrada estándar) del segundo comando.
- · Por ejemplo:
 - ls | more
 - Se ejecuta el comando 1s y la salida del mismo, es enviada como entrada del comando more
- Se pueden anidar tantos pipes como se deseen:
- ¿Cómo haríamos si quisiéramos contar la cantidad de usuarios del sistema que en su nombre de usuario aparece una letra a?

cat /etc/passwd | cut -d: -f1 | grep a | wc -l









¿Como creamos un shell script?

- · Creamos un archivo con cualquier editor de texto
- Opcionalmente, en la primer línea indicamos cual será el interprete del comando (en nuestro caso bash) → #!/bin/bash
- o Si no especificamos el intérprete explícitamente, se asume el intérprete por defecto del usuario que ejecuta el script. ¿Qué problemas puede traer esto?
- El resto del archivo se compone de líneas de comandos, tal cual como si los estuviéramos usando de forma interactiva.
- · Al guardar el script (por convención) utilizaremos .sh a modo de terminación de los archivos para indicar que son shell scripts.



Primer ejemplo - Hola mundo

- #!/bin/bash
- # Si la primer línea de mi script comienza #con la cadena #! se interpretará como el #path al intérprete a utilizar (podría ser #python, perl, php, etc...)
- # Ahora el script en sí echo "Hola mundo"

Ejemplo: 1holaMundo.sh



Ejecución del script anterior

Para ejecutar un script tenemos varias formas de hacerlo:

- Desde la línea de comandos podemos ejecutarlo pasándoselo al comando bash como argumento:
 - \$ bash miscript.sh
- Le podemos dar permisos de ejecución e invocarlo utilizando
 - su path relativo:
- \$./miscript.sh ...o mediante su path absoluto:
- \$ /home/yo/miscript.sh
- O podemos también ejecutarlo en modo debug (depuración)
 - \$ bash -x miscript.sh

¿Por que hay que ejecutarlo usando ruta relativa o completa?

→ ¿Qué ocurre si en /bin se agrega un script con el



Variables

- · bash no es fuertemente tipado.
 - o Las variables pueden cambiar su tipo durante la ejecución de nuestros scripts.
- · Las variables tienen básicamente dos tipos:
 - o String
 - Array
- · No precisan declaración, se las utiliza directamente.
- Las asignaciones tienen la forma (notar que no hay espacios entre el signo = y los operandos):

variable=valor

- Como en el resto de las situaciones referentes al shell, bash es case sensitive.
- Para referenciar su valor se utiliza el símbolo \$ y opcionalmente {}







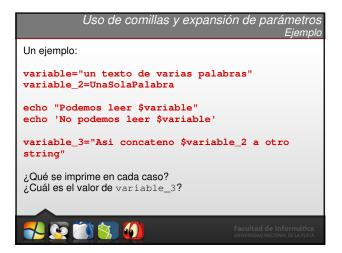
Variables: Arreglos Creación: arreglo_a=() # Se crea vacío arreglo_b=(1 2 3 5 8 13 21) # Inicializado Asignación de un valor en una posición concreta: arreglo_b[2]=spam Acceso a un valor del arreglo (En este caso las llaves no son opcionales): \${arreglo_b[2]} Acceso a todos los valores del arreglo (Es indistinto usar @ ó * como índice): \${arreglo[@]} # == \${arreglo[*]}



Variables: Arreglos (cont) Tamaño del arreglo: \${\#\arreglo[@]} \# == \\${\#\arreglo[*]}\$ Borrado de un elemento (no elimina la posición, solamente la deja vacía): unset arreglo[2] Los índices en los arreglos comienzan en 0



Uso de comillas y expansión de parámetros · En ocasiones, se pueden omitir las comillas al ingresar valores string. Por ejemplo: o si el string se compone de una sola palabra, o en el parámetro para el comando echo, o o cuando sabemos que el valor de una variable no tiene espacios o es un número. A la hora de utilizar las comillas, tenemos dos tipos de comillas que se comportan distinto: o Comillas dobles: los string delimitados por estas comillas son evaluados y se reemplazan en ellos expresiones o se expanden variables. o Comillas simples: Se toman como literales, no se efectúa ningún reemplazo en ellas. Ejemplo: nombre=Pepe echo "\$nombre" # imprime Pepe echo '\$nombre' # imprime \$nombre



Sustitución de comandos

- Permite utilizar la salida de un comando como si fuese una cadena de texto normal.
- Permite guardarlo en variables o utilizarlos directamente.
 Se la puede utilizar de dos formas, cada una con distintas realas:
 - o \$ (comando_valido)
 o `comando_valido`

Nota: La primer forma resulta más clara y posee reglas de anidamiento de comandos más sencillas.

Ejemplo:

arch="\$(ls)" # == arch="\ls\" == arch=\ls\
mis_archivos="\$(ls /home/\$(whoami))"



Facultad de Informática

Estructuras de control •Selección de alternativas: Selección: oDecisión: case \$variable in if [condition] "valor 1") then block block "valor 2") block ;; block ;; esac



Facultad de Informática

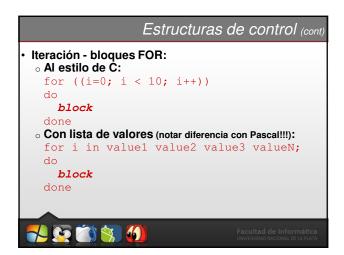
```
Menú de opciones:

o select variable in alternatival alternativa2
alternativa3
do
# en $variable está el valor elegido
block
done

Ejemplo:
select action in new exit
do
case $action in
"new")
echo "Selected option is NEW"
;;
"exit")
exit 0
exit 0
exit 0

i) new
exit 0
exit 0
exit 0
exit 0

facultad de Informática
duntacado Macoula of Largada
```







Estructuras de control: ejemplos	
<pre>if ["\$USER" == root] then echo "super user" else echo "Ud. es \$USER" fi</pre>	for archivo in \$(1s) do echo "- \$archivo" done for ((i=0; i<5; i++)) do echo \$i done
done Adicionalmente: break [n] corta la ejecución de n niveles de bucles continue [n] salta a la siguiente iteración del enésimo loop que contiene esta instrucción Facultad de Informática	

#!/bin/bash # Imprime los numeros del 1 al 5 # (no es un código para nada elegante) i=0 # true es un comando que siempre retorna 0 while true do let i++ # Incrementa i en 1 if [\$i -eq 6]; then break # Corta el loop (while) fi echo \$i done Facultad de Informática

```
i=0
while true
do
let i++
if [ $i -eq 6 ]; then
break # Corta el while
elif [ $i -eq 3 ]; then
continue # Salta una iteración
fi
echo $i
done

¿Qué hace el script anterior?

Facultad de Informática
UNIVERDAD MACINAL DE LA FLATA
```



Argumentos y valor de retorno • Los scripts pueden recibir argumentos en su invocación • Para accederlos, se utilizan variables especiales: • \$0 contiene la invocación al script • \$1, \$2, \$3, ... contienen cada uno de los argumentos • \$# contiene la cantidad de argumentos recibidos • \$* contiene la lista de todos los argumentos • \$? contiene en todo momento el valor de retorno del último comando ejecutado if [\$# -ne 2]; then exit 1 # Error else echo "Nombre: \$1, Apellido: \$2" fi exit 0 # Funcionó correctamente

Función exit

- Causa la terminación de un script
- Puede devolver cualquier valor entre 0 y 255:
 - El valor 0 indica que el script se ejecutó con éxito
 - Un valor distinto indica un código de error
- · Se puede consultar el exit status imprimiendo la variable \$?



Funciones

- Se pueden declarar de 2 formas o function nombre { block } o nombre() { block }
- Con la sentencia return se retorna un valor entre 0 y 255
- El valor retornado se puede evaluar con la variable
- Reciben argumentos en las variables \$1, \$2, etc.



Funciones: ejemplos

```
# recibe 2 argumentos y retorna:
  1 si el primero es el mayor
# 0 en caso contrario
mayor() {
   echo "Se van a comparar los valores $*"
  if [ $1 -gt $2 ]; then echo "$1 es el mayor"
    return 1
  echo "$2 es el mayor"
  return 0
mayor 5 6 # Invocación
echo $? # Imprime el resultado
```



Variables: alcance y visibilidad Las variables no inicializadas son reemplazadas por nulo o 0, según el contexto de evaluación Por defecto las variables son globales Una variable local a una función se define con local test() { local variable } Las variables de entorno son heredadas por los procesos hijos Para exponer una variable global a los procesos hijos se usa export variables global a los procesos hijos se usa export variables de entorno a variables de # comando # comando verá entre sus variables de # entorno a Variable_GLOBAL Facultad de Informática de la proceso de la para de