

Actividad 4: Introducción a la programación de los intérpretes de comandos

Melissa Matrecitos Avila

26 de Febrero de 2018

1 Introducción

El siguiente texto es el reporte de la actividad 4 del curso de Física Computacional 1, en la cual se trabajó con un interprete de comandos llamado Shell, el es una interfaz de usuario para acceder a los servicios de un sistema operativo, en este caso LINUX.

En esta actividad trabajamos las distintas formas de interactuar y hacer scripts (programas) con el Shell (/bin/sh) y el Again Shell (/bin/bash). Para llevarla a cabo se utilizaron los comandos cat, chmod, echo, grep, less, ls y wc, los cuales se explicaran en los resultados de la práctica.

Por último se realizó una síntesis sobre las notas de Steve Parker "Shell Script Tutorial", de los cuales se incluyen ejemplos de las secciones que allí vienen.

2 Síntesis

En esta sección se presentará la síntesis sobre las notas de Steve Parker "Shell Script Tutorial" con algunos ejemplos.

2.1 Introducción y Filosofía

Este tutorial está escrito para ayudar a las personas a comprender algunos de los conceptos básicos de la programación de scripts de shell. Hay una serie de factores que pueden incluirse en scripts de shell buenos, limpios y rápidos.

- Los criterios más importantes deben ser un diseño claro y legible.
- La segunda es evitar comandos innecesarios.

Un diseño claro hace la diferencia entre un guión de shell que aparece como "magia negra" y uno que se mantiene y comprende fácilmente.

2.2 Un Primer Script

Para el primer script se escribirá uno que diga "Hello World", esto se logra, primero que nada especificando por quien debe ser ejecutado el script y con el comando echo se escribe en la pantalla los argumentos que lo acompañan. Es importante saber que echo pondrá automáticamente un espacio entre los argumentos, para evitarlo, el argumento se debe escribir entre comillas, otra cosa importante es saber que el símbolo # es para agregar comentarios, excepto cuando es acompañado de ! ya que #! significa que lo que sigue debe ser interpretado por el shell de Bourne.

```
#!/bin/bash
#Comentario
echo Hello World

#!/bin/bash
#Comentario
echo"Hola    Mundo"
```

2.3 Variables Parte 1

Casi todos los lenguajes de programación existentes tienen el concepto de variables , un nombre simbólico para un trozo de memoria al que podemos asignar valores, leer y manipular sus contenidos. Tenga en cuenta que no debe haber espacios alrededor del signo "=", el cual tarta el comando como una asignación de variable.

Al shell no le importan los tipos de variables; pueden almacenar cadenas, enteros, números reales, cualquier cosa que te guste. Podemos establecer de manera interactiva nombres de variables usando el readcomando.

No es necesario declarar las variables en el shell Bourne, como sí lo hacen en lenguajes como C. Pero si intenta leer una variable no declarada, el resultado es la cadena vacía. No obtienes advertencias o errores.

```
#!/bin/bash
echo "Hola, ¿Cómo te llamas?"
read Nombre
echo "¿Cuál es tu color favorito, $Nombre?"
read Color
echo "El color favorito de $Nombre es $Color"

#!/bin/bash
echo "Hola, para saber el año en el que naciste escribe el año actual:"
read Actual
echo "¿Cuál será tu edad este año?"
read Edad
echo "El año en que naciste es"
expr $Actual-$Edad
```

2.4 Comodines

Los comodines no son muy útiles en los scripts de shell. Esta sección es realmente solo para mover y editar archivos en la terminal. Son más utilizados cuando hay ciclos.

```
#!/bin/bash
#Copiar archivos de a hacia b
cp / tmp / a / * / tmp / b /

#!/bin/bash
#Escribir lo que contiene la carpeta a
echo /tmp/a/*
```

2.5 Caracteres de Escape

La mayoría de los caracteres (*, ', etc) no se interpretan (es decir, que se toman literalmente) a menos de que sean colocados entre comillas dobles ("). Pero para los caracteres ", ', y *siendo interpretado como código a pesar de las comillas, para esto se utiliza el carácter de escape* el cual hace que el intérprete no los interprete, sino que los pase al comando que se esta utilizando.

```
#!/bin/bash
echo "Tienes que pagar \$ 10. "
```

```
#!/bin/bash
echo "Mi nombre es \"Melissa\" "
```

2.6 Ciclos

La mayoría de los lenguajes tienen el concepto de bucles: si queremos repetir una tarea veinte veces, no queremos tener que escribir el código veinte veces, con un ligero cambio cada vez. Como resultado, tenemos `for` y `while` bucles en el shell Bourne.

Los bucles "for" iteran a través de un conjunto de valores hasta que se agote la lista:

```
#!/bin/bash
for i in 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
do
    echo "Summer"
done
echo "What time is it ? Summertime!"
```

Los bucles "while" iteran hasta que se cumpla una condición de salida, éste tipo de bucles se dice que son más "divertidos" debido a que el usuario decide si va a seguir dentro o va a salir del ciclo:

```
#!/bin/bash
salida=Halo
while [ "$salida" != "Tschüss" ]
do
    echo "Escribe algo (Tschüss para salir)"
    read salida
    echo "Escribiste: $salida"
done
```

2.7 Prueba (test)

"[" es un enlace simbólico de test, solo para hacer que los programas shell sean más legibles. Esto significa que "test" es en realidad un programa, al igual que ls u otros programas. Test a menudo se invoca indirectamente a través de las instrucciones if y while. También es la razón por la que tendrás problemas si creas un programa llamado testy tratas de ejecutarlo, ya que se llamará a este intérprete de comandos en lugar de tu programa.

```
#!/bin/bash
echo -en "Please guess the magic number: "
read X
echo $X | grep "[^0-9]" > /dev/null 2>&1
if [ "$?" -eq "0" ]; then
    # If the grep found something other than 0-9
    # then it's not an integer.
    echo "Sorry, wanted a number"
else
    # The grep found only 0-9, so it's an integer.
    # We can safely do a test on it.
    if [ "$X" -eq "7" ]; then
        echo "You entered the magic number!"
    fi
fi
```

```
#!/bin/bash
X=0
while [ -n "$X" ]
do
    echo "Enter some text (RETURN to quit)"
    read X
    if [ -n "$X" ]; then
        echo "You said: $X"
    fi
done
```

2.8 Caso

La "case" declaración guarda pasar por un conjunto completo de if .. then .. else declaraciones. Pueden ser una herramienta muy útil y poderosa. A menudo se utilizan para analizar los parámetros pasados a un script de shell, entre otros usos.

```
#!/bin/bash
echo "Please talk to me ..."
while :
do
    read INPUT_STRING
    case $INPUT_STRING in
hello)
echo "Hello yourself!"
;;
bye)
echo "See you again!"
break
;;
*)
echo "Sorry, I don't understand"
;;
esac
done
echo
echo "That's all folks!"
```

2.9 Variables Parte 2

Ya hay un conjunto de variables establecidas para usted, y la mayoría de ellas no pueden tener valores asignados. Estos pueden contener información útil, que el script puede utilizar para conocer el entorno en el que se está ejecutando. El primer conjunto de variables que veremos son \$0 .. \$9 y \$#. La variable \$0 es el nombre base del programa como se lo llamó. \$1 .. \$9 son los primeros 9 parámetros adicionales con los que se invocó el script.

La variable @\$ es todos los parámetros \$1 .. cualquiera. La variable \$*, es similar, pero no conserva ningún espacio en blanco, y las comillas, por lo que "Archivo con espacios" se convierte en "Archivo" "con" "espacios". Como regla general, usa @\$y evita \$*.

\$# es la cantidad de parámetros con los que se invocó el script. Otra variable es interesante IFS. Este es el separador de campo interno . El valor predeterminado es SPACE TAB NEWLINE, pero si lo está cambiando, es más fácil tomar una copia Otra variable especial es \$?. Esto contiene el valor de salida del último comando de ejecución, esto ayuda a que los scripts sean robustos y más inteligentes.

```
#!/bin/bash
old_IFS="$IFS"
IFS=:
echo "Please input some data separated by colons ..."
read x y z
IFS=$old_IFS
echo "x is $x y is $y z is $z"

#!/bin/bash
/usr/local/bin/my-command
if [ "$?" -ne "0" ]; then
    echo "Sorry, we had a problem there!"
fi
```

2.10 Variables Parte 3

Como mencionamos en Variables - Parte I , las llaves alrededor de una variable evitan confusiones. Sin embargo, eso no es todo, estos brackets de lujo tienen otro uso mucho más poderoso. Podemos tratar con problemas de variables indefinidas o nulas.

```
#!/bin/bash
foo=sun
echo $fooshine      # $fooshine is undefined
echo ${foo}shine    # displays the word "sunshine"

#!/bin/bash
echo -en "What is your name [ 'whoami' ] "
read myname
if [ -z "$myname" ]; then
    myname='whoami'
fi
echo "Your name is : $myname"
```

2.11 Programas externos

Los programas externos a menudo se usan en scripts de shell; hay algunas órdenes internas (echo, which, y test son comúnmente incorporados), pero muchos comandos útiles son en realidad utilidades Unix, tales como tr, grep, ex, y cut. El backtick (`) también se asocia a menudo con comandos externos, se usa para indicar que el texto adjunto se debe ejecutar como un comando.

```
#!/bin/bash
grep "^${USER}:" /etc/passwd | cut -d: -f5

#!/bin/bash
MYNAME=$(grep "^${USER}:" /etc/passwd | cut -d: -f5)
echo $MYNAME
```

2.12 Funciones

Una característica que a menudo se pasa por alto de la programación de guiones de shell de Bourne es que puede escribir fácilmente funciones para usar en su secuencia de comandos. Esto generalmente se hace de una de dos maneras; con un script simple, la función simplemente se declara en el mismo archivo como se llama. Sin embargo, al escribir un conjunto de secuencias de comandos, a menudo es más fácil escribir una "biblioteca" de funciones útiles, y el origen de ese archivo al inicio de los otros scripts que utilizan las funciones. Una función puede devolver un valor en una de cuatro formas diferentes:

- Cambiar el estado de una variable o variables
- Use el exit comando para finalizar el script de shell
- Utilice el return comando para finalizar la función y devolver el valor proporcionado a la sección de llamada del script de shell
- echo output to stdout, que será capturado por la persona que lo llama al igual que `c='expr $ a + $ b'` es llamado.

La diferencia es que una función de shell no puede cambiar sus parámetros, aunque puede cambiar los parámetros globales.

```
#!/bin/bash

myfunc()
{
    echo "I was called as : $@"
    x=2
}

### Main script starts here

echo "Script was called with $@"
x=1
echo "x is $x"
myfunc 1 2 3
echo "x is $x"

#!/bin/bash

factorial()
{
    if [ "$1" -gt "1" ]; then
        i='expr $1 - 1'
        j='factorial $i'
        k='expr $1 \* $j'
        echo $k
    fi
}
```

```

    else
        echo 1
    fi
}

```

```

while :
do
    echo "Enter a number:"
    read x
    factorial $x
done

```

2.13 Consejos y Sugerencias

En esta sección, cómo lo dice su título, se nos brindan algunas sugerencias o tips sobre como mejorar los scripts, como los dos ejemplos que se muestran a continuación:

```

#!/bin/bash
steves = `grep -i steve / etc / passwd | cut -d: -f1`
echo "Todos los usuarios con la palabra" steve "en su passwd"
echo "Las entradas son:"
echo "$ steves" | tr ' ' '\ 012'

```

```

#!/bin/bash
NO_LINES = `wc -l archivo | awk '{print $ 1}' `

```

Se les dio mayor importancia a los que eran de tipo:

- CGI Scripting
- Códigos de salida y control de flujo
- Simple esperar reemplazo
- Usar la trampa para saber cuándo te han interrumpido, como CTRL-C, etc.
- Solución para la dicotomía 'echo -n' versus 'echo c'

2.14 Referencia Rápida

Se muestra una guía de referencia rápida sobre el significado de algunos de los comandos y códigos menos fáciles de adivinar de los scripts de shell. Por su naturaleza, también son bastante difíciles de encontrar usando los motores de búsqueda.

Mando	Descripción	Ejemplo
&	Ejecuta el comando anterior en el fondo	<code>ls &</code>
&&	Lógico Y	<code>if ["\$foo" -ge "0"] && ["\$foo" -le "9"]</code>
	O lógico	<code>if ["\$foo" -lt "0"] ["\$foo" -gt "9"] (no en el shell Bourne)</code>
^	Inicio de línea	<code>grep "^foo"</code>
ps	Fin de la línea	<code>grep "foo\$"</code>
=	Igualdad de cadenas (cf. -eq)	<code>if ["\$foo" = "bar"]</code>
!	Lógico NO	<code>if ["\$foo" != "bar"]</code>
\$\$	PID de la carcasa actual	<code>echo "my PID = \$\$"</code>
ps	PID del último comando de fondo	<code>ls & echo "PID of ls = \$!"</code>
ps	estado de salida del último comando	<code>ls ; echo "ls returned code \$?"</code>
\$0	Nombre del comando actual (como se llama)	<code>echo "I am \$0"</code>
\$1	Nombre del primer parámetro del comando actual	<code>echo "My first argument is \$1"</code>
\$9	Nombre del noveno parámetro del comando actual	<code>echo "My ninth argument is \$9"</code>
ps	Todos los parámetros del comando actual (conservando el espacio en blanco y las comillas)	<code>echo "My arguments are \$@"</code>
ps	Todos los parámetros del comando actual (no preserva el espacio en blanco y las comillas)	<code>echo "My arguments are \$*"</code>
-eq	Igualdad numérica	<code>if ["\$foo" -eq "9"]</code>
-nordeste	Numeric Inequality	<code>if ["\$foo" -ne "9"]</code>
-lt	Menos que	<code>if ["\$foo" -lt "9"]</code>
-le	Menor o igual	<code>if ["\$foo" -le "9"]</code>
-gt	Más grande que	<code>if ["\$foo" -gt "9"]</code>
-ge	Mayor que o igual	<code>if ["\$foo" -ge "9"]</code>
-z	La cadena es de longitud cero	<code>if [-z "\$foo"]</code>
-norte	La cadena no es de longitud cero	<code>if [-n "\$foo"]</code>
-Nuevo	Más nuevo que	<code>if ["\$file1" -nt "\$file2"]</code>
Testamento		
-re	Es un directorio	<code>if [-d /bin]</code>
-F	Es un archivo	<code>if [-f /bin/ls]</code>
-r	Es un archivo legible	<code>if [-r /bin/ls]</code>
-w	Es un archivo escribible	<code>if [-w /bin/ls]</code>
-X	Es un archivo ejecutable	<code>if [-x /bin/ls]</code>
(...)	Definición de función	<code>function myfunc() { echo hello }</code>

2.15 Shell Interactivo

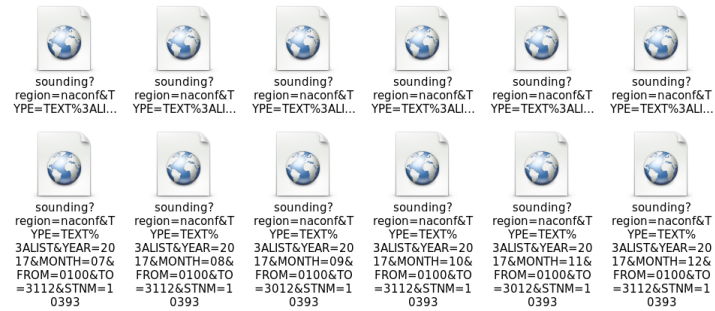
Aquí hay algunas sugerencias rápidas para usar el shell UNIX o Linux de forma interactiva.

- **bash:** tiene algunas herramientas de búsqueda de historia muy prácticas; las teclas de flecha hacia arriba y hacia abajo se desplazarán por el historial de comandos anteriores. Más útilmente, `Ctrl + r` hará una búsqueda inversa, haciendo coincidir cualquier parte de la línea de comando. Presione `ESC` y el comando seleccionado se pegará en el shell actual para que pueda editarlo según sea necesario.
- **ksh:** Puede hacer que ksh sea más útil agregando comandos de historial, ya sea en `vi` o `emacs`. Hay varias maneras de hacerlo, dependiendo de las circunstancias exactas: `"set -o vi"`, o `"ksh -o vi"`, o `"exec ksh -o vi"` (donde `"vi"` podría reemplazarse por `"emacs"` si prefiere el modo `emacs`). Además si golpeas `ESC` y `k`, al golpear repetidamente `k`, retrocedes en el historial de comandos. Puede usar los comandos `vi command-mode` y `entry-mode` para editar los comandos.

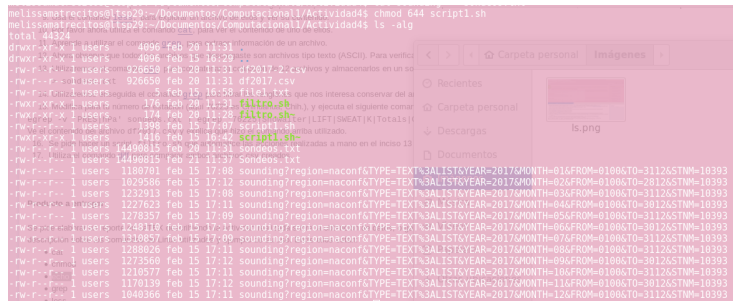
3 Resultados de la práctica

A continuación se describirá la practica realizada: Después de crear la carpeta "Actividad 4" en la cual se guardarían todos los archivos utilizados, se decargó el archivo proporcionado por el profesor llamado "script1.sh", en el cual se hicieron

los cambios pertinentes para descargar los datos de la estación 10393, correspondiente a Linderburg en Alemania. Después con el comando **"chmod"** se creó el ejecutable de dicho script que se confirmó utilizando el comando **"ls -alg"**, que después se ejecutó en la terminal. Los 12 archivos descargados fueron:

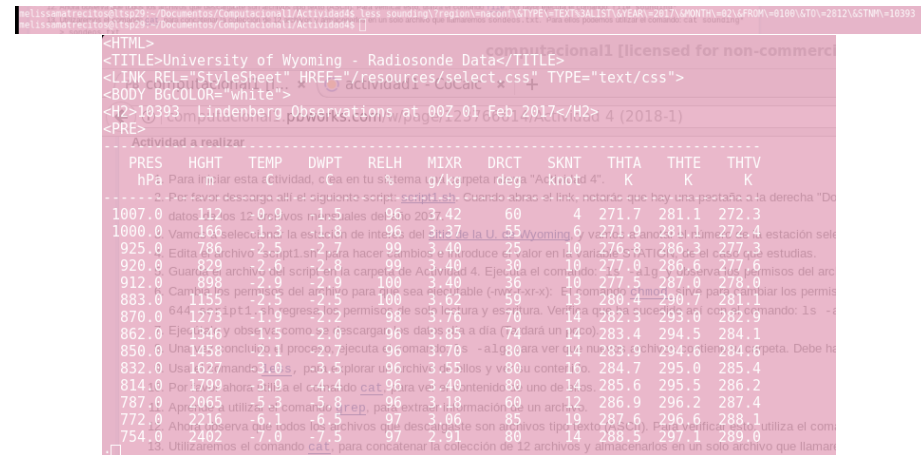


Archivos descargados por el script1.sh



Comandos "chmod" y "ls-alg" en la terminal

Posteriormente con el comando **"less"** se exploró el archivo correspondiente al mes de febrero, este mostró el archivo en la terminal, ubicando el puntero al inicio del archivo:



Comando less

Siguiendo con la actividad se utilizó el comando **"cat"** par explorar un archivo, a diferencia del less, este te mostraba rápidamente el archivo y dejaba el puntero al final del archivo:

```
hellissam@treccitos01isp29:~/Documentos/Computacional1/Actividad4$ cat sounding7region/wacont1.txt
<FORM>
<INPUT CLASS="button" TYPE="button" VALUE="Close this window"
onClick="window.close();"> actividad1 - CoCalc
<INPUT CLASS="button" TYPE="button" VALUE="Select another map"
onClick="window.blur();">
</FORM>
<HR SIZE="1">
<I> descarga allí el siguiente script: script1.sh. Cuando abras el link, notarás que hay una pestaña a la derecha "Downlo
</I>Interested in graduate studies in atmospheric science?
Check out our program at the de Interés del sitio de la U. de Wyoming, y vamos a anotar el número de la estación seleccio
<a href="http://www.uwyo.edu/atsc/howtoapply/"> de el valor en la variable STATION, de el caso que estudias.
target= topUniversity of Wyoming de Actividad 4. Ejecuta el comando: ls -alig y observa los permisos del archivo
</a></I>
<HR SIZE="1"><FONT SIZE="1">
<I> Continúa leyendo el archivo para que sea ejecutable (-rwx-r-xr-x). El comando chmod sirve para cambiar los permisos d
</I>Continúa leyendo el archivo para que sea ejecutable (-rwx-r-xr-x). El comando chmod sirve para cambiar los permisos d
Questions about the weather data provided by this site can be
addressed to <A HREF="mailto:tdoolan@uwyo.edu">tdoolan@uwyo.edu (redará un poco)
Larry Doolan (tdoolan@uwyo.edu)</A></FONT>
<HR SIZE="1">
<I> Ejecutando less, para explorar un archivo de ellos y ver su contenido.
</I>
<SCRIPT TYPE="text/javascript">
<!-- 11. Aprende a utilizar el comando grep para extraer información de un archivo.
window.focus();
// --> Utilizaremos el comando cat para concatenar la colección de 12 archivos y almacenarlos en un solo archivo que llamaremos
sondeos.txt
</SCRIPT>
</BODY>
</HTML>
hellissam@treccitos01isp29:~/Documentos/Computacional1/Actividad4$
```

Comando cat

Otro comando utilizado fue el **"echo"**, el cual sirvió para escribir texto en un archivo, en este caso el archivo "file1":

```
hellissam@treccitos01isp29:~/Documentos/Computacional1/Actividad4$ echo "sondeos.txt: Para leer podemos utilizar el comando: cat 'sounding'"
14450 134280 1029586 sounding7region/wacont1.txt
hellissam@treccitos01isp29:~/Documentos/Computacional1/Actividad4$
```

Comando echo

También se utilizó el comando **"wc"**, con este contamos las palabras de un archivo:

```
hellissam@treccitos01isp29:~/Documentos/Computacional1/Actividad4$ wc sounding7region/wacont1.txt
14450 134280 1029586 sounding7region/wacont1.txt
hellissam@treccitos01isp29:~/Documentos/Computacional1/Actividad4$
```

Comando wc

Después se hizo uso del comando `["grep"]`, el cual sirve para extraer información específica de los archivos. También se utilizó el comando **"file"** para para verificar que los archivos descargados fueran tipo ASCII. Posteriormente se utilizó la instrucción:

```
cat sounding* > sondeos.txt
```

para almacenar los 12 archivos descargados en uno solo de nombre sondeos, aquí el redirector **">"** sirvió para enviar los 12 archivos al archivo nuevo, siguiendo con la practica se volvió a utilizar comando `grep`, luego se utilizó la siguiente entrada:











```
egrep -v 'PRES|hPa' sondeos.txt | egrep '10393|Showalter|LIFT|SWEAT|K|Totals|CAPE|CINS|LFCT
```

[illegible]

```
melissamatrecitos@ltsp29:~/Documentos$ cd Computacional1
melissamatrecitos@ltsp29:~/Documentos/Computacional1$ cd Actividad4
melissamatrecitos@ltsp29:~/Documentos/Computacional1/Actividad4$ emacs filtro.sh
melissamatrecitos@ltsp29:~/Documentos/Computacional1/Actividad4$ chmod 755 filtro.sh
melissamatrecitos@ltsp29:~/Documentos/Computacional1/Actividad4$ ./filtro.sh
melissamatrecitos@ltsp29:~/Documentos/Computacional1/Actividad4$ diff df2017-2.csv
melissamatrecitos@ltsp29:~/Documentos/Computacional1/Actividad4$
```

emacs@ltsp29.example.com

File Edit Options Buffers Tools Sh-Script Help

     Save  Undo    

```
#!/bin/bash
cat sounding* > sondeos.txt
egrep -v 'PRES|hPa' sondeos.txt | egrep '10393|Showalter|LIFT|SWEAT|K|Totals|CAP|
E|CINS|LFCT|CAPV|Temp|Pres|thick|Precip' > df2017-2.csv
```

```
melissamatrecitos@lisp29: ~/Documentos/Computacional/Actividad4$ diff df2017.csv df2017-2.csv
melissamatrecitos@lisp29: ~/Documentos/Computacional/Actividad4$
```

Al inicio de la actividad me pareció muy complicado entender que era lo que estábamos haciendo, pero conforme fui avanzando las dudas se resolvieron, descubriendo así muchas herramientas que ayudan en la optimización de procesos, sobre todo aquellos donde se tienen que trabajar con muchos datos.

- Shell (computing). (2018). En.wikipedia.org. Recuperado el 20 Febrero 2018, de [https://en.wikipedia.org/wiki/Shell_\(computing\)](https://en.wikipedia.org/wiki/Shell_(computing))
- Parker, S. (2017). Shell Scripting Tutorial. Shellsript.sh. Recuperado el 21 Febrero 2018, de <https://www.shellsript.sh/index.html>
- Atmospheric Soundings. (2018). Weather.uwyo.edu. Recuperado el 21 Febrero 2018, de <http://weather.uwyo.edu/upperair/sounding.html>

6 Apéndice

1. ¿Qué fue lo que más te llamó la atención en esta actividad?
La manera tan sencilla en que se pueden automatizar procesos que manejan gran cantidad de datos ya que solo conocía comandos muy básicos.
2. ¿Qué consideras que aprendiste?
Como automatizar procesos por medio de scripts, así como los comandos que estos utilizan.
3. ¿Cuáles fueron las cosas que más se te dificultaron?
Al inicio batallé mucho para saber que era lo que estaba haciendo, ya que no sabía para que eran los comandos. Creo que esto se debe a que un nuevo "lenguaje".
4. ¿Cómo se podría mejorar en esta actividad?
Me hubiera gustado una introducción teórica a lo que íbamos hacer, para no estar tan perdida y así poder agilizar el desarrollo de la actividad.
5. ¿En general, cómo te sentiste al realizar en esta actividad?
Al inicio estaba muy confundida, pero al final me gustó mucho todo lo que hicimos.