Introducción Shell-UNIX y Programación en BASH

Semana Internacional de la Ciencia

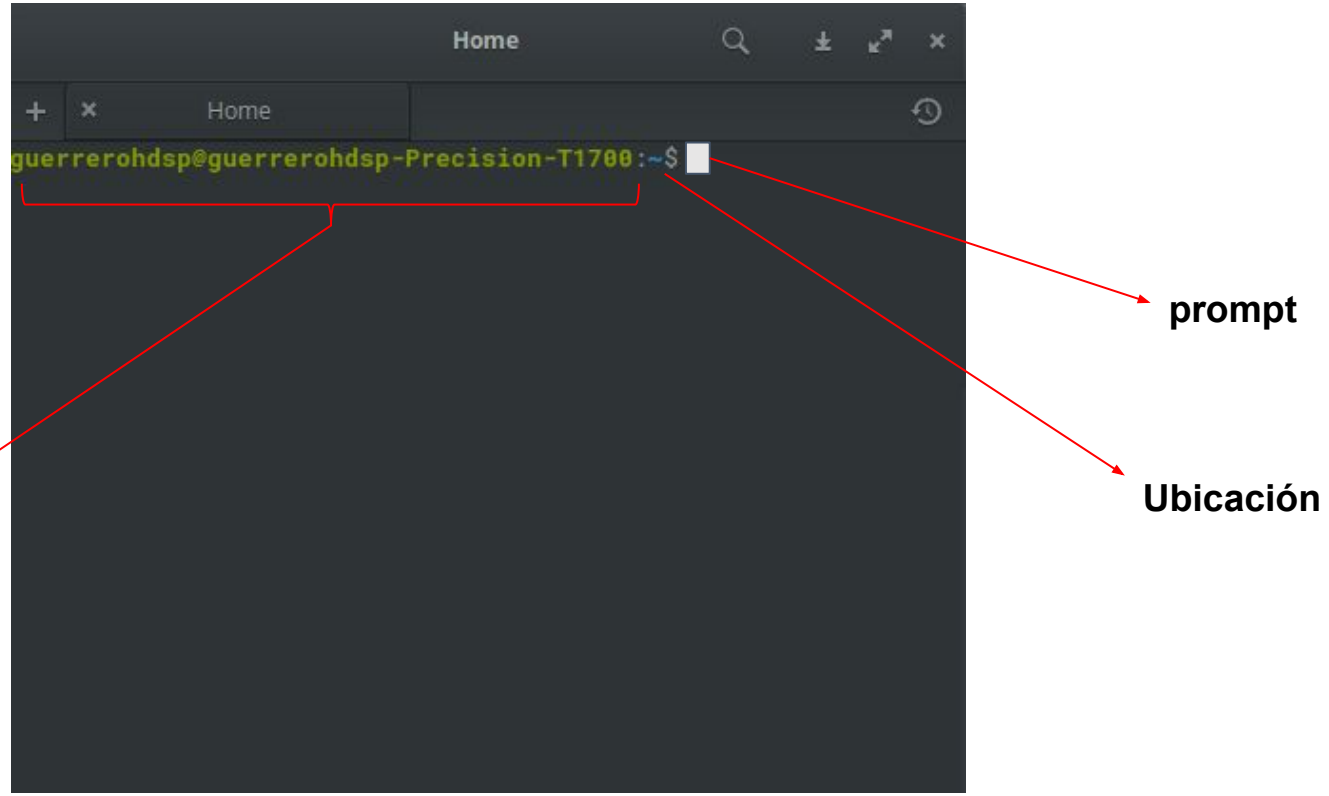
Facultad de Ciencias
Escuela de Física

19 de septiembre de 2018

Estructura del Shell

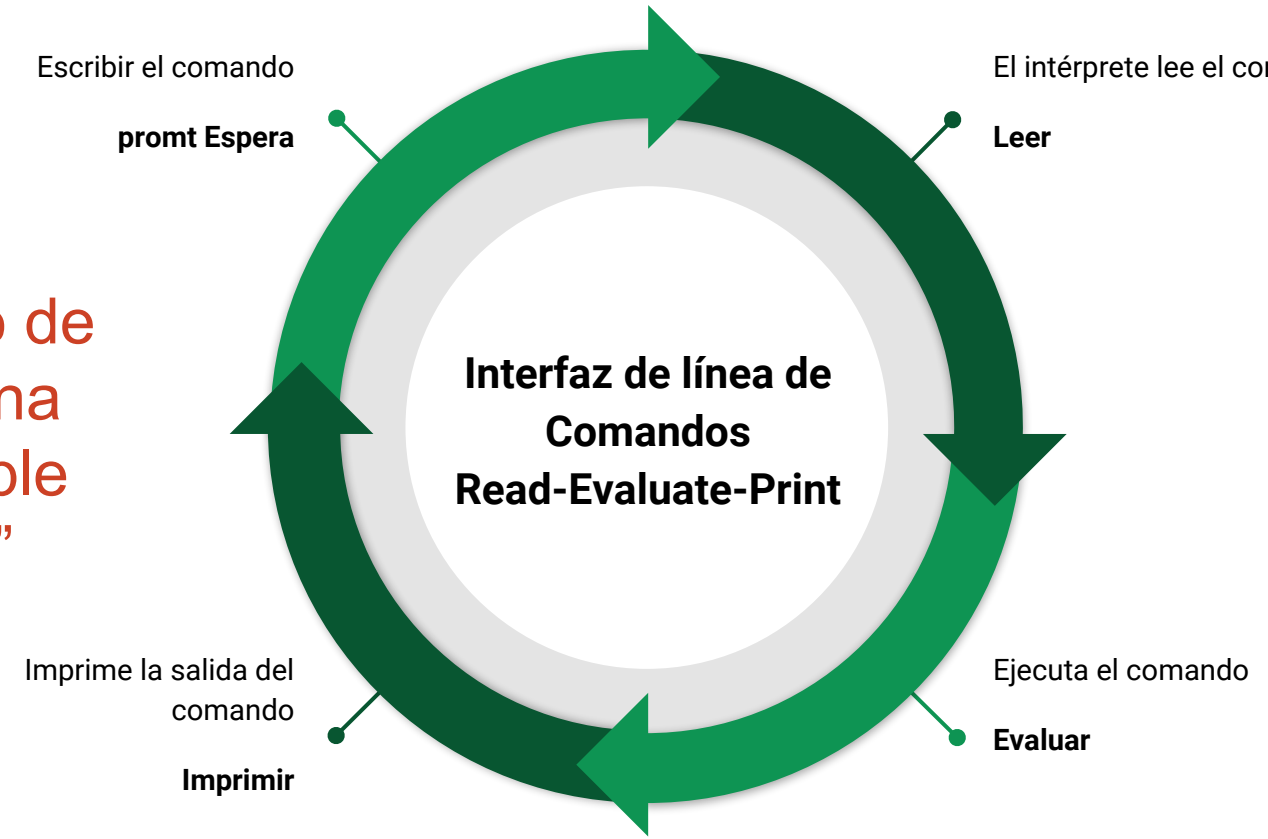
Abrir Terminal

Terminal



Presentando el Shell

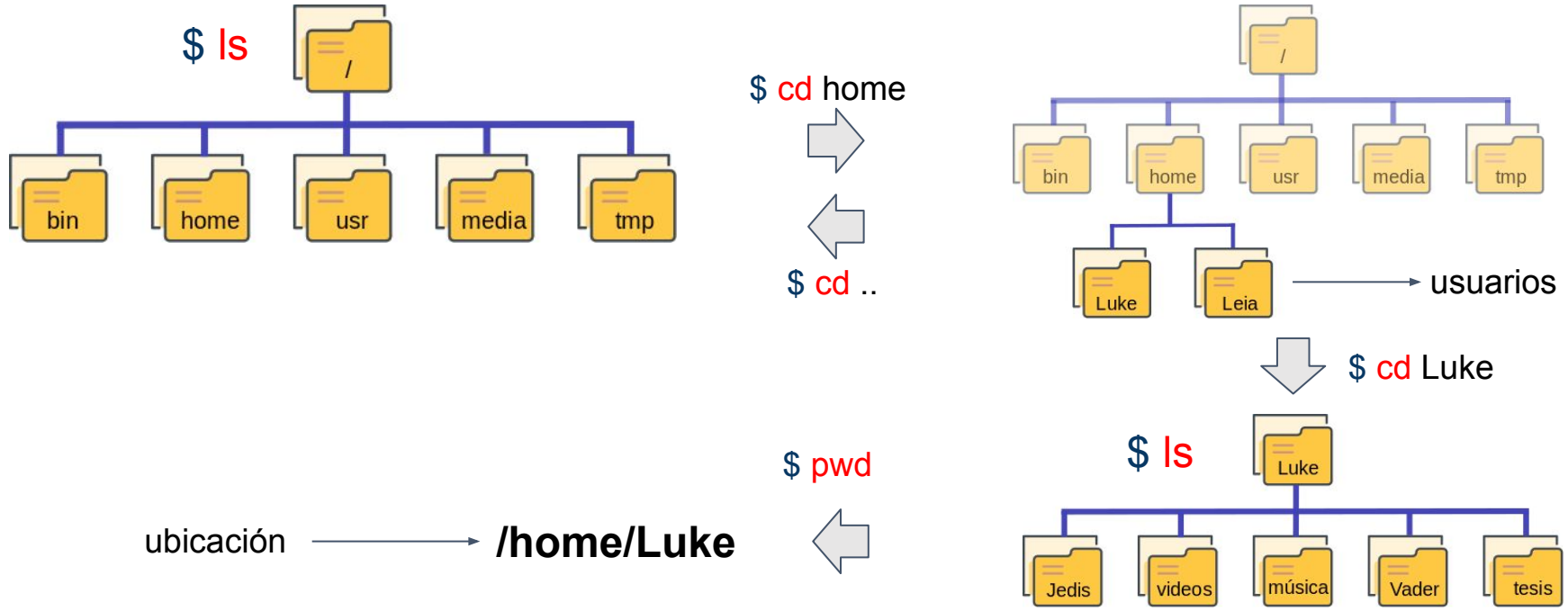
"Un vocabulario de comandos y una gramática simple para usarlos"



¿Por qué usarlo?

Sistemas de archivos

Comandos: pwd, ls, cd, cd..



IDEAS... bashrc

$ gedit ~/.bashrc

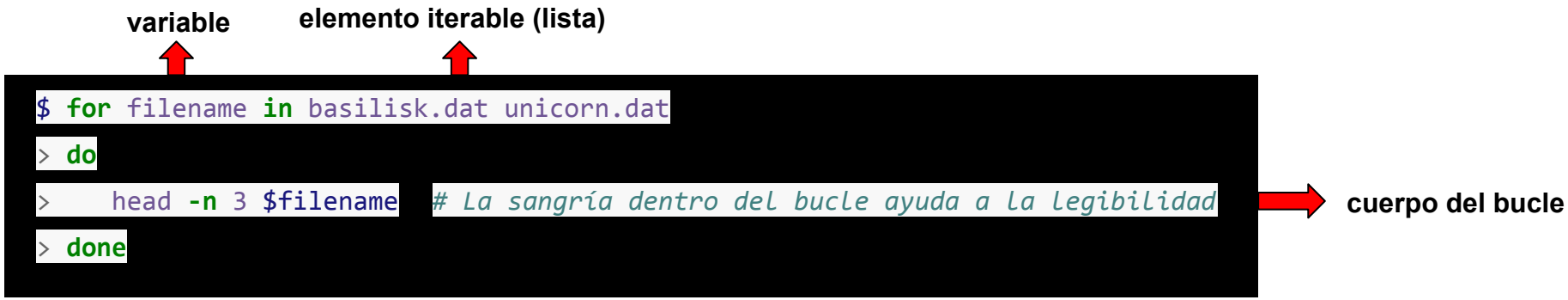
Archivos y Directorios

Tuberías y Filtros (“*Pipes and Filters*”)

Bucles

Los bucles (loops en inglés) son fundamentales para mejorar la productividad a través de la automatización, debido a que nos permiten ejecutar comandos de forma repetitiva.

- Sintaxis de un bucle `for`



Nota:

- `$` - se refiere al prompt, pero también se utiliza para pedir que la terminal obtenga el valor de una variable.
- `>` - se refiere al prompt dentro del bucle, pero también se utiliza para redirigir la salida de un comando.
- `;` - se utiliza para separar dos comando escritos en una sola línea.

EJEMPLO 1

En este ejemplo, usaremos el directorio creatures que sólo tiene dos archivos de ejemplo, pero los principios se pueden aplicar a muchos más archivos a la vez. Nos gustaría modificar estos archivos, pero también guardar una versión de los archivos originales, nombrando las copias `original-basilisk.dat` y `original-unicorn.dat`.

No se puede usar:

```
$ cp *.dat original-*.dat
```

Puesto que se expandiría a:

```
$ cp basilisk.dat unicorn.dat original-*.dat
```

Esto no respalda nuestros archivos, en su lugar obtenemos un error:

```
cp: target `original-*.dat' is not a directory
```

En cambio, podemos usar un **bucle** para ejecutar una operación a la vez sobre cada cosa en una lista:

```
$ for filename in *.dat
> do
>     cp $filename original-$filename
> done
```

Este bucle ejecuta el comando `cp` una vez para cada nombre de archivo:

- `cp basilisk.dat original-basilisk.dat`
- `cp unicorn.dat original-unicorn.dat`

EJEMPLO 2

Aquí un ejemplo sencillo que muestra las tres primeras líneas de cada archivo de una sola vez:

```
$ for filename in basilisk.dat unicorn.dat
> do
>     head -n 3 $filename
> done
```

```
COMMON NAME: basilisk
CLASSIFICATION: basiliscus vulgaris
UPDATED: 1745-05-02
COMMON NAME: unicorn
CLASSIFICATION: equus monoceros
UPDATED: 1738-11-24
```

Teniendo en cuenta el bucle anterior, hemos llamado a la variable en este bucle `filename` con el fin de hacer su propósito más claro para los lectores humanos. A la terminal no le importa el nombre de la variable; si escribimos este bucle como:

```
$ for x in basilisk.dat unicorn.dat
> do
>     head -n 3 $x
> done
```

```
$ for temperature in basilisk.dat unicorn.dat
> do
>     head -n 3 $temperature
> done
```

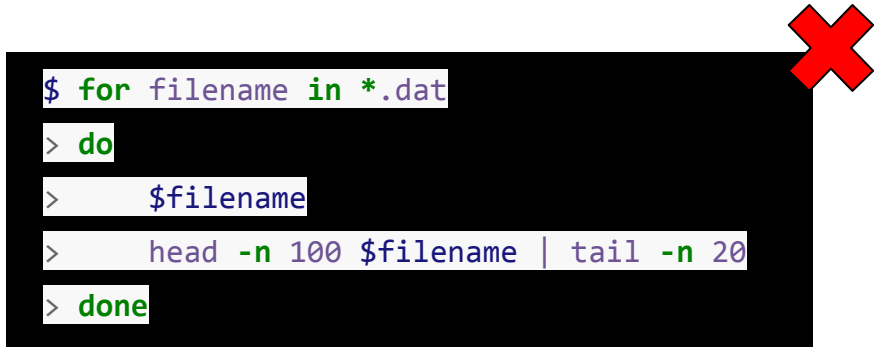
Nota:
Evitar el uso nombres sin sentido `x` o nombres engañosos `temperature`.

EJEMPLO 3

He aquí un bucle un poco más complicado:

```
$ for filename in *.dat
> do
>     echo $filename
>     head -n 100 $filename | tail -n 20
> done
```

En este caso, ya que la terminal expande `$filename` para que sea el nombre de un archivo, `echo $filename` sólo imprime el nombre del archivo `hello there`. Ten en cuenta que no podemos escribir esto como:



La primera vez a través del bucle, cuando `$filename` se expande a `basilisk.dat`, la terminal intentará ejecutar `basilisk.dat` como un programa.

PIPELINE DE NELLE: PROCESANDO ARCHIVOS

Nelle ahora está lista para procesar sus archivos de datos. Dado que todavía está aprendiendo cómo utilizar la terminal, decide construir los comandos requeridos en etapas. Su primer paso es asegurarse de que puede seleccionar los archivos correctos (recuerda, aquellos cuyos nombres terminan en 'A' o 'B', en lugar de 'Z'). Posicionada en su directorio home, Nelle teclea:

```
$ cd north-pacific-gyre/2012-07-03
$ for datafile in NENE*[AB].txt
> do
>     echo $datafile
> done
```

```
NENE01729A.txt
NENE01729B.txt
NENE01736A.txt
...
NENE02043A.txt
NENE02043B.txt
```

Su siguiente paso es decidir cómo llamar a los archivos que creará el programa de análisis `goostats`. Prefijar el nombre de cada archivo de entrada con "stats" parece simple, así que modifica su bucle para hacer eso:

```
$ for datafile in NENE*[AB].txt
> do
>     echo $datafile stats-$datafile
> done
```

```
NENE01729A.txt stats-NENE01729A.txt
NENE01729B.txt stats-NENE01729B.txt
NENE01736A.txt stats-NENE01736A.txt
...
NENE02043A.txt stats-NENE02043A.txt
NENE02043B.txt stats-NENE02043B.txt
```

REDIRECCIONAMIENTO DE LA SALIDA

```
for alkanes in *.pdb
do
    echo $alkanes
    cat $alkanes > alkanes.pdb
done
```

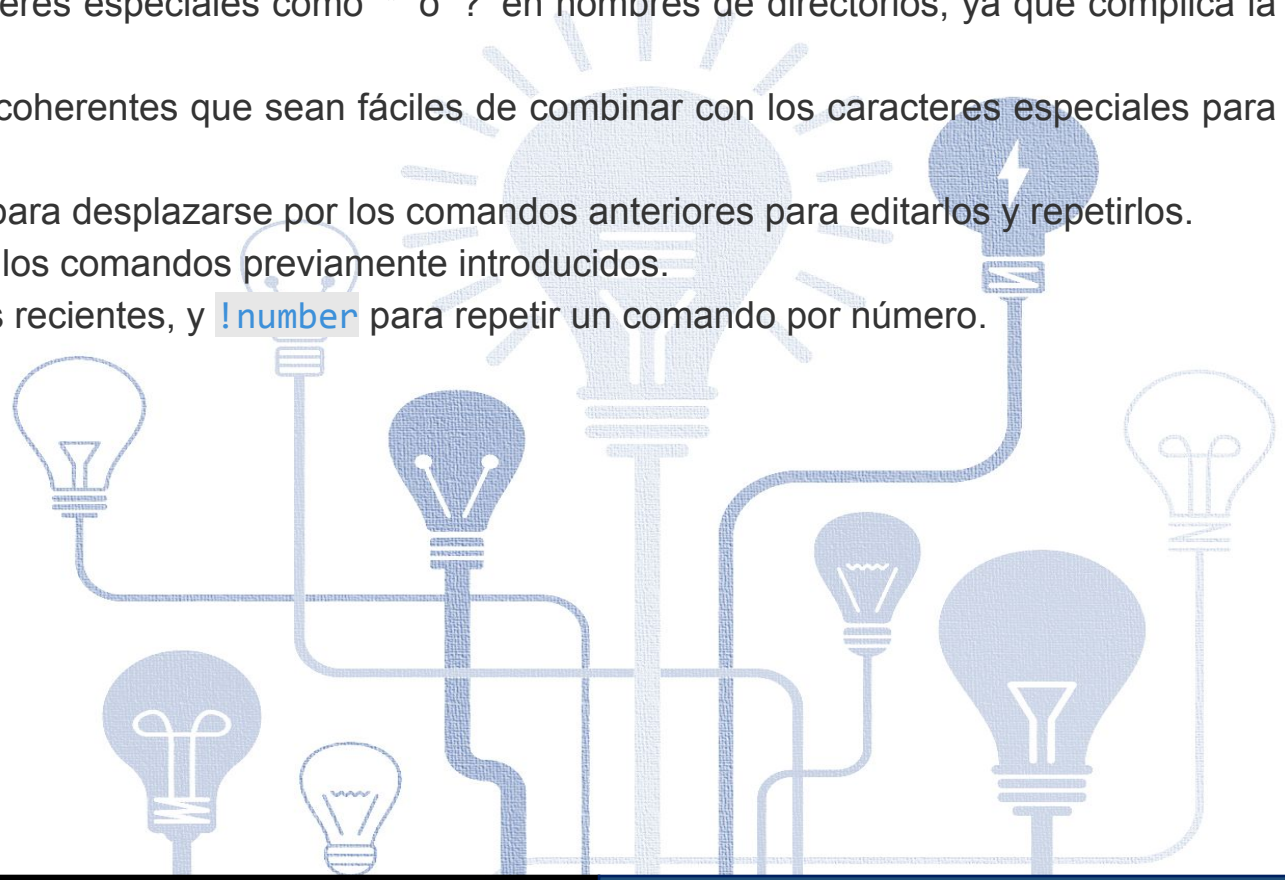
El texto de cada archivo se escribe (uno a la vez) en `alkanes.pdb`. Sin embargo, el archivo se sobrescribe en cada iteración del bucle.

```
for datafile in *.pdb
do
   cat $datafile >> all.pdb
done
```

`>>` concatena en un archivo, en lugar de sobrescribirlo con la salida del comando. Dado que la salida del comando `cat` ha sido redirigida, nada se imprime en pantalla.

RESUMEN: BUCLES

- Un bucle `for` repite comandos una vez para cada elemento de una lista.
- Cada bucle `for` necesita una variable para referirse al elemento en el que está trabajando actualmente.
- Uso de `$name` para expandir una variable (es decir, obtener su valor). También se puede usar `${name}`.
- No utilizar espacios, comillas o caracteres especiales como '*' o '?' en nombres de directorios, ya que complica la expansión de variables.
- Proporcionar a los archivos nombres coherentes que sean fáciles de combinar con los caracteres especiales para facilitar la selección de los bucles.
- Utilizar la tecla de flecha hacia arriba para desplazarse por los comandos anteriores para editarlos y repetirlos.
- Usar `Ctrl-R` para buscar a través de los comandos previamente introducidos.
- Usar `history` para mostrar comandos recientes, y `!number` para repetir un comando por número.



Scripts de Shell

Vamos a tomar los comandos que repetimos con frecuencia y los vamos a guardar en archivos, de modo que, podemos volver a ejecutar todas esas operaciones escribiendo un sólo comando. Por razones históricas, a un conjunto de comandos guardados en un archivo se le llama un **script** de la terminal.

Comencemos por volver a `molecules/` creando un nuevo archivo, `middle.sh`, que se convertirá en nuestro **script** de la terminal:

```
$ cd molecules
$ gedit middle.sh
```

Podemos usar el editor de texto para editar directamente el archivo - simplemente insertaremos la siguiente línea:

```
head -n 15 octane.pdb | tail -n 5
```

Una vez que hayamos guardado el archivo, podemos pedirle a la terminal que ejecute los comandos que contiene. Nuestra terminal se llama `bash`, por lo que ejecutamos el siguiente comando:

```
$ bash middle.sh
```

```
ATOM      9  H           1      -4.502    0.681    0.785  1.00  0.00
ATOM     10  H           1      -5.254   -0.243   -0.537  1.00  0.00
ATOM     11  H           1      -4.357    1.252   -0.895  1.00  0.00
ATOM     12  H           1      -3.009   -0.741   -1.467  1.00  0.00
ATOM     13  H           1      -3.172   -1.337    0.206  1.00  0.00
```

GENERALIZACIÓN DE SCRIPTS

¿Qué pasa si queremos seleccionar líneas de un archivo arbitrario? Podríamos editar middle.sh cada vez para cambiar el nombre de archivo, pero eso probablemente llevaría más tiempo que simplemente volver a escribir el comando. En cambio, editemos middle.sh y hagamos que sea más versátil:

```
$ gedit middle.sh
```

Ahora, dentro de "gedit", reemplaza el texto `octane.pdb` con la variable especial denominada `$1`:

```
head -n 15 "$1" | tail -n 5
```

Dentro de un **script** de la terminal,`$1` significa "el primer nombre de archivo (u otro parámetro) en la línea de comandos". Ahora podemos ejecutar nuestro **script** de esta manera:

```
$ bash middle.sh octane.pdb
```

```
ATOM      9  H           1      -4.502    0.681    0.785  1.00  0.00
ATOM     10  H           1      -5.254   -0.243   -0.537  1.00  0.00
ATOM     11  H           1      -4.357    1.252   -0.895  1.00  0.00
ATOM     12  H           1      -3.009   -0.741   -1.467  1.00  0.00
ATOM     13  H           1      -3.172   -1.337    0.206  1.00  0.00
```

```
$ bash middle.sh pentane.pdb
```

```
ATOM      9  H           1       1.324    0.350   -1.332  1.00  0.00
ATOM     10  H           1       1.271    1.378    0.122  1.00  0.00
ATOM     11  H           1      -0.074   -0.384    1.288  1.00  0.00
ATOM     12  H           1      -0.048   -1.362   -0.205  1.00  0.00
ATOM     13  H           1      -1.183    0.500   -1.412  1.00  0.00
```

GENERALIZACIÓN DE SCRIPTS

Aún así necesitamos editar `middle.sh` cada vez que queramos ajustar el rango de líneas. Vamos a arreglar esto usando las variables especiales `$2` y `$3` para el número de líneas que se pasarán respectivamente a `head` y `tail`:

```
$ gedit middle.sh
```

```
head -n "$2" "$1" | tail -n "$3"
```

Ahora podemos ejecutar:

```
$ bash middle.sh pentane.pdb 15 5
```

```
ATOM      9  H           1       1.324   0.350  -1.332  1.00  0.00
ATOM     10  H           1       1.271   1.378   0.122  1.00  0.00
ATOM     11  H           1      -0.074  -0.384   1.288  1.00  0.00
ATOM     12  H           1      -0.048  -1.362  -0.205  1.00  0.00
ATOM     13  H           1      -1.183   0.500  -1.412  1.00  0.00
```

```
$ bash middle.sh pentane.pdb 20 5
```

```
ATOM     14  H           1      -1.259   1.420   0.112  1.00  0.00
ATOM     15  H           1      -2.608  -0.407   1.130  1.00  0.00
ATOM     16  H           1      -2.540  -1.303  -0.404  1.00  0.00
ATOM     17  H           1      -3.393   0.254  -0.321  1.00  0.00
TER      18              1
```

PIPELINE DE NELLE: CREANDO UN SCRIPT

El supervisor de Nelle insistió en que todos sus análisis deben ser reproducibles. Nelle se da cuenta que debería haber proporcionado un par de parámetros adicionales a `goostats` cuando procesó sus archivos. La forma más fácil de capturar todos los pasos es en un **script**. Ella ejecuta el editor y escribe lo siguiente:

```
# Calculate reduced stats for data files.
for datafile in "$@"
do
    echo $datafile
    bash goostats $datafile stats-$datafile
done
```

Guarda esto en un archivo llamado `do-stats.sh` para que ahora pueda volver a hacer la primera etapa de su análisis escribiendo:

```
$ bash do-stats.sh NENE*[AB].txt
```

Por otra parte, el script podría haberlo escrito así:

```
# Calculate stats for Site A and Site B data
files.
for datafile in NENE*[AB].txt
do
    echo $datafile
    bash goostats $datafile stats-$datafile
done
```

DEPURACIÓN (DEBUGGING) DE SCRIPTS

Supongamos que se ha guardado el siguiente **script** en un archivo denominado `do-errors.sh` en el directorio `north-pacific-gyre/2012-07-03` de Nelle:

```
# Calcular las estadísticas de los archivos de datos.
for datafile in "$@"
do
    echo $datafile; bash goostats $datafile stats-$datafile
done
```

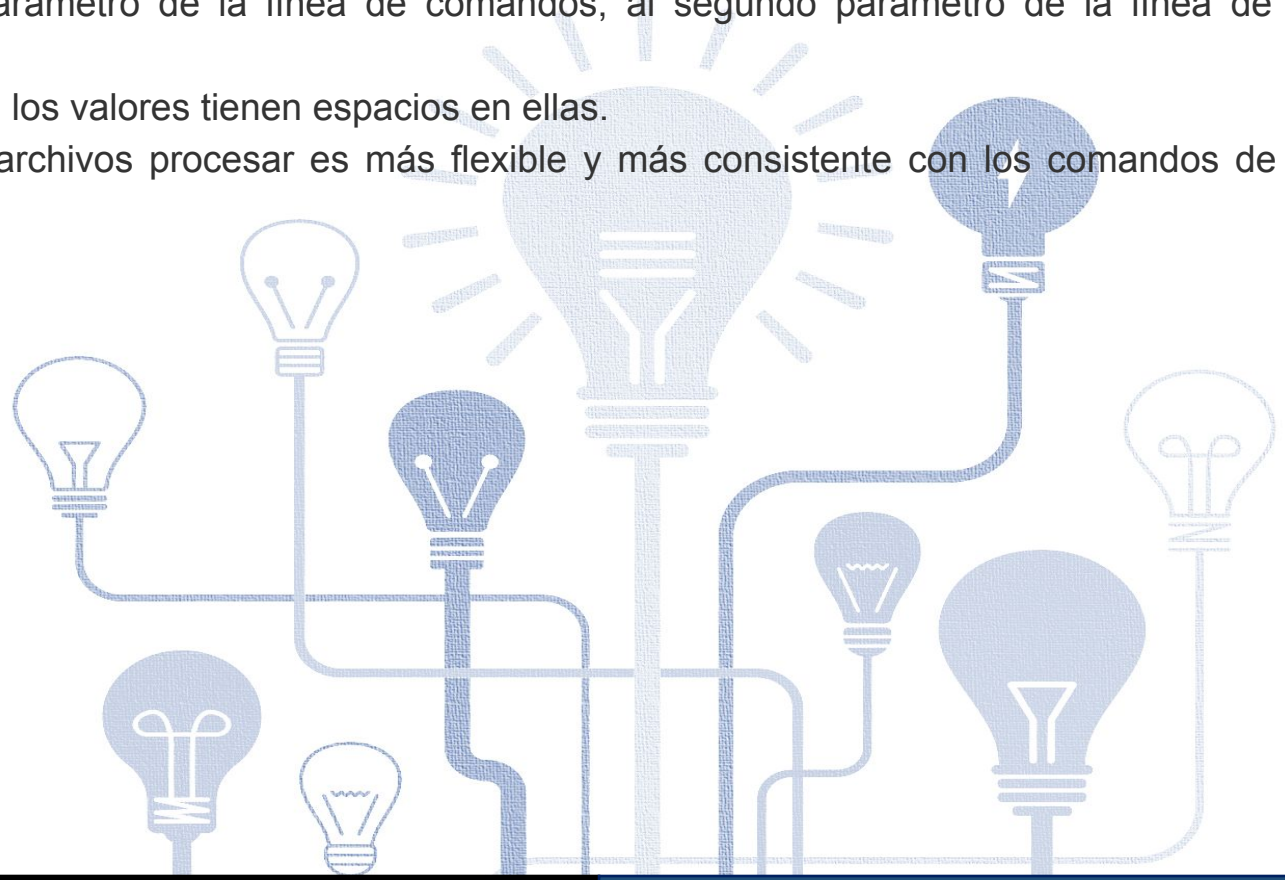
Ahora, ejecuta el **script** utilizando la opción `-x`

```
$ bash -x do-errors.sh NENE*[AB].txt
```

Nota:
El indicador `-x` hace que `bash` se ejecute en modo de depuración.

RESUMEN: SCRIPTS

- Guardar comandos en archivos (normalmente llamados **scripts** de la terminal) para su reutilización.
- `bash filename` ejecuta los comandos guardados en un archivo.
- `$@`se refiere a todos los parámetros de la línea de comandos de un **script** de la terminal.
- `$1`, `$2`, etc., se refieren al primer parámetro de la línea de comandos, al segundo parámetro de la línea de comandos, etc.
- Coloque las variables entre comillas si los valores tienen espacios en ellas.
- Dejar que los usuarios decidan qué archivos procesar es más flexible y más consistente con los comandos de Unix.



Encontrando archivos: grep

El comando `grep` encuentra e imprime líneas en archivos que coinciden con un patrón. Para nuestros ejemplos, usaremos un archivo que contenga tres haikus publicados en 1998 en la revista *Salon*. Para este conjunto de ejemplos, Vamos a estar trabajando en el subdirectorio “writing”:

```
$ cd
$ cd writing
$ cat haiku.txt
```

```
The Tao that is seen
Is not the true Tao, until
You bring fresh toner.

With searching comes loss
and the presence of absence:
"My Thesis" not found.

Yesterday it worked
Today it is not working
Software is like that.
```

EJEMPLOS

Busquemos líneas que contengan la palabra “not”:

```
$ grep not haiku.txt
```

```
Is not the true Tao, until
"My Thesis" not found
Today it is not working
```

Vamos a probar un patrón distinto: “The”.

```
$ grep The haiku.txt
```

```
The Tao that is seen
"My Thesis" not found.
```

Para restringir los aciertos a las líneas que contienen la palabra “The” por sí sola, podemos usar `grep` con el indicador `-w`. Esto limitará los coincidencias a palabras.

```
$ grep -w The haiku.txt
```

```
The Tao that is seen
```

A veces, queremos buscar una frase, en vez de una sola palabra. Esto puede hacerse fácilmente con `grep` usando la frase entre comillas.

```
$ grep -w "is not" haiku.txt
```

```
Today it is not working
```

EJEMPLOS

Otra opción útil es `-n`, que numera las líneas que coinciden:

```
$ grep -n "it" haiku.txt
```

```
5:With searching comes loss
9:Yesterday it worked
10:Today it is not working
```

Podemos combinar la opción `-w` para encontrar las líneas que contienen la palabra “the” con `-n` para numerar las líneas que coinciden:

```
$ grep -n -w "the" haiku.txt
```

```
2:Is not the true Tao, until
6:and the presence of absence:
```

Ahora queremos usar la opción `-i` para hacer que nuestra búsqueda sea insensible a mayúsculas y minúsculas:

```
$ grep -n -w -i "the" haiku.txt
```

```
1:The Tao that is seen
2:Is not the true Tao, until
6:and the presence of absence:
```

EJEMPLOS

Ahora, queremos usar la opción `-v` para invertir nuestra búsqueda, es decir, queremos que obtener las líneas que NO contienen la palabra “the”.

```
$ grep -n -w -v "the" haiku.txt
```

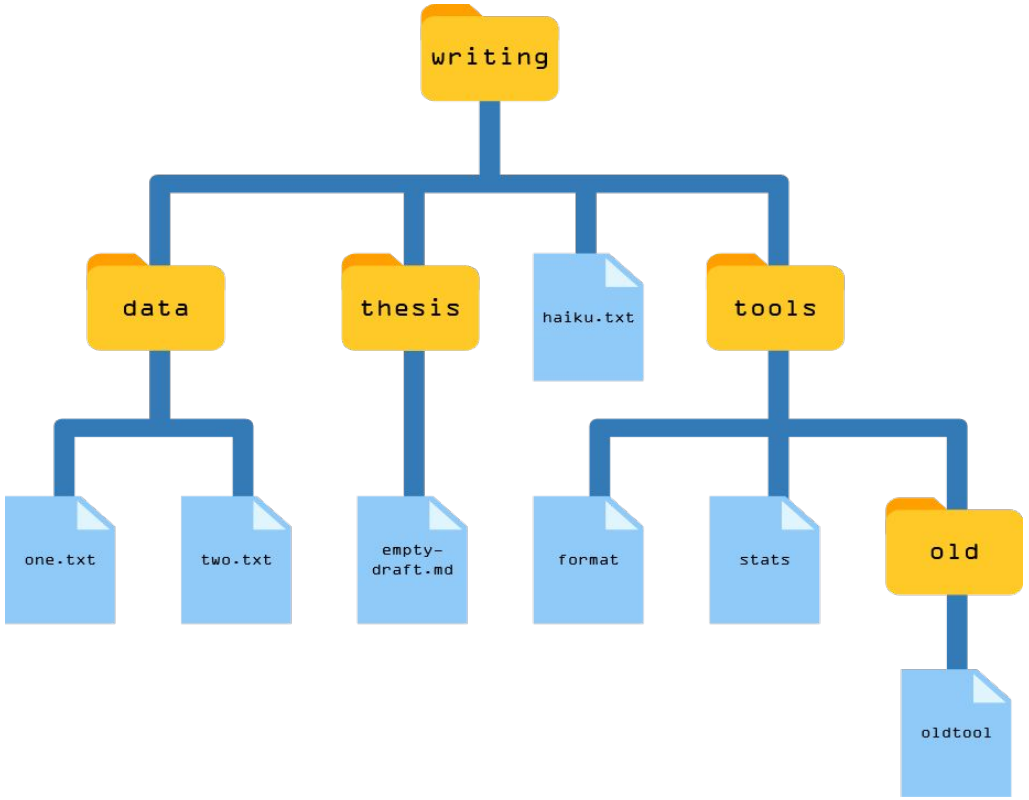
```
1:The Tao that is seen
3:You bring fresh toner.
4:
5:With searching comes loss
7:"My Thesis" not found.
8:
9:Yesterday it worked
10:Today it is not working
11:Software is like that.
```

`grep` tiene muchas otras opciones. Para averiguar cuáles son, podemos escribir:

```
$ grep --help
```

Encontrando archivos: find

Mientras `grep` encuentra líneas en los archivos, El comando `find` busca los archivos. Una vez más, tienes muchas opciones. Para mostrar cómo funcionan las más simples, utilizaremos el árbol de directorios que se muestra a continuación.



EJEMPLOS

La primera opción en nuestra lista es `-type d` que significa “encontrar directorios”.

```
$ find . -type d
```

```
./
./data
./thesis
./tools
./tools/old
```

Si cambiamos `-type d` por `-type f`, recibimos una lista de todos los archivos:

```
$ find . -type f
```

```
./haiku.txt
./tools/stats
./tools/old/oldtool
./tools/format
./thesis/empty-draft.md
./data/one.txt
./data/LittleWomen.txt
./data/two.txt
```

EJEMPLOS

Ahora tratemos de buscar por nombre:

```
$ find . -name *.txt
```

```
./haiku.txt
```

El problema es que el shell amplía los caracteres comodín como `*` *antes* de ejecutar los comandos. El comando que ejecutamos era:

```
$ find . -name haiku.txt
```

Para conseguir lo que queremos, vamos a hacer lo que hicimos con `grep`: escribe `* txt` entre comillas simples para evitar que el shell expanda el comodín `*`

```
$ find . -name '*.txt'
```

```
./data/one.txt
./data/LittleWomen.tx
t
./data/two.txt
./haiku.txt
```

LISTAR Y BUSCAR

`ls` y `find` se pueden usar para hacer cosas similares dadas las opciones correctas, pero en circunstancias normales, `ls` enumera todo lo que puede, mientras que `find` busca cosas con ciertas propiedades y las muestra. ¿Cómo podemos combinar eso con `wc -l` para contar las líneas en todos esos archivos?

La forma más sencilla es poner el comando `find` dentro de `$()`:

```
$ wc -l $(find . -name '*.txt')
```

```
11 ./haiku.txt
300 ./data/two.txt
21022
./data/LittleWomen.txt
70 ./data/one.txt
21403 total
```

RESUMEN: ENCONTRANDO ARCHIVOS

- `find` encuentra archivos con propiedades específicas que coinciden con los patrones especificados.
- `grep` selecciona líneas en archivos que coinciden con los patrones especificados.
- `--help` es un indicador usado por muchos comandos bash y programas que se pueden ejecutar desde dentro de Bash, se usa para mostrar más información sobre cómo usar estos comandos o programas.
- `man command` muestra la página del manual de un comando.
- `$(comando)` contiene la salida de un comando.