Programming for Data Science

Adolfo De Unánue

4 de septiembre de 2019

Índice general

Lectures

Línea de comandos

La computadora

Las computadoras sólo hacen cuatro cosas:

- Ejecutan programas
- Almacenan datos
- Se comunican entre sí para hacer las cosas recién mencionadas.
- Interactúan con nosotros.
  - La interacción puede ser gráfica (como están acostumbrados) conocida también como **GUI** (*Graphical User Interface*) vía el ratón u otro periférico, o desde la línea de comandos, llamada como **CLI** (*Command Line Interface*).

Introducción

El `shell` de Unix (en su caso particular es un `shell` de `GNU/Linux`), es más viejo que todos nosotros. Y el hecho de que siga activo, y en uso, se debe a que es una de las invenciones humanas más exitosas para usar la computadora de manera eficiente.
De una manera muy rápida el `shell` puede hacer lo siguiente:

- Un intérprete interactivo: lee comandos, encuentra los programas correspondientes, los ejecuta y despliega la salida.
  - Esto se conoce como **REPL**: *Read, Evaluate, Print, Loop*
- La salida puede ser redireccionada a otro lugar además de la pantalla. (Usando > y <).
- Una cosa muy poderosa (y en la que está basada –como casi todo lo actual–) es combinar comandos que son muy básicos (sólo hacen una sola cosa) con otros para hacer cosas más complicadas (esto es con un **pipe** `|`).
- Mantiene un histórico que permite rejecutar cosas del pasado.
- La información es guardada jerárquicamente en carpetas o directorios.
- Existen comandos para hacer búsquedas dentro de archivos (`grep`) o para buscar archivos (`find`) que combinados pueden ser muy poderosos.
  - Uno puede hacer **data analysis** solamente con estos comandos, así de poderosos son.
- Las ejecuciones pueden ser pausadas, ejecutadas en el **fondo** o en máquinas remotas.
- Además es posible definir variables para usarse por otros programas.
- El `shell` cuenta con todo un lenguaje de programación, lo que permite ejecutar cosas en **bucles**, **condicionales**, y hasta cosas en paralelo.

¿Por qué?

En muchas ocasiones, se verán en la necesidad de responder muy rápido y en una etapa muy temprana del proceso de **big data** . Las peticiones regularmente serán cosas muy sencillas, como estadística univariable y es aquí donde es posible responder con las herramientas mágicas de UNIX.

Línea de comandos

La línea de comandos es lo que estará entre nosotros y la computadora casi todo el tiempo en este curso. De hecho, una lectura obligada[1] es In the beginning…was de command line de **Neal Stephenson**,

[1] No es de tarea, pero debería de serlo

el escritor de **Criptonomicon**[2].

La **CLI** es otro programa más de la computadora y su función es ejecutar otros comandos. El más popular es `bash`, que es un acrónimo de **Bourne again shell**. Aunque en esta clase también usaremos `zsh`[3].

[2] Otra lectura recomendada

[3] Que significa Z shell. Lo sé, algo decepcionante.

¡Ten cuidado!

La primera regla de la línea de comandos es: *ten cuidado con lo que deseas, por que se te va a cumplir*. La computadora hará exactamente lo que le digas que haga, pero recuerda que los humanos tienen dificultadas para expresarse en *lenguaje de computadoras*.

Esta dificultad puede ser muy peligrosa, sobre todo si ejecutas programas como `rm` (*borrar*) o `mv` (*mover*).

Creando archivos de prueba

Puedes crear archivos *dummy* para este curso usando el comando `touch`:

```
touch space\ bars\ .txt
```

Nota que usamos el caracter \ para indicar que queremos un espacio en el nombre de nuestro archivo. Si no lo incluyes

```
touch space bars .txt
```

... la computadora creará tres archivos separados: `space`, `bars`, and `.txt`[4].

[4] Ve la advertencia en la sección anterior

Archivos y directorios

La computadora guarda la información de una manera ordenada. El sistema encargado de esto es el `file system`, el cual es básicamente un árbol de información[5] que guarda los datos en una abstracción que llamamos **archivos** y ordena los archivos en **carpetas** o **directorios**, los cuales a su vez pueden contener otros **directorios**.

[5] Aunque hay varios tipos de `file systems` (`ext3`, `ext4`, `xfs`, `bfs`, etc) que pueden utilizar modificaciones a esta estructura de datos, lo que voy a decir aplica desde el punto de vista de usuario del file system no su especificación técnica.

> **Info: TODO** en los sistemas operativos `*nix` (como Unix, GNU/Linux, FreeBSD, MacOS, etc) es un *archivo*.

Muchos de los comandos del **CLI** o `shell` tienen que ver con la manipulación del `file system`.

Filosofía UNIX

Creada originalmente en 1978 por **Doug McIlroy**, la filosofía de UNIX es un acercamiento al diseño de software que enaltece el software modular y minimalista.

Han existido varias adaptaciones[6], pero la que más me gusta es la de **Peter H. Salus**,

Es importante tener estos principios en mente, ya que ayuda a enmarcar los conceptos que siguen.

- Escribe programas que hagan una cosa y que la hagan bien
- Escribe programas que puedan trabajar en conjunto
- Escribe programas que puedan manipular *streams* de texto, ya que el texto es la interfaz universal.

[6] Una discusión larga y detallada se encuentra en The Art of Unix Programming de Eric Steven Raymond.

Conocer los alrededores

Navegación en la terminal

Moverse rápidamente en la **CLI** es de vital importancia. Teclea en tu *terminal*

```
Anita lava la tina
```

Y ahora intenta lo siguiente:

`Ctrl + a` Inicio de la línea
`Ctrl + e` Fin de la línea
`Ctrl + r` Buscar hacia atrás[7]
`Ctrl + b` Mueve el cursor hacia atrás una letra a la vez
`Alt + b` Mueve el cursor hacia atrás una palabra a la vez
`Ctrl + f` Mueve el cursor hacia adelante una letra a la vez
`Alt + f` Mueve el cursor hacia adelante una palabra a la vez
`Ctrl + k` Elimina el resto de la línea (en realidad corta y pone en el búfer circular)
`Ctrl + y` Pega la último del búfer.
`Alt + y` Recorre el búfer circular.
`Ctrl + d` Cierra la terminal
`Ctrl + z` Manda a *background* el programa que se está ejecutando
`Ctrl + c` Intenta cancelar
`Ctrl + l` Limpia la pantalla

[7] Elimina el nefasto y tardado flechita arriba

Atención: Estas combinaciones de teclas (*keybindings*) son universales. Te recomiendo que las practiques y configures tus otras herramientas con estos mismas combinaciones, por ejemplo RStudio ó JupyterLab .

Pregunta 1

¿Qué hacen las siguientes combinaciones?

- `Alt + t`
- `Alt + d`
- `Ctrl + j`
- `Alt + 2 Alt + b`

Para tener más información sobre los *bindings* consulta aquí.

¿Quién soy?

```
whoami
```

¿Quién está conmigo?

```
who
```

¿Dónde estoy?

Imprime el nombre del *directorio* actual

```
pwd
```

Cambia el directorio un nivel arriba (a el directorio *padre* [8])

```
cd ..
```

[8] En inglés es parent directory, no se me ocurrió otra traducción ¿Alguna sugerencia?

Si quieres regresar al directorio anterior

```
cd -
```

Hogar dulce, hogar

Cambia el directorio `$HOME` (tu directorio) utilizando ~

```
cd ~
```

o bien, no pasando ningún argumento

```
cd
```

¿Qué hay en mi directorio (*folder*) ?

`ls` Lista los contenidos (archivos y directorios) en el directorio actual, pero no los archivos *ocultos*.

```
ls
```

Lista los contenidos en formato *largo* (`-l`), muestra el tamaño de los archivos, fecha de último cambio y permisos

```
ls -l
```

Lista los contendios en el directorio actual y todos los sub-directorios en una estructura de *árbol*

```
tree
```

Límita la expansión del *árbol* a dos niveles

```
tree -L 2
```

Muestra los archivosshows file sizes (`-s`) in human-readable format (`-h`)

```
tree -hs
```

¿Qué hay en mi archivo?

Muestra el principio (*head*) del archivo, `-n` especifica el número de líneas (`10`).

```
head -n10 $f
```

Muestra la final (*tail*) del archivo.

```
tail -n10 $f
```

Muestra la parte final del archivo cada segundo (usando `watch`)

```
tail -n10 $f | watch -n1
```

"seguir" (*follows*) (`-f`) la parte final del archivo, cada vez que hay cambios

```
tail -f -n10 $f
```

> **Info:** Seguir archivos es útil cuando estás ejecutando un programa que guarda información a un archivo, por ejemplo un *log*

Cuenta las palabras, caracteres y líneas de un archivo

```
wc $f
```

¿Dónde está mi archivo?

Encuentra el archivo por nombre

```
find -name "<lost_file_name>" -type f
```

Encuentra directorios por nombre

```
find -name "<lost_dir_name>" -type d
```

Caveats con `git`

Mover archivos puede confundir a `git`. Si estás trabajando con archivos en `git` usa lo siguiente:

```
# Para mover o renombrar
git mv /source/path/$move_me /destination/path/$move_me

# Para eliminar
git rm $remove_me
```

Practiquemos un poco

Enciende la máquina virtual

```
vagrant up
```

Conéctate a la máquina virtual con `vagrant`

```
vagrant ssh
```

Teclea `whoami` y luego presiona **enter**. Este comando te dice que usuario eres.
Teclea `cd /`
Para saber donde estamos en el `file system` usamos `pwd` (de *print working directory*).

Info: Estamos posicionados en la raíz del árbol del sistema, el cual es simbolizada como `/`.

Para ver el listado de un directorio usamos `ls`

Info: Ahora estás observando la estructura de directorios de `/`.

Los comandos (como `ls`) pueden tener modificadores o **banderas** (*flags*), las cuales modifican (vaya sorpresa) el comportamiento por omisión del comando. Intenta lo siguiente: `ls -l`, `ls -a`, `ls -la`, `ls -lh`, `ls -lha`. Discute con tu compañero junto a tí las diferencias entre las banderas.
Para obtener ayuda puedes utilizar `man` (de *manual*) y el nombre del comando.

Pregunta 2

¿Cómo puedes aprender que hace `ls`?

Puedes buscar dentro de `man page` para `ls` (o de cualquier otro manual) si tecleas `/` y escribiendo la palabra que buscas, luego presiona `enter` para iniciar la búsqueda. Esto te mostrará la primera palabra que satisfaga el criterio de búsqueda. `n` te mostrará la siguiente palabra. `q` te saca del `man page`.
Busca la bandera para ordenar (*sort*) el listado de directorios por tamaño.
Muestra el listado de archivos de manera ordenada por archivo.
Otro comando muy útil (aunque no lo parecerá ahorita) es `echo`.
Las variables de sistema (es decir globales en tu sesión) se pueden obtener con el comando `env`. En particular presta atención a `HOME`, `USER` y `PWD`.
Para evaluar la variable podemos usar el signo de moneda `$`,
Imprime las variables con `echo`, e.g.

```
echo $USER
```

Pregunta 3

¿Qué son las otras variables `HOME`, `PWD`?

El comando `cd` permite cambiar de directorios (¿Adivinas de donde viene el nombre del comando?)
La sintáxis es `cd nombre_directorio`.

Pregunta 4

¿Cuál es la diferencia si ejecutas `ls -la` en ese directorio?

Info: Las dos líneas de hasta arriba son `.` y `..` las cuales significan **este directorio** (`.`) y el directorio padre (`..`) respectivamente. Los puedes usar para navegar (i.e. moverte con `cd`)

Pregunta 5

¿Puedes regresar a raíz?

Pregunta 6

En raíz (/) ¿Qué pasa si ejecutas `cd $HOME`?

Info: Otras maneras de llegar a tu `$HOME` son `cd ~` y `cd` (sin argumento).

Verifica que estés en tu directorio (¿Qué comando usarias?) Si no estás ahí, ve a él.

Para crear un directorio existe el comando `mkdir` que recibe como parámetro un nombre de archivo.

Crea la carpeta `test`. Entra a ella. ¿Qué hay dentro de ella?

Vamos a crear un archivo de texto, para esto usaremos `nano`[9]. Por el momento teclea

```
nano hola.txt
```

y presiona enter.

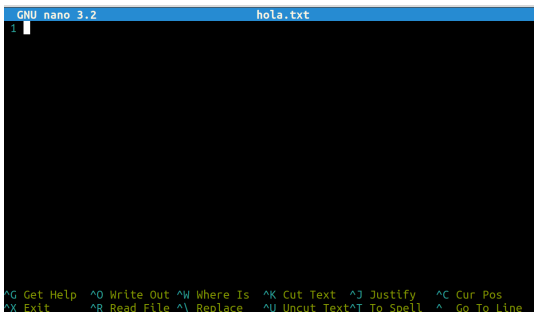

Figura 0.1: Editor `nano`, mostrando el archivo recién creado `hola.txt` .

Teclea en `nano`

```
¡Hola Mundo!
```

y luego presiona la siguiente combinación de teclas: `Ctrl+O` [10]para guardar el archivo (Te va a preguntar *dónde* guardarlo).

[9] Otras opciones son: `GNU Emacs` (un editor de textos muy poderoso. Es el que estoy usando) o `vi`. No importa cual escojas, aprende a usarlo muy bien. Recuerda, queremos disminuir el dolor.

[10] Las combinaciones de las teclas están desplegadas en la parte inferior de la pantalla

`Ctrl+X` para salir de `nano`. Esto te devolverá a la línea de comandos.
Verifica que esté el archivo.
Para ver el contenido de un archivo tienes varias opciones (además de abrir `nano`): `cat`, `more`, `less`

```
cat hola.txt
```

Para borrar usamos el comando `rm` (de *remove*).
Borra el archivo `hola.txt`.

> Pregunta 7
>
> ¿Cómo crees que se borraría un directorio?
> ¿Puedes borrar el directorio `test`? ¿Qué falla? ¿De dónde puedes obtener ayuda?

Crea otra carpeta llamada `tmp`, crea un archivo `copiame.txt` con `nano`, escribe en él:

```
Por favor cópiame
```

Averigua que hacen los comandos `cp` y `mv`.
Copia el archivo a uno nuevo que se llame `copiado.txt`.
Borra `copiame.txt`.
Modifica `copiado.txt`, en la última línea pon

```
¡Listo!
```

Renombra `copiado.txt` a `copiame.txt`.
Por último borra toda la carpeta `tmp`.
Desconéctate de la máquina virtual con `Ctrl+D` y luego "apaga" la máquina virtual

```
vagrant halt
```

Wildcards: Globbing

La línea de comandos permite usar comodines (*wildcards*)[11] para encontrar archivos:
El primer *glob* que veremos es `*`:

```
echo *
```

El `shell` expandió `*` para identificarlo con todos los archivos del directorio actual.
Obvio lo puedes usar con otros comandos, como `ls`: Listar todos los archivos que tienen una extensión `txt`

```
ls *.txt
```

Listar todos los archivos que contienen `a` en el nombre y extensión `txt`

```
ls *a*.txt
```

`*` no es el único carácter especial. `?` hace *match* con cualquier carácter individual.
Listar todos los archivos que tienen 5 caracteres en el nombre:

[11] También permite expresiones regulares (regular expressions, a.k.a. regexp), pero es importante saber que NO son iguales. Globs y regexps son usadas en diferentes contextos y significan diferentes cosas. Por ejemplo el símbolo `*`, es un modificador de cantidad en regexp, pero expande cuando es usado como globs. Más adelante veremos un ejemplo.

```
ls ?????,txt
```

Datos para jugar

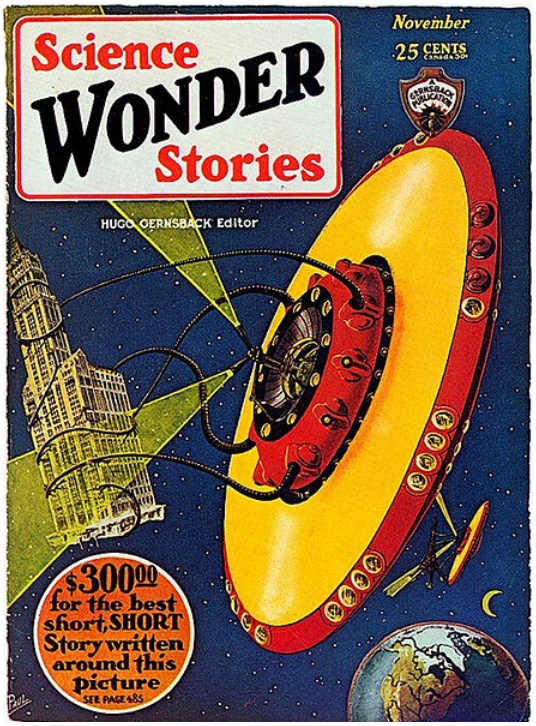

Figura 0.2: Portada de la revista Science Wonder Stories (1929). Imagen tomada de Wikipedia

- Para los siguientes ejemplos trabajaremos con los archivos encontrados en **The National UFO Reporting Center Online Database**[12]
- Estos datos representan los *avistamientos* de OVNIS en EUA.
- Usaremos como ejemplo la descarga el mes de Noviembre y Diciembre de 2014
- Se encuentran en la carpeta `data/ufo` en la máquina virtual.

[12] Por cierto, no creo que haya vida extraterreste, principalmente debido a la paradoja de Fermi (también ver aquí). Relacionado con esto, creo que la mejor solución a esta paradoja es la teoría del Gran Filtro (más información). Si prefieres vídeos este playlist tiene todo lo que quieres saber.

Conectando comandos

Entubando

El símbolo | (*pipe*) "entuba" la salida de un comando al siguiente comando. Por ejemplo:

```
ls -la | wc -l
```

La salida de `ls -la` en lugar de ser impresa en pantalla[13] es enviada a `wc -l`
El siguiente ejemplo utiliza `grep` para buscar y seleccionar cadenas o patrones

[13] ¿Recuerdas el REPL?

```
seq 50 | grep 3
```

Veremos más sobre `seq` y `grep` más adelante.

`stdin`, `stdout` y `stderr`

`stdin` (entrada estándar), `stdout` (salida estándar) y `stderr` (error estándar) son *canales* de interacción de la terminal. En tu terminal, todos apuntan a la pantalla, pero es posible redireccionarlos hacia otros lados.

Los tres canales están asignados a los siguientes *file descriptors*[14]

0 `stdin`, el teclado
1 `stdout`, la pantalla
2 `stderr`, la pantalla

[14] Recuerda, en `GNU/Linux` todo es un archivo, incluído el hardware

Es posible redireccionar por ejemplo, todos los mensajes de error a un archivo

```
rm este_archivo_no_existe 2> error.txt
```

Si quieres ignorar los errores, puedes mandarlos a un agujero negro[15] estándar, i.e. `/dev/null`:

[15] El sistema operativo está lleno de construcciones similares, por ejemplo `/dev/random`

```
algun_comando 2> /dev/null
```

o apuntar `stdout` y `stderr` al mismo lado (configuración por *default*)

```
algun_comando 2>&1
```

Redireccionando *hacia*

Los símbolos >, » Redireccionan la salida de los comandos a un sumidero (*sink*), e.g. un archivo, o la pantalla o la impresora.
La salida de `ls` se guarda en el archivo `prueba.bat`.

```
ls >> prueba.dat
```

Similar al ejemplo anterior

```
seq 10 > numeros.txt
```

Pregunta 8

¿Cuál es la diferencia?

TIP: Ejecuta varias veces los ejemplos anteriores

Redireccionando *desde*

< Redirecciona desde el archivo

```
sort < prueba.dat # A la línea de comandos acomoda con sort,
sort < prueba.dat > prueba_sort.dat # Guardar el sort a un archivo.
```

Incluso puedes hacer

```
< prueba.dat wc -l
```

Info: Esto suena muy similar a

```
cat prueba.dat | wc -l
```

`< prueba.dat wc -l` es más eficiente, ya que no está generando un subproceso (lo cual puede ser muy importante en procesamiento intensivo.

Substitución de comandos

En muchas ocasiones quieres la salida estándar (`stdoout`) de un comando para reusarla en algún *script* u otro comando.

```
echo "Hoy es $(date)"
```

Puedes guardar el resultado del comando en una variable:

```
NUMBER_OF_LINES=$(wc -l data/*.txt |  tail -1 | cut -d' ' -f 2)
```

El operador `$()` se le conoce como *command substitution*

Substitución de procesos

Existe otro operador, `<()`, llamado *process substitution*. A diferencia del operador `$()` que sustituye la **salida** del proceso ejecutado dentro del `$()`, el operador `<()` sustituye un **archivo que contiene la salida** del proceso que se ejecutó dentro de `<()`

El ejemplo paradigmático del operador `<()` ocurre cuando tienes que pasar varios *outputs* a un solo programa, por ejemplo `diff`[16],

[16] `diff` muestra la diferencia entre dos archivos

```
diff <(ls /data) <(ls /vagrant)
```

o si estás usando muchos archivos temporales, por ejemplo, en lugar de hacer

```
curl http://www.nuforc.org/webreports/ndxe201908.html > 201908.txt
curl http://www.nuforc.org/webreports/ndxe201907.html > 201907.txt
cat 201908.txt 201907.txt > 2019.txt
rm 20190?.txt
```

puedes hacer

```
cat <(curl http://www.nuforc.org/webreports/ndxe201908.html) \
    <(curl http://www.nuforc.org/webreports/ndxe201907.html) > 2019.txt
```

Una ventaja de este último ejemplo, es que el `shell` ejecutará los procesos en **paralelo** (!!).

Ejecución condicional

`&&` es un AND, sólo ejecuta el comando que sigue a `&&` si el primero es exitoso.

```
ls && echo "Hola"
```

```
lss && echo "Hola"
```

Si sólo quieres ejecutar varios comandos, sin preocuparte por que todos sean exitosos, cambia el `&&` por `;` (punto y coma).

Algunos comandos útiles

seq

Genera secuencias de números

```
seq 5
```

La sintaxis es: `seq inicio step final`. Por ejemplo

```
seq 1 2 10
```

genera la secuencia de 1 al 10 de dos en dos.
Usando otro separador (`-s`) que no sea es caracter de espacio

```
seq -s '|' 10
```

Agregando *padding*

```
seq -w 1 10
```

tr

Cambia, reemplaza o borra caracteres del `stdin` al `stdout`

```
echo "Hola mi nombre es Adolfo De Unánue"  | tr '[:upper]' '[:lower]'
```

```
echo "Hola mi nombre es Adolfo De Unánue"  | tr -d ' '
```

```
echo "Hola mi nombre es Adolfo De Unánue"  | tr -s ' ' '_'
```

Ejercicio 1

Transforma el archivo de `data/ufo` de tabuladores a `|`, cambia el nombre con terminación `.psv`.

wc

`wc` significa *word count*. Este comando cuenta las palabras,renglones, bytes en el archivo.
En nuestro caso nos interesa la bandera `-l` la cual sirve para contar líneas.

```
seq 30 | grep 3 | wc -l
```

Ejercicio 2

- ¿Cuantos avistamientos existen Noviembre 2014? ¿Y en Diciembre 2014?
- ¿En total?

head, tail

`head` y `tail` sirven para explorar visualmente las primeras diez (*default*) o las últimas diez (*default*) renglones del archivo, respectivamente.

```
head UFO-Dic-2014.tsv

tail -3 UFO-Dic-2014.tsv
```

cat

`cat` concatena archivos y/o imprime al `stdout`

```
echo 'Hola mundo' >> test

echo 'Adios mundo cruel' >> test

cat test

rm test
```

Podrías hacer lo mismo, sin utilizar `echo`

```
cat >> test

# Teclea Hola mundo
# Teclea Adios mundo cruel

# Ctrl-C
```

También podemos concatenar archivos

```
cat UFO-Nov-2014.tsv UFO-Dic-2014.tsv  > UFO-Nov-Dic-2014.tsv
wc -l UFO-Nov-Dic-2014.tsv
```

En el siguiente ejemplo redireccionamos al `stdin` el archivo como entrada del `wc -l` sin generar un nuevo proceso

```
< numeros.txt wc -l
```

split

`split` hace la función contraria de `cat`, divide archivos. Puede hacerlo por tamaño (bytes, `-b`) o por líneas (`-l`).

```
split -l 500 UFO-Nov-Dic-2014.tsv
wc -l UFO-Nov-Dic-2014.tsv
```

cut

Con `cut` podemos dividir el archivo pero por columnas. Las columnas puede estar definidas como campo (`-f`, `-d`), carácter (`-c`) o bytes (`-b`).
Creemos unos datos de prueba

```
echo "Adolfo|1978|Físico" >> prueba.psv
echo "Paty|1984|Abogada" >> prueba.psv
```

Ejecuta los siguientes ejemplos, ¿Cuál es la diferencia?

```
cut -d'|' -f1 prueba.psv
cut -d'|' -f1,3 prueba.psv
cut -d'|' -f1-3 prueba.psv
```

Ejercicio 3

- ¿Qué pasa con los datos de avistamiento? Quisiera

las columnas 2, 4, 6 ó si quiero las columnas Fecha, Posted, Duración y Tipo (en ese orden).

- ¿Notaste el problema? Para solucionarlo requeriremos comandos más poderosos…
- Lee la documentación (`man cut`), ¿Puedes ver la razón del problema?

uniq

- `uniq` Identifica aquellos renglones consecutivos que son iguales.
- `uniq` puede contar (`-c`), eliminar (`-u`), imprimir sólo las duplicadas (`-d`), etc.

sort

- `sort` Ordena el archivo, es muy poderoso, puede ordenar por

columnas (`-k`), usar ordenamiento numérico (`-g`, `-h`, `-n`), mes (`-M`), random (`-r`) etc.

```
sort -t "," -k 2  UFO-Nov-Dic-2014.tsv
```

uniq y sort

- Combinados podemos tener un `group by`:

```
# Group by por estado y fecha
cat UFO-Dic-2014.tsv |\
    cut -d$'\t' -f1,3 |\
    tr '\t' ' ' |\
    cut-d' ' -f1,3 |\
    sort -k 2 -k 1 |\
    uniq -c |\
    sort -n -r -k 1 |\
    head
```

Ejercicio 4

- ¿Cuál es el top 5 de estados por avistamientos?
- ¿Cuál es el top 3 de meses por avistamientos?

Hacer varias cosas a la vez

```
# Ejecuta un servidor web en el puerto 8888
python -m http.server 8888
```

Ve a la dirección mostrada en la terminal en tu navegador

Si presionas `Ctrl+z` *suspenderás* la ejecución de ese programa, pero no será **cancelado**. Observa que el `shell` te devolvió el control de la terminal. Trata de ver de nuevo la lista de archivos en el navegador … y el navegador se quedará en *Waiting for 0.0.0.0 ...*

Regresa a la terminal y ejecuta

```
jobs
```

`jobs` muestra que programas están ejecutándose y su estatus. En particular, el servidor `http` está detenido.

Para activarlo usa

```
fg
```

`fg` significa *foreground*. Interrumpe la ejecución con `Ctrl+c`.

Si quieres ejecutar el servidor y **no** bloquear la terminal agrega `&` al final.

```
python -m http.server 8888 &
```

De esta manera el servidor está ejecutándose en el *background*.

Ejercicio 5

En la misma terminal ejecuta el editor `nano` y escribe algo en él

- ¿Cómo lo mandas al *background*?
- ¿Cómo lo traes de nuevo en ejecución?
- ¿Cuál es el estatus de los *jobs*?

Atención: Es importante saber que estos procesos o *jobs* estarán ejecutándose mientras tu sesión esté activa. Si te desconectas los procesos serán terminados.

Más adelante veremos como solucionar esto.

Otros comandos útiles

`file -i` Provee información sobre el archivo en cuestion

```
file -i UFO-Dic-2014.tsv
```

`iconv` Convierte entre encodings, charsets etc.

```
iconv -f iso-8859-1 -t utf-8 UFO-Dic-2014.tsv  > UFO-Dic_utf8.tsv
```

`od` Muestra el archivo en octal y otros formatos, en particular la bandera `-bc` lo muestra en octal seguido con su representación ascii. Esto sirve para identificar separadores raros.

```
od -bc UFO-Nov-2014.tsv   | head -4
```

Descargar datos

Es posible hacer peticiones de HTTP (*http requests*)[17] desde la línea de comandos.
El comando para hacerlo es `curl`.

```
curl http://www.gutenberg.org
```

Info: Si hay redirecciones puedes usar `-L` para seguir la redirección (por ejemplo en los casos donde hay un *url shortener*, como `bit.ly`.

La respuesta (*response*) incluye el *body* (lo que ves en el navegador) y el *header* (metainformación sobre la petición y respuesta). Si sólo quieres ver el *header*

```
curl -I https://duckduckgo.com
```

El *header* contiene un pedazo de información crucialÑ: el status code. El *status code* te sirve para ver el estado de la página:

- ¿La página respondió correctamente? `200 OK`
- ¿El recurso solicitado no existe? `404 File Not Found`
- etc

```
curl -I https://duckduckgo.com 2>/dev/null | head -n 1 | cut -d$' ' -f2
```

donde:

- `2>/dev/null`, redirecciona `stderr` a un agujero negro
- `head -n 1`, lee la primera línea únicamente
- `cut -d$' ' -f2`, separa la línea usando espacios (`' '`) y toma el segundo campo, que contiene el código de estatus `HTTP`.

[17] `HTTP` es el protocolo de comunicación usado por el navegador para solicitar y recibir documentos guardados en otras computadoras o servidores (páginas web, les dicen). Más adelante en el curso lo discutiremos en profundidad.

Expresiones regulares

> In computing, regular expressions provide a concise and flexible means for identifying strings of text of interest, such as particular characters, words, or patterns of characters. Regular expressions (abbreviated as regex or regexp, with plural forms regexes, regexps, or regexen) are written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.
>
> Wikipedia: Regular Expressions

Regexp: Básicos

- Hay varios tipos POSIX, Perl, PHP, GNU/Emacs, etc. Nosotros nos enfocaremos en `POSIX`.

Atención: Conocer que tipo de expresiones regulares estás utilizando te quitará muchos dolores de cabeza.

Lee la documentación de tu lenguaje favorito.

- Pensar en patrones (*patterns*).
- Operadores básicos

  OR `gato|gata` hará match con gato o gata.

  Agrupamiento o precedencia de operadores `gat(a|o)` tiene el mismo significado que `gato|gata`.

- Cuantificadores

  ? o ó 1

  + uno o más

  ***** cero o más[18]

- Expresiones básicas

  . Cualquier carácter.

  [ ] Cualquier carácter incluido en los corchetes, e.g. `[xyz]`, `[a-zA-Z0-9-]`.

  [^] Cualquier caracter individual que `n` esté en los corchetes, e.g. `[^abc]`. También puede indicar inicio de líinea (fuera de los corchetes.).

\(\) ó ( ) crea una subexpresión que luego puede ser invocada con `\n` donde `n` es el número de la subexpresión.

{m,n} Repite lo anterior un número de al menos `m` veces pero no mayor a `n` veces.

\b representa el límite de palabra.

[18] Como discutimos anteriormente el glob `*` es diferente al operador `*`

Regexp: Ejemplos

- username: `[a-z0-9 -]{3,16}`
- contraseña: `[a-z0-9 -]{6,18}`
- IP address:

```
(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?\.){3}(25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)
```

Ejercicio 6

- fecha (dd/mm/yyyy): ???
- email (adolfo@itam.edu) : ???
- URL (`http://gmail.com`): ???

Regexp: Expresiones de caracteres

- `[:digit:]` Dígitos del 0 al 9.
- `[:alnum:]` Cualquier caracter alfanumérico 0 al 9 OR A a la Z OR a a la z.
- `[:alpha:]` Caracter alfabético A a la Z OR a a la z.
- `[:blank:]` Espacio o TAB únicamente.

Regexp: ¿Quieres saber más?

- Learn Regular Expressions in 20 minutes
- The 30 minute regex tutorial

Analizando datos[19]: Comandos avanzados

[19] Principalmente texto, para ser justos

grep

`grep` nos permite buscar líneas que tengan un patrón específico. Es el equivalente a un filtro.

```
grep "CA" UFO-Nov-Dic-2014.tsv
grep "HOAX" UFO-Nov-Dic-2014.tsv
grep -v "18:" UFO-Nov-Dic-2014.tsv
grep -E "18:|19:|20:" UFO-Nov-Dic-2014.tsv
grep -E "[B|b]lue|[O|o]range" UFO-Nov-Dic-2014.tsv \
      | grep -v "Orangebug" |
grep -i -E "[blue|orange]" UFO-Dic-2014.tsv
grep -c -o -E "[B|b]lue|[O|o]range" UFO-Nov-Dic-2014.tsv  # Ejecuta una bandera a la vez
grep -o -E "[B|b]lue|[O|o]range" UFO-Nov-Dic-2014.tsv | sort | uniq -c
grep "\/[0-9]\{1,2\}\/" UFO-Dic-2014.tsv # Seleccionamos dias
grep -v "\/[0-9]\{4\}" UFO-Dic-2014.tsv  # Año mal formateado
grep -E "([aeiou]).*\1" names.txt # Vocal caracteres Misma vocal
echo "Hola grupo ¿Cómo están?" | grep -oE '\w+'
```

Ejercicio 7

Usando los archivos de la carpeta `grep`

- Selecciona las líneas que tienen exactamente cinco dígitos.
- Selecciona las que tienen más de 6 dígitos.
- Cuenta cuántos *javier*, *romina* o *andrea* hay.

awk

`awk` es un lenguaje de programación muy completo, orientado a archivos de texto que vengan en columnas, i.e. *dataframes*

Un programa de `awk` consiste en una secuencia de enunciados del tipo patrón-acción:

```
awk  'search pattern { action statement }' [file]
```

`awk` define algunas variables especiales:

`$1,$2, $3, ...` Valores de las columnas

`$0` toda la línea

`FS` separador de entrada

`OFS` separador de salida

`NR` número de la línea actual

`NF` número de campos en la línea (record)

```
 awk 'END { print NR }'  UFO-Nov-Dic-2014.tsv  # Lo mismo que wc -l
# Número de columnas
 awk 'BEGIN{FS = "\t" }; { print NF }' UFO-Nov-Dic-2014.tsv \
      | sort -n | uniq |
 awk -F"[\t]" '{ print NF }' UFO-Nov-Dic-2014.tsv \
      | sort -n | uniq |
 awk 'BEGIN{ FS = "\t" }; { if(NF =≠ 7){ print >> "UFO_fixme.tsv"} \
 else { print >> "UFO_OK.tsv" } }' UFO-Nov-Dic-2014.tsv
# Limpia el archivo con columnas de más
 awk -F"[\t]" '{ print NF ":" $0 }' UFO-Nov-Dic-2014.tsv
 awk -F"[\t]" '{ $2 = ""; print }' UFO-Nov-Dic-2014.tsv
 awk 'BEGIN{ FS = "," }; {sum += $1} END {print sum}' data.txt
# Suma de una columna (aunque aquí no tiene mucho sentido)
 awk -F, '{sum1+=$1; sum2+=$2;mul+=$2*$3} END {print sum1/NR,sum2/NR,mul/NR}' numbers.dat
# Promedios de varias columnas
```

```
awk '/CA/ { n++ }; END { print n+0 }' UFO-Nov-Dic-2014.tsv
awk -F, '$1 > max { max=$1; maxline=$0 }; END { print max, maxline }' numbers.dat
awk '{ sub(/FL/,"Florida"); print }' UFO-Nov-Dic-2014.tsv
awk '{ gsub(/foo/,"bar"); print }' UFO-Nov-Dic-2014.tsv
awk '/baz/ { gsub(/foo/, "bar") }; { print }' UFO-Nov-Dic-2014.tsv
awk '!/baz/ { gsub(/foo/, "bar") }; { print }' UFO-Nov-Dic-2014.tsv
awk 'a != $0; { a = $0 }' # Como uniq
awk '!a[$0]++' # Remueve duplicados que no sean consecutivos
awk -F"[\t]" '$4 ~/Circle/' UFO-Nov-Dic-2014.tsv
awk -F"[\t]" 'BEGIN { conteo=0;} \
       $4 ~/Circle/ { conteo++; }
END { print "Número de avistamientos circulares en el dataset =",conteo;}' UFO-Nov-Dic-2014.tsv
```

Para saber más consulta el manual **GNU awk** *Effective AWK Programming*

sed

- `sed` significa **stream editor** . Permite editar archivos de manera automática.
- El comando tiene cuatro **espacios**
  - Flujo de entrada
  - Patrón
  - *Búfer*
  - Flujo de salida
- Entonces, `sed` lee el **flujo de entrada** hasta que encuentra `\n`. Lo copia al **espacio patrón**, y es ahí donde se realizan las operaciones con los datos. El **búfer** está para su uso, pero es opcional, es un búfer, vamos. Y finalmente copia al **flujo de salida**.

```
sed 's/foo/bar/' data3.txt   # Sustituye foo por bar
sed -n 's/foo/bar/' data3.txt # Lo mismo pero no imprime a la salida
sed -n 's/foo/bar/; p' data3.txt # Lo mismo pero el comando "p", imprime
sed -n 's/foo/bar/' -e p data3.txt # Si no queremos separar por espacios
sed '3s/foo/bar/'  data3.txt # Sólo la tercera linea
sed '3!s/foo/bar/'  data3.txt # Excluye la tercera línea
sed '2,3s/foo/bar/' data3.txt # Con rango
sed -n '2,3p' data3.txt  # Imprime sólo las líneas de la 2 a la 3
sed -n '$p' # Imprime la última línea
sed '/abc/,/-foo-/d' data3.txt # Elimina todas las líneas entre "abc" y "-foo-"
sed '/123/s/foo/bar/g'  data3.txt
# Sustituye globalmente "foo" por "bar" en las líneas que tengan 123
sed 1d data2.txt # Elimina la primera línea del archivo
sed -i 1d data2.txt  # Elimina la primera línea del archivo de manera interactiva
```

Ejercicio 8

Elimina los `headers` repetidos con `sed` en los archivos `UFO`.

Bash programming

La mayor parte del tiempo usaremos el `shell`, para hacer pequeños *scripts*, pero existen ocasiones en las cuales es necesario tratar al `shell` como un lenguaje de programación[20]

[20] En este punto es bueno preguntarse si no deberías hacerlo mejor en otro lenguaje de programación, como `python`.

Estructuras de datos

Las variables son declaradas

```
nombre="Adolfo"
```

Nota que no hay espacios alrededor del signo de igual. El valor de la variable se obtiene con el signo de dólares

```
echo $nombre
```

Es posible definir variables que no sean escalares, llamadas arreglos (`arrays`).

```
array=(abc 123 def "programming for data science")
```

Para acceder a elementos en el arreglo usa la posición e.g. `3`:

```
echo ${array[3]}
```

Tambien es posible usar *globs*

```
echo ${array[*]}
```

Para conocer el número de elementos del arreglo

```
echo ${#array}
```

Atención: Los arreglos es un punto donde los diferentes *shells* ejecutan diferente. `bash` tiene índices basados en `0`, `zsh` en `1`.

Bucles de ejecución, (*Loops*)

Los `for-loops` en `bash` tienen una estructura muy similar a los de `python`:

```
for var in iterable; do
instrucción
instrucción
...
done
```

Donde `iterable` puede construirse con *globs*

```
for i in *; do
  echo $i;
done
```

listas,

```
for i in hola adios mundo cruel; do
  echo $i;
done
```

arreglos,

```
for i in $array; do
  echo $i;
done
```

E inclusive el horrible formato de C:

```
for (( i = 0; i < 10; i++ )) do
    echo $i;
done
```

Info: No tiene que estar en varias líneas: `for i in {a..z}; do echo $i; done`

Tricks

También es posible hacer lo siguiente:

```
for i in {a..z}; do
  echo $i;
done
```

En el ejemplo anterior utilizamos *brace expansion*:

```
echo {0..9}
echo {0..10..2}
echo data.{txt,csv,json,parquet}
```

Condicionales

Existen dos operadores para hacer pruebas en bash: `[` y `[[`.

De preferencia usa `[[`

```
[[ 1 = 1 ]]
echo $?    # Imprime el resultado del comando previo (true → 0)
```

Existen algunos operadores para hacer comparaciones los más comunes son `-a, -d, -z` (unarios) y `-lt, -gt, -eq, -neq` (binarios).

Teniendo los *tests* es posible crear estructuras `if-then-else`

```
if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi
```

Funciones

Las funciones son símbolo de buena programación, ya que encapsulan comportamiento.

La sintaxis es muy simple:

```
function function_name {
    # cuerpo de la función
}
```

A diferencia de otros lenguajes de programación, no se definen los argumentos de la función.
Para ejecutar la función

```
function_name "Hola mundo" 123
```

En este caso `"Hola mundo"` y `123` son pasados como argumentos a la función. Estos pueden ser usados en el cuerpo de la función usando la posición de los mismos e.g. `$1` es `"Hola mundo"`, `$2` es `123`, etc.

Ejecutar scripts

Para cualquier archivo *script* es importante que la primera línea del archivo le diga al `shell` que comando usar para ejecutarlo.
A la primera línea se conoce como **shebang** y se representa por `#!` seguido de la ruta al ejecutable, eg. `#! /usr/bin/python`, `#! /bin/bash`, `#! /usr/bin/env Rscript`
Por ejemplo:
Sin el `shebang`

```
python ejemplo.py
```

con el `shebang`

```
./ejemplo.py
```

Atención: Para que el ejemplo funcione es necesario dar permisos de ejecución al archivo

```
chmod u+x ejemplo.py
```

¿Cómo conecto estos *scripts* con los demás usando `|`, `>`, etc?
Sencillo: Hay que modificar nuestros *scripts* de `python`, `R` y `bash` para leer del `stdin`.

Python, leyendo de `stdin`

Un ejemplo mínimo de `python` es el siguiente:

```
#!/usr/bin/env python

import sys

def process(linea):
    linea = int(linea.strip())
    print(f"El triple de {linea} es {linea*3}")

for linea in sys.stdin:
    process(linea)
```

Abre `nano`, copia este código y guarda el archivo como `script.py`. Un ejemplo de uso es el que sigue:

```
seq 1 1000 | script.py
```

R, leyendo de `stdin`

El ejemplo en `R` se ve así:

```
#!/usr/bin/env Rscript

f ← file("stdin")

x ← c()

open(f)

while(length(line ← readLines(f, n = 1)) > 0) {

  x ← c(x, as,numeric(line))
  print(summary(x))
}


close(f)

print("Final summary:")
summary(x)
```

Ejemplo de uso:

```
< /data/numbers.txt script.R
```