***Curso de Terminal y Linea de Comandos.***

**¿Qué es y cómo funciona la terminal?**

* La terminal es un programa que recibe instrucciones, las traduce a lenguaje del computador, las ejecuta y muestra resultados.
* Utiliza un entorno 100% texto por lo tanto es muy eficiente.
* Permite la automatización de tareas repetitivas.
* Se compone de:

1. **El prompt:** que es un conjunto de caracteres al principio de cada línea y muestra información propia del sistema en que se esté utilizando.
2. **El cursor:** es un “underscore” que titila en la pantalla del terminal, y en conjunto con el prompt, indican que están a la espera de recibir órdenes.

* En la terminal se ingresan instrucciones, también conocidas como “comandos”.
* Estos comandos, a su vez, se componen de: nombre programa parámetro modificadores.
* La terminal también cuenta con otras utilidades, tales como:

1. Comodines.
2. Combinación de teclas.
3. Sustitución de comandos.

**Shortcuts con teclas de Windows.**

### Explorador de Archivos.

**Windows + E**.

### Mostrar Escritorio.

**Windows + D**

### Comando ejecutar

**Windows + R**

### Mostrar aplicaciones abiertas

**Windows + Tabulador**

### Configuración de Windows

**Windows + I**

### Crear escritorio virtual

**Windows + Ctrl + D**

### Cerrar escritorio virtual actual

**Windows + Ctrl + F4**

### Cambiar de escritorio

**CTRL + Windows + Flecha izquierda / derecha**

### Administrador de tareas

**Ctrl + Shift + Esc**

### Abrir el menú contextual del botón Inicio

**Windows + X**

# Comandos para organización de archivos.

# 

# 

# 

# 

# Tratamiento de texto.

# 

Anteriormente aprendimos cómo crear y organizar nuestras carpetas. Ahora vamos a trabajar archivos que, por supuesto, debemos guardar en estos directorios que previamente creamos.

**touch**: nos permite crear archivos.**cat**: nos permite visualizar todo el contenido de nuestros archivos.



**head**: es muy parecido al comando **cat**. También nos permite visualizar el contenido de nuestros archivos, pero debemos indicarle cuántas líneas nos debe mostrar. Por defecto nos mostrará las primeras 10.

**tail**: funciona igual que el comando **head**, pero al revés. También debemos indicarle cuántas líneas nos debe mostrar, la diferencia es que no las mostrará de abajo hacia arriba. Por defecto nos mostrará las últimas 10.

## 

## **Búsqueda y tratamiento de texto.**

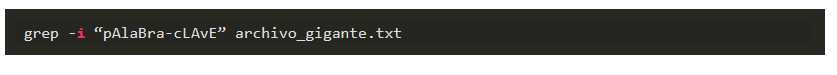
No solo podemos visualizar nuestros archivos (o parte de nuestros archivos) tal cual como escribimos, también podemos filtrar y cambiar el contenido que podemos ver en los archivos.

**Por ejemplo:** imagina que tenemos un archivo gigante, con cientos o incluso miles de líneas. Si imprimiéramos el contenido de todo el archivo sería muy difícil encontrar el nombre de una persona o elemento específico.

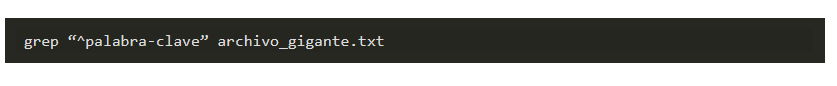
Y se vuelve aún más complicado si necesitamos que las palabras que buscamos cumplan ciertas condiciones, como solo mayúsculas o minúsculas, que la siguiente o anterior palabra cumpla ciertas condiciones, etc.

En estos casos podemos utilizar el comando **grep** para filtrar las líneas que queremos visualizar utilizando (o no) expresiones regulares:

Si nos da igual si la palabra clave incluye mayúsculas o minúsculas podemos utilizar el flag **-i**:

También podemos verificar si la línea incluye esta palabra clave al final:

O si la incluye al principio:



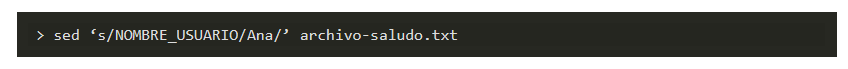
También hay situaciones donde necesitamos modificar un poco la información que obtenemos de un archivo de texto.

Por ejemplo, imagina que nuestro archivo contiene un poema, frase o saludo para responderle a los usuarios de nuestra aplicación. El problema es que cada usuario tiene un nombre diferente.

¡Hola, NOMBRE\_USUARIO! Felicitaciones por completar tu desafío con PUNTOS\_USUARIO puntos.

No queremos editar este archivo. Solo necesitamos cambiar los caracteres **NOMBRE\_USUARIO** por el verdadero nombre del usuario.

Para esto podemos utilizar el comando **sed(Script Editor)**. Solo debemos indicarle que queremos realizar una sustitución (**s/**), la palabra que vamos a cambiar (**NOMBRE\_USUARIO**), la nueva palabra que vamos a incluir (**Ana**) y cerrar con el símbolo **/**.

Ahora imagina que, además del nombre, debemos cambiar también la puntuación que obtuvo el usuario:



**awk:** Trataminento de texto delimitado, este comando sirve para trabajar con archivos de textos delimitados por comas.  
**Ejemplos:**

1. awk -F ‘;’ ‘{ print $1}’ nuevasPelis.csv

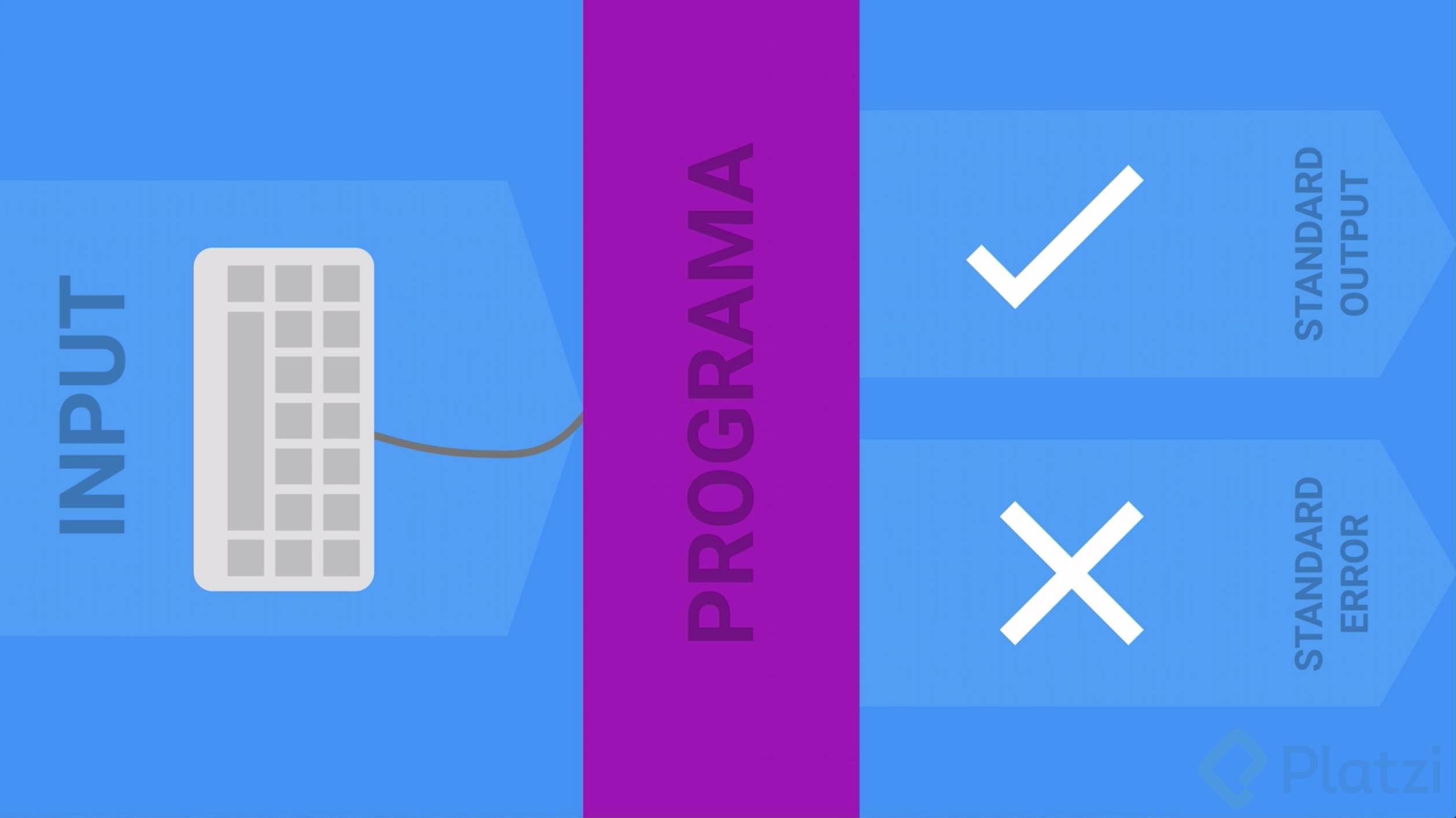
***USAR VIM***

* Ingresamos el comando archivo ***vim archivo.txt***
* Dentro una vez se abra el editor presionamos la tecla I para poder modificar el texto.
* Con ***Esc***  salgo de la edición del archivo.
* Escribo dos puntos para activar el modo de comandos
* Luego de los dos puntos, con ***w*** guardo el contenido del archivo. “ :w “
* ***Control + Q*** salgo del editor de VIM.

# Flujos estándar y cómo los Pipes nos pueden dar super poderes.

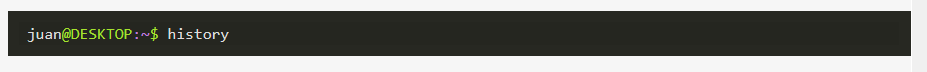
**Antes de enseñarte los poderes que nos podrían proporcionar los Pipes**, me gustaría mencionarte y recordarte que en Linux/Unix cada nuevo proceso generado, es decir, cada nueva aplicación ejecutada, es inicializada con tres canales de datos, canales conocidos cómo streams.

* Standar Input
* Standar Output
* Standar Error



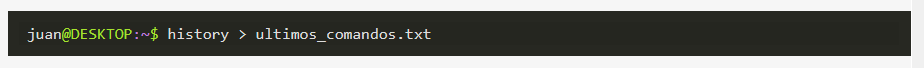
## **TRABAJEMOS CON LOS INPUTS Y OUTPUTS**

Ahora, veamos un ejemplo de cómo podemos sacarle provecho a los stream. El comando history nos permite conocer los últimos comandos ejecutados en nuestra sesión. Basta con ejecutar *history* en la terminal.

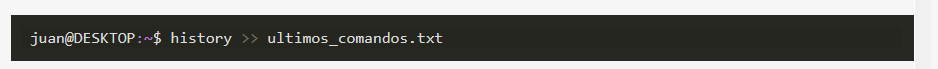


Una vez ejecutado estaremos visualizando en pantalla un listado de todos los comandos que hemos utilizado. Todo el texto que visualizamos (los comandos) se encuentran en el stream del Standard output.

Si nosotros así lo deseamos, podemos almacenar esa salida en un archivo de texto plano. Basta con utilizar el signo mayor **>** que, seguido del nombre del archivo

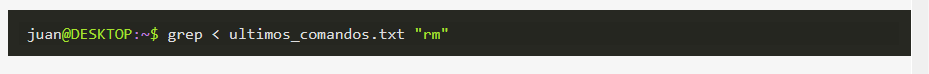


Con el signo mayor que, crearemos el archivo si este no existe, y colocaremos el output del programa allí; En caso que el archivo exista, y esté almacene algún tipo de contenido, el contenido será reemplazado por el output del programa. Si nosotros no queremos perder la contenido que posee un archivo, lo que podemos hacer es utilizar el doble signo mayor que.



Esto hará que el output sea agregado al final del archivo, sin reemplazar nada.

Una vez tenemos el output almacenado, hagamos algo interesante, utilicemos el comando grep para conocer qué archivos o folders has eliminado recientemente.



Al ejecutar esta sentencia estaremos visualizando los últimos comandos que cuentan con rm en su ejecución.

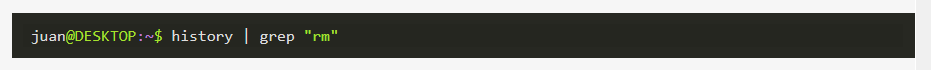
Con el signo menor que, estamos colocando todo el contenido del archivo cómo Standar Input del comando grep. A partir de estos inputs, el programa grep ya puede buscar **rm**

## **PIPES - Un gran poder conlleva una gran responsabilidad 💪**

Existen comandos para muchas cosas, sin embargo, no existe un comando en la actualidad que lo haga todo. Si queremos hacer tareas complejas, es probable que tengamos que apoyarnos de más de un comando.

Regresemos al ejemplo anterior, donde nosotros necesitábamos conocer los archivos o folders eliminados recientemente. Para solucionar la problemática tuvimos que ejecutar el comando history y su output almacenarlo en un archivo, el cual nos sirvió cómo input del programa grep. Es Aquí donde entran los **PIPES** a salvar el día, y hacer nuestro trabajo cómo administrador de servidores más sencillo.

Ejecutemos la siguiente sentencia.



**Obtuviste el mismo resultado que en el ejemplo anterior?** Super interesante, ¿no lo crees? obtuvimos el mismo resultado solo que ahora con una sentencia mucho más corta, legible y sin tener que crear nuevos archivos en nuestro disco duro.

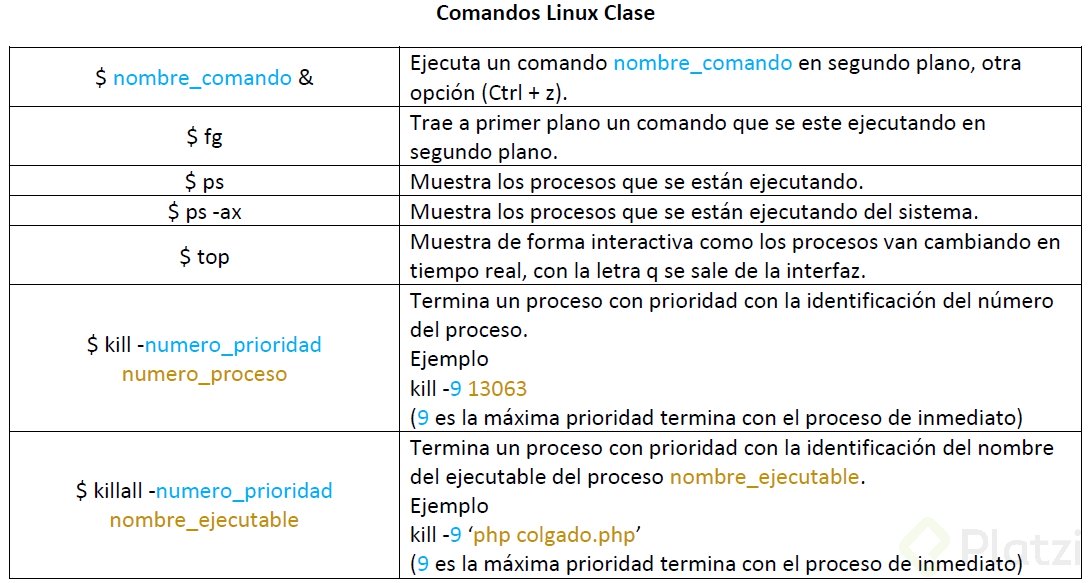
El **PIPE** nos permitirá encadenar la ejecución de programas, pasando el output de uno cómo el input de otro. En este caso estamos colocando el output de history cómo el input de grep. El orden en el que se ejecutarán los programas es de izquierda a derecha y de esa misma manera es cómo se estarán pasados los outputs.

Y así cómo encadenamos la ejecución de dos comandos podemos hacerlo con tres, cinco, o con la ***n*** cantidad que necesitemos. Nosotros no estamos limitados únicamente a encadenar programas Unix/Linux, también podemos encadenar nuestros propios programas, no importa si están escritos en Java, Python, PHP etc… claro, siempre y cuando hayamos tomado en cuentos los streams en su codificación.

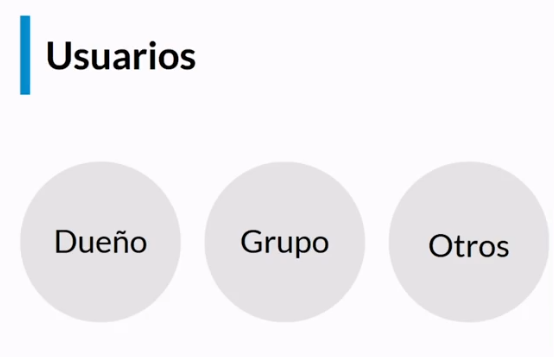
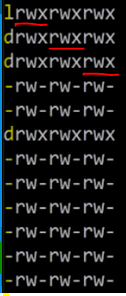
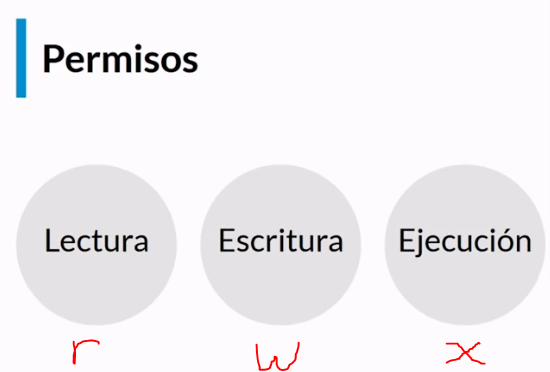
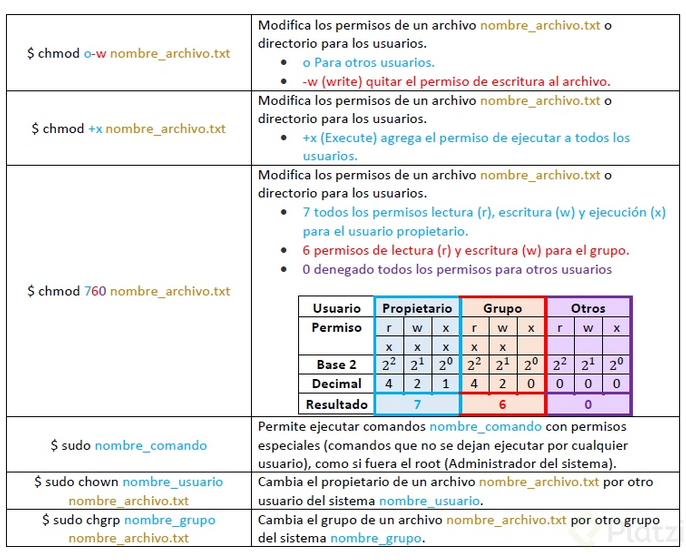
# Administración de procesos en background y foreground.

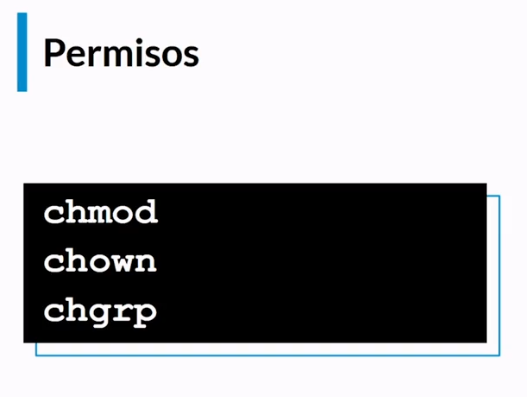
# 

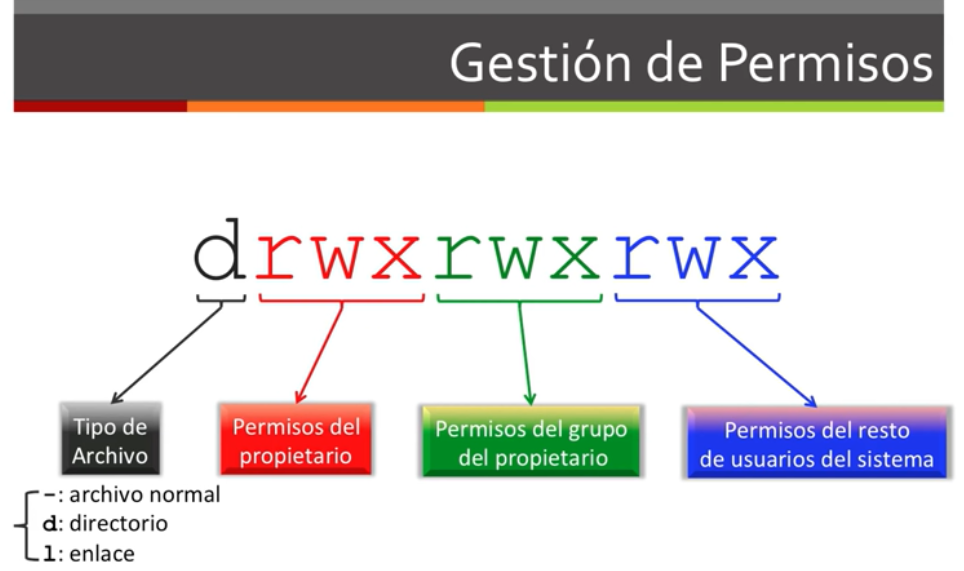
* Procesos
* **En primer plano**: Son aquellos que mientras estén en ejecución, la terminal no permitirá realizar ninguna otra acción.
* **Procesos en segundo plano**: Son aquellas actividades que se están ejecutando mientras nosotros estamos realizando otras actividades en la terminal.
* Para colocar un proceso en segundo plano desde el inicio de su ejecución, se coloca al **final** del toda la línea de comando, el **símbolo de ampersand (&).**
* Para colocar un proceso en segundo plano durante su ejecución, se debe utilizar la combinación teclas: **Ctrl + Z.**
* Para volver a colocar un proceso en primer plano, se debe ingresar el comando “fg”.
* Comando para ver procesos que se estén ejecutando: “ps” o con modificador para ver también procesos del sistema: “ps ax”.
* Comando para ver procesos ejecutando en el sistema en tiempo real: “top”. Para salir de la ejecución del comando se debe oprimir la tecla “q”.
* Cuando sea necesario detener un proceso mientras está en ejecución: Se debe usar la combinación de teclas Ctrl + C.
* Si el proceso está en segundo plano y se requiere detener o es un proceso que está colgado y se debe detener: ejecutar el comando “ps ax” para averiguar el número del comando (Campo PID) y después se ejecuta el comando: “kill -9 #proceso”, lo cual detendrá obligatoriamente el proceso indicado.



# Permisos sobre archivos: El sistema de permisos octal

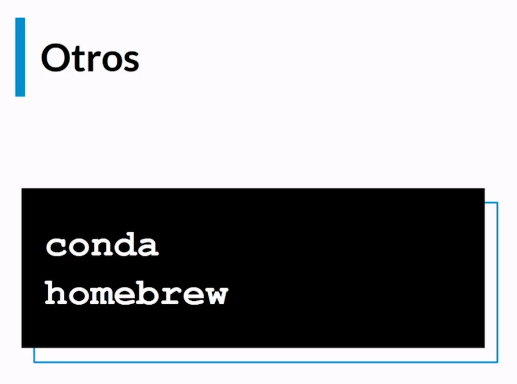
 



***Para información detallada consultar el archivo:***

*Administracion-usuarios-servidores-linux.docx*

# Sistemas de manejo de paquetes.



# Herramientas de compresión y combinación de archivos.

## Generar archivos .gz:

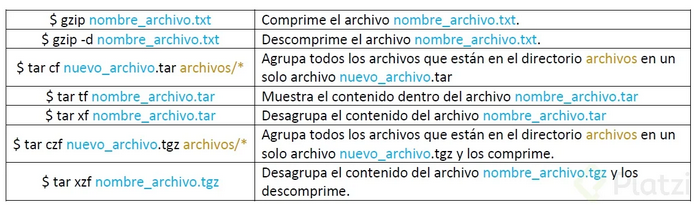
**Comprimir:** gzip archivo.txt  
**Descomprimir:** gzip -d archivo.txt.gz

## Generar Archivos .tar:

**Empaquetar:** tar cf paquete.tar /carpeta/a/empaquetar/  
**Ver contenido del paquete:** tar tf paquete.tar  
**Empaquetar y ver contenido empaquetado:** tar -cvf paquete.tar /carpeta/a/empaquetar/  
**Desempaquetar:** tar xf paquete.tar

## Archivos .tar.gz:

**Empaquetar y Comprimir:** tar czf empaquetado.tar.gz /carpeta/a/empaquetar/y/comprimir  
**Descomprimir:** tar xzf archivo.tar.gz



# Herramientas de búsqueda de archivos.

# 

## **Find un comando con mucho poder 💪**



**Ruta**  
Si no se indica una ruta se toma en cuenta entonces el directorio donde se esté actualmente, es decir el directorio de trabajo actual, que es lo mismo que indicar con un punto **“.”**.  
   
Es posible asignar más de una ruta de búsqueda también como por ejemplo:



**Búsquedas básicas** 👍  
Algunas banderas que podemos utilizar para buscar:

* -name = Busca nombre de un archivo
* iname = Igual que name pero este no toma en consideración si tiene alguna mayúscula
* -user = El usuario propietario
* -group = El grupo propietario
* -type = tipo de archivo, f para directorios

**Búsquedas a través del tiempo** ⏰

* -mmin = Tiempo en minutos
* -mtime = Periodos de 24 horas  
     
  **-exec; El poder aumenta** 👊  
     
  -exec Permite ejecutar acciones sobre el resultado de cada línea o archivo devuelto por find, ejemplo:



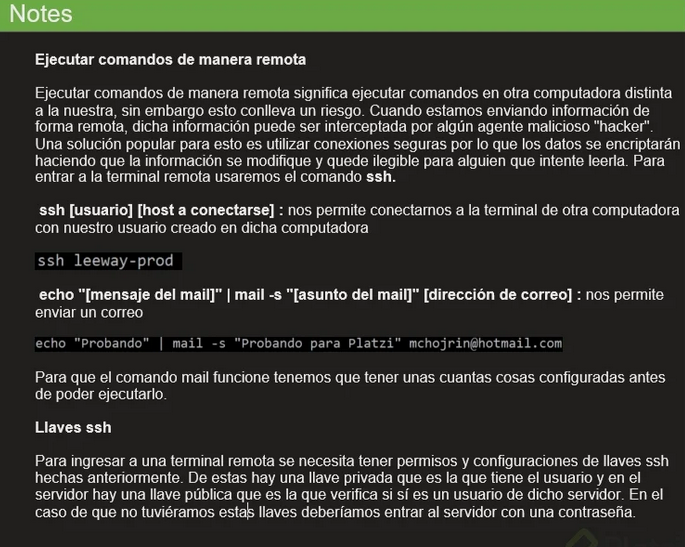
# Herramientas para interactuar a través de HTTP

# 

# CURL – Peticiones en crudo

# WGET – Descargas

# Acceso seguro a otras computadoras.



***Para realizar las configuraciones necesarias para el envio de emails consultar:***

*Configuración de un servicio de mailing.docx*

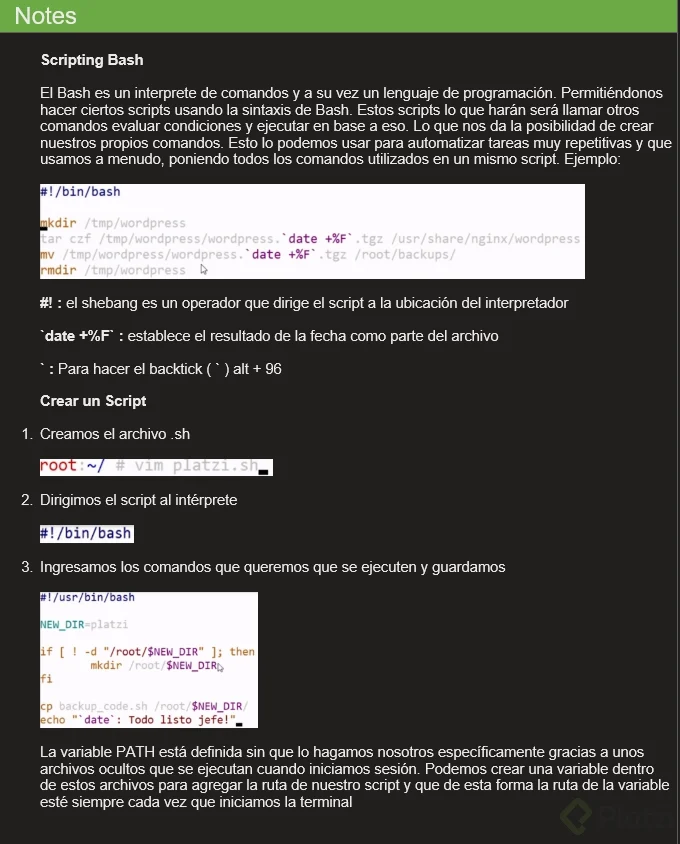
# Qué son y cómo se utilizan las variables de entorno.

# 

Es un definición global a la que todos los procesos tienen acceso. Esta toma más información de lo que se esté typeando.  
Ejemplo: **echo $PATH** = se encuentran todos los comandos ejecutables

**Asignación de las variables de entorno**  
**export:** Este comando se utiliza para asignar a toda la sesión  
Ejemplo: export MI\_VAR = mauro, si luego escribimos **echo $MI\_VAR** se mostrará mauro en la terminal. *(Este permanecerá mientras dure mi sesión)*  
.

# Cómo y para qué escribir scripts en Bash



# Cómo y para qué dejar tareas programadas.

# 

# Para realizar la automatización de tareas usaremos primero *at,* en el que introduciremos primeroel tiempo en el que se va a ejecutar esta tarea y luego las instrucciones, quedando de la siguiente manera: (Primero debemos configurar los servicios ATD, CRON que se ven en la tabla inferior)

# 

# 

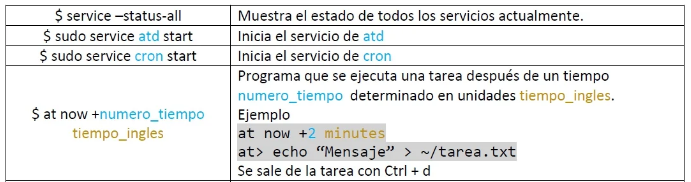
# *Control + D* para salir y tenemos ya nuestra tarea programada.

# Ahora usaremos *cron,* para ello haremos lo siguiente.

# Ejecutamos *crontab -e* que muestra las tareas que están programadas y en donde las podremos editar. Nos abrirá el editor de texto donde ingresaremos nuestra tarea agregando al inicio minuto-hora-dia\_mes-mes-dia\_semana y luego el comando.

# 

# 

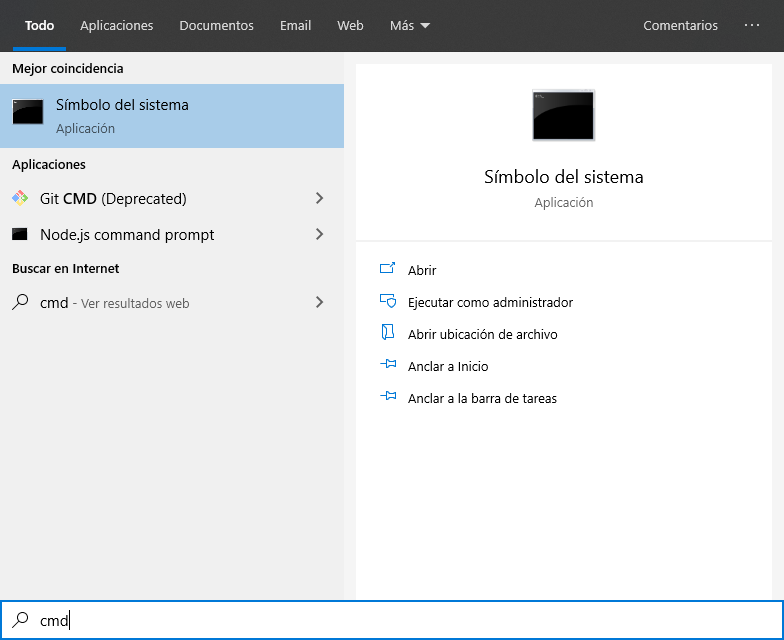


# Cómo se ejecutan los comandos estudiados en Windows y MacOS .

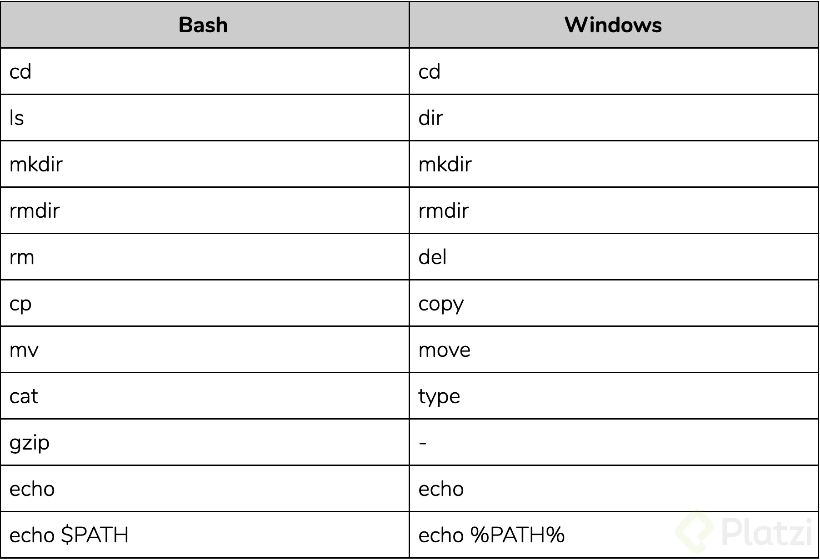
Además del terminal que utilizamos durante el curso ([bash](https://es.wikipedia.org/wiki/Bash)), existen muchos otros. Solamente en ambientes tipo \*nix (Linux, BSD, etc…) puedes encontrarte con [Korn-Shell](https://es.wikipedia.org/wiki/Korn_shell) y [Zsh](https://es.wikipedia.org/wiki/Zsh).

Se trata de otros intérpretes que incorporan diferentes atajos, pero que esencialmente permiten realizar las mismas tareas, ya que en el fondo simplemente ejecutan los mismos comandos que están presentes en el sistema.

Una terminal bastante diferente de la que vimos en el curso es la que viene de fábrica junto con Windows: el programa CMD.

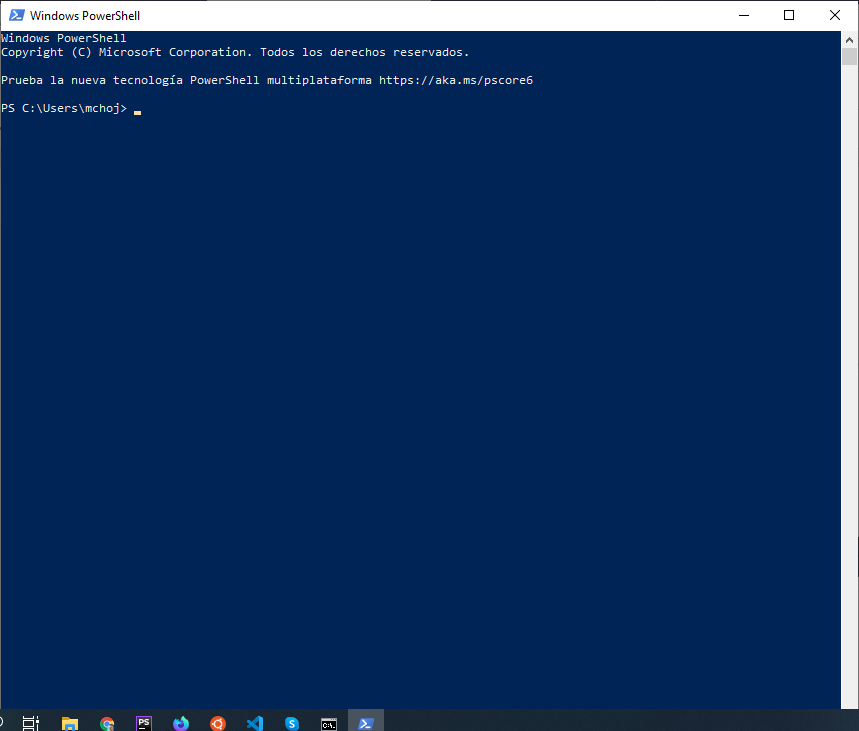


Esta terminal está basada en el viejo [DOS](https://es.wikipedia.org/wiki/MS-DOS), algunos comandos son iguales, otros son similares:



En esta consola puedes también crear scripts, claro que no serán basados en la sintaxis de bash, sino en la sintaxis [batch](http://www.cristalab.com/tutoriales/programacion-batch-con-archivos-.bat-c48410l/).

Otra consola interesante que trae Windows es PowerShell:



Una vez allí puedes utilizar todas las herramientas básicas de la terminal CMD, además de una serie de herramientas avanzadas que puedes encontrar con sólo escribir get-command y dar enter.

Puedes ver más información sobre esta terminal [aquí](https://esgeeks.com/como-usar-windows-powershell-guia-basica/).

Los comandos básicos de MacOS son los mismos que viste en el curso, aunque podrían tener ligeras variaciones en cuanto a sus modificadores.

Lo importante es que hayas aprendido una nueva forma de interactuar con tu computadora, mucho más eficiente y, quién te dice, cómoda. 😃

***Curso de Expresiones Regulares.***

**¿Qué son las expresiones regulares? (RegEx)**

Las expresiones regulares son patrones de caracteres que te permiten ir seleccionando o descartando datos en un archivo de texto como por ejemplo csv, o en una línea o un input, según coincidan o no con este patrón.

Prácticamente todos los lenguajes de programación tienen librerías o módulos para manejar expresiones regulares.

Las expresiones regulares pueden ser muy complejas, pero no son nada difíciles de entender.

A través de este curso, sin tecnicismos y con ejemplos puntuales, vamos a aprender a utilizarlas para que sean esa herramienta que siempre nos ayude, y sea la primera para solucionar problemas de grandes cantidades de datos en string.

Las expresiones regulares pueden llegar a ser muy raras por la forma en la que se ven, pero son muy útiles. Aprender a usarlas nos ayuda en pocas palabras, a buscar. Se diferencia del CTRL+F porque éste busca textos precisos y te arroja el match. Con expresiones regulares es más complejo, por ejemplo, se pueden buscar patrones (buscar todas las palabras que estén entre dos espacios, palabras que empiecen con mayúscula, encontrar la primera palabra de cada línea, etc.)

Se usa mucho en logs de servidores que son archivos enormes para analizarlos

Las expresiones regulares son usadas tanto en frontend como en backend.

# El carácter (.)

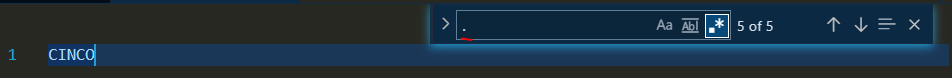
¿Qué es un archivo de texto, por ejemplo un CSV?  
¿Qué es una cadena de caracteres?

Cada espacio en una cadena de texto se llena con un carácter, esto lo necesitamos tener perfectamente claro para comenzar a trabajar con expresiones regulares

Abriremos nuestro editor, qué en este curso recomendamos ATOM, vamos a presionar CTRL + F y podremos buscar por match idénticos.

**Cadena de Caracteres:** Es un carácter ASCII generalmente, seguido de otro carácter y de otro. No todos son visibles, el espacio por ejemplo.

**El carácter (.):** Encuéntrame todo lo que sea un carácter



# Cheat Sheet:

[*https://cheatography.com/davechild/cheat-sheets/regular-expressions/*](https://cheatography.com/davechild/cheat-sheets/regular-expressions/)

***Para consultar:***

[*http://w3.unpocodetodo.info/utiles/regex.php*](http://w3.unpocodetodo.info/utiles/regex.php)

# Las clases predefinidas y construidas

Las búsquedas en las expresiones regulares funcionan en múltiplos de la cantidad de caracteres que explícitamente indicamos.

**Dígitos:** \d

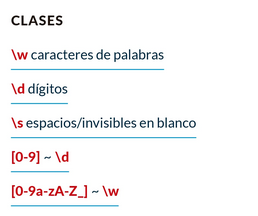
* Encuentra todos los dígitos de 0 a 9.
* \d es equivalente a poner [0-9]. En este caso “[0-9]” sería una clase construida que abarca esos números de 0 a 9.
* Si en vez de \d, usamos por ejemplo [0-2] nos encontrará solamente los dígitos de 0 a 2.
* Podemos usar “\D” para encontrar justo lo contrario, todo lo que no son dígitos.

**Palabras:** \w

* Encuentra todo lo que puede ser parte de una palabra, tanto letras (minúsculas o mayúsculas) como números.
* \w es equivalente a poner [a-zA-Z0-9\_].
* Si en vez de \w, usamos por ejemplo [a-zA-Z] nos encontrará solamente las letras.
* Podemos usar “\W” para encontrar justo lo contrario, todos los caracteres que no son parte de palabras.

**Espacios:** \s

* Encuentra todos los espacios (los saltos de línea también son espacios).
* Podemos usar “\S” para encontrar justo lo contrario, todo lo que no son espacios.

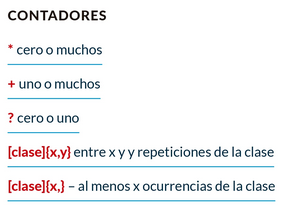


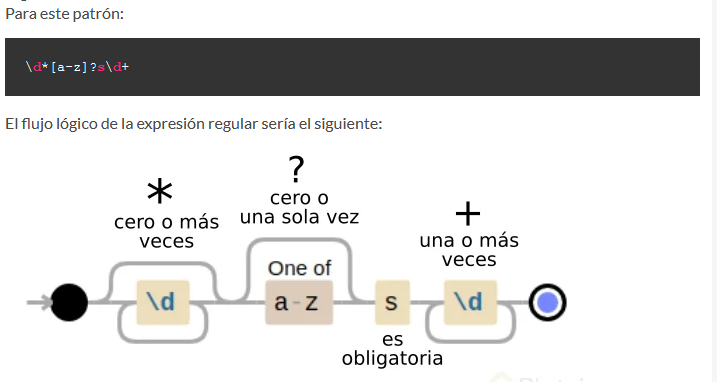
# Los delimitadores: \* , ? , +

* (\*) : Cero o más veces
* (?): Cero o una sola vez
* (+): una o más veces.

Aplican al carácter o sentencia que preceden

* **[a-z]?** : Esto es que puede estar ***una sola vez*** o ***no estar*** una letra minúscula de la **(a)** a la **(z)**.
* **\d**\*: Esto es que puede estar ***muchas veces*** o ***no estar*** un **digito**.
* **\d+**: Esto es que puede estar ***muchas veces*** o ***una sola vez*** un **digito**.





# Los contadores {1,4}

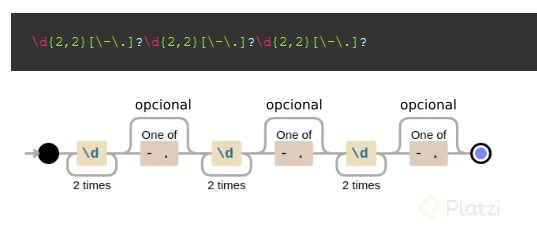
Lo que vamos a aprender en esta clase es comenzar a generalizar nuestras búsquedas, a ser específicos cubriendo grandes cantidades de caracteres sin tener que escribir de forma repetitiva como sería poner por ejemplo “\d\d\d\d\d\d\d\d…”

Aplicando a un carácter que lo preceda se puede colocar entre llaves de esta forma, para indicarle que busque la cantidad deseada de caracteres.

{**Cota inferior**, **Cota superior**}

**Ejemplo:**

* **\d{0,2}:** Esto buscara 0, 1, 2 dígitos
* **\d{2,2}:** Esto buscara 2 dígitos
* **\d{2}:** Esto buscara 2 dígitos
* **\d{2,}:** Esto buscara 2 o más dígitos.
* **\d{2,4}:** Esto buscara 2, 3, 4 dígitos.



# El caso de (?) como delimitador

El ? indica al patrón que encuentre las coincidencias de manera rápida (o lazy); es decir, devolviendo el resultado más pequeño que haga match hasta donde se encuentra el delimitador, y esto lo haga tantas veces como sea posible dentro de la cadena.

\*? Coincide con el elemento anterior cero o más veces, pero el menor número de veces que sea posible.  
+? Coincide con el elemento anterior una o más veces, pero el menor número de veces que sea posible.   
?? Coincide con el elemento anterior cero o una vez, pero el menor número de veces que sea posible.

\*\*La función de (?) como delimitador es justamente en delimitar a la menor cantidad posible de los matches \*\*

Los matches los hace lo más pequeños posibles.  
Es decir: Haz el match, pero los divides en grupos pequeños.

Ejemplo:



Encuentra todos los caracteres y hace matches pequeños.

# Not (^) y su uso.

Este carácter nos permite negar una clase o construir “anticlases”, vamos a llamarlo así, que es: toda la serie de caracteres que no queremos que entren en nuestro resultado de búsqueda. Para esto definimos una clase, por ejemplo: [ 0-9 ], y la negamos [ ^0-9 ] para buscar todos los caracteres que coincidan con cualquier carácter que no sea 0,1,2,3,4,5,6,7,8 o 9

**\S:** Va a encontrar todo lo que NO sea carácter visible

**\W**: Me va a encontrar todo lo que no sea alfanumérico

**\D**: Me va a encontrar todo lo que no sea un dígito

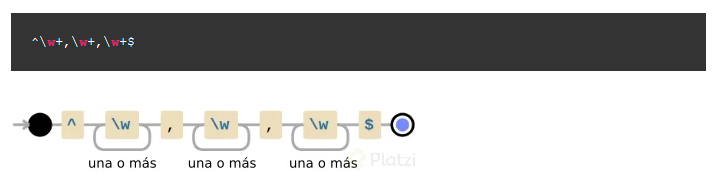
**MAS CLASES PREDETERMINADAS**

**\t** — Representa un tabulador.  
**\r** — Representa el “retorno de carro” o “regreso al inicio” o sea el lugar en que la línea vuelve a iniciar.  
**\n** — Representa la “nueva línea” el carácter por medio del cual una línea da inicio. Es necesario recordar que en Windows es necesaria una combinación de \r\n para comenzar una nueva línea, mientras que en Unix solamente se usa \n y en Mac\_OS clásico se usa solamente \r.  
**\a** — Representa una “campana” o “beep” que se produce al imprimir este carácter.  
**\e** — Representa la tecla “Esc” o “Escape”  
**\f** — Representa un salto de página  
**\v** — Representa un tabulador vertical  
**\x** — Se utiliza para representar caracteres ASCII o ANSI si conoce su código. De esta forma, si se busca el símbolo de derechos de autor y la fuente en la que se busca utiliza el conjunto de caracteres Latin-1 es posible encontrarlo utilizando “\xA9”.  
**\u** — Se utiliza para representar caracteres Unicode si se conoce su código. “\u00A2” representa el símbolo de centavos. No todos los motores de Expresiones Regulares soportan Unicode. El .Net Framework lo hace, pero el EditPad Pro no, por ejemplo.  
**\d** — Representa un dígito del 0 al 9.  
**\w** — Representa cualquier carácter alfanumérico.  
**\s** — Representa un espacio en blanco.  
**\D** — Representa cualquier carácter que no sea un dígito del 0 al 9.  
**\W** — Representa cualquier carácter no alfanumérico.  
**\S** — Representa cualquier carácter que no sea un espacio en blanco.  
**\A** — Representa el inicio de la cadena. No un carácter sino una posición.  
**\Z** — Representa el final de la cadena. No un carácter sino una posición.  
**\b** — Marca la posición de una palabra limitada por espacios en blanco, puntuación o el inicio/final de una cadena.  
**\B** — Marca la posición entre dos caracteres alfanuméricos o dos no-alfanuméricos.

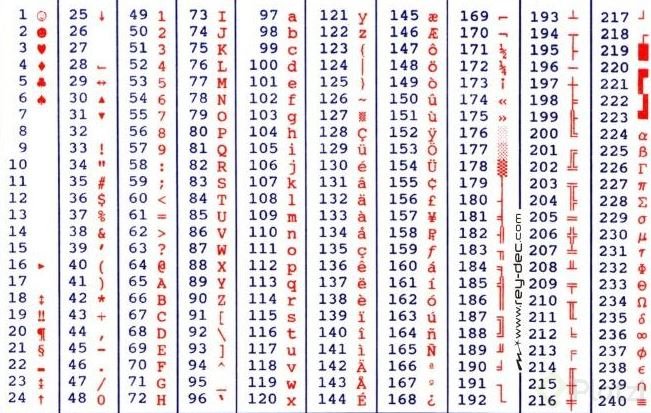
# Principio (^) y final de línea ($)

Estos dos caracteres indican en qué posición de la línea debe hacerse la búsqueda:  
el ^ se utiliza para indicar el principio de línea  
el $ se utiliza para indicar final de línea

^ ------------- $

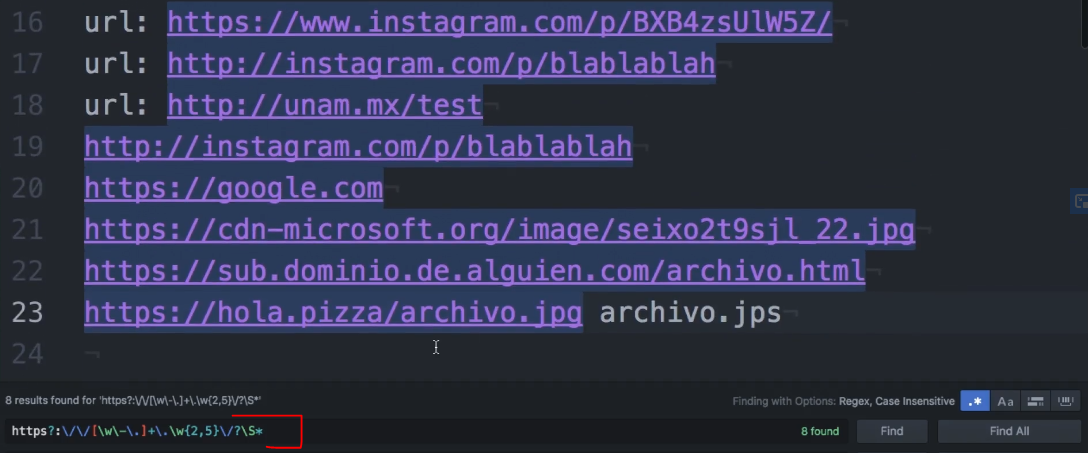


**Códigos ASCII.**



# URLs

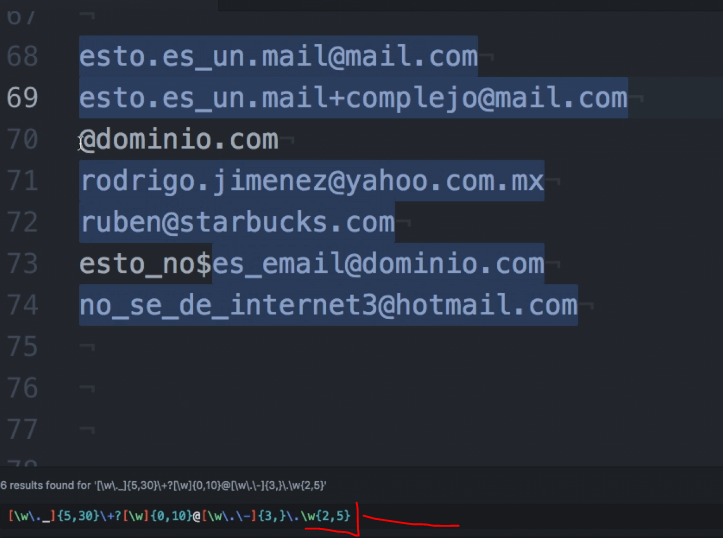
Una de las cosas que más vamos a usar en la vida, seamos frontend o backend, serán directamente dominios o direcciones de internet; ya sea direcciones completas de archivo (una url) o puntualmente dominios para ver si es correcto un mail o no.

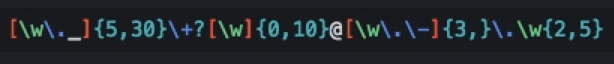




# Mails

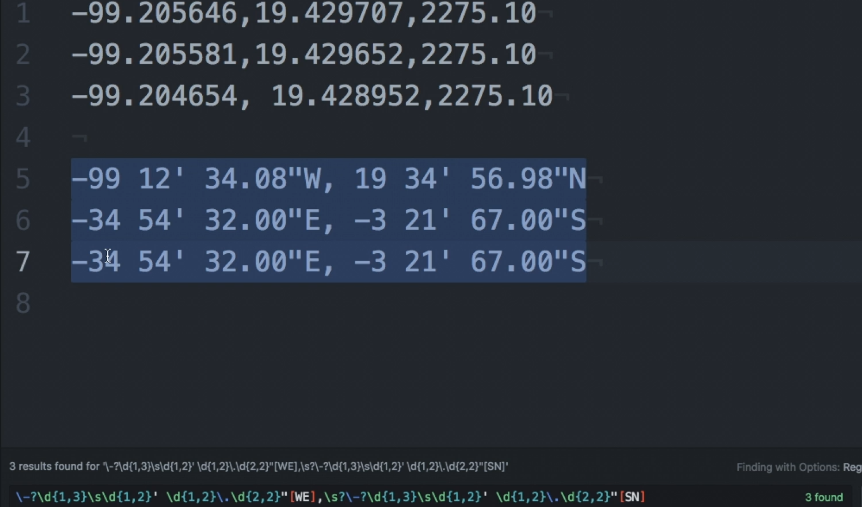
Quedamos en que ya podemos definir URLs, y dentro de las URLs están los dominios. No es infalible, pero es muy útil para detectar la gran mayoría de errores que cometen los usuarios al escribir sus emails.





# Locaciones

Esta clase nos va a servir para ver unos tips comunes de qué hacer y sobre todo qué no hacer con expresiones regulares, usando como ejemplo datos de posicionamiento en el mapa: latitud y longitud.

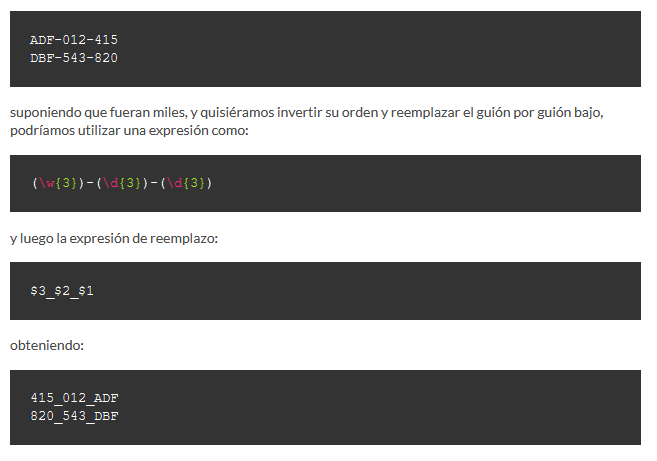




# Búsqueda y reemplazo

Al igual que una navaja suiza, las expresiones regulares son una herramienta increíblemente útil pero tienes que darle la importancia y las responsabilidades adecuadas a cada una, ya que no son la panacea, no solucionan todos los problemas.

El uso más conveniente de las expresiones regulares es buscar coincidencias o matches de cadenas en un texto, y si es necesario, reemplazarlas con un texto diferente.



***Para practicar:*** [***https://regexone.com/***](https://regexone.com/)

## **Banderas**

Las expresiones regulares pueden tener banderas que afectan la búsqueda, éstas deberán de estar hasta el final de la línea.

[**Listado de Banderas en js:**](https://javascript.info/regexp-introduction#flags)  
**i**  
Con este indicador, la búsqueda no distingue entre mayúsculas y minúsculas: no hay diferencia entre A y a

**g**  
Con esta marca, la búsqueda busca todas las coincidencias, sin ella, solo se devuelve la primera coincidencia.

**m**  
Modo multilínea

**s**  
Habilita el modo “dotall”, que permite un punto. para que coincida con el carácter de nueva línea \ n

**u**  
Permite el soporte completo de Unicode. La bandera permite el procesamiento correcto de pares sustitutos.

**y**  
Modo “adhesivo”: búsqueda en la posición exacta del texto

# `grep` y `find` desde consola

En los sistemas operativos basados en UNIX podemos utilizar expresiones regulares a través de la consola mediante los comandos grep y find.

* **grep:** Nos ayuda a buscar dentro de los archivos, textos muy puntuales utilizando una versión muy reducida de expresiones regulares.
* **find:** Se utiliza para encontrar archivos en un determinado directorio a partir de diversas reglas de búsqueda que incluyen expresiones regulares.





**Importants Q&A**

El símbolo que denota cualquier carácter es**: “ . ” (dot)**

Si quisiera hacer que una expresión no distinga entre mayúsculas y minúsculas, debo agregarle la bandera: **-i**

El patrón *a.* significa exactamente: **Una *a* o cualquier otro carácter**

El patrón a\* significa exactamente: **Cero o más *a***

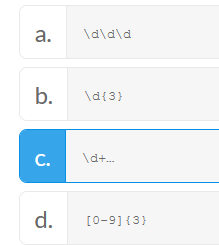
El patrón a? significa exactamente: **Cero o solo una *a***

El patrón a+ significa exactamente: **Una o más *a***

La clase \w es equivalente a: **[a-zA-Z0-9]**

El carácter que denota cualquier fin de línea es: **\$**

Si quiero encontrar grupos de sólo 3 caracteres numéricos, cuál de las siguientes expresiones NO funciona:

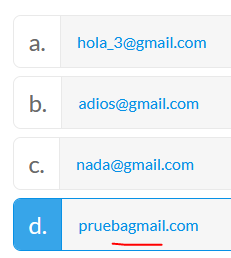


Una expresión greedy significa que se intentará encontrar el patrón: **Tantas veces como sea posible**

La expresión ^1.\* encuentra: **Una línea que empiece con 1**

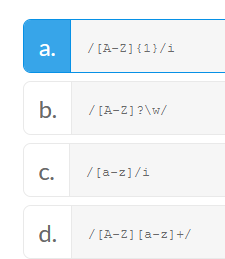
Cuál de las siguientes líneas SÍ hará match con la expresión



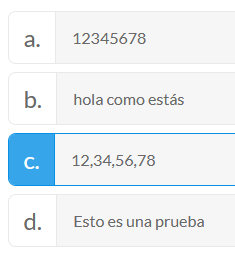


Cuando tenemos un problema y lo planeamos solucionar con expresiones regulares, entonces: **Tardaremos un poco más en solucionarlo, pero tendremos una solución robusta y duradera.**

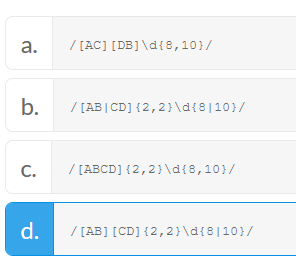
¿Cuál de las siguientes expresiones sería útil para encontrar palabras que empiecen con letra mayúscula?



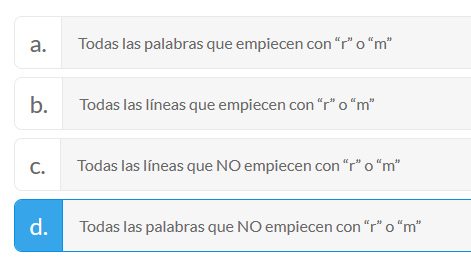
Cuál de las siguientes líneas NO hará match con la expresión /^[\w\s]+$/:



Supongamos que tenemos un archivo csv con códigos de producto, todos los códigos son dos caracteres seguidos por 8 a 10 dígitos, pero sólo queremos los productos que empiecen con AB o CD, ¿qué expresión usarías?



La expresión [^rm][a-z]+ encontrará:



El patrón /(\d+?),.\*/ qué match tendría? (representando al match de la forma [ab]):

