Bash

TB022 - Esteban - Riesgo - Kristal

Qué es Bash?

Bash es el *shell* por defecto (y más usado) de los sistemas Linux modernos.

Es un acrónimo de Bourne Again SHell, desarrollado en 1989 a partir de la base de Bourne Shell (creado en 1979).

Es básico, pero tiene todo lo necesario para armar scripts complejos y poderosos.

Todo lo que veamos en la materia va a estar cubierto por Bash. Sin embargo, …

ZSH

Es otro *shell* basado en Bourne Shell creado en 1990 que además toma cosas de ksh y tcsh (otros shell menos conocidos).

Tiene features muy útiles como auto completado programable, syntax highlighting, perfiles, entre otras, que la hace (en algunos aspectos) superior a Bash en cuanto a la experiencia de usuario.

Es cada vez más usado. Es el shell por default en macOS desde macOS Catalina y en Kali Linux (distribución diseñada para la ciencia forénsica informática y *penetration testing*).

Recomendamos usar ZSH en vez de Bash (configurando algunas cosas).

Instalar ZSH

Para usar ZSH es necesario instalar dos cosas:

ZSH

El shell en sí mismo

Guía de instalación

Oh My Zsh

Oh My Zsh es un framework *open source* para manejar la configuración de Zsh. Permite setear temas de estilo, plugins y más para facilitar el uso y personalizar el espacio propio al máximo.

Guía de instalación

Themes (opcional)

Hay muchos, los más conocidos hoy en día son powerlevel10k (ya no está en mantenimiento) y Starship

ZSH

Disclaimer:

Tiene un par de diferencias con Bash (por ejemplo, los índices de arrays empiezan en 1), pero son configurables.

Además para correr scripts con el Shebang podemos asegurarnos de que corra bash, mientras que para operaciones que solemos hacer en la consola nos es más práctico.

Como saber si estoy usando ZSH correctamente?

Correr en la terminal

```
$ echo $SHELL
```

ZSH

ZSH +
Oh My Zsh +
powerlevel10k

vs

BASH

Variables

Podemos crear y asignar valor a variables usando el operador `=`

Notar la falta de espacios

```
$ VARIABLE="Hello World"
```

Las `"` son opcionales si no hay espacios en el valor de la variable.

Para referenciar una variable debemos usar el operador `$`

```
$ echo $VARIABLE
Hello World
```

Si la variable no existe y se referencia, no habrá error, simplemente tendrá valor vacío `""`

```
$ echo $NOEXISTE
```

Podemos insertar el valor de una variable en una cadena

```
$ VARIABLE="Juani"
$ echo "Hola $VARIABLE"
Hola Juani
```

Estas variables tienen como scope el shell en el cual se definan

Para imprimir por stderror: `>&2 echo "Mensaje de error”`

Variables

Hay distintos tipos:

cadenas

```
$ var="cualquier cosa entre comillas"
```

números

```
$ var=123
```

arrays

```
$ var=(1 dos 3)
```

Para operar números debemos usar *compound notation*

```
$ var=$((1 + 2))
$ echo $var
3
```

Las variables se suelen escribir en minúscula y `snake_case`

Arrays

Declarar

```
ARRAY=(uno 2 tres 4 mil)
```

Ver la longitud

```
${#ARRAY[@]}
```

Ver elemento en posición *i*

```
${ARRAY[i]}
```

Agregar un elemento al final

```
ARRAY+=(2000)
```

Exit Status

Todos los comandos al finalizar le indican al SO si la operación fue exitosa o no.

Si el exit status de un comando es `0` significa que terminó correctamente. Si es cualquier otro número (el rango va de 0 a 255) significa que hubo un error. El manual nos dirá qué significa cada número de error.

Para ver el valor de salida de un comando debemos usar `$?`

```
$ ls -d /usr/bin
/usr/bin
$ echo $?
0
$ ls -d /bin/usr
ls: cannot access /bin/usr: No such file or directory
$ echo $?
2
```

Control de Flujo: IF

```
if [ comando ]; then
    <hacer algo>
elif [ comando ]; then
    <hacer algo>
else
    <hacer algo>
fi
```

`[]` en realidad es azúcar sintáctico para un comando llamado `test`

`test` devuelve `0` si la expresión es verdadera y `1` si no lo es

No existe el tipo booleano. Hay dos comandos, `true` y `false` que siempre devuelven 0 y 1 respectivamente.

`[[]]` es algo nuevo que introdujo Bash (no presente en sh) y nos da más libertades para facilitarnos el desarrollo. Más explicación acá.

Recomendamos siempre usar `[[` en vez de `[`.

Control de Flujo: IF operadores

`=` o `==` para chequear igualdad en **strings**

`-eq`, `-lt`, `-le`, `-gt`, `-ge`, `-ne` para comparar valores numéricos

`-e` valida si el archivo o directorio existe

`-f` valida si el archivo existe y es regular

`-r`, `-w` valida si el ejecutor del comando tiene permisos de lectura o escritura

`-a`, `-o` and y or

Para ver la lista entera y ver cómo usarlos entrar al manual `man test` o ver en este link

Hay precedencia y cortocircuito como en Python.

Control de Flujo: FOR

```
for <var> in {INICIO..FIN..PASO}
do
  <hacer algo>
done
```

Crea un rango de valores desde INICIO hasta FIN **incluidos** avanzando de a PASO cada vez *(el ..PASO existe sólo a partir de bash4)

```
for <var> in 1 2 3
```

`break` y `continue` se pueden usar para controlar el flujo. Terminar prematuramente, o pasar a la siguiente ejecución.

```
for <var> in file1 file2 file3
```

Control de Flujo: FOR

Iterar un array

```
for elem in ${ARRAY[@]}; do
     echo $elem
done
```

Control de Flujo: WHILE & UNTIL

```
until [ <condicion> ]; do
    <hacer algo>
done
```

`<hacer algo>` se ejecutará *hasta que* `[<condicion>]` devuelva un valor de `0`, es decir, *mientras que* `[<condicion>]` devuelva `1`.

```
while [ <condicion> ]; do
    <hacer algo>
done
```

`<hacer algo>` se ejecutará *mientras que* `[<condicion>]` devuelva `0`.

`break` y `continue` se pueden usar para controlar el flujo. Terminar prematuramente, o pasar a la siguiente ejecución.

While: leer archivos

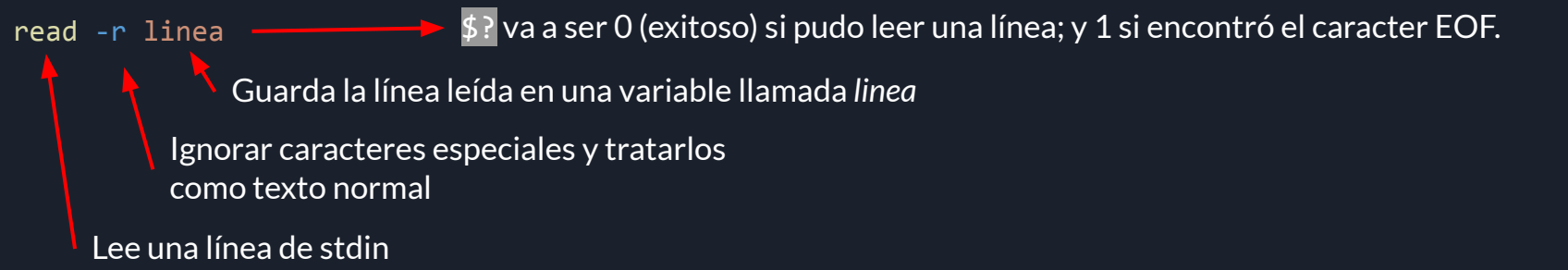
Uno de los principales usos que le podemos dar al ciclo while es para leer las líneas de un archivo (o de stdin) una por una.

Un archivo es una cadena de texto que termina con un caracter especial “invisible” llamado **EOF**.

Si estamos en *stdin*, lo podemos insertar usando Ctrl + D.

Bash no lo muestra y hasta se ve que imprime al lado. En Zsh se puede configurar que lo marque con un # y que agregue un salto de línea.

El comando read que permite leer una línea de stdin:



While: leer archivos

Uno de los principales usos que le podemos dar al ciclo while es para leer las líneas de un archivo (o de stdin) una por una.

Para lograrlo necesitamos:

- Usar el comando read que permite leer una línea de stdin
- Como read solo lee de stdin, hay que redireccionarle el contenido de un archivo usando <
- Considerar que el contenido del archivo puede tener caracteres especiales como \ o espacios/líneas en blanco, por lo que debemos indicarle al comando que los procese adecuadamente

```
while IFS= read -r linea; do
    <algo>
done < "$file"
```

Setear el separador de campo como vacío, para no ignorar espacios en blanco ni líneas vacías.

Redirigimos el contenido del archivo al stdin.

* En cualquier momento puedo correr en bash: `VARIABLE=algo comando` y hace que esa variable valga eso sólo durante la ejecución de ese comando para lo que ve ese comando.

IMPORTANTE: asi como está, se saltea la última línea del archivo a leer. *`|| [[ -n $linea ]]` para leer una no vacía.

Leer de archivo o STDIN

Es común para un comando o script que si no se especifica de qué archivo leer, lea de stdin.

Para lograr esto podemos usar **parameter expansion** especificando un valor default

```
${VAR:-default_value}
```

Si la variable VAR no está definida o es vacía, se usa "default_value"

En nuestro caso podemos hacer lo siguiente:

```
while IFS= read -r linea; do
    <algo>
done < "${file:-/dev/stdin}"
```

Si no le pasamos un archivo válido, lee de standard input (el archivo `/dev/stdin` es un archivo especial que referencia al stdin)

Generación de un bash script

1. Crear un archivo nuevo que va a contener el script (a veces es convención que el archivo termine en `.sh`). Los editores de texto permiten crear archivos, si no se puede usar el comando `touch`
2. Darle permisos de ejecución al archivo con `chmod +x` si no los tiene
3. Editar el archivo usando un editor de texto
4. Indicar en la primer línea del archivo qué programa queremos usar para interpretar el script (*shebang*)

   `#!/usr/bin/env bash` para que funcione en cualquier máquina con cualquier OS.
5. Ejecutar el archivo indicando el path del mismo

Cada script tendrá como *exit status* el estado del último comando ejecutado.

Funciones: declaración

```
function <nombre> {
   <hacer algo>
}
```

```
<nombre>() {
   <hacer algo>
}
```

```
hola_mundo() {
   echo "Hola Mundo!"
}
```

```
$ hola_mundo
Hola Mundo!
```

Se ejecutan llamando a su nombre sin nada especial

Funciones: uso de variables

```
var1='A'
var2='B'

mi_func () {
    local var1='C'
    var2='D'
    echo "Adentro: var1: $var1, var2: $var2"
}

echo "Antes: var1: $var1, var2: $var2"

mi_func

echo "Despues: var1: $var1, var2: $var2"
```

Adentro: var1: C, var2: D

Antes: var1: A, var2: B

Despues: var1: A, var2: D

Funciones: devolver valores

El valor de retorno o exit status de una función indica si la ejecución fue exitosa o no. No podemos usarlo para devolver valores que luego sean usados en el flujo del programa.

Podemos asignar variables globales

```
mi_func () {
   res="resultado"
}

mi_func
echo $res
```

Podemos usar `$()` para hacer *command substitution*

```
mi_func () {
   local res="resultado"
   echo "$res"
}

res="$(mi_func)"
echo $res
```

command substitution significa "correr este comando y lo que salga por *stdout* colocar aquí

Command substitution

command substitution significa "correr este comando y lo que salga por *stdout* colocar aquí

```
mi_func () {
    local res="resultado"
    echo "$res"
}

res="$(mi_func)"
echo $res
```

```
nombre=Juani
echo "Hola, ($nombre)asd"
```

Si no pusieramos los paréntesis, intentaría buscar el valor de la variable `nombreasd` y devolvería vacío.

NO confundir `$(var)` con `($var)`:
`($var)` busca el valor de la variable var y lo pone entre los paréntesis
`$(var)` ejecuta el comando var

Funciones: argumentos

```
var1='A'
var2='B'

mi_func () {
   echo "Adentro: var1: $1, var2: $2"
}

mi_func $var1 $var2
```

Adentro: var1: A, var2: B

Los argumentos son SIEMPRE POSICIONALES

Y `$0` ? Qué valor tiene?

`$#` contiene el número de argumentos usados
`$*` contiene la *lista* de argumentos usados

Es el nombre del scope que está siendo ejecutado. En este caso `mi_func`

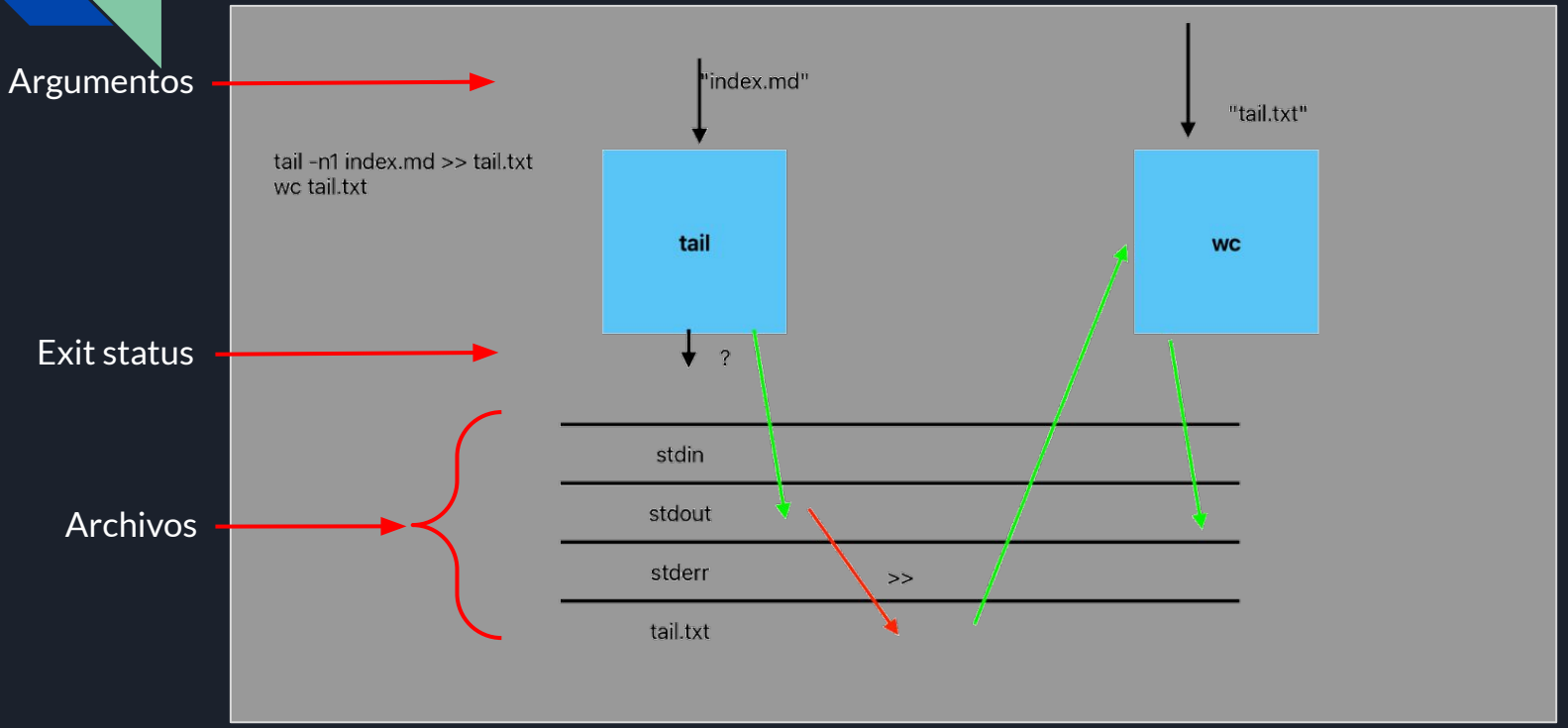
Argumentos

Los scripts también pueden recibir argumentos

```
$ ./mi_script.sh $var1 $var2
```

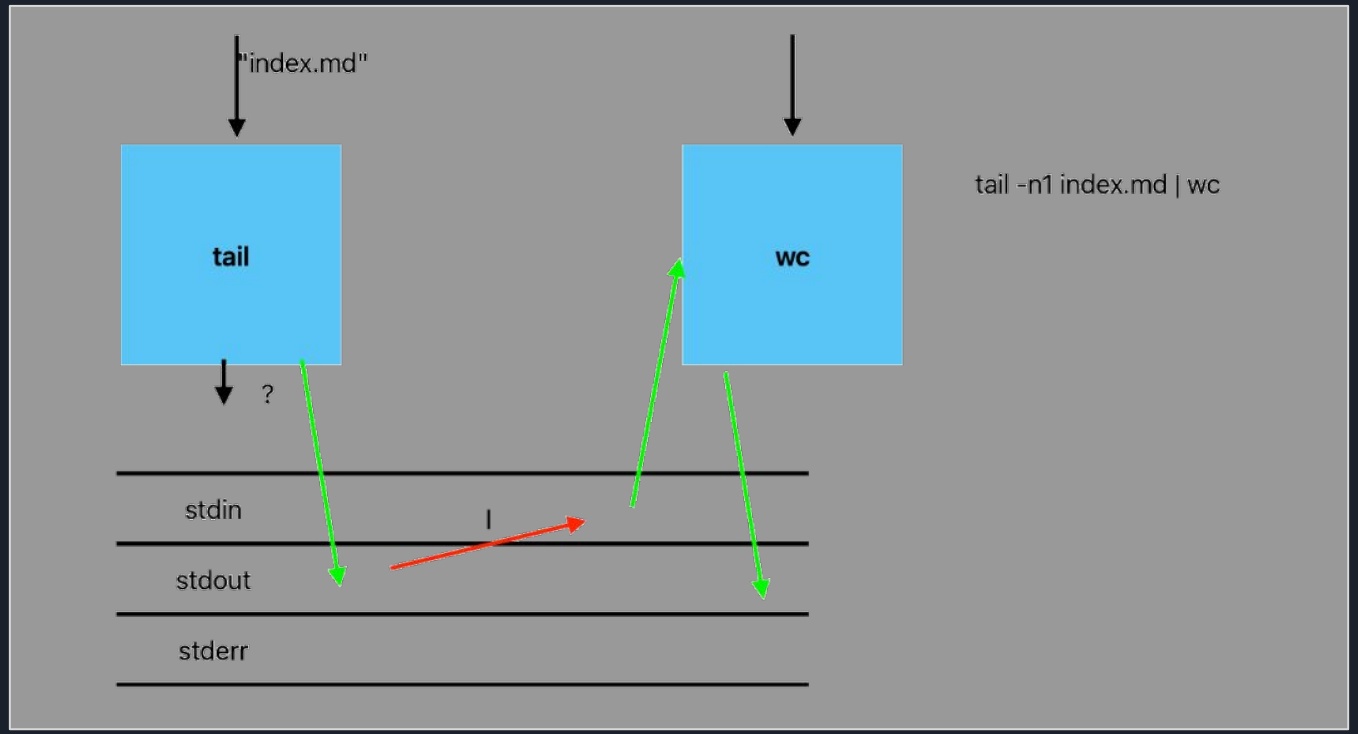
Se cumplen las mismas reglas ya mencionadas para funciones.

Resumen comunicación entre procesos



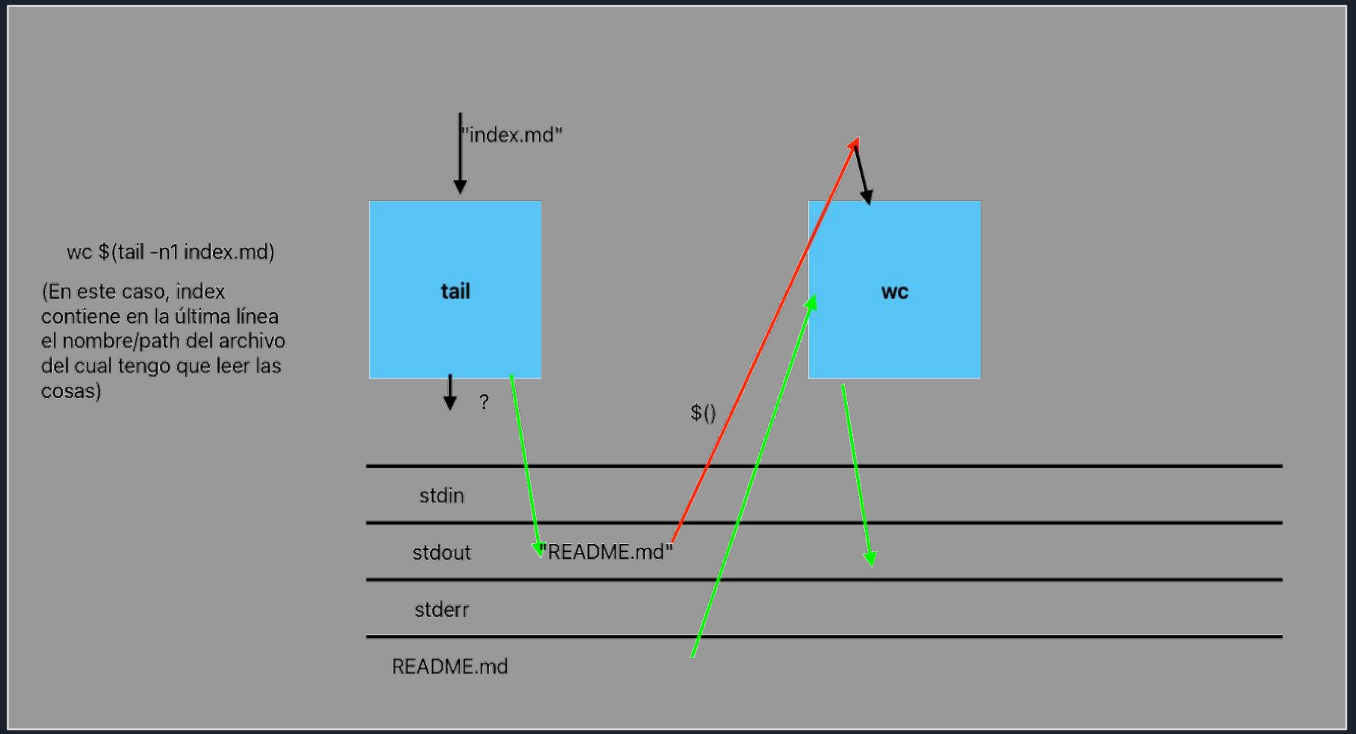
Redirección: > o >>

Resumen comunicación entre procesos



Piping: |

Resumen comunicación entre procesos



Command substitution: $()

Procesamiento de archivos

Bash es una herramienta muy útil para procesar archivos de manera rápida y eficiente. Los comandos vistos hasta el momento son suficientes para generar flujos complejos y procesar casi cualquier requisito.

Los siguientes comandos son comandos avanzados cuyo uso específico es el de facilitar este procesamiento:

- Grep
- Cut
- Sed

Grep: file pattern searcher

`grep [argumentos] <patron> <archivo> [archivo2] [archivoN]`

Permite buscar patrones en el contenido de archivo.

Sobre los archivos pasados, busca linea por linea cuales cumplen con el patrón a buscar y devuelve por stdout todas las líneas de el/los archivo/s que lo cumplen.

Algunos argumentos útiles para usar:

- `-i`: case insensitive
- `-m <num>`: devuelve los primeros <num> matcheos
- `-r`: búsqueda recursiva en todos los subdirectorios
- `-n`: imprime además el número de línea del archivo
- `-w <pattern>`: busca el patrón como palabra entera y no parte de otra palabra
- `-o`: sólo devuelve la parte de la línea que matchea, si hay múltiples matches por línea, los devuelve separado

Return code

- 0 si hubo matches
- 1 si no hubo ningún match
- >1 si hubo un error

Cut: cut out portions of each line

`cut <argumentos> <lista> <archivo> [archivo2] [archivoN]`

Permite "cortar" los fragmentos especificados en <lista> de cada línea de los archivos pasados.

Es útil para seleccionar parte de los datos, los que sean necesarios, de archivos que tengan todas las líneas con un mismo formato (CSV por ejemplo).

Algunos argumentos útiles:

`-d <delim>`: usa <delim> como delimitador (por default usa tabs)

`-f <lista>`: la lista ahora pasa a indicar las posiciones a seleccionar de la lista de elementos que surge de dividir la línea usando el delimitador.

`-c <lista>`: la lista ahora pasa a indicar los caracteres a seleccionar de cada línea

`-s`: ignora líneas que no tengan al menos una aparición del delimitador (por default se muestran completas

Return code

0 si no hubo error

>0 si hubo un error

Sed: stream editor

`sed [<argumentos>] <comando> <archivo>`

Modifica el archivo siguiendo los comandos especificados. El resultado es mostrado por stdout.

El comando completo es muy complejo y tiene muchas variaciones y acciones posibles, lo vamos a usar de manera simplificada:

`sed 's/<pattern_old>/<pattern_new>/' <archivo>`

`s` indicará que queremos hacer una substitution (reemplazar un patrón por otro).

Los patrones siguen las mismas reglas que para los comandos anteriores.

Podemos indicar la cantidad de sustituciones que queremos hacer agregando al final `N` (con N siendo la cantidad, o `g` si queremos que reemplace todas las ocurrencias que encuentre en la misma línea).

Regular Expressions

También llamadas regex, regexp son secuencias de caracteres que especifican un *match pattern* en un texto dado.

Todos los comandos que vimos que recibían un patrón, pueden recibir una regex.

De hecho, ya vimos algunas cuestiones de las mismas: `*`, `.` son algunos de los caracteres especiales de regex que nos permiten hacer patrones complejos.

Regular Expressions

Tokens más usados:

`[abc]`: un único caracter a, b o c (si se usa [^abc] busca lo opuesto, que no sea ninguno)

`[a-z]`: un único caracter, cualquier letra entre a y z (sólo minúsculas). Se puede usar [A-Z] o combinar [a-zA-Z].

`.`: un único caracter cualquiera.

`\d`: cualquier dígito (también se puede usar [0-9])

`a?`: `0` o 1 apariciones de a (reemplazar a por el token que se desee)

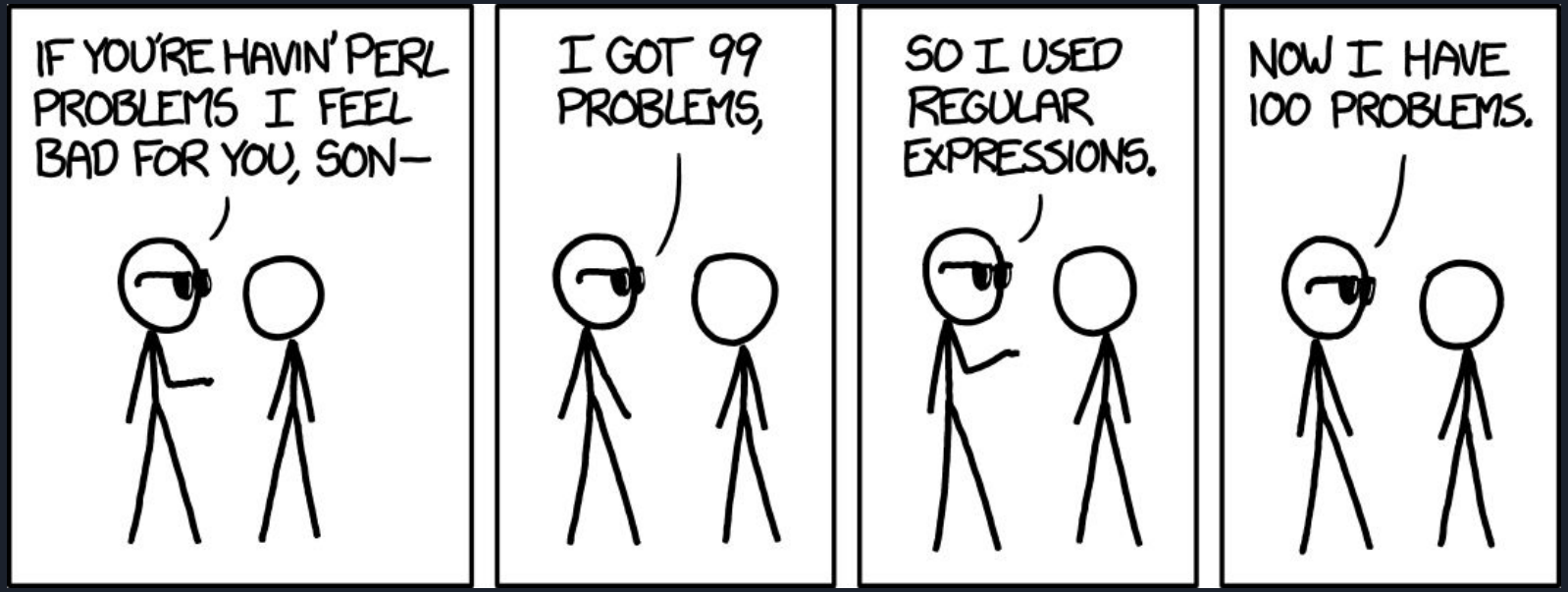
`a*`: `0` o más apariciones de a (reemplazar a por el token que se desee)

`a{N}`: exactamente N apariciones de a

`a{N,M}`: entre N y M (si se obvia M, será simplemente mayor o igual o N apariciones)

Sitio muy útil para practicar https://regex101.com/

Regular Expressions



xkcd 1171

Variables de Entorno

Son variables que están definidas de modo que estén siempre disponible en cada nueva sesión del shell que se inicie (por ejemplo, una *env var* puede ser definida en un script y usada en otro). Se suelen usar para configuración.

Hay muchas *env vars* ya definidas que son usadas por muchos comandos:

`$HOME, $PATH`

`export ENV_VAR=valor` Nos permite declarar *env vars* temporales. Se usa `SNAKE_CASE` con mayúsculas

`$PATH` **es una env var especial. Si el path a un ejecutable o script está incluido en esta variable, será posible ejecutarlo desde cualquier parte del sistema sin indicar el path al mismo.**

Para declarar *env vars* permanentes entre distintas sesiones (al reiniciar el sistema) debemos usar unos archivos de configuración.

Archivos de configuración

```
/bin/bash
      The bash executable
/etc/profile
      The systemwide initialization file, executed for login shells
~/.bash_profile
      The personal initialization file, executed for login shells
~/.bashrc
      The individual per-interactive-shell startup file
~/.bash_logout
      The individual login shell cleanup file, executed when a login shell exits
~/.inputrc
      Individual readline initialization file
```

Si usamos ZSH los archivos son los mismos cambiando `bash` por `zsh`

Bonus Tracks

Archivos especiales

`/dev/null`

Archivo usado para redireccionar “basura” o cosas que queramos que mueran.

Sirve para redireccionar stdout o stderr de comandos usados dentro de un script para tener control sobre lo que sí quiero que muestre mi script.

`/dev/stdin`

Puedo leer de stdin con: `var=$(</dev/stdin)`. *Esta es la forma “fácil” pero no ideal porque puede que falle en ciertas situaciones, sobre todo con Linux. Lo más seguro es leer línea a línea como el resto de los archivos.*

Fijarse que sólo estoy ejecutando el comando redireccionar como input del archivo de stdin.

`/dev/stdout y /dev/stderr también existen.`

Redirección de error

`>&2`

Para imprimir al stderr en vez del stdout: `>&2 echo "Esto es un error"`

Fin