5. Programación de Scripts

> 🖊 **Nota**
>
> Este tema originalmente habla también del shell y programar con bash, yo no explico porque no entra en el examen. Aunque es bastante importante saber como usar cada herramienta. En SOI se suele explicar (pero de forma muy precaria), si os interesa los apuntes originales de la asignatura lo explica bastante bien. Los temas tratados son:
>
> - Línea de comandos: El interprete de comandos (shell), variables de shell, expansiones del shell, redirección de la entrada/salida, orden de evaluación y ficheros de inicialización de bash
> - Scritps de administración: programación shell-script, entrada/salida, tests, estructura if...then...else, expresiones, control de flujo y funciones
>
> De todas formas, yo seguramente hable de algunos temas recogidos aquí con tal de aclarar cosa que considero poco obvias.

5.1 Expresiones regulares

Las **expresiones regulares** permiten reconocer en texto una serie de cadas de caracteres que siguen un patrón:

- Los ficheros de configuración y logs de Unix son normalmente ficheros de texto.
- Muchos comandos de procesamiento y búsqueda de texto como `grep`, `egrep`, `sed`, `awk` o `vi` usan expresiones regulares

No hay que confundir las expresiones regulares con los comodines, que son utilizados por el shell para referenciar ficheros. Se pueden usar con ficheros o con cualquier otro de tipo de entrada, como la entrada estándar, pipes, etc. Para evitar confusiones conviene escribir las expresiones regulares entre comillas, simples o dobles.

Se distinguen las expresiones regulares básicas y las extendidas, más completas.

> ⓘ **Info**
>
> - **Los comodines (Wildcards/Globbing):** Son caracteres especiales (como `*` o `?`) interpretados por la **Shell** (antes de ejecutar el comando) para coincidir con **nombres de ficheros y directorios**.

- **Las expresiones regulares (Regex):** Son patrones interpretados por el **comando o programa** (como `grep`) para buscar coincidencias dentro del **contenido del texto**.
- **La entrada estándar (stdin):** Es el flujo de datos de entrada predeterminado de un programa (Descriptor de fichero 0). Si no se especifica un archivo, comandos como `grep` leerán lo que escribas en el teclado o lo que le llegue desde otro comando.
- **Los pipes (tuberías `|`):** Son el mecanismo que conecta la salida estándar (stdout) de un comando con la entrada estándar (stdin) del siguiente, permitiendo filtrar con expresiones regulares datos generados "al vuelo".

5.1.1 Comandos egrep y `sed -r`

Estos comandos permiten utilizar **Expresiones Regulares Extendidas (ERE)**, lo que simplifica la escritura de patrones complejos (como grupos `()` o alternancias `|`) sin necesidad de "escapar" tantos caracteres con la barra invertida `\`.

El comando `egrep:` busca parones en ficheros de texto.

- Lee el texto línea por línea. Si encuentra el patrón en una línea, imprime la línea completa en la salida estándar (pantalla). Si no, la descarta.
- **Diferencia clave:** `egrep` es equivalente a ejecutar `grep -E`. Interpreta los metacaracteres extendidos (`?`, `+`, `{}`, `|`, `()`) directamente.

```shell
egrep [opciones] 'patrón' archivo
```

El comando `sed -r:` Es un editor de flujo de texto. Aunque tiene muchas funciones, su uso más frecuente con expresiones regulares es para **sustituir** texto.

- **Opción `-r`:** Habilita las expresiones regulares extendidas (en versiones modernas se usa `-E`, pero `-r` es el clásico en GNU/Linux).
- **Sintaxis de sustitución:** `s` (substitute), `patrón` (lo que busco), `reemplazo` (lo nuevo) y `g` (global - todas las veces que aparezca en la línea).

```shell
sed -r 's/patrón/sustitución/g' archivo
```

Ambos comandos siguen la filosofía Unix:

- **Entrada:** Pueden leer de un fichero o de la **entrada estándar (stdin)** a través de una tubería (`|`).
- **Uso de Comillas:** Es vital proteger la expresión regular para que la Shell no la interprete antes de tiempo.

Tipo de Comillas	Descripción	Ejemplo
Simples `'...'`	**Recomendado.** Todo lo de adentro es literal. La shell no toca nada.	`grep '$USER'` (Busca la palabra literal "$USER")
Dobles `"..."`	Permite a la shell expandir variables antes de ejecutar el comando.	`grep "$USER"\` (Busca tu nombre de usuario actual, ej: "juan")

Ejemplos:

SHELL

```
# Busca la palabra error O la palabra warning
egrep 'error|warning' /var/log/syslog
```

SHELL

```
# Cambia '2023' por '2024' en todas las apariciones
sed -r 's/2023/2024/g' config.txt
```

SHELL

```
IP="192.168.1.1"

# INCORRECTO: Busca la palabra literal $IP
egrep '$IP' conexiones.log

# CORRECTO: La shell cambia $IP por 192.168.1.1 antes de buscar
egrep "$IP" conexiones.log
```

Importante: no debemos confundir las expresiones regulares con los como- dines para la sustitución de nombres de ficheros (glob). Por ejemplo, el asterisco * como comodín indica cualquier cadena en nombres de fichero, mientras que como expresión regular es repetición

Característica	Comodines (Globbing)	Expresiones Regulares (Regex)
Quién lo interpreta	La Shell (bash, zsh, sh)	La herramienta (grep, sed, awk, vim)
Objetivo	Encontrar archivos en el disco	Encontrar texto dentro de archivos/flujos
Ejemplo `*`	`*.txt` (Todos los archivos acabados en .txt)	`a*` (La letra 'a' cero o más veces)
Cuándo ocurre	Antes de que el comando se ejecute	Mientras el comando se ejecuta

Si no especificamos fichero, egrep y sed usan la entrada estándar. Las comillas dobles permiten utilizar variables y comandos dentro de las expresiones regulares.

5.1.2 Expresiones regulares extendidas

Los sistemas UNIX actuales admiten extensiones a las expresiones regulares básicas (BRE). Para usar estas extensiones, generalmente necesitamos herramientas que las soporten explícitamente (como `egrep`, `grep -E` o `sed -r`).

La regla de oro es: **La mayoría de los caracteres son literales**, es decir, concuerdan consigo mismos.

- **Ejemplo:** La regex `casa` busca exactamente la secuencia "c", "a", "s", "a".

La excepción son los **metacaracteres**, símbolos con significado especial que definen el patrón.

Símbolo	Función	Significado	Ejemplo	Coincide con...
`.`	Comodín	Cualquier carácter (excepto salto de línea).	`a.c`	"abc", "axc", "a@c"
`[ ]`	Clase	Cualquiera de los caracteres dentro.	`[abc]`	"a", "b", "c"
`[^ ]`	Negación	Cualquier carácter **que no esté** dentro.	`[^abc]`	"d", "z", "1" (pero no a, b o c)
`^`	Ancla	Inicio de línea.	`^abc`	Líneas que empiezan por "abc"
`$`	Ancla	Final de línea.	`abc$`	Líneas que terminan en "abc"
`\`	Escape	Quita el significado especial al siguiente carácter.	`\.`	Un punto literal "."
**`	`**	Lógica	Operador OR (alternancia).	`a
`( )`	Grupo	Agrupa patrones y permite capturarlos.	`(ab)+`	"ab", "abab", "ababab"

Cuantificadores de repetición: Estos afectan al carácter o grupo que tienen **inmediatamente a su izquierda**

Símbolo	Significado	Mnemotecnia	Ejemplo a...
`*`	**0** o más veces.	"El opcional múltiple"	`a*` (vacío, a, aa, aaa...)

Símbolo	Significado	Mnemotecnia	Ejemplo a...
`+`	**1** o más veces.	"El obligatorio"	`a+` (a, aa, aaa...)
`?`	**0** o **1** vez.	"El opcional único"	`a?` (vacío o a)
`{n}`	Exactamente *n* veces.	Exacto	`a{3}` (aaa)
`{n,}`	Al menos *n* veces.	Mínimo	`a{2,}` (aa, aaa...)
`{n,m}`	Entre *n* y *m* veces.	Rango	`a{2,4}` (aa, aaa, aaaa)

Dentro de una clase de caracteres (corchetes), la mayoría de metacaracteres **pierden su significado especial** y se vuelven literales.

- `[a.]c` → Busca una 'a' o un punto literal '.', seguido de 'c'. (Aquí el punto no es "cualquier carácter").

Si quieres buscar literalmente un corchete, un guion o un circunflejo dentro de una clase, su posición importa:

1. **El guion `-`:** Ponlo al final de la lista. `[a-z-]` (letras o un guion).
2. **El cierre `]`:** Ponlo al principio de la lista. `[]abc]` (un corchete cerrado, a, b, o c).
3. **El circunflejo `^`:** Ponlo en cualquier sitio **menos al principio** (si va al principio niega). `[a^b]` (a, circunflejo, o b).

Ejemplos:

- `a..c`: Una 'a', dos caracteres cualquiera, una 'c'. (Ej: *azxc*, *a12c*).
- `[a-z]`: Cualquier letra minúscula.
- `ab*c`: Una 'a', cero o muchas 'b', una 'c'. (Ej: *ac*, *abc*, *abbbc*).
- `ab+c`: Una 'a', una o muchas 'b', una 'c'. (Ej: *abc*, *abbc*). **No coincide con *ac***.
- `.*`: Cualquier cadena de texto (muy común).
- `^#.*\.$`: Línea que es un comentario (empieza por `#`) y termina en un punto literal.
- `(a|b)c`: Puede ser *ac* o *bc*.
- `x(abc)*x`: Una 'x', seguida de la secuencia "abc" repetida 0 o más veces, y otra 'x'. (Ej: *xx*, *xabcx*, *xabcabcx*).

Conceptos avanzados:

Etiquetado y Retroreferencias (Backreferences):
Lo que pongas entre paréntesis `( )` se guarda en memoria temporalmente. Puedes "pegar" lo que se encontró usando `\1` (para el primer paréntesis), `\2` (para el segundo), etc.

Ejemplo de detección:

- Patrón: `(.)oo\1`

- Explicación: Busca cualquier carácter (grupo 1), seguido de "oo", seguido de **lo mismo que se encontró en el grupo 1**.
- Coincide con: `moom`, `noon`.
- No coincide con: `moon` (porque la 'n' final no es igual a la 'm' inicial).

Ejemplo de sustitución con `sed`: Podemos reordenar el texto usando las referencias.

```shell
# Intercambiar el orden de dos caracteres
sed -r "s/(.)(.)/\2\1/g"
# Si la entrada es "ab", el grupo 1 captura 'a', el grupo 2 captura 'b'.
# La salida \2\1 imprime primero 'b' y luego 'a' -> "ba".
```

El carácter `&` en `sed`:
En la parte de sustitución, el símbolo `&` representa **toda la cadena que coincidió con el patrón**.

```shell
sed -r "s/hola/[&]/g"
# Si encuentra "hola", lo sustituye por "[hola]".
```

Concordancia Codiciosa (Greedy Matching):
Las expresiones regulares son "codiciosas" por naturaleza: intentan abarcar la **mayor cantidad de texto posible** que cumpla el patrón.

- Texto: `yyyaxaxayyy`
- Patrón: `a(.*)a` (una 'a', lo que sea, y otra 'a')
- Resultado: Detectará `axaxa` entero.
- *Nota:* No se detendrá en la segunda 'a' (axa), sino que irá hasta la última posible.

Porrazos

Ejemplo 1:

SHELL

```
 #concuerda con direcciones de e-mail
 \w+@\w+\.\w+((\.\w+)*)?
  ```


`\w` (word character) es un atajo para no tener que escribir `[a-zA-Z0-9_]`.
- **`\w`** = Cualquier letra (a-z), número (0-9) o guion bajo (_).
- **`\w+`** = Una o más letras/números seguidos.

El concepto avanzado aquí es la **Agrupación Anidada** para estructuras repetitivas.
- **El problema:** Un dominio puede tener una extensión (`.com`), dos (`.co.uk`), tres (`.pntic.mec.es`), etc. No sabemos cuántas.
- **La solución avanzada:** `((\.\w+)*)?`
    1. **Grupo interno `(\.\w+)`**: Define la unidad mínima que se repite (un punto seguido de texto).
    2. **Cuantificador interno `*`**: Dice "esta unidad puede aparecer 0, 1 o mil veces".
    3. **Grupo externo con `?`**: Dice "y todo este lío de repeticiones es, en sí mismo, opcional".

Ejemplo 2:
```shell
# concuerda con fechas en el formato dd-mm-yyyy (años entre el 1900 y 2099)
(0[1-9]|[12][0-9]|3[01])-(0[1-9]|1[012])-(19|20)[0-9]{2}
```

Este es el ejemplo más importante para un administrador de sistemas. **Las expresiones regulares NO saben matemáticas**. No saben que 32 es mayor que 31. Solo ven caracteres.

Para validar un rango numérico (como los días del 1 al 31), tenemos que **descomponer el número en patrones de texto** usando el operador **OR (`|`)**.ç

Caso	Regex	Explicación	Números que cubre
A	`0[1-9]`	Empieza por 0, sigue un dígito del 1 al 9.	01, 02... 09
B	`[12][0-9]`	Empieza por 1 o 2, sigue cualquier dígito.	10, 11... 29
C	`3[01]`	Empieza por 3, sigue un 0 o un 1.	30, 31

Desglose del AÑO `(19|20)[0-9]{2}`:

- `(19|20)`: Restringimos el siglo. Solo aceptamos 19 o 20.
- `[0-9]{2}`: Aceptamos cualquier combinación para los dos últimos dígitos.

- **Resultado:** Valida desde 1900 hasta 2099.

Ejemplo 3:

```shell
concuerda con números en punto flotante (con o sin exponente)
[-+]?([0-9]*\.)?[0-9]+([eE][-+]?[0-9]+)?
```

- **El Signo Opcional:** `[-+]?`
- **La parte decimal (El truco del asterisco):** `([0-9]*\.)?`
  - Aquí hay algo curioso: `[0-9]*` (asterisco, no más). Esto significa "cero o más dígitos".
  - **¿Por qué?** Para permitir escribir `.5` (sin el cero delante). Si pusiéramos `+`, obligaríamos a escribir `0.5`.
  - Todo el grupo es opcional `?` porque un entero (50) no lleva punto.
- **La parte obligatoria:** `[0-9]+`
  - Al menos debe haber un dígito en algún lado.
- **Notación Científica (Exponente):** `([eE][-+]?[0-9]+)?`
  - Acepta `e` o `E`.
  - Acepta un signo opcional para el exponente (`e-5`).
  - Obliga a que haya dígitos detrás de la 'e' (no vale `10e`).

5.2 Procesamiento de textos

En UNIX/Linux, estos comandos se conocen como **filtros**. Su filosofía es recibir datos por la **entrada estándar** (stdin) o de un fichero, procesarlos, y enviar el resultado a la **salida estándar** (stdout).

Flujo típico: `entrada` → `filtro` → `salida`

> **La potencia de las tuberías**
>
> Estos comandos brillan cuando se combinan.
>
> Ejemplo: sort < fichero.txt | head -3 > top3.txt (Ordena el fichero y guarda las 3 primeras líneas en uno nuevo).

Comando	Función Principal
head / tail	Muestra el principio o final de un fichero.
tac / rev	Invierte el fichero (por líneas o por caracteres).
wc	Cuenta líneas, palabras y bytes (*Word Count*).
sort	Ordena las líneas (alfabética o numéricamente).

Comando	Función Principal
uniq	Elimina líneas repetidas **consecutivas**.
tr	Traduce o elimina caracteres (solo desde stdin).
cut	Corta y extrae columnas (campos) de texto.
paste	Pega ficheros lado a lado (concatena líneas).
join	Cruce de bases de datos (SQL JOIN) en ficheros de texto.
split	Divide un fichero grande en trozos pequeños.

> **Nota**
>
> Para el final no hay que saberse ninguna opción

5.2.1 Visualización Parcial (`head`, `tail`, `tac`, `rev`)

`head` y `tail`

Muestran las primeras o últimas líneas. Son fundamentales para ver logs o cabeceras de archivos.

```shell
head [opciones] fichero
```

Opción	Descripción
`-n X`	Muestra X líneas (por defecto son 10).
`-f`	*(Solo tail)* **Follow**. Se queda esperando nuevos datos (ideal para logs en tiempo real).

Ejemplos:

```shell
# Muestra las 2 primeras líneas con cabecera (-v)
$ head -n 2 -v quijote.txt
==> quijote.txt <==
En un lugar de la Mancha, de cuyo nombre
no quiero acordarme, no ha mucho tiempo

# Muestra las 2 últimas líneas
$ tail -n 2 quijote.txt
astillero, adarga antigua, rocín flaco y
galgo corredor.
```

`tac` y `rev`

- `tac`: Es `cat` al revés. Imprime la última línea primero, luego la penúltima...
- `rev`: Invierte el orden de los caracteres de cada línea (efecto espejo).

```shell
$ cat archivo.txt
Hola
Mundo

$ tac archivo.txt   |   $ rev archivo.txt
Mundo               |   aloH
Hola                |   odnuM
```

5.2.2 Estadísticas y Ordenación (`wc`, `sort`, `uniq`)

`wc` (Word Count)

Cuenta elementos en el fichero. Si no se ponen opciones, muestra: líneas, palabras y bytes.

- **Opciones:** `-l` (líneas), `-w` (palabras), `-c` (bytes/caracteres).

```shell
$ wc -l quijote.txt
5 quijote.txt
```

`sort`

Ordena las líneas. **Importante:** Por defecto ordena alfabéticamente (ASCII), donde 10 va antes que 2.

```shell
sort [opciones] fichero`
```

Opción	Descripción
`-n`	Orden numérico (para que 2 vaya antes que 10).
`-r`	Orden inverso (Reverse).
`-f`	Ignorar mayúsculas/minúsculas (Case insensitive).
`-k N`	Ordenar por la columna número N.

```shell
# Ordenar ignorando mayúsculas (-f) por la segunda columna (-k 2)
$ sort -f -k 2 nombres.txt
luis Andión
Adriana Gómez
```

`uniq`

Filtra líneas repetidas.

> ⚠ **¡Cuidado!**
>
> uniq solo detecta duplicados si son adyacentes (están juntos). Por eso SIEMPRE se suele usar sort antes que uniq.
>
> Uso típico: sort fichero | uniq

- **Opciones:** `-c` (cuenta las repeticiones), `-u` (muestra solo las únicas), `-d` (muestra solo las repetidas).

```shell
$ sort nombres.txt | uniq -c
   1 Enrique Pena
   2 Celia Fernández  <-- Aparecía 2 veces
```

5.2.3 Manipulación de Contenido (`tr`, `cut`)

`tr` (Translate)

Sustituye o elimina caracteres.

- **Nota:** `tr` **NO** acepta nombres de archivo como argumento. Funciona solo con la entrada estándar (`<` o `|`).

```shell
# Cambiar minúsculas a mayúsculas
$ tr 'a-z' 'A-Z' < quijote.txt

# Eliminar (-d) espacios en blanco
$ tr -d ' ' < quijote.txt

# Sustituir múltiples caracteres (a->p, u->k)
$ tr 'au' 'pk' < quijote.txt
# "En un lugar" se convierte en "En kn lkgpr"
```

`cut`

Corta columnas verticales de un fichero.

Opción	Descripción
`-c 1-5`	Corta por posición de carácter (del 1 al 5).
`-d 'x'`	Define el delimitador (separador) como 'x' (por defecto es tabulador).
`-f N`	Elige el campo (field) número N.

```shell
# Fichero: Luis Andión (separado por espacio)
# Queremos solo el nombre (campo 1) usando el espacio como separador
$ cut -d ' ' -f 1 nombres-ord.txt
Luis
Adriana
```

5.2.4 Gestión de múltiples ficheros (`paste`, `join`, `split`)

`paste` (Concatenación horizontal)

Pega líneas de dos ficheros una al lado de la otra.

```shell
$ paste nombres.txt apellidos.txt
Luis    Andión
Adriana Gómez
```

`join` (Cruce relacional)

Combina dos ficheros basándose en un **campo común** (como una ID o clave).

> ⚠ **Requisito Obligatorio**
>
> Para que join funcione, ambos ficheros deben estar ordenados previamente por el campo que se va a usar para unir.

```shell
# Unir fichero1 y fichero2 usando el campo 2 (-j 2) como clave
$ join -j 2 nombres1.txt nombres2.txt
Andión Luis Pedro  <-- (Clave común) (Resto f1) (Resto f2)
```

`split`

Divide un archivo grande en trozos pequeños.

```
split [opciones] fichero [prefijo_salida]
```
SHELL

- **Opciones:** `-l N` (cada N líneas), `-b N` (cada N bytes), `-d` (usar sufijos numéricos 00, 01... en vez de aa, ab...).

```
# Dividir quijote.txt en archivos de 2 líneas cada uno
$ split -l 2 -d quijote.txt quij_
# Genera: quij_00, quij_01, quij_02...
```
SHELL

5.3 `awk`

`awk` es, posiblemente, la herramienta más potente que existe. A diferencia de `cut` o `grep`, `awk` **es un lenguaje de programación completo** diseñado para procesar datos estructurados (columnas).

AWK (Aho, Weinberger y Kernighan) está diseñado para procesar texto organizado en tablas (filas y columnas), como los logs del sistema, `/etc/passwd` o la salida de comandos como `ls -l`.

- **Filosofía:** Lee la entrada **línea a línea**.
- **Automático:** Divide cada línea en **campos** (columnas) automáticamente sin que tengas que decírselo.
- **Sintaxis:** Muy similar al lenguaje C.

5.3.1 Estructura de un programa `awk`

Un script de `awk` tiene forma de "sándwich". Tiene tres partes, aunque no es obligatorio usarlas todas:

Sección	Sintaxis	¿Cuándo se ejecuta?	Uso típico
1. Cabecera	`BEGIN { ... }`	**Una sola vez**, ANTES de leer la primera línea.	Inicializar variables, definir separadores (`FS`), imprimir títulos.
2. Cuerpo	`/patrón/ { ... }`	**Una vez por cada línea** del fichero.	Filtrar datos, cálculos, imprimir columnas.
3. Cierre	`END { ... }`	**Una sola vez**, DESPUÉS de leer la última línea.	Imprimir totales, promedios o mensajes de despedida.

Lo más habitual es usarlo directamente en la terminal con comillas simples `'...'` para proteger el código del shell.

```shell
awk 'BEGIN { setup }  /filtro/ { accion }  END { conclusion }' fichero
```

5.3.2 Manejo de Campos

Cuando `awk` lee una línea, la trocea automáticamente basándose en los espacios o tabuladores (por defecto).

- `$0`: La línea completa (sin trocear).
- `$1`: El primer campo (columna 1).
- `$2`: El segundo campo.
- `$N`: El enésimo campo.

> **Visualización** Imagina la frase: `Hola mundo linux`
>
> - `$0` = "Hola mundo linux"
> - `$1` = "Hola"
> - `$2` = "mundo"

Ejemplo básico:

```shell
# Imprime la columna 1 y la 3
echo "uno dos tres cuatro" | awk '{ print $1, $3 }'
# Salida: uno tres
```

5.3.3 Variables de Control

`awk` tiene variables internas que controlan cómo lee y escribe los datos. Las más importantes para un administrador son:

Variable	Significado	Descripción
`FS`	*Field Separator*	Qué carácter separa las columnas en la **entrada**. (Defecto: espacio).
`OFS`	*Output Field Separator*	Qué carácter separará las columnas en la **salida** (al hacer `print`).
`NR`	*Number of Records*	Número de línea actual (contador de líneas procesadas).
`NF`	*Number of Fields*	Número total de columnas que tiene la línea actual.

Ejemplo vital: Leer `/etc/passwd` Como este archivo usa dos puntos (`:`) y no espacios, debemos cambiar el `FS`.

```shell
# Cambiamos el separador de entrada (FS) a ":"
# Cambiamos el separador de salida (OFS) a una flecha "->"
awk 'BEGIN { FS=":"; OFS="->" } { print $1, $6 }' /etc/passwd

# Salida:
# root->/root
# daemon->/usr/sbin
```

5.3.4 Filtrado (El "cerebro" del cuerpo)

En la parte central del script, puedes decidir a qué líneas aplicas las acciones.

1. **Sin filtro:** `{ print $1 }` → Afecta a TODAS las líneas.
2. **Con Regex:** `/Error/ { print $0 }` → Solo líneas que contengan "Error".
3. **Con Lógica:** `$3 > 500 { print $1 }` → Solo si el valor de la columna 3 es mayor de 500.

```shell
df -h | awk '
  # Si la línea empieza por "/dev/sd" (es un disco físico)...
  /^\/dev\/sd/ {
      # Imprime nombre de partición ($1) y porcentaje ($5)
      print "Partición:", $1, "Ocupación:", $5
  }'
```

5.3.5 Programación Avanzada (Variables y Operaciones)

Al ser un lenguaje similar a C, puedes usar variables, sumas, condicionales (`if`) y formato avanzado (`printf`).

Ejemplo Completo: Sumar el tamaño de archivos Queremos sumar el tamaño (columna 5 de `ls -l`) de todos los archivos y mostrar el total al final.

```shell
ls -l | awk '
  BEGIN {
      suma = 0;   # Inicializamos variable (opcional, en awk empiezan en 0)
      print "--- Iniciando Suma ---"
  }

  # Cuerpo: Se ejecuta para CADA archivo
  {
      suma = suma + $5;              # Acumulamos el tamaño
      printf("Archivo: %s - %d bytes\n", $9, $5); # printf tipo C
  }

  END {
      print "----------------------";
      print "Total bytes:", suma;
  }
'
```

5.3.6 Formas de Ejecución

Tienes 3 formas de correr `awk`, igual que `bash` o `python`:

1. **Línea de comandos (One-liner):** Para cosas rápidas.

```shell
awk '{print $1}' fichero.txt
```

2. **Fichero de script (`-f`):** Cuando el código es largo.

```shell
awk -f mi_programa.awk datos.txt
```

3. **Ejecutable (Shebang):**
   - Creas un fichero con la primera línea: `#!/usr/bin/awk -f`
   - Le das permisos: `chmod +x script.awk`
   - Ejecutas: `./script.awk datos.txt`

5.4 Pyhton

Python es un lenguaje de **tipado dinámico**: no declaras el tipo de variable (como `int a;` en C), sino que el valor asignado define el tipo.

5.4.1 Tipos de Datos y Operadores

Tipos Básicos

- **Enteros (`int`):** Precisión ilimitada (puedes guardar números gigantes).
- **Flotantes (`float`):** Equivalente al `double` de C.
- **Cadenas (`str`):** Se delimitan por comillas simples `' '` o dobles `" "`.
- **Booleanos (`bool`):** `True` o `False` (Ojo, la primera letra va en mayúscula).

Entrada y Salida:

- `input()`: Siempre devuelve un **string**. Si quieres números, debes convertirlo (cast).
- `print()`: Imprime cualquier cosa.

```python
# Conversión de tipos (Casting)
a = int(input("Dame un número: "))  # De str a int
b = str(10.5)                       # De float a str
```

Operadores

Python tiene una sintaxis muy limpia para operaciones matemáticas y lógicas.

Tipo	Operadores	Notas
Aritméticos	`+`, `-`, `*`, `/`	La división `/` siempre devuelve float.
División Entera	`//`	Devuelve solo la parte entera.
Resto (Módulo)	`%`	
Potencia	`**`	`2 ** 3` es 8.
Bit a Bit	`|` (OR), `&` (AND), `^` (XOR), `~` (NOT), `<<`, `>>`	Operaciones binarias a bajo nivel.
Lógicos	`and`, `or`, `not`	Se usan palabras en inglés, no `&&` o `

5.4.2 Estructuras de Datos (Colecciones)

Esta es la parte más importante para manejar datos. Se dividen en cuatro grandes grupos:

Listas (`list`)

Secuencias ordenadas y **mutables** (se pueden cambiar).

- **Definición:** Corchetes `[]`
- **Slicing (Troceado):** `lista[inicio:fin]`. (El `fin` no se incluye).
- **Índices negativos:** `-1` es el último elemento.

```python
frutas = ['naranjas', 'uvas', 123]
frutas.append('peras')      # Añadir al final
frutas.pop()                # Sacar el último
frutas[0:2]                 # Elementos 0 y 1 (el 2 no entra)
```

Tuplas (`tuple`)

Secuencias ordenadas e **inmutables** (una vez creadas, no se tocan). Son más rápidas y seguras que las listas.

- **Definición:** Paréntesis `()` (o a veces sin nada, separadas por comas).

Conjuntos (`set`)

Colecciones desordenadas de elementos **únicos** (no admite duplicados). Ideales para operaciones matemáticas de conjuntos.

- **Definición:** `set()` o llaves `{}` con valores sueltos.

Operación	Símbolo	Descripción
Unión	`a | b`	Elementos en A o en B (o en ambos).
Intersección	`a & b`	Elementos que están en A y también en B.
Diferencia	`a - b`	Elementos de A quitando los que estén en B.

Diccionarios (`dict`)

Colecciones de pares **Clave-Valor**. Funcionan como una base de datos rápida o una tabla hash.

- **Definición:** Llaves `{ clave: valor }`.

```python
user = {'nombre': 'Ana', 'edad': 19}
print(user['nombre'])     # Acceder: Ana
user['edad'] = 20         # Modificar
```

5.4.3 El problema de las Referencias (Vital)

En Python, las variables son "etiquetas" que apuntan a objetos en memoria.

- **Tipos Simples (int, str):** Si copias `b = a` y cambias `a`, `b` **NO** cambia. (Se crea un nuevo objeto).
- **Colecciones (listas, dict):** Si copias `b = a`, ambas apuntan a la **misma lista**. Si tocas `a`, **cambias** `b`.

¿Cómo copiar de verdad?

Para tener listas independientes, hay que clonarlas explícitamente:

```python
lista_a = [1, 2, 3]
lista_b = lista_a[:]  # El slicing [:] crea una copia nueva
```

5.4.4 Control de Flujo y Funciones

Bucles y Condicionales

- `if / elif / else`: Importante la indentación (tabulaciones).
- `for`: Itera sobre colecciones directamente.
- `range(start, stop, step)`: Genera secuencias numéricas.

Funciones (`def`)

Python permite argumentos con valores por defecto y argumentos arbitrarios:

- `*args`: Recibe una tupla de argumentos extra (sin nombre).
- `kwargs`**: Recibe un diccionario de argumentos extra (con nombre clave=valor).

Estructura del Script (`if __name__ == "__main__":`)

Este bloque es crucial en scripts reutilizables.

- **Sin esto:** Si importas el fichero desde otro script, se ejecutaría todo el código.
- **Con esto:** El código dentro del `if` solo se ejecuta si lanzas el script directamente (`python script.py`), pero no si lo importas (`import script`).

5.4.5 Programación Orientada a Objetos (POO)

Python permite definir clases para encapsular datos y funciones.

- **Constructor:** `__init__(self, ...)` se ejecuta al crear el objeto.
- `self`: Referencia al propio objeto (como `this` en Java/C++). Debe ser el primer argumento de cada método.
- **Privacidad:** Si una variable empieza por `__` (doble guion bajo), es privada y no se puede acceder desde fuera fácilmente.
- **Herencia Múltiple:** Una clase puede heredar de varias "madres": `class Hija(Madre1, Madre2)`.

PYTHON

```python
class Servidor:
    def __init__(self, ip):
        self.ip = ip           # Atributo público
        self.__estado = "OFF"  # Atributo privado

    def encender(self):
        self.__estado = "ON"
```