

# Uso de Shell

≡ Laboratorio

Laboratorio 1

+ Añadir una propiedad

Comentarios



Añadir un comentario...

## ▼ Comandos básicos Shell

### ▼ 🌱 PARTE 1: Comandos básicos — `date`, `cal`, `echo`

#### 📌 `date` — Mostrar fecha y hora

¿Para qué sirve?

Muestra la fecha y hora actual del sistema, o una fecha simulada.

✍️ Sintaxis general:

```
date [opciones]
```

#### 🔧 Opciones comunes de `date`:

Opción	Descripción	Ejemplo
<code>-d "FECHA"</code>	Muestra una fecha simulada	<code>date -d "27 Oct"</code>
<code>+FORMATO</code>	Personaliza la salida con símbolos de formato	<code>date -d "27 Oct" "+%A %d de %B de %Y"</code>

⬆️ Ejemplo:

```
date -d "27 Oct" "+%A %d de %B de %Y"
```

🖨️ Salida:

```
domingo 27 de octubre de 2024
```

#### 📌 `cal` — Mostrar calendarios

¿Para qué sirve?

Muestra el calendario actual, un mes específico o el año completo.

✍️ Sintaxis general:

```
cal [mes] [año]  
cal [opciones]
```



### 🔧 Opciones comunes de `cal`:

Opción	Descripción	Ejemplo
<code>-y</code>	Muestra el calendario del año actual	<code>cal -y</code>

#### ⬆ Ejemplo:

```
cal 7 2025
```

#### 🖨 Salida:

Calendario del mes de julio de 2025.



### 📌 `echo` — Mostrar texto o variables

#### ¿Para qué sirve?

Imprime texto simple o formateado en pantalla.

#### 📝 Sintaxis general:

```
echo [opciones] [texto]
```

### 🔧 Opciones comunes de `echo`:

Opción	Descripción	Ejemplo
<code>-e</code>	Interpreta secuencias de escape ( <code>\n</code> , <code>\t</code> )	<code>echo -e "hola\nmundo"</code>
comillas ( <code>" "</code> )	Preservan espacios en blanco	<code>echo "hola mundo"</code>

#### ⬆ Ejemplo:

```
echo -e "hola\ncómo estás"
```

#### 🖨 Salida:

```
hola  
cómo estás
```



## ▼ 🌟 PARTE 2: Manejo de directorios — `pwd`, `cd`, `mkdir`, `rmdir`

### 📌 `pwd` — Mostrar el directorio actual

#### ¿Para qué sirve?

Muestra la ruta completa del directorio en el que te encuentras.

#### 📝 Sintaxis general:

```
pwd
```

⬆ Ejemplo:

```
pwd
```

⬆ Salida:

```
/home/usuario/Documentos
```

## 🔧 **cd** — Cambiar de directorio

### ¿Para qué sirve?

Permite **moverte entre carpetas** del sistema de archivos.

### ⌚ Sintaxis general:

```
cd [ruta]
```

## 🔧 Opciones comunes / rutas especiales:

Ruta / símbolo	Acción que realiza	Ejemplo
..	Subir un nivel (directorio padre)	<code>cd ..</code>
.	Quedarse en el directorio actual	<code>cd .</code>
~	Ir al directorio personal ( \$HOME )	<code>cd ~</code>
-	Ir al directorio anterior	<code>cd -</code>

⬆ Ejemplo:

```
cd ~/Descargas
```

⬆ Te mueve al directorio `Descargas`.

## 🔧 **mkdir** — Crear directorios

### ¿Para qué sirve?

Crea carpetas (una o varias).

### ⌚ Sintaxis general:

```
mkdir [opciones] nombre_directorio...
```

## 🔧 Opciones comunes de `mkdir`:

Opción	Descripción	Ejemplo
-p	Crear directorios parentales si no existen	<code>mkdir -p carpeta1/.../carpetaN</code>

`-p` Crea jerarquías de subdirectorios completas `mkdir -p dir1/dir1.1/dir1.1.1`

#### ⬆ Ejemplo simple:

```
mkdir proyecto
```



#### ⬆ Ejemplo con `-p`:

```
mkdir -p trabajo/codigos/python
```



### 📌 `rmdir` — Borrar directorios vacíos

¿Para qué sirve?

Elimina únicamente carpetas vacías.

#### ✍ Sintaxis general:

```
rmdir [opciones] nombre_directorio...
```

#### ⬆ Ejemplo:

```
rmdir carpeta_vacia
```

⚠ Si el directorio tiene archivos o subdirectorios, mostrará:

⚠ Si el directorio tiene archivos o subdirectorios, mostrará:

```
rmdir: failed to remove 'carpeta': Directory not empty
```



## ▼ 🌿 PARTE 3: Listar archivos y directorios — `ls`

### 📌 `ls` — Listar archivos

¿Para qué sirve?

Muestra el contenido de un directorio. Puedes ver nombres, permisos, tamaños, fechas, etc.

#### ✍ Sintaxis general:

```
ls [opciones] [ruta]
```

Si no se indica ruta, lista el contenido del directorio actual.



#### 📌 Opciones comunes de `ls`:

Opción	Descripción	Ejemplo
<code>-l</code>	Listado largo (muestra permisos, dueño, fecha, tamaño, etc.)	<code>ls -l</code>
<code>-a</code>	Incluye archivos ocultos (los que comienzan con <code>.</code> )	<code>ls -a</code>

<code>-h</code>	Tamaños legibles (en KB, MB) (útil con <code>-l</code> )	<code>ls -lh</code>
<code>-t</code>	Ordena por fecha de modificación más reciente	<code>ls -lt</code>
<code>-r</code>	Invierte el orden de la lista	<code>ls -lr</code>
<code>-d</code>	Muestra información del directorio en sí, no de su contenido	<code>ls -ld /etc</code>
<code>-S</code>	Ordena por tamaño de archivo (mayor primero)	<code>ls -ls</code>
<code>-R</code>	Muestra contenido de subdirectorios (recursivo)	<code>ls -lR /etc</code>
<code>-i</code>	Muestra el número de inodo de cada archivo	<code>ls -i archivo.txt</code>



### ⬆ Ejemplo 1: listado largo

```
ls -l
```



⬇ Salida:

```
-rw-r--r-- 1 usuario usuario 1234 Jul 22 10:00 archivo.txt
```



### ⬆ Ejemplo 2: listar ocultos y normales

```
ls -la
```



⬇ Salida incluye:

```
.bashrc
.escondido
documento.txt
```



### ⬆ Ejemplo 3: ordenar por tamaño

```
ls -ls
```



⬇ Salida: el archivo más grande aparece primero.



### 🧠 ¿Cuándo usar `ls`?

- Para explorar archivos rápidamente.
- Ver detalles como permisos o tamaños.
- Para encontrar archivos ocultos o recientes.



## ▼ 🌟 PARTE 4: Globbing y Comodines — `*`, `?`, `[ ]`

## 📌 ¿Qué es el globbing?

¿Para qué sirve?

Permite buscar o trabajar con múltiples archivos usando patrones de nombres en lugar de escribirlos uno por uno.

### 📎 Sintaxis general:

comando patrón

Por ejemplo:

```
ls *.txt
```

### 🎯 Comodines más comunes:

Comodín	¿Qué hace?	Ejemplo	Coincide con...
*	Cualquier número de caracteres (incluye ninguno)	ls u*	usuario, ubuntu, u
?	Exactamente un carácter cualquiera	ls file?.txt	file1.txt, fileA.txt
[abc]	Uno de esos caracteres	ls file[12].txt	file1.txt, file2.txt
[0-9]	Un solo número entre 0 y 9	ls file[0-9].txt	file1.txt, file7.txt
[^0-9]	Cualquier carácter <b>excepto</b> los del rango indicado	ls file[^0-9].txt	fileA.txt, fileX.txt, file_.txt

### ⬆ Ejemplo 1: archivos que comienzan con `u` en `/etc`

```
ls /etc/u*
```

📍 Coincide con:

```
/etc/udev  
/etc/ufw
```

### ⬆ Ejemplo 2: archivos que terminan en `.d`

```
ls /etc/*.d
```

📍 Coincide con:

```
etc/systemd  
/etc/network.d
```

### ⬆ Ejemplo 3: archivos tipo `recX.d` donde X es un número:

```
ls /etc/rec[0-9].d
```

⬇ Coincide con:

```
/etc/rec1.d  
/etc/rec7.d
```



### 🧠 ¿Cuándo usar globbing?

- Para trabajar con muchos archivos a la vez.
- Para hacer búsquedas más flexibles.
- Para scripts o automatizaciones.

## ▼ ⚡ PARTE 5: Buscar archivos — `find`

### ⚡ `find` — Buscar archivos según criterios

¿Para qué sirve?

Permite **buscar archivos y carpetas** por nombre, tipo, tamaño, fecha, etc., dentro de un directorio y sus subdirectorios.



📝 Sintaxis general:

```
find [ruta_inicial] [opciones] [criterios]
```



### 🔧 Opciones y criterios comunes:

Opción	¿Qué hace?	Ejemplo
<code>-name</code>	Buscar por nombre (puede usar comodines *)	<code>find /etc -name "*.conf"</code>
<code>-type f</code>	Buscar solo archivos	<code>find /usr -type f -name "*.sh"</code>
<code>-type d</code>	Buscar solo directorios	<code>find . -type d -name "*temp*"</code>
<code>-size +N[kM]</code>	Archivos mayores a N (k = KB, M = MB)	<code>find . -size +10k</code>
<code>-empty</code>	Archivos o carpetas vacías	<code>find ~ -empty</code>
<code>-newer archivo</code>	Archivos más recientes que otro	<code>find . -newer referencia.txt</code>
<code>-delete</code>	Borra lo que se encuentre	<code>find . -type f -empty -delete</code>

### ⬆ Ejemplo 1: Archivos grandes

```
find /etc -type f -size +10k
```



⬇ Encuentra archivos en `/etc` mayores a 10 KB.

## ⬆ Ejemplo 2: Archivos vacíos

```
find / -empty 2>/dev/null
```

⬇ Muestra archivos o carpetas vacías. `2>/dev/null` oculta errores de "permiso denegado".



## ⬆ Ejemplo 3: Buscar archivos `.c` menores a 10 KB

```
find / -type f -name "*.c" -size -10k 2>/dev/null
```

## ⬆ Ejemplo 4: Borrar carpetas vacías en tu home

```
find ~ -type d -empty -delete
```

## 🧠 ¿Cuándo usar `find` ?

- Cuando necesitas **buscar algo profundo** en el sistema.
- Cuando quieres **automatizar limpiezas** (ej. borrar archivos temporales).
- Para aplicar acciones a archivos encontrados (`exec`, `delete`).

## ▼ 🌱 PARTE 6: Ver detalles de archivos — `stat`

### 📌 `stat` — Mostrar metadatos de un archivo

¿Para qué sirve?

Muestra información detallada de un archivo o directorio, como:

- Tamaño
- Tipo
- Fechas de acceso/modificación
- Permisos
- Número de inodo y bloques

### 📎 Sintaxis general:

```
stat [archivo_o_directorio]
```

## ⬆ Ejemplo 1: Ver detalles de un archivo binario

```
stat /usr/bin/haddock
```

⬇ Salida (resumen):

```
File: /usr/bin/haddock
Size: 20920      Blocks: 48          IO Block: 4096   regular file
```

```
Device: 802h/2050d Inode: 394876      Links: 1
Access: 2025-07-22 ...
Modify: 2025-07-01 ...
Change: 2025-07-01 ...
```

### 🔧 ¿Qué significan los campos más importantes?

Campo	Significado
<code>Size</code>	Tamaño en bytes del archivo
<code>Blocks</code>	Número de bloques lógicos usados
<code>IO Block</code>	Tamaño de cada bloque lógico
<code>Inode</code>	Identificador único del archivo
<code>Access</code>	Última vez que se leyó el archivo
<code>Modify</code>	Última vez que se modificó el contenido
<code>Change</code>	Última vez que se cambió algún atributo (como permisos)

### 🧠 ¿Cuándo usar `stat`?

- Para ver cuándo se modificó un archivo por última vez.
- Para conocer el tamaño exacto y detalles técnicos.
- Para verificar inodos al trabajar con enlaces (`ln`).

## ▼ 🌱 PARTE 7: Mostrar contenido de archivos — `cat`

### 📌 `cat` — Concatenar y mostrar archivos

¿Para qué sirve?

Muestra el contenido de uno o más archivos directamente en la terminal. También permite combinar archivos o crear nuevos.

### ✍️ Sintaxis general:

```
cat [opciones] [archivo1] [archivo2] ...
```

### 🔧 Opciones comunes de `cat`:

Opción	¿Qué hace?	Ejemplo
<code>-n</code>	Numera las líneas del archivo	<code>cat -n archivo.txt</code>
<code>&gt;</code>	Redirige salida para crear o sobrescribir archivo	<code>cat archivo1 archivo2 &gt; todo.txt</code>
<code>&gt;&gt;</code>	Redirige salida agregando al final	<code>cat archivo &gt;&gt; resumen.txt</code>

### ⬆️ Ejemplo 1: Ver un archivo en pantalla

```
cat archivo.txt
```

## ⬆ Ejemplo 2: Ver dos archivos juntos

```
cat archivo1.txt archivo2.txt
```

↳ Une ambos contenidos y los muestra uno tras otro.



## ⬆ Ejemplo 3: Numerar líneas

```
cat -n poema.txt
```

↳ Salida:

```
1 En un lugar de la Mancha  
2 De cuyo nombre no quiero acordarme...
```



## ⬆ Ejemplo 4: Crear un archivo desde terminal

```
cat > nuevo.txt
```

✍ Luego puedes escribir lo que quieras y presionar **Ctrl + D** para guardar.

### 🌐 ¿Cuándo usar **cat**?

- Para leer rápidamente archivos pequeños.
- Para combinar múltiples archivos de texto.
- Para crear archivos manualmente desde terminal.

## ▼ 🌱 PARTE 8: Contar líneas, palabras y caracteres — **wc**

### 📌 **wc** — Word Count

¿Para qué sirve?

Cuenta la cantidad de líneas, palabras y caracteres (o bytes) de un archivo o entrada.

### ✍ Sintaxis general:

```
wc [opciones] [archivo]
```

### 🔧 Opciones comunes de **wc**:

Opción	¿Qué hace?	Ejemplo
<b>-l</b>	Cuenta solo las líneas	<code>wc -l archivo.txt</code>
<b>-w</b>	Cuenta solo las palabras	<code>wc -w archivo.txt</code>
<b>-m</b>	Cuenta los caracteres (UTF-8)	<code>wc -m archivo.txt</code>

	incluidos)	
-c	Cuenta los bytes (ojo: ≠ caracteres con tildes o emojis)	wc -c archivo.txt

### ⬆ Ejemplo 1: Contar todo (líneas, palabras y bytes)

```
wc archivo.txt
```

⬇ Salida:

```
12 48 310 archivo.txt
```

Significa:

- 12 líneas
- 48 palabras
- 310 bytes

### ⬆ Ejemplo 2: Contar solo líneas

```
wc -l archivo.txt
```

⬇ Salida:

```
12 archivo.txt
```

### ⬆ Ejemplo 3: Contar la longitud de la línea más larga (truco con `awk`)

`wc` no tiene una opción directa, pero puedes usar:

```
awk '{ print length }' archivo.txt | sort -n | tail -1
```

⬇ Muestra la cantidad de caracteres de la línea más larga.

### ⬆ También funciona con entrada desde teclado o pipes:

```
ls | wc -l
```

⬇ Cuenta cuántos archivos hay en el directorio actual.

## 🔴 ¿Cuándo usar `wc` ?

- Para saber cuántas líneas tiene un archivo (útil en logs, scripts, datos).
- Para contar palabras o caracteres de textos.
- Para automatizar estadísticas de texto.

## ▼ PARTE 9: Entrada/salida estandar y redirecciones —

### 📌 ¿Qué es la E/S estándar?

En Linux, todo programa usa tres canales por defecto:

Canal	Código	¿Para qué sirve?	Por defecto va a...
Entrada		Lo que recibe el programa	El teclado
Salida		Lo que imprime el programa	La pantalla
Error		Mensajes de error	La pantalla



## 🔧 Operadores de redirección

### 📌 — Redirige la salida estándar (crea o sobrescribe archivo)

```
ls > lista.txt
```

Crea `lista.txt` con la salida de `ls`. Si ya existe, lo sobrescribe.

### 📌 — Redirige agregando al final del archivo

```
ls >> lista.txt
```

Añade la salida de `ls` al final de `lista.txt`.

### 📌 — Usa un archivo como entrada

```
wc -l < archivo.txt
```

Lee el archivo como si lo estuvieras escribiendo tú por teclado.

### 📌 — Entrada heredada (a través de un bloque manual)

```
wc -l << FIN
Hola
Cómo estás
Todo bien
FIN
```

Resultado: `3` (3 líneas escritas manualmente)

### 📌 — Pipe: pasa la salida de un comando a otro

```
ls | wc -l
```

Cuenta cuántos archivos hay.

📌 `tee` — Guarda la salida en un archivo y también la muestra

```
date | tee fsalida
```

⬇️ Muestra la fecha en pantalla y la guarda en `fsalida`.

## 🧠 ¿Cuándo usar redirecciones?

- Para guardar resultados sin copiar a mano
- Para automatizar procesos
- Para trabajar con muchos comandos conectados

## ▼ 🌱 PARTE 10: Identificar tipo de archivo — `file`

📌 `file` — Detectar el tipo real de un archivo

¿Para qué sirve?

Muestra el **tipo de contenido** de un archivo (texto, imagen, ejecutable, etc.), sin fijarse en su extensión.

✍️ Sintaxis general:

```
file [archivo1] [archivo2] ...
```

### ⬆️ Ejemplo 1: Archivo de texto

```
file documento.txt
```

⬇️ Salida:

```
documento.txt: ASCII text
```

### ⬆️ Ejemplo 2: Script de Bash

```
file script.sh
```

⬇️ Salida:

```
script.sh: Bourne-Again shell script, ASCII text executable
```

### ⬆️ Ejemplo 3: Archivo binario

```
file /usr/bin/ls
```

 Salida:

```
/usr/bin/ls: ELF 64-bit LSB executable, x86-64
```

### Ver particiones del disco

```
file /dev/sd*
```

 Muestra cosas como:

```
/dev/sda1: Linux rev 1.0 ext4 filesystem data  
/dev/sda2: swap file
```

### ¿Cuándo usar `file`?

- Para saber si un archivo es realmente lo que parece.
- Para verificar si un archivo binario, de texto, script o multimedia está bien formado.
- Para detectar archivos mal renombrados.

## ▼ PARTE 11: Copiar archivos y directorios — `cp`

### `cp` — Copiar archivos o carpetas

¿Para qué sirve?

Permite duplicar archivos o copiar carpetas completas, conservando o modificando sus nombres.

### Sintaxis general:

```
cp [opciones] origen destino
```

### Opciones comunes de `cp`:

Opción	¿Qué hace?	Ejemplo
<code>-i</code>	Pregunta antes de sobrescribir archivos	<code>cp -i a.txt b.txt</code>
<code>-f</code>	Fuerza el copiado, sobrescribiendo sin preguntar	<code>cp -f a.txt b.txt</code>
<code>-r</code>	Copia directorios de forma recursiva	<code>cp -r carpeta1 carpeta2</code>
<code>-p</code>	Conserva permisos, fechas, propietario	<code>cp -p a.txt respaldo.txt</code>

### Ejemplo 1: Copiar un archivo

```
cp informe.txt copia.txt
```

 Crea una copia exacta con otro nombre.

### Ejemplo 2: Copiar múltiples archivos a una carpeta

```
cp a.txt b.txt c.txt ~/Documentos/
```

 Copia los 3 archivos al directorio `Documentos`.

### Ejemplo 3: Copiar una carpeta completa

```
cp -r proyecto/ respaldo/
```

 Copia toda la estructura de la carpeta `proyecto`.

### Ejercicio del PDF:

Copiar todos los `.h` de `/usr/include` al home:

```
cp /usr/include/*.h ~
```

Copiar `/etc` completo a `~/Descargas`:

```
cp -r /etc ~/Descargas
```

### ¿Cuándo usar `cp`?

- Para hacer respaldos.
- Para duplicar archivos antes de modificarlos.
- Para reorganizar carpetas sin perder el contenido original.

## ▼ PARTE 12: Mover o renombrar archivos — `mv`

### — Mover o renombrar archivos/directorios

¿Para qué sirve?

- Renombra archivos o carpetas
- Mueve archivos de un lugar a otro

### Sintaxis general:

```
mv [opciones] origen destino
```

### Opciones comunes de `mv`:

Opción	¿Qué hace?	Ejemplo
<code>-i</code>	Pregunta antes de sobrescribir	<code>mv -i viejo.txt nuevo.txt</code>

<code>-f</code>	Fuerza la operación, sobrescribe sin preguntar	<code>mv -f archivo.txt backup/</code>
<code>-u</code>	Mueve solo si el archivo origen es más reciente	<code>mv -u actual.txt copia/</code>

### ⬆ Ejemplo 1: Renombrar un archivo

```
mv informe.txt informe_final.txt
```

⬇ Cambia el nombre pero **no** la ubicación.



### ⬆ Ejemplo 2: Mover un archivo a otra carpeta

```
mv tarea.doc ~/Documentos/
```

⬇ Mueve el archivo a tu carpeta `Documentos/`.



### ⬆ Ejemplo 3: Mover y sobrescribir sin preguntar

```
mv -f config.conf /etc/
```



### ⬆ Ejercicio del PDF:

Cambiar la extensión `.h` a `.h.bak`:

```
for file in *.h; do mv "$file" "${file}.bak"; done
```

Luego moverlos a `Descargas`:

```
mv *.h.bak ~/Descargas/
```



### 🧠 ¿Cuándo usar `mv`?

- Para renombrar archivos fácilmente.
- Para mover archivos o carpetas entre ubicaciones.
- Para reorganizar tu sistema de archivos sin duplicar datos.

## ▼ 🌱 PARTE 13: Eliminar archivos y directorios — `rm`

### 🔗 `rm` — Remover archivos o carpetas

¿Para qué sirve?

Elimina archivos y directorios de forma permanente (⚠ no se envían a la papelera).

📝 Sintaxis general:



```
rm [opciones] archivo_o_directorio
```

### 🔧 Opciones comunes de `rm`:

Opción	¿Qué hace?	Ejemplo
<code>-i</code>	Pregunta antes de borrar cada archivo	<code>rm -i archivo.txt</code>
<code>-f</code>	Fuerza el borrado sin preguntar (ni mostrar errores)	<code>rm -f archivo.txt</code>
<code>-r</code>	Borra carpetas y su contenido recursivamente	<code>rm -r carpeta/</code>
<code>-rf</code>	Borrado forzado y recursivo (⚠️ muy peligroso si mal usado)	<code>rm -rf ~/temp</code>

### ⬆ Ejemplo 1: Borrar un archivo

```
rm notas.txt
```

### ⬆ Ejemplo 2: Borrar varios archivos con confirmación

```
rm -i tarea1.txt tarea2.txt
```

👉 Pregunta uno por uno:

```
rm: remove regular file 'tarea1.txt'? y
```

### ⬆ Ejemplo 3: Borrar un directorio completo

```
rm -r proyecto/
```

### ⬆ Ejemplo 4: Borrar todos los archivos que empiezan con "t" en [Descargas](#)

```
rm ~/Descargas/t*
```

### ⬆ Ejemplo 5: Con confirmación por cada uno

```
rm -i ~/Descargas/t*
```

⚠️ ¡Cuidado! Un comando como este puede destruir tu sistema si eres root:

```
rm -rf /
```

Nunca lo uses.

### 🧠 ¿Cuándo usar `rm`?

- Para limpiar archivos temporales o innecesarios.
- Para automatizar eliminación de archivos vacíos o viejos.
- Siempre con precaución, especialmente con `rf`.

## ▼ 🌟 PARTE 14: Cambiar permisos — `chmod`

### 📌 `chmod` — Cambiar permisos de archivos o directorios

¿Para qué sirve?

Permite dar o quitar permisos de lectura, escritura y ejecución para el propietario, grupo y otros usuarios.

### ✍️ Sintaxis general:

```
chmod [quién][+|-][permiso] archivo
```

### 🔧 Símbolos de uso:

Componente	Significado
u	usuario (propietario)
g	grupo
o	otros
a	todos (u + g + o)
r	lectura
w	escritura
x	ejecución
+ / -	agregar / quitar permisos

### ⬆️ Ejemplo 1: Dar permiso de escritura a todos

```
chmod a+w archivo.txt
```

### ⬆️ Ejemplo 2: Quitar permiso de ejecución a todos

```
chmod a-x script.sh
```

### ⬆️ Ejemplo 3: Dar solo a grupo el permiso de lectura

```
chmod g+r documento.pdf
```

### ⬆ Ejercicio del PDF:

1. Dar permiso de escritura a todos los archivos en `Descargas`:

```
chmod a+w ~/Descargas/*
```

1. Crear un directorio `foo` y bloquear escritura:

```
mkdir foo  
chmod a-w foo
```

1. Intentar copiar algo a `foo` (falla):

```
cp /etc/passwd foo
```

1. Dar permisos otra vez:

```
chmod a+w foo
```

1. Quitar permiso de ejecución para `foo` (no podrás entrar):

```
chmod a-x foo  
cd foo # da error
```

### 🌐 ¿Cuándo usar `chmod`?

- Para proteger archivos importantes de modificaciones accidentales.
- Para habilitar la ejecución de scripts (`chmod +x archivo.sh`).
- Para configurar permisos correctos en servidores o scripts compartidos.

## ▼ 🌟 PARTE 15: Crear enlaces — `ln`

### 🔗 `ln` — Crear enlaces (nombres alternativos para archivos)

¿Para qué sirve?

Crea enlaces físicos (hard links) o enlaces simbólicos (soft links) a archivos o carpetas.

- Hard link: otro nombre para el **mismo archivo real** (comparten el mismo inodo).
- Soft link (simbólico): es como un **acceso directo** que apunta al archivo original.

📎 Sintaxis general:

#### ◆ Enlace duro:

```
ln [archivo_origen] [nuevo_nombre]
```

### ◆ Enlace simbólico:

```
ln -s [ruta_origen] [enlace_simbólico]
```

#### ⬆ Ejemplo 1: Crear un enlace duro

```
ln datos.txt copia.txt
```

⬇ `datos.txt` y `copia.txt` apuntan al mismo contenido, incluso si borras uno, el otro sobrevive.



#### ⬆ Ejemplo 2: Crear un enlace simbólico

```
ln -s /etc/passwd passwd
```

⬇ `passwd` es un acceso directo a `/etc/passwd`.



#### ⬆ Comprobar si dos archivos son hard links

```
ls -li archivo1 archivo2
```

⬇ Si los números de inodo son iguales, son el mismo archivo.



#### 🧠 Diferencias clave:

Característica	Hard Link	Soft Link
¿Mismo inodo?	✓ Sí	✗ No
¿Funciona si se borra el original?	✓ Sí (contenido sigue)	✗ No (enlace roto)
¿Puede enlazar carpetas?	✗ No	✓ Sí
¿Entre particiones?	✗ No	✓ Sí

#### ✍ Ejercicio del PDF:

```
ln -s /etc/passwd passwd  
file passwd
```

⬇ Te muestra que `passwd` es un enlace simbólico a `/etc/passwd`.



#### 🧠 ¿Cuándo usar `ln` ?

- Para acceder a un archivo con otro nombre o desde otra ubicación.
- Para crear accesos directos a herramientas o scripts.
- Para usar un mismo archivo compartido en varias rutas sin duplicarlo.



## ▼ Uso del Shell Guia 2

### ▼ Parte 1: ¿Qué es el shell?

- El shell es un programa que permite interactuar con el sistema operativo.
- Es un intérprete de comandos: lee lo que escribes, lo interpreta y ejecuta.
- También es un lenguaje de programación: tiene condicionales, bucles, funciones y variables.
- Hay varios tipos de shell: `bash`, `sh`, `zsh`, `ksh`, etc.
- Puedes verificar cuál usas con `echo $SHELL` o `ps -p $$`.

### ▼ Parte 2: ¿Qué es un shell script?

- Un shell script es un archivo de texto plano que contiene comandos de shell.
- Se ejecuta línea por línea de forma interpretada.
- Se puede escribir en editores como `nano`, `vim`, `emacs`, etc.
- Para crear uno:

```
nano mi_script.sh
```

- Para ejecutarlo:

1. Dar permisos: `chmod +x mi_script.sh`
2. Ejecutar: `./mi_script.sh`

- El `./` se usa porque el directorio actual (`.`) no está en el `$PATH` por seguridad.

### ▼ Parte 3: Shebang (#!) y Comentarios

- El shebang es la primera línea del script y le dice al sistema qué intérprete usar.

```
#!/bin/bash
```

- Sin el shebang, el script se ejecutará con el shell por defecto (que puede no ser Bash).
- Es muy importante para garantizar que el script se interprete correctamente.

#### Comentarios en Bash:

- Cualquier línea que comienza con `#` (excepto el shebang) es un comentario.
- Los comentarios no se ejecutan, sirven para documentar.

```
# Esto es un comentario
echo "Hola mundo" # También puede ir al final de una línea
```

## ▼ Parte 4: Variables en Shell

#### Variables escalares (simples)

- Se declaran así (sin espacios alrededor del `=`):

```
nombre=Mayo
```



- Para acceder a su valor:

```
echo $nombre # Muestra: Max
```

### ✓ Leer datos del usuario con `read`

```
echo "Ingresa tu nombre:"  
read nombre  
echo "Hola, $nombre"
```



#### ⚠ Reglas para nombres de variables:

- Pueden contener letras, números y guiones bajos (`_`)
- Deben comenzar con una letra o guion bajo
- Ejemplos válidos: `edad`, `_usuario1`, `nombre_completo`

## ▼ █ Parte 5: Arreglos, variables de solo lectura y `unset`

### ✓ Arreglos en Bash

Un arreglo almacena **múltiples valores** con un solo nombre:

```
frutas=(manzana uva piña)
```



O por índice:

```
frutas[0]=manzana  
frutas[1]=uva
```

#### 📌 Acceder a los valores:

- Un valor:

```
echo ${frutas[1]} # uva
```



- Todos los valores:

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

### 🔒 Variables de solo lectura

Una vez definidas, no pueden ser modificadas:

```
readonly PI=3.1416
```

## ☒ Eliminar variables con `unset`

- Para borrar una variable (si no es de solo lectura):

```
unset frutas
```

## ▼ █ Parte 6: Variables del entorno y del shell

### ✓ Variables locales

- Definidas normalmente:

```
nombre=Max
```

- Solo existen en el shell actual.
- No se heredan por otros procesos.

### ✓ Variables de entorno

- Se usan para pasar valores a otros programas o shells hijos.
- Se crean así:

```
export nombre=Max
```

- Ver todas las variables del entorno:

```
env
```

### ✓ Variables del shell

Son predefinidas por el shell para su funcionamiento.

Variable	Significado
<code>\$PWD</code>	Directorio actual
<code>\$PATH</code>	Rutas donde buscar comandos
<code>\$HOME</code>	Carpeta personal del usuario

- Puedes ver todas las variables del shell con:

```
set
```

## ▼ █ Parte 7: Sustitución de variables

Permite usar valores por defecto, hacer asignaciones condicionales, y manejar errores cuando las variables no existen.

### ✓ 1. Valor por defecto (solo se muestra)

```
echo ${nombre:-Invitado}
```

Si `nombre` no está definida → imprime "Invitado"  
(No cambia el valor de `nombre`)

## ✓ 2. Asignar valor si está vacía

```
echo ${nombre:=Invitado}
```

Si `nombre` está vacía o no existe → le asigna "Invitado"

## ✓ 3. Error si no está definida

```
: ${nombre:?Debes definir tu nombre}
```

Si `nombre` no está definida → muestra mensaje y termina el script

## ✓ 4. Sustituir solo si está definida

```
echo ${DEBUG:+Modo Depuración Activado}
```

Solo imprime el texto si `DEBUG` está definida

## ▼ █ Parte 8: Sustitución con patrones

( `{param:offset:len}` , `#` , `##` , `%` , `%%` , `^` , `,` , `,` , `,` )

Permite modificar cadenas quitando texto des de el inicio o final, según patrones.

### a) Substring (extraer parte de una cadena)

```
 ${param:offset}  
 ${param:offset:len}
```

- `offset` → posición inicial (empieza en 0).
- `len` → longitud opcional a extraer.

Ejemplo:

```
msg="abcdef"  
echo ${msg:2}      # cdef  
echo ${msg:2:3}    # cde
```

### b) Longitud de la cadena

```
 ${#param}
```

Devuelve la longitud de `param`.

Ejemplo:

```
msg="hola"  
echo ${#msg} # 4
```



### Eliminar desde el inicio (# y ##)

#### ◆ Ejemplo práctico

```
texto="abc123123xyz"
```



#### 1. Con # y ## (inicio)

```
echo "${texto##*1}"
```



👉 quita desde el inicio hasta el primer 1:

```
23123xyz
```

```
echo "${texto##*1}"
```



👉 quita desde el inicio hasta el último 1:

```
23xyz
```



```
mensaje="holafeomundo"
```

Expresión	Resultado	Qué hace
<code> \${mensaje##hola}</code>	<code>feomundo</code>	Quita <code>hola</code> desde el inicio
<code> \${mensaje##*feo}</code>	<code>mundo</code>	Quita todo hasta <code>feo</code> (primer match)
<code> \${mensaje##*o}</code>	<code>mundo</code>	Quita todo hasta el último <code>o</code>



### Eliminar desde el final (% y %%)

#### 2. Con % y %% (final)

```
echo "${texto%3*}"
```



👉 quita desde el último 3 hasta el final:

```
abc12312
```

```
echo "${texto%3*}"
```

👉 quita desde el primer 3 hasta el final:

```
abc12
```

```
archivo="backup_2025.txt"
```

Expresión	Resultado	Qué hace
<code> \${archivo%.txt}</code>	<code>backup_2025</code>	Quita <code>.txt</code> del final (match corto)
<code> \${archivo%_*}</code>	<code>backup</code>	Quita todo desde el primer <code>_</code> (match largo)

🧠 \* es un comodín:

- = "cualquier cosa"
- Si no usas , solo coincide exactamente con texto literal

## ◆ Operadores de *Case Modification* en Bash

### 1. Mayúscula inicial

```
 ${var^}
```

Convierte la primera letra de la variable en mayúscula.

Ejemplo:

```
palabra="maribel"
echo "${palabra^}"    # Maribel
```

### 2. Todas mayúsculas

```
 ${var^^}
```

Convierte toda la cadena a mayúsculas.

Ejemplo:

```
palabra="Jose Miguel"
echo "${palabra^^}"    # JOSE MIGUEL
```

### 3. Minúscula inicial

```
 ${var,}
```



Convierte la **primera letra** de la variable en minúscula.

Ejemplo:

```
palabra="Maribel"
echo "${palabra,,}" # maribel
```

### 4. Todas minúsculas

```
 ${var,,}
```



Convierte **toda la cadena** a minúsculas.

Ejemplo:

```
palabra="JOSE MIGUEL"
echo "${palabra,,}" # jose miguel
```

## ◆ Combinaciones

Se pueden combinar con subcadenas o usar dentro de un bucle como ya hiciste:

- Primera letra en mayúscula, resto en minúscula:

```
inicial=${palabra:0:1}
resto=${palabra:1}
echo "${inicial^}${resto,,}"
```



- Forzar que todo quede en “capitalizado”:

```
for palabra in $nombres; do
    echo "${palabra^${palabra,,}}"
done
```



## ◆ Resumen rápido

- `^` → primera letra a MAYÚSCULA.
- `^,` → todo MAYÚSCULA.
- `,` → primera letra a minúscula.
- `,,` → todo minúscula.



## ▼ ■ Parte 9: Sustitución de órdenes y aritmética

### ✓ 1. Sustitución de órdenes

Permite ejecutar un comando y **reemplazar** la expresión por su salida.

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

- ◆ Ejecuta `date` y muestra su salida.



### Sintaxis válida:

- Moderna: `$(comando)`
- Antigua (menos recomendable): ``comando``

## ✓ 2. Sustitución aritmética

Permite hacer operaciones matemáticas con enteros:

```
echo $((2 + 3))      # Resultado: 5
```

- ◆ Se usa para sumar, restar, multiplicar, etc.



### Operadores comunes:

Operador	Significado
<code>+ - * / %</code>	Básicos
<code>**</code>	Potencia
<code>++ --</code>	Incrementos
<code>== != &gt; &lt; &gt;= &lt;= &amp;&amp;</code>	Comparaciones

🧠 `$(expresión) ≠ $(comando)`

- `$( )` → hace cuentas
- `$( )` → ejecuta comandos



## ✍ ¿Cómo probarlos?

Puedes usar líneas como estas en la terminal:

```
x=8
echo $((x ** 2))      # Potencia: 64

x=15
((x %= 4))            # Módulo y asignar: x = 3
echo $x

a=3; b=2
echo ${a:b?1:0}        # Operador ternario: imprime 1
```



## ▼ 📄 Parte 10: Entrecomillado (Quoting)

El shell interpreta caracteres especiales como `$`, `*`, `;`, `&`, etc.

El quoting sirve para protegerlos y evitar errores.

## ✓ 1. Backslash (\) → Protege un solo carácter

```
echo Hola\;Mundo
```

- ◆ El `;` no se interpreta como "fin de comando"

## ✓ 2. Comillas simples (`'`) → Protege todo el contenido

```
echo '<-$1250.*>; (update?) [y|n]'
```

- ◆ Nada se interpreta: ni `$`, ni `*`, ni `;`, etc.

## ✓ 3. Comillas dobles (`"`) → Protege todo excepto `$`, `\` y `$(...)`

```
echo "$USER tiene \$100 [$(date +%Y-%m-%d)]"
```

- ◆ El `$USER` y `$(date ...)` sí se interpretan
- ◆ El resto queda protegido

### 📌 Regla de oro:

Quoting	¿Evalúa <code>\$</code> , <code>\$(...)</code> ?	¿Protege texto?
<code>'texto'</code>	✗ No	<input checked="" type="checkbox"/> Totalmente
<code>"texto"</code>	<input checked="" type="checkbox"/> Sí	<input checked="" type="checkbox"/> Parcialmente
<code>\carácter</code>	✗ No	<input checked="" type="checkbox"/> Solo ese carácter

## ▼ █ Parte 11: Estructuras de control – `if`, `test`, `[ ]`

Permiten evaluar condiciones y tomar decisiones en los scripts.

### ✓ Sintaxis básica:

```
if [ condición ]; then
    # instrucciones si la condición es verdadera
else
    # si es falsa
fi
```

### ✓ Uso de `test` o `[ ]`:

Son equivalentes:

```
test "$a" = "$b"
[ "$a" = "$b" ]
```

### ⚠ Espacios son obligatorios:

<code>[ "\$a" = "\$b" ]</code>	<input checked="" type="checkbox"/>
<code>["\$a"="\$b"]</code>	✗

## Tipos de condiciones:

### ◆ a) Archivos

Expresión	Significado
<code>-e archivo</code>	¿Existe el archivo o carpeta?
<code>-f archivo</code>	¿Es un archivo regular?
<code>-d carpeta</code>	¿Es un directorio?
<code>-r archivo</code>	¿Tiene permisos de lectura?
<code>-x archivo</code>	¿Es ejecutable?

### ◆ b) Cadenas (texto)

Expresión	Significado
<code>-z "\$var"</code>	¿Cadena vacía?
<code>-n "\$var"</code>	¿Cadena NO vacía?
<code>"\$a" = "\$b"</code>	¿Son iguales?
<code>"\$a" != "\$b"</code>	¿Son diferentes?

### ◆ c) Números

Expresión	Significado
<code>-eq</code>	Igual a
<code>-ne</code>	No igual
<code>-lt , -le</code>	Menor, menor o igual
<code>-gt , -ge</code>	Mayor, mayor o igual

## 1. Expresiones compuestas: complemento, AND y OR

En Bash puedes combinar condiciones en una misma expresión con:

### a) Complemento (negación): `!`

```
if [ ! -z "$nombre" ]; then
    echo "La variable NO está vacía"
fi
```

`!` niega el resultado de la condición.

### b) AND lógico: `&&`

```
if [ "$edad" -ge 18 ] && [ "$pais" = "Peru" ]; then
    echo "Eres mayor de edad y vives en Perú"
fi
```

Ambas condiciones deben ser verdaderas para que se ejecute el bloque.

### c) OR lógico: ||

```
if [ "$dia" = "sábado" ] || [ "$dia" = "domingo" ]; then
    echo "Es fin de semana"
fi
```

Se ejecuta si al menos una condición es verdadera.

### Alternativa avanzada (dobles paréntesis):

También puedes usar `[[ ... ]]` en lugar de `[ ... ]`, lo cual permite más flexibilidad:

```
if [[ -n "$nombre" && "$edad" -ge 18 ]]; then
    echo "Nombre válido y mayor de edad"
fi
```

## 2. Sentencia case

Es una forma **ordenada** de hacer muchas comparaciones de valores (como un `switch` en otros lenguajes).

### Sintaxis:

```
case $variable in
    valor1)
        # comandos
        ;;
    valor2)
        # comandos
        ;;
    *)
        # comandos si no coincide ningún caso
        ;;
esac
```

### Ejemplo:

```
read -p "Ingrese una opción [a/b/c]: " opcion

case $opcion in
    a)
        echo "Elegiste A"
        ;;
    b)
        echo "Elegiste B"
        ;;
    c)
        echo "Elegiste C"
        ;;
    *)
        echo "Opción no válida"
        ;;
esac
```

## 📌 Notas:

- Los `;;` marcan el fin de cada caso.
- El `El` actúa como "default" (si no coincide con ninguno).

## ▼ █ Parte 12: Lazos (`while`, `until`, `for`)

Los lazos permiten repetir instrucciones automáticamente.

### ✓ 1. `while` → mientras la condición sea verdadera

```
contador=1
while [ $contador -le 3 ]; do
    echo "Contador: $contador"
    contador=$((contador + 1))
done
```

- ◆ Repite mientras `$contador` sea menor o igual que 3.

### ✓ 2. `until` → mientras la condición sea falsa

```
contador=1
until [ $contador -gt 3 ]; do
    echo "Contador: $contador"
    contador=$((contador + 1))
done
```

- ◆ Funciona igual que el anterior, pero con lógica inversa.

### ✓ 3. `for` → para cada elemento en una lista

```
for fruta in manzana uva pera; do
    echo "Fruta: $fruta"
done
```

- ◆ Imprime cada elemento de la lista.

### ✓ Contar con `for` y `seq`

```
for i in $(seq 1 5); do
    echo "Número: $i"
done
```

## ▼ █ Parte 13: Lazos anidados, infinitos, `break` y `continue`

### ✓ 1. Lazos anidados (`while` dentro de `while`)

Útiles para imprimir estructuras o recorrer matrices.

```
i=0
while [ $i -le 3 ]; do
    j=$i
    while [ $j -ge 0 ]; do
```

```

        echo -n "$j "
        j=$((j - 1))
    done
    echo
    i=$((i + 1))
done

```

◆ Imprime:

```

0
1 0
2 1 0
3 2 1 0

```

## ✓ 2. Lazo infinito

```

while true; do
    echo "Infinito..."
done

```

## ✓ 3. `break` → salir del lazo

```

while true; do
    read -p "¿Salir? (s/n): " respuesta
    [ "$respuesta" = "s" ] && break
done

```

## ✓ 4. `continue` → saltar a la siguiente iteración

```

for i in $(seq 1 5); do
    [ $i -eq 3 ] && continue
    echo "Número: $i"
done

```

◆ Salta el número 3.

## ▼ ■ Parte 14: Parámetros del script (`$0`, `$1`, `$@`, etc.)

Cuando ejecutas un script con argumentos:

```
bash script.sh uno dos tres
```

Bash los guarda en variables especiales:

## ✓ Parámetros posicionales

Variable	Significado
<code>\$0</code>	Nombre del script ( <code>script.sh</code> )
<code>\$1</code> , <code>\$2</code> , <code>\$3</code> ...	Argumentos recibidos ( <code>uno</code> , <code>dos</code> , ...)

## ✓ Variables especiales

Variable	¿Qué hace?
<code>\$#</code>	Número de argumentos
<code>\$*</code>	Todos los argumentos como una cadena
<code>\$@</code>	Todos los argumentos como lista individual
<code>\$?</code>	Código de salida del último comando
<code>\$\$</code>	ID del proceso del script actual

### 📝 Ejemplo:

```
echo "Script: $0"
echo "Argumentos: $*"
echo "Número de argumentos: $#"
```

Con:

```
bash ejemplo.sh hola mundo
```

Resultado:

```
Script: ejemplo.sh
Argumentos: hola mundo
Número de argumentos: 2
```

## ✓ Diferencia entre `$*` y `$@`

Cuando los usas entre comillas:

- `"$*"` → todo como una sola cadena
- `"$@"` → cada argumento como unidad separada

## 💡 Diferencia entre `$*` y `$@`

Ambos representan todos los argumentos pasados al script o función.

La diferencia solo aparece cuando los pones entre comillas dobles (`" "`).

## ✓ `$*` — todos los argumentos como una sola cadena

```
#!/bin/bash
for arg in "$*"; do
    echo "ARG: $arg"
done
```

Si ejecutas:

```
bash script.sh uno dos tres
```

◆ Resultado:

```
ARG: uno dos tres
```



### ✓ `$@` — todos los argumentos como varios elementos independientes

```
#!/bin/bash
for arg in "$@"; do
    echo "ARG: $arg"
done
```

◆ Resultado:

```
ARG: uno
ARG: dos
ARG: tres
```



### 📌 En resumen:

Expresión	¿Cómo lo interpreta el <code>for</code> si hay " " ?
<code>"\$*"</code>	Un solo argumento: <code>"uno dos tres"</code>
<code>"\$@"</code>	Tres argumentos separados: <code>"uno"</code> <code>"dos"</code> <code>"tres"</code>

## ▼ █ Parte 15: Funciones en Bash



### ✓ ¿Qué es una función?

Es un bloque de código reutilizable dentro de un script.

Sirve para organizar mejor el código, evitar repeticiones y modularizar tareas.

### ✓ Sintaxis:

```
nombre_funcion() {
    # instrucciones
}
```

Y para invocarla:

```
nombre_funcion
```



### ✍ Ejemplo simple:

```
saludar() {
    echo "Hola, $1"
}

saludar Max
saludar Ana
```

- ◆ Usa `$1` como primer argumento de la función.



## ✓ Retorno de valores

### ✓ Forma 1: usando variables globales

```
sumar() {
    resultado=$(( $1 + $2 ))
}
sumar 3 4
echo "Resultado: $resultado"
```

### ✓ Forma 2: con `return` (solo valores 0–255)

```
sumar() {
    return $(( $1 + $2 ))
}
sumar 3 4
echo $? # Muestra: 7
```

⚠ No sirve para textos ni para números grandes.

✓ Puedes usar `$1`, `$2`, `$@`, `$#` dentro de funciones igual que en scripts.

