Comandi Shell

- mkdir

  Crea una directory

  ```
  mkdir dir
  ```

- rmdir

  Cancella la directory se vuota

  ```
  rmdir dir
  ```

- cd

  Modifica la directory corrente

  ```
  cd dir
  ```

- pwd

  Mostra la directory corrente

  ```
  pwd
  ```

- ls

  Mostra l'elenco dei file/directory

  ```
  ls [dir/file]
  ```

- cp

  Copia il file Src nel file Dst, se Dst è directory copia il file Src dentro la directory Dst

  ```
  cp Src Dst
  ```

- ln

  Crea un link tra file (a seconda delle opzioni può creare un link hardware o software)

  I link puntano alla stessa area di memoria, quindi allo stesso file

```
#per creare link hardware
ln file1 file2

#per creare un link software
ln -s file1 file2
```

link software: se elimino o sposto il file originale il link non funziona più

link hardware: se elimino o sposto file originale il link resta funizionante(incrementa contatore i-node [il file viene eliminato del tutto quando il contatore va a 0])

- mv

  Sposta file e directory in una directory oppure serve per rinominare file o directory

  ```
  mv file/dir file/dir
  ```

- rm

  Elimina il link file se il contatore è uguale a 0 elimina il file

  ```
  rm file/dir

  #per chiedere conferma all'utente
  rm -i file/dir

  #per eliminare il contenuto di una cartella ricorsivamente
  rm -r dir
  ```

- cat

  Il comando `cat` (abbreviazione di concatenate) è utilizzato per leggere, creare e concatenare file. Stampa il contenuto di uno o più file sulla shell.

  ```
  # Legge il contenuto di un file
  cat file

  # Crea un nuovo file e digita il contenuto
  cat > nuovo_file

  # Concatena due file e stampa il contenuto
  cat file1 file2
  ```

  Di seguito sono riportate alcune delle opzioni più comuni:

  - `n` o `-number` aggiunge numeri di riga a tutte le righe.
  - `b` o `-number-nonblank` aggiunge numeri di riga solo alle righe non vuote.
  - `s` o `-squeeze-blank` sostituisce più linee vuote consecutive con una sola linea vuota.
  - `E` o `-show-ends` mostra il carattere `$` alla fine di ogni riga.

- `T` o `-show-tabs` mostra i caratteri di tabulazione come `^I`.

Esempi:

```
# Aggiunge numeri di riga
cat -n file

# Aggiunge numeri di riga alle righe non vuote
cat -b file

# Sostituisce più linee vuote con una sola linea vuota
cat -s file

# Mostra il carattere $ alla fine di ogni riga
cat -E file

# Mostra i caratteri di tabulazione
cat -T file
```

- more

  Funziona come cat ma per ogni file ci sono 3 righe di intestazione

```
more file1 ... fileN
```

- chmod

  Cambia i permessi di un file

```
chmod [u g o a] [+ -] [rwx]

#i primi parametri servono per indicare il soggetto a cui cambia il permesso
#rispettivamente u=user, g=group, o=other, a=all
#secondo parametro per dare o togliere permesso
#terzo per decidere quale permesso dare: r=read, w=write, x=execution
```

- chown

  Cambia proprietario di una directory/file

```
chown nome_utente file/dir
```

- chgrp

  Cambia gruppo proprietario di una directory/file

```
chgrp nome_gruppo file/dir
```

- sh

  Per usare una shell

  ```
  sh
  #se voglio eseguire un file .sh senza usare prima chmod
  sh file.sh
  ```

- bash

  Per usare la bash

  ```
  bash
  ```

- ps

  Il comando `ps` in Bash è un'utilità potente per visualizzare informazioni sui processi in esecuzione nel sistema.

  ### Esempi di utilizzo:

  1. Mostra tutti i processi con una visualizzazione dettagliata:

  ```
  ps -ef
  ```

  1. Mostra solo i processi del tuo utente:

  ```
  ps -u $USER
  ```

  1. Visualizza solo i processi con un determinato nome di comando:

  ```
  ps -C sshd
  ```

  1. Visualizza solo le informazioni PID specifiche:

  ```
  ps -p 1234 5678
  ```

  1. Personalizza il formato dell'output per visualizzare solo PID, nome del comando e stato del processo:

  ```
  ps -o pid,cmd,state
  ```

  ### Opzioni del comando `ps`:

  1. **e, -A**: Mostra tutti i processi.

    - `e` è compatibile con POSIX, mentre `A` è un'estensione GNU.
2. **f**: Visualizza una visualizzazione dettagliata dei processi.
    - Questa opzione fornisce informazioni estese come l'utente proprietario, l'ID del processo, l'ID del processo genitore, il tempo di CPU utilizzato, lo stato del processo, il comando completo e altro ancora.
3. **u**: Mostra informazioni dettagliate sull'utente proprietario dei processi.
    - Questa opzione include il nome utente, l'ID utente, l'ID del gruppo, l'ID del gruppo effettivo, il tempo di CPU utilizzato e altro ancora.
4. **p**: Mostra informazioni specifiche per i processi elencati nell'elenco di PID fornito.
    - È possibile specificare uno o più PID separati da spazi per visualizzare solo le informazioni relative a quei processi.
5. **o formato**: Specifica il formato dell'output personalizzato.
    - Con questa opzione è possibile specificare quali colonne visualizzare nell'output e il loro ordine.
6. **C comando**: Mostra i processi che corrispondono al nome del comando specificato.
    - Ad esempio, `ps -C bash` mostrerà tutti i processi il cui comando è "bash".

- login

  Eseguire il Login

  ```
  login
  ```

- logout

  Eseguire il Logout

  ```
  logout
  ```

- exit

  Per uscire dalla shell e interrompere il processo

  ```
  exit
  #molto utile con case per gestire i casi in cui dobbiamo interrompere il
  programma
  case $# in
  0) echo OK;;
  *) echo Error
           exit 1;;
  esac
  #il numero dopo l'exit serve per differenziare le varie casistiche di errore
  ```

- tee

  Legge l'input standard e lo scrive sia sullo standard output che in uno o più file.

  ```
  echo 'ciao' | tee file.txt
  ```

- env

  Mostra l'elenco delle variabili di ambiente o esegue un comando in un ambiente modificato.

```
# Per visualizzare tutte le variabili d'ambiente
env

# Per eseguire un comando in un ambiente specifico
env VAR1="valore1" VAR2="valore2" comando
```

- export

  Il comando `export` in Bash viene utilizzato per esportare variabili nell'ambiente, in modo che siano disponibili per i processi figli.

```
export [opzioni] [NOME=VALORE ...]
```

```
# Esporta una variabile nell'ambiente
export MIO_NOME="Mario"
```

- expr

  Il comando `expr` in Bash è utilizzato per valutare espressioni aritmetiche o di stringhe e stampare il risultato.

```
expr [options] expression
```

  Esempio:

```
# Valutazione di un'espressione aritmetica
risultato=$(expr 5 + 3)
echo "Il risultato è $risultato"
```

- eval

  Il comando `eval` in Bash viene utilizzato per eseguire una stringa come un comando Bash. Questo può essere utile quando si desidera eseguire dinamicamente del codice generato o memorizzato in una variabile.

```
eval [options] arguments
```

  Esempio:

```
# Definizione di una stringa contenente una pipeline
pipeline="ls -l | grep .txt"
```

```
# Esecuzione della stringa come un comando usando eval
eval $pipeline
```

- set

  Il comando `set` in Bash viene utilizzato per modificare il comportamento del shell e impostare variabili di shell, opzioni e argomenti della riga di comando

  ```
  set [options][arguments]
  ```

  Esempio:

  ```
  # Imposta una variabile di shell
  set var=Hello
  ```

  ### Uso delle opzioni:

  - `e`: Interrompe lo script se si verifica un errore (uscita non zero).
  - `u`: Interrompe lo script se viene utilizzata una variabile non definita.
  - `x`: Stampa ogni comando prima di eseguirlo.

- shift

  il comando `shift` viene utilizzato per spostare gli argomenti della riga di comando (posizionali) in avanti, facendo sì che l'argomento successivo diventi il primo. Questo è utile quando si desidera iterare attraverso un elenco di argomenti di lunghezza variabile.

  ```
  shift [n]
  ```

  Esempio:

  ```
  # Mostra tutti gli argomenti iniziali
  echo "Argomenti iniziali: $*"

  # Sposta gli argomenti di due posizioni
  shift 2

  # Mostra gli argomenti spostati
  echo "Argomenti dopo lo shift: $*"
  ```

- if

  Il comando `if` è utilizzato per creare condizioni nei script bash. Puoi utilizzare il comando `if` per eseguire comandi basati su condizioni logiche.

  Ecco uno schema di base del comando `if`:

```
if [ condizione ]
then
  comando
fi
```

Le condizioni possono essere espressioni di confronto numeriche o di stringhe. Alcuni operatori comuni includono:

- `eq`: uguale a
- `ne`: diverso da
- `gt`: maggiore di
- `ge`: maggiore o uguale a
- `lt`: minore di
- `le`: minore o uguale a
- `=`: uguale a (per le stringhe)
- `!=`: diverso da (per le stringhe)

Esempi:

```
# Confronto numerico
if [ "$a" -eq "$b" ]
then
  echo "a è uguale a b"
fi

# Confronto di stringhe
if [ "$a" = "$b" ]
then
  echo "a è uguale a b"
fi
```

Puoi anche utilizzare `else` e `elif` (else if) per aggiungere più condizioni:

```
if [ condizione ]
then
  comando1
elif [ altra_condizione ]
then
  comando2
else
  comando3
fi
```

- case

Il comando `case` è utilizzato per eseguire comandi specifici in base al valore di una variabile o di un'espressione. È simile all'istruzione `switch` in altri linguaggi di programmazione.

Ecco uno schema di base del comando `case`:

```
case espressione in
  pattern1)
    comandi
    ;;
  pattern2)
    comandi
    ;;
  *)
    comandi_default
    ;;
esac
```

Nell'esempio precedente, se `espressione` corrisponde a `pattern1`, verranno eseguiti i comandi sotto `pattern1`. Se `espressione` corrisponde a `pattern2`, verranno eseguiti i comandi sotto `pattern2`. Se `espressione` non corrisponde a nessuno dei pattern, verranno eseguiti i `comandi_default`.

Esempio:

```
nome="Mario"

case $nome in
  "Mario")
    echo "Ciao, Mario!"
    ;;
  "Luigi")
    echo "Ciao, Luigi!"
    ;;
  *)
    echo "Non riconosco il tuo nome."
    ;;
esac
```

In questo esempio, se la variabile `nome` è uguale a "Mario", verrà stampato "Ciao, Mario!". Se `nome` è uguale a "Luigi", verrà stampato "Ciao, Luigi!". In tutti gli altri casi, verrà stampato "Non riconosco il tuo nome.".

- head

Il comando `head` in Bash è utilizzato per leggere le prime righe di un file. Di default, mostra le prime 10 righe.

```
# Visualizza le prime 10 righe del file
head file
```

Di seguito sono riportate alcune delle opzioni più comuni:

- `n` o `number` visualizza le prime `n` righe del file.
- `c` o `bytes` visualizza i primi `bytes` byte del file.
- `q` o `quiet` o `silent` non visualizza i nomi dei file quando si lavora con più file.
- `v` o `verbose` visualizza i nomi dei file quando si lavora con più file.

Esempi:

```
# Visualizza le prime 5 righe del file
head -n5 file

# Visualizza i primi 20 byte del file
head -c20 file

# Non visualizza i nomi dei file quando si lavora con più file
head -q file1 file2 file3

# Visualizza i nomi dei file quando si lavora con più file
head -v file1 file2 file3
```

- tail

Il comando `tail` in Bash è utilizzato per leggere le ultime righe di un file. Di default, mostra le ultime 10 righe.

```
# Visualizza le ultime 10 righe del file
tail file
```

Di seguito sono riportate alcune delle opzioni più comuni:

- `n` o `number` visualizza le ultime `n` righe del file.
- `f` o `follow` visualizza le righe aggiunte al file mentre è aperto.
- `q` o `quiet` o `silent` non visualizza i nomi dei file quando si lavora con più file.
- `v` o `verbose` visualizza i nomi dei file quando si lavora con più file.

Esempi:

```
# Visualizza le ultime 5 righe del file
tail -n5 file

# Visualizza le righe aggiunte al file mentre è aperto
tail -f file

# Non visualizza i nomi dei file quando si lavora con più file
tail -q file1 file2 file3

# Visualizza i nomi dei file quando si lavora con più file
tail -v file1 file2 file3
```

- read

Il comando `read` in Bash è utilizzato per leggere l'input da tastiera (standard input). È spesso utilizzato in un ciclo `while` per leggere linee di testo da un file o da un flusso di dati.

Ecco un esempio base:

```
echo "Inserisci il tuo nome:"
read nome
echo "Ciao, $nome"
```

In questo esempio, il comando `read` attende che l'utente inserisca del testo. Quando l'utente preme Invio, il testo inserito viene assegnato alla variabile `nome`.

Alcune delle opzioni più comuni includono:

- `p`: permette di visualizzare un prompt prima dell'input dell'utente.
- `a`: legge l'input in un array.
- `n`: legge solo un determinato numero di caratteri.
- `t`: imposta un timeout per l'input dell'utente.
- `s`: nasconde l'input dell'utente (utile per le password).

Esempi:

```
# Utilizza un prompt per l'input dell'utente
read -p "Inserisci il tuo nome: " nome
echo "Ciao, $nome"

# Legge l'input in un array
echo "Inserisci tre numeri:"
read -a numeri
echo "Hai inserito: ${numeri[0]}, ${numeri[1]}, ${numeri[2]}"

# Legge solo i primi due caratteri
read -n2 -p "Inserisci due lettere: " lettere
echo
echo "Hai inserito: $lettere"

# Imposta un timeout per l'input dell'utente
read -t5 -p "Hai 5 secondi per inserire il tuo nome: " nome
echo
echo "Hai inserito: $nome"

# Nasconde l'input dell'utente
read -s -p "Inserisci la tua password: " password
echo
echo "Password segreta inserita."
```

- rev

Il comando `rev` di bash è utilizzato per ribaltare le linee di testo. Stampa le linee con i caratteri in ordine inverso.

```
# Stampa la linea "ciao" con i caratteri in ordine inverso
echo 'ciao' | rev
```

Questo comando restituirà "oaic", che è "ciao" scritto al contrario.

`rev` non ha molte opzioni. Puoi usare `-V` o `--version` per ottenere informazioni sulla versione, e `-h` o `--help` per ottenere un aiuto sull'uso del comando:

```
# Stampa informazioni sulla versione di rev
rev -V

# Stampa aiuto sull'uso del comando rev
rev -h
```

- grep

  Il comando grep ricerca un file per una stringa di testo e stampa ogni linea che contiene quella stringa. Il nome "grep" deriva da "global regular expression print", che è una funzione disponibile nell'editor di testo ed.

  ```
  grep 'testo da cercare' file
  ```

  Di seguito sono riportate alcune delle opzioni più comuni:

  - `i` rende la ricerca insensibile al maiuscolo/minuscolo.
  - `r` o `R` ricerca in modo ricorsivo.
  - `v` inverte la ricerca, restituendo solo le linee che non contengono la stringa di testo.
  - `l` restituisce solo i nomi dei file che contengono la stringa di testo.
  - `n` stampa il numero della linea con l'output.
  - `c` conta il numero di volte che la stringa specificata appare nel file.

  Esempi:

  ```
  # Ricerca insensibile al maiuscolo/minuscolo
  grep -i 'testo da cercare' file

  # Ricerca ricorsiva
  grep -r 'testo da cercare' directory

  # Ricerca invertita
  grep -v 'testo da cercare' file

  # Stampa solo i nomi dei file
  grep -l 'testo da cercare' *

  # Stampa il numero della linea con l'output
  grep -n 'testo da cercare' file

  # Conta il numero di volte che la stringa appare nel file
  grep -c 'testo da cercare' file
  ```

  Le espressioni regolari (o RegEx) sono una potente strumento per cercare e manipolare stringhe di testo utilizzando determinati modelli. Si possono utilizzare in molti comandi di shell, come `grep`.

Ecco alcuni esempi di utilizzo delle espressioni regolari:

```
# Cerca tutte le linee che iniziano con 'a'
grep '^a' file

# Cerca tutte le linee che terminano con 'b'
grep 'b$' file

# Cerca tutte le linee che contengono un numero
grep '[0-9]' file

# Cerca tutte le linee che contengono una lettera
grep '[a-zA-Z]' file

# Cerca tutte le linee che contengono uno spazio
grep ' ' file

# Cerca tutte le linee che contengono una parola di esattamente tre lettere
grep '\\b[a-zA-Z]{3}\\b' file

# Cerca tutte le linee che contengono una parola che inizia con 'a' e finisce
con 'b'
grep '\\ba[a-zA-Z]*b\\b' file

# Cerca tutte le linee che non contengono numeri
grep -v '[0-9]' file
```

- sort

  Ordina le righe di tutti i file specificati in input.

```
# Ordina le righe del file in ordine alfabetico.
sort file

# Ordina le righe del file in ordine alfabetico inverso.
sort -r file

# Ordina le righe del file in base al secondo campo (campi separati da spazi).
sort -k2 file

# Ordina le righe del file in base al secondo campo (campi separati da
virgola).
sort -t, -k2 file

# Ordina le righe del file in base al secondo campo (campi separati da virgola)
in ordine numerico.
sort -t, -k2 -n file

# Ordina le righe del file ignorando le maiuscole.
sort -f file
```

- test

Il comando `test` è utilizzato per verificare e confrontare vari valori. Puoi utilizzarlo per controllare se una variabile ha un certo valore, se esiste un file, se le dimensioni di un file sono maggiori di un certo valore, ecc.

```
# Verifica se il file esiste
test -e file

# Verifica se la directory esiste
test -d directory

# Verifica se il file è leggibile
test -r file

# Verifica se il file è scrivibile
test -w file

# Verifica se il file è eseguibile
test -x file

# Verifica se la stringa è vuota
test -z stringa

# Verifica se la stringa non è vuota
test -n stringa

# Verifica se la stringa1 è uguale alla stringa2
test stringa1 = stringa2

# Verifica se la stringa1 è diversa dalla stringa2
test stringa1 != stringa2

# Verifica se il numero1 è uguale al numero2
test numero1 -eq numero2

# Verifica se il numero1 è diverso dal numero2
test numero1 -ne numero2

# Verifica se il numero1 è minore del numero2
test numero1 -lt numero2

# Verifica se il numero1 è maggiore del numero2
test numero1 -gt numero2
```

In alternativa, puoi utilizzare il comando `test` senza la parola `test`, utilizzando solo le parentesi quadre

Nell'esempio seguente, il comando `if` viene utilizzato in combinazione con `test` per verificare se un file esiste.

```
file="test.txt"

if test -e $file
then
  echo "Il file esiste."
else
  echo "Il file non esiste."
```

```
fi
```

In questo esempio, se il file "test.txt" esiste, verrà stampato "Il file esiste.". Se il file "test.txt" non esiste, verrà stampato "Il file non esiste.".

Un altro esempio utilizza `if` e `test` per controllare se una stringa è vuota.

```
stringa=""

if test -z $stringa
then
  echo "La stringa è vuota."
else
  echo "La stringa non è vuota."
fi
```

In questo esempio, se la variabile `stringa` è vuota, verrà stampato "La stringa è vuota.". Se la variabile `stringa` non è vuota, verrà stampato "La stringa non è vuota.".