

# Bash Guide & Command Cheatsheet — A dummy-to-dummy summary

- ✍️ **Nicholas Magi**, [nicholas.magi@studio.unibo.it](mailto:nicholas.magi@studio.unibo.it),
  - con il contributo di **Gioele Foschi** - [gioele.foschi@studio.unibo.it](mailto:gioele.foschi@studio.unibo.it)
- Corso di **Sistemi Operativi** @ CdL in Ingegneria e Scienze Informatiche, Università di Bologna — Campus di Cesena.
- Riassunto e schematizzazione delle dispense del prof. **Vittorio Ghini**.
- Scritto su: **Obsidian** — consigliato il suo utilizzo poiché alcuni contenuti non sono direttamente fruibili da altri interpreti di Markdown.
- Link utile: <https://www.docstomarkdown.pro/markdown-table-of-contents-generator-free/> — *generatore di indice dei contenuti*.

## Indice dei contenuti

- [Bash Guide & Command Cheatsheet — A dummy-to-dummy summary](#)
  - [Nozioni per uso del terminale: Metacaratteri](#)
  - [Nozioni per uso del terminale: Espansioni](#)
  - [Nozioni per uso del terminale: Variabili](#)
    - [Variabili note: PATH](#)
    - [Variabili note: \\$BASHPID e \\$\\$](#)
    - [Riferimenti indiretti a variabili](#)
    - [Manipolare il contenuto della variabile](#)
      - [Lunghezza del contenuto di una variabile](#)
      - [Rimozione di suffissi](#)
      - [Rimozione di prefissi](#)
      - [Sostituzione](#)
        - [Variazioni di comportamento: sostituzione TOTALE](#)
        - [Variazioni di comportamento: sostituzione all'INIZIO](#)
        - [Variazioni di comportamento: sostituzione alla FINE](#)
      - [Substring](#)
      - [Espansione di nomi di variabili corrispondenti ad un prefisso](#)
  - [Nozioni per uso del terminale: Permessi di file e directory](#)
  - [Nozioni per uso del terminale: Wildcards](#)
    - [Wildcard elenco](#)
  - [Nozioni per uso del terminale: Parameter Expansion](#)

- Nozioni per uso del terminale: *Valutazione Aritmetica*
  - Nozioni per uso terminale: *Espressioni condizionali*
  - Valori di verità
  - Operatori logici
  - Versioni
  - Operatori utili
    - Su FILES
    - Su STRINGHE
- Nozioni per uso del terminale: *Liste di comandi*
- Nozioni per uso del terminale: *Compound Commands*
  - Cicli — costruito iterazione
  - If — costruito selezione
- Nozioni per uso del terminale: *Command substitution*
- Nozioni per uso del terminale: *Ridirezionamenti di Stream I/O*
- Nozioni per uso del terminale: *Raggruppamento di comandi*
- Nozioni per uso del terminale: *Processi*
  - Processi in *foreground*
  - Processi in *background*
  - Jobs
  - Job control
  - Processi Zombie & Orfani
- Nozioni per uso del terminale: *Precedenza degli operatori*
  - Terminatori di una sequenza di comandi
  - "Concatenatori" di una sequenza di comandi
  - Ordine
- Cheatsheet comandi
  - Generics
  - Lettura & gestione file descriptor
    - Visualizzare i file descriptor associati ad un processo
    - read
    - exec
    - Manipolazione di stringhe & informazioni testuali
      - head
      - tail
      - tee
      - cut
      - grep

- [sed — Stream Editor](#)
- [Gestione processi](#)
  - [disown](#)
  - [nohup](#)

---

## Nozioni per uso del terminale: *Metacaratteri*

METACARATTERI	SIGNIFICATO
> >> <	redirezione I/O
\	pipe
* ? [...]	wildcards
`command`	command substitution (use <b>backticks</b> )
;	esecuzione <b>sequenziale</b> - <b>separatore di comandi</b>
\  \  &&	esecuzione <b>condizionale</b>
(...)	raggruppamento comandi
&	esecuzione in <b>background</b>
" " ' '	quoting
#	commento
\$	espansione di variabile
\	carattere di escape *
<<	"here document"

---

## Nozioni per uso del terminale: *Espansioni*

In ordine di effettuazione

ESPANSIONE	ESEMPIO
1) history expansion	!123
2) brace expansion	a{damn,czk,bubu}e
3) tilde expansion	~/nomedirectory

ESPANSIONE	ESEMPIO
4) parameter and variable expansion	<code>\$1 \$? \$! \${var}</code>
5) arithmetic expansion	<code>\$(( ))</code>
6) command substitution LTR <sup>1</sup>	<code>\$( )</code>
7) word spitting	
8) pathname expansion	<code>* ? [...]</code>
9) quote removal	quoting

1 — LTR: *Left To Right*

## Nozioni per uso del terminale: *Variabili*

ESEMPIO	SIGNIFICATO
<code>MYVAR=PAROLA</code>	<code>MYVAR</code> ha valore <code>PAROLA</code> ; <b>unico modo di assegnare un valore ad una variabile</b> , non mettere spazi prima/dopo o in mezzo!
<code>echo \${MYVAR}</code>	stampo il valore di <code>MYVAR</code> , quindi <code>PAROLA</code> .
<code>echo \${MYVAR}x</code>	stampo il valore di <code>MYVAR</code> seguito dal carattere <code>x</code> , quindi visualizzerò <code>PAROLAx</code> .
<code>echo \$MYVAR</code>	stampo il valore di <code>MYVAR</code> , quindi <code>PAROLA</code> .
<code>echo \$MYVARx</code>	la shell non è in grado di trovare una variabile <code>MYVARx</code> , quindi stampa <b>stringa vuota</b> .
<code>echo \$MYVAR x</code>	la shell individua correttamente la variabile <code>MYVAR</code> , quindi visualizzerà <code>PAROLA x</code> .

## Variabili note: `PATH`

`PATH` è una variabile d'ambiente che contiene i percorsi assoluti di alcuni eseguibili utilizzati molto spesso.

### Info

Esempio di un possibile valore di path:

```
/bin:/sbin:/usr/bin:/usr/local/bin:/home/nickolausen
```

## Variabili note: `$BASHPID` e `$$`

Ogni processo è identificato da un codice numerico identificativo detto **PID** (**P**rocess **I**Dentifier);  
Per visualizzare il PID di un processo si fa riferimento alla variabile `$$`;

```
> echo $$  
2606
```

### ⚠ Eccezione di comportamento!

`$$` si riferisce al PID della shell padre di un processo. Se voglio vedere il PID di una sub-shell lanciata da un gruppo di comandi, devo utilizzare la variabile `$BASHPID`!

### ⚠ Attenzione alla portabilità

`$BASHPID` è definita solo in bash e solo per le versioni di bash > 4.0!

```
echo "PID fuori $$" ; ( echo "PID dentro $$" ; echo -n "" )
```

```
# Output SENZA USO DI $BASHPID  
PID fuori 1760  
PID dentro 1760
```

```
echo "PID fuori $$" ; ( echo "PID dentro $BASHPID" ; echo -n "" )
```

```
# Output CON USO DI $BASHPID  
PID fuori 1760  
PID dentro 2042
```

## Riferimenti indiretti a variabili

Operatore `!`:

- se `IDX=1`, `${!IDX} == $1`;
- se `IDX=2`, `${!IDX} == $2`;
- se `IDX=3`, `${!IDX} == $3`;
- ecc...

## Manipolare il contenuto della variabile

### Warning

Le seguenti espansioni ***non vanno a modificare la variabile***, ma mostrano solamente la modifica apportata.

Supponiamo di avere la variabile `VAR="[13] qualcosa con [ 0 ] fine"`

## Lunghezza del contenuto di una variabile

```
${#VAR}
```

- Ottengo la lunghezza della stringa memorizzata in `VAR` ;
- `echo ${#VAR}` stampa in output: `28`

## Rimozione di *suffissi*

```
${VAR%%pattern}
```

- Rimuovo il ***più lungo*** *suffisso* che fa match con la stringa originale
- `echo ${VAR%%}*}` stampa in output: `[13]`

```
${VAR%pattern}
```

- Rimuovo il ***più corto*** *suffisso* che fa match con la stringa originale
- `echo ${VAR%}*}` stampa in output: `[13] qualcosa con [ 0`

## Rimozione di *prefissi*

```
${VAR##pattern}
```

- Rimuovo il ***più lungo*** *prefisso* che fa match con la stringa originale
- `echo ${VAR##[*]}` stampa in output: `0 ] fine`

```
${VAR#pattern}
```

- Rimuovo il ***più corto*** *prefisso* che fa match con la stringa originale
- `echo ${VAR#[*]}` stampa in output: `13] qualcosa con [ 0 ] fine`

## Sostituzione

```
${VAR/pattern/string}
```

- Cerca nel contenuto di `VAR` la *sottostringa più lunga* che fa match con il `pattern` fornito (anche con Wildcards) e lo *sostituisce* con `string`

### Variazioni di comportamento: sostituzione TOTALE

```
${VAR//pattern/string}
```

Come per una normale sostituzione, ma questa viene effettuata su **tutte le occorrenze** di `pattern`, non solo sulla prima trovata.

### Variazioni di comportamento: sostituzione all'INIZIO

```
${VAR/#pattern/string}
```

Come per una normale sostituzione, ma questa viene effettuata **solo se** `pattern` **si trova all'inizio della variabile**.

### Variazioni di comportamento: sostituzione alla FINE

```
${VAR/%pattern/string}
```

Come per una normale sostituzione, ma questa viene effettuata **solo se** `pattern` **si trova alla fine della variabile**.

## Substring

```
${VAR:offset:length}
```

- Mostra la **sottostringa** lunga `length` che parte dal carattere numero `offset` del contenuto di `VAR`

```
${VAR:offset}
```

- Mostra la **sottostringa** che parte dal carattere numero `offset` del contenuto di `VAR`

## Espansione di nomi di variabili corrispondenti ad un prefisso

```
${!VarNamePrefix*}
```

Restituisce un elenco con tutti i nomi delle variabili il cui nome inizia con il prefisso specificato `VarNamePrefix`.

*Esempio:* data l'esistenza delle seguenti variabili

```
BASH=/bin/bash
BASH_ALIASES=()
BASH_ARGC=()
BASH_ARGV=()
BASH_CMDS=()
BASH_LINENO=()
BASH_SOURCE=()
BASH_VERSION='4.1.17(9)-release'
CYG_SYS_BASHRC=1
```

Per vedere le variabili il cui nome inizia con `BASH_AR` devo digitare il comando:

```
echo ${!BASH_AR*}
```

```
# Output: BASH_ARGC BASH_ARGV
```

---

## Nozioni per uso del terminale: *Permessi di file e directory*

Ogni file appartiene ad un utente e ad un gruppo di cui l'utente fa parte: per cambiare owner, usare

```
chown <nuovoproprietario> <nomefile>
```

I permessi di un file sono identificati dai seguenti codici **letterali** e **ottali**, divisi nelle 3 categorie assegnabili: USER, GROUP e OTHERS.

USER			GROUP			OTHERS		
R	W	X	R	W	X	R	W	X
4	2	1	4	2	1	4	2	1



Esempio di assegnazione contemporanea di più permessi **mediante formato numerico**:

```
chmod 764 ./miofile.txt
```

Dove:

- 7 = 4 (lettura) + 2 (scrittura) + 1 (esecuzione) per livello **USER**;
- 6 = 4 (lettura) + 2 (scrittura) per livello **GROUP**;
- 4 (lettura) per **OTHERS**;

---

## Nozioni per uso del terminale: *Wildcards*

- \* -> sostituito con una **qualunque stringa di caratteri**;
- ? -> sostituito con un **qualunque carattere**;
- [ *elenco* ] -> sostituito con un qualunque carattere specificato in *elenco*;

### Wildcard *elenco*

WILDCARD	SOSTITUZIONE
[[[:digit:]]	una sola cifra, 0-9
[[[:upper:]]	un solo carattere <b>MAIUSCOLO</b>
[[[:lower:]]	un solo carattere <b>minuscolo</b>
[c-f]	un solo carattere compreso tra 'c' ed 'f'; <i>in questo caso, {c, d, e, f}</i>
[1-7]	un solo carattere compreso tra 1 e 7; <i>in questo caso, {1, 2, 3, 4, 5, 6, 7}</i>
[abk]	un solo carattere tra 'a', 'b' o 'k'

---

## Nozioni per uso del terminale: *Parameter Expansion*

Siamo sulla bash, chiamiamo il nostro script passandogli qualche argomento:

```
./faiqualcosa.sh argo1 mamma2 soreta4
```

Dentro a `faiqualcosa.sh` possiamo accedere a diverse informazioni identificate come:

- `$#` : numero di argomenti ricevuti (\*nell'esempio, **3\***);
- `$0` : nome del processo in esecuzione (\*nell'esempio,  `./faicalcosa.sh` \*)
- `$1`  - primo argomento,  `$2`  - secondo argomento... (nell'esempio,  `$1 = argo1` ,  `$2 = mamma2` ,  `$3 = soreta4` )
- `$*` : tutti gli argomenti ricevuti, concatenati e separati da spazi bianchi (nell'esempio,  `argo1 mamma2 soreta4` )
- `@$` : come per  `$*` , ma gli argomenti sono quotati separatamente (nell'esempio,  `"argo1" "mamma2" "soreta4"` )

In particolare:

```
$* == @$ ---> $1 $2 $3 ... $n

# Se quoto le due variabili, ottengo:
"$*" ---> "$1 $2 $3 ... $n"
"$@" ---> "$1" "$2" "$3" ... "$n"
```

## Nozioni per uso del terminale: *Valutazione Aritmetica*

**TUTTA LA RIGA** — Assegno alla variabile  `NUM`  il risultato di  `3+2` :

```
(( NUM=3+2 ))
```




### Warning

Non usare  `$`  prima dell'espressione!

**PARTE DI RIGA** — faccio riferimento, per esempio, ad una variabile  `PIPP05`

```
echo PIPP0$( (3+2) )
```

Le espressioni aritmetiche possono contenere:

-  operatori aritmetici (+, -, \*, /, %)
-  assegnamenti
-  parentesi tonde per modificare la precedenza dei calcoli

## Info

Le espressioni aritmetiche possono essere usate anche come condizione di `while` e `if`.

# Nozioni per uso terminale: *Espressioni condizionali*

## Valori di verità

- `true`: "exit status di un `command`" = 0
- `false`: "exit status di un `command`"  $\neq$  0

## Operatori logici

NOT: `!`





AND: `&&`, `-a`

OR: `||`, `-o`




## Versioni

### Warning

In nessuna versione è supportato:

-  **WORD SPLITTING**
-  **BRACE EXPANSION**
-  **PATHNAME EXPANSION**
-  **INSERIMENTO COMANDI** (ad eccezione della *command substitution* con ``` per generare **operandi**, non operatori!)

1. Con doppie parentesi quadre `[[ condiz ]]` — *enhanced brackets*

-  SI a tutte le versioni di operatori logici
-  SI a composizione di espressioni mediante parentesi
-  SI ad andata a capo con carattere **backslash** (`\`)

2. Con singole parentesi quadre `[ condiz ]` — *single brackets* o mediante istruzione `test`:  
`test condiz`

- **✗** NO versioni degli operatori logici C-like — `&&` e `||`
- **✗** NO composizione di espressioni mediante parentesi
- **✗** NO ad andata a capo con carattere **backslash** (`\`)

## Operatori utili

### Su FILES

OPZIONE	SPIEGAZIONE
<code>-d file</code>	True if <code>file</code> <i>exists and is a directory</i> .
<code>-e file</code>	True if <code>file</code> <i>exists</i> .
<code>-f file</code>	True if <code>file</code> <i>exists and is a regular file</i> .
<code>-h file</code>	True if <code>file</code> <i>exists and is a symbolic link</i> .
<code>-r file</code>	True if <code>file</code> <i>exists and is readable</i> .
<code>-s file</code>	True if <code>file</code> <i>exists and has a size greater than zero</i> .
<code>-t fd</code>	True if <i>file descriptor</i> <code>fd</code> <i>is open and refers to a terminal</i> .
<code>-w file</code>	True if <code>file</code> <i>exists and is writable</i> .
<code>-x file</code>	True if <code>file</code> <i>exists and is executable</i> .
<code>-0 file</code>	True if <code>file</code> <i>exists and is owned by the effective user id</i> .
<code>-G file</code>	True if <code>file</code> <i>exists and is owned by the effective group id</i> .
<code>file1 -nt file2</code>	True if <i>file1</i> <i>is newer (according to modification date) than file2</i> , or if <i>file1</i> <i>exists and file2</i> <i>does not</i> .
<code>file1 -ot file2</code>	True if <i>file1</i> <i>is older (according to modification date) than file2</i> , or if <i>file2</i> <i>exists and file1</i> <i>does not</i> .

### Su STRINGHE

OPZIONE	SPIEGAZIONE
<code>-z string</code>	True if <i>the length of string</i> <i>is zero</i> .
<code>-n string</code>	True if <i>the length of string</i> <i>is non-zero</i> .
<code>string1 == string2</code> , <code>string1 = string2</code> , <code>string1 -eq string2</code>	True if <i>the strings are equal</i> .
<code>string1 != string2</code> , <code>string1 -ne string2</code>	True if <i>the strings are not equal</i> .
<code>string1 &lt; string2</code> , <code>string1 -lt string2</code>	True if <i>string1</i> <i>sorts before string2</i> <i>lexicographically</i> .

OPZIONE	SPIEGAZIONE
<code>string1 &gt; string2</code> , <code>string1 -gt string2</code>	True if <i>string1</i> sorts after <i>string2</i> lexicographically.

### Info

Esistono anche le varianti `-le` (\*less or equal to\*) e `-ge` (\*greater or equal to\*).

## Nozioni per uso del terminale: *Liste di comandi*

In una command line posso scrivere:

1. Un semplice comando

```
cd ./mydir
./myscript.sh argument1
```

2. Un'espressione valutata aritmeticamente

```
(( VAR=(5+4)*({VAR}-1)))
```

3. Una sequenza di comandi connessi da | (*pipe*)

```
cat $FILE | grep thisword
```

4. Una sequenza di comandi condizionali, in particolare

**4.1.** Eseguo un comando `B` **se e solo se** il comando `A` è andato a buon fine (exit status == 0);  
[per esempio: lancio l'eseguibile `myscript.exe` **SE E SOLO SE** la sua compilazione è andata a buon fine]

```
gcc myscript.c && ./myscript.exe
```

**4.2** Eseguo un comando `B` **se e solo se** il comando `A` è terminato con un errore (exit status != 0);

[per esempio: creo la cartella `mysecretfiles` SE E SOLO SE provando ad entrarci non ha avuto successo]

```
cd ./mysecretfiles || mkdir mysecretfiles
```

5. Un raggruppamento di comandi

```
( cat file1.txt ; cat file2.txt )
```

6. Un'espressione condizionale

```
[[ -e file.txt && $1 -gt 13 ]]
```

#### Info

L'**exit status di una lista di comandi** corrisponde all'exit status dell'**ultimo comando eseguito!**

---

## Nozioni per uso del terminale: *Compound Commands*

### Cicli — costruito iterazione

1. Equivalente di un `foreach`

```
for FILE in `ls ./Documents` ; do echo $FILE ; done
```

2. Equivalente di un `for`

```
for ((IDX=1;IDX<= $#;IDX=IDX+1)) ; do echo ${!IDX} ; done
```

[nell'esempio: stampo a video tutti gli argomenti ricevuti dallo script in esecuzione]

3. Costrutto `while`

```
while read ROW ; do  
    echo "Ho letto $ROW"
```

```
done
```

**N.B.:** Posso comporre anche in diverso modo le condizioni di un ciclo, ad esempio:

```
IDX=1
while read ROW ; [[ $? == 0 && $IDX -leq 10 ]] ; do
    echo "Idx:${IDX}\t${ROW}"
    ((IDX=IDX+1))
done
```

[nell'esempio: leggo al massimo 10 righe da `stdin`]

## If — costruito selezione

```
FILE=./appunti.txt
if [[ -e $FILE ]] ; then
    echo "${FILE} esiste!"
fi
```

---

## Nozioni per uso del terminale: *Command substitution*

Utilizzo dei **backticks (o backquotes)** (```): cattura l'output di un programma - utilizzato per memorizzare i risultati di uno script in una variabile.

```
OUT=`.\printnumber.sh`
echo "Catturato l'output $OUT"
```

Utilizzo di **double quotes** (`"`):

- ❌ NO ; come separatore tra comandi
- ❌ NO sostituzione wildcards
- ✅ SI visualizzazione contenuto variabili
- ✅ SI esecuzione comandi (se racchiusi da ```)

```
echo "Dear $USER today is `date`"
```

```
stdout: Dear nickolausen today is Fri Dec 13 11:20:55 CET 2024
```

Utilizzo di **single quotes** (`'`):

- ❌ NO ; come separatore tra comandi
- ❌ NO sostituzione wildcards
- ❌ NO visualizzazione contenuto variabili
- ❌ NO esecuzione comandi (anche se racchiusi da ``)

```
echo 'Dear $USER today is `date`'
stdout: Dear $USER today is `date`
```

## Nozioni per uso del terminale: *Ridirezionamenti di Stream I/O*

RIDIREZIONAMENTO	SIGNIFICATO
<	riceve input da file.
>	manda <code>stdout</code> verso un file, <b>eliminando il vecchio contenuto del file.</b>
>>	manda <code>stdout</code> verso un file, <b>aggiungendo in coda al vecchio contenuto del file.</b>
	manda <code>stdout</code> del primo comando come <code>stdin</code> del secondo.

### Ricorda

Per terminare input da tastiera: **Ctrl + D**

- Ridirezionamento di **input** e **output** possono essere fatti **contemporaneamente**:

```
./myscript.sh < ./input.txt > ./output.txt

# oppure, analogamente:
./myscript.sh > ./output.txt < ./input.txt
```

- Ridirezionamento di `stdout` e `stderr` possono essere fatti **contemporaneamente**:

```
./myscript.sh &> ./out.txt
```



- Ridirezionamento di `stdout` e `stderr` possono essere fatti in un colpo solo su due file diversi:

```
./myscript.sh 2> ./error.txt > ./output.txt
```

- È possibile ridirigere uno stream `N` sullo stream `M` tramite il comando:

```
N>&M
```

- È possibile ridirigere contemporaneamente `stderr` e `stdout` di un programma `program1` sullo `stdin` di un programma `program2` tramite pipe:

```
./program1.sh |& ./program2.sh
```

---

## Nozioni per uso del terminale: *Raggruppamento di comandi*

Una sequenza di comandi racchiusa da una coppia di parentesi tonde viene eseguita in una sub-shell. L'exit status di quella sequenza corrisponde all'exit status dell'ultimo comando eseguito.

 **Questo accade se la sequenza è composta da più di 1 comando!**

In caso contrario, non viene creata nessuna subshell.

Esempio di sequenza di comandi:

```
( pwd; ls -al; whoami ) > ./out.txt
```

 **Warning**

Durante l'esecuzione, `stdin`, `stdout` e `stderr` dei singoli comandi vengono concatenati.

---

# Nozioni per uso del terminale: *Processi*

## Processi in *foreground*

- Processi che controllano il terminale da cui sono stati lanciati: ne condividono `stdin`, `stdout` e `stderr`; non permettono l'esecuzione di altri programmi.
- In ogni istante di tempo al più **1 processo** può essere in *foreground*.

## Processi in *background*

- Processi eseguiti in parallelo rispetto all'esecuzione della bash; detengono una copia dei *file descriptor* associati alla shell che li ha lanciati, quindi condividono `stdin / stderr / stdout`; questo comporta che la chiusura del terminale causi a sua volta la terminazione di tutti i processi — affinché questi possano continuare a prescindere dalla vita del terminale, occorre "sganciarli" da esso.

## Jobs

- Processi in background o sospesi solo figli di quella shell.

## Job control

- Spostamento di un processo `background` ↔ `foreground`

## Processi Zombie & Orfani

Si parla di queste 2 categorie di processi quando viene chiamato il comando `wait` (*vedi sotto, in Command Cheatsheet*).

```
wait $PROC_PID
```

Ricordiamo che ogni processo è composto da un PID (identificativo del processo) + un PCB (*Process Control Block*, insieme di attributi del processo)

1. Se la shell padre termina senza aver effettuato una chiamata `wait $PROC_PID`, allora il processo figlio viene definito **processo orfano**. Tutti i processi orfani vengono adottati dal processo **init**, colui che detiene il PID = 1, che provvederà a chiamare `wait` per i nuovi arrivati — così da far rilasciare il PCB.
2. Se la shell figlia termina prima dell'esecuzione della shell padre, allora il figlio viene detto **processo zombie**. Il processo è terminato, ma il suo PCB è ancora presente nelle tabelle del sistema operativo. Questo viene eliminato solo quando viene effettuata una chiamata dal padre a `wait $PROC_PID`.

---

# Nozioni per uso del terminale: *Precedenza degli operatori*

## Terminatori di una sequenza di comandi

```
; & newline # andata a capo
```

## "Concatenatori" di una sequenza di comandi

```
&& || ; & |
```

## Ordine

### Warning

Ordine per **precedenza decrescente**

1. { ... ; }, ( ... ) — con le parentesi graffe è **obbligatorio** il ; finale!
2. |
3. &&, ||
4. &, ;

---

## Cheatsheet comandi

### Generics

ESEMPIO	SIGNIFICATO
<code>source</code> <code>./myscript.sh</code>	Esegue <code>myscript.sh</code> <b>senza creare una subshell dedicata</b> . Le variabili create dentro <code>myscript.sh</code> sono visibili anche dalla <b>shell chiamante</b> . Equivale a <code>./myscript.sh</code> .
<code>unset MYVAR</code>	Elimino <code>MYVAR</code> , che sia vuota oppure no.
<code>history</code>	Mostra sul terminale lo storico dei comandi eseguiti, associando a ciascuno un numero <b>immutabile</b> che lo <b>identifica</b> .
<code>!181</code>	Lancio il comando identificato dal numero <b>181</b> ( <i>guarda history</i> ).

ESEMPIO	SIGNIFICATO
<code>set</code>	Lanciato senza parametri, mostra tutte le variabili (locali e d'ambiente) della shell corrente e tutte le funzioni implementate.
<code>set -a</code>	Specifica che tutte le variabili create da quel momento in poi verranno rese d'ambiente, ereditate quindi dalle shell figlie. Comportamento contrario: <code>set +a</code> . <b>N.B.:</b> Se voglio dichiarare una variabile locale dopo aver eseguito <code>set -a</code> , allora utilizzo il comando <code>export -n MYVAR</code> .
<code>set +o history</code>	Disabilita la memorizzazione nella <b>history</b> di ulteriori comandi eseguiti. Comportamento contrario: <code>set -o history</code> .
<code>pwd</code>	" <i>Print Working Directory</i> ": visualizza il percorso corrente.
<code>cd mydir</code>	" <i>Change directory</i> ", abbreviativo di <code>chdir</code> , naviga nella cartella <code>mydir</code> .
<code>mkdir nuovadir</code>	" <i>Make directory</i> ", crea una nuova cartella <code>nuovadir</code> .
<code>rmdir todelete</code>	" <i>Remove directory</i> ", elimina <code>todelete</code> <b>SOLO SE QUESTA È VUOTA</b> . Equivalentemente (e senza condizioni) si può optare per <code>rm -rf todelete</code> (" <i>remove -recursive -forced todelete</i> ")
<code>echo MSG</code>	Stampa in stdout <code>MSG</code> . Se specificato <code>-n</code> , non mette il carattere <code>\n</code> a fine messaggio (utile per la stampa nei file).
<code>cat myfile.txt</code>	Stampa in stdout il contenuto del file <code>myfile.txt</code> .
<code>env</code>	Visualizza in stdout tutte le variabili d'ambiente.
<code>which python</code>	Visualizza il percorso dell'eseguibile <code>python</code> .
<code>mv src dst</code>	" <i>Move</i> ", sposta il file <code>src</code> in <code>dst</code> . Spesso utilizzato per <b>rinominare files/directories</b> .
<code>ps aux</code>	Stampa informazioni sui processi in esecuzione.
<code>du mydir</code>	" <i>Disk Usage</i> ": stampa l'utilizzo del disco della cartella <code>mydir</code> .
<code>kill -9 PID</code>	Termina il processo con ID <code>PID</code> .
<code>killall nomeProc</code>	Elimina tutti i processi di nome <code>nomeProc</code> .
<code>bg</code>	" <i>Background</i> ": manda il job corrente in background.
<code>fg</code>	" <i>Foreground</i> ": ripristina il job più recente in primo piano.
<code>df</code>	" <i>Disk free</i> ": mostra spazio libero dei filesystem montati.
<code>touch newfile.txt</code>	Crea un file <code>newfile.txt</code> .
<code>more myfile.txt</code>	Mostra poco alla volta il contenuto di <code>myfile.txt</code>
<code>man grep</code>	Mostra la documentazione del comando <code>grep</code> ( <i>preso come esempio</i> ).
<code>true</code>	Restituisce exit status 0 (vero).

ESEMPIO	SIGNIFICATO
false	Restituisce exit status 1 (falso).

## Lettura & gestione file descriptor

### Visualizzare i file descriptor associati ad un processo

Ricordiamo che in `/proc/` esiste una subdirectory per ogni processo in esecuzione; posso riferirmi alla subdirectory del processo corrente con `/proc/$$` (`$$` = **PID del processo corrente**).

✓ I file descriptor associati ad un processo sono collocati nella cartella

`/proc/$$/fd/`

#### read

```
read RIGA
```

Di default, legge il contenuto in `stdin` e lo memorizza in `RIGA`; se vengono specificate più variabili, il contenuto letto viene spezzato in più parole - ciascuna "incastrata" nella variabile corrispondente (*prima parola* → *prima variabile*, *seconda parola* → *seconda variabile* ecc...). Nel caso in cui vengono dichiarate meno variabili rispetto alle parole da leggere, **l'ultima variabile contiene il testo mancante**

OPZIONI UTILI	
<code>-u \$FD</code>	specifica il file descriptor <code>FD</code> da cui leggere.
<code>-N 4</code>	legge esattamente <code>4</code> caratteri da <code>stdin</code> .
<code>-r</code>	il carattere <b>backslash</b> ( <code>\</code> ) non viene interpretato come interruttore di linea.

#### exec

Ridireziona un file descriptor su un altro.

```
bash-3.2$ exec > file
bash-3.2$ date
bash-3.2$ exit
bash-3.2$ cat file
Thu 18 Sep 2014 23:56:25 CEST
```

Nell'esempio, ridireziona qualsiasi messaggio stampato su `stdout` all'interno di `file` fino al comando `exit`. Può essere utilizzato in diverse modalità:

MODO APERTURA	Utente sceglie il numero associabile al file descriptor ( <code>n</code> : numero identificativo del file descriptor)	Sistema sceglie file descriptor libero
Read-only	<code>exec n&lt; ./myfile.txt</code>	<code>exec {MYVAR}&lt; ./myfile.txt</code>
Write-only	<code>exec n&gt; ./myfile.txt</code>	<code>exec {MYVAR}&gt; ./myfile.txt</code>
Append	<code>exec n&gt;&gt; ./myfile.txt</code>	<code>exec {MYVAR}&gt;&gt; ./myfile.txt</code>
Read&Write	<code>exec n&lt;&gt; ./myfile.txt</code>	<code>exec {MYVAR}&lt;&gt; ./myfile.txt</code>

### Ricorda!

Bash assegna ad alcuni numeri dei file descriptor di **default**:

- `0` → `stdin`
- `1` → `stdout`
- `2` → `stderr`

### Warning

Qualunque sia la modalità di apertura di un file descriptor, questo deve essere chiuso con il comando:

```
exec n>&-  
# n = identificativo del file descriptor in questione  
  
# oppure, se aperto tramite variabile  
exec {MYVAR}>&-
```

## Manipolazione di stringhe & informazioni testuali

`head`

Seleziona un certo numero di righe di un testo a partire dal suo **inizio**.

ARGS UTILI	ESEMPIO	DESCRIZIONE
<code>-n \$NUM</code>	<code>cat ./myfile.txt \   head -n 2</code>	Indica il numero <code>\$NUM</code> di righe da selezionare.
<code>\$FILE</code>	<code>head -n 4 \$FILE</code>	Indica il file <code>\$FILE</code> da cui selezionare le prime 4 righe.
<code>-c \$NUM</code>	<code>cat ./myfile.txt \   head -c 10</code>	Stampa i primi <code>\$NUM</code> caratteri del testo da manipolare.

## tail

Seleziona un certo numero di righe di un testo a partire dalla sua **fine**.

ARGS UTILI	ESEMPIO	DESCRIZIONE
<code>-n \$NUM</code>	<code>cat ./myfile.txt \   tail -n 2</code>	Indica il numero <code>\$NUM</code> di righe da selezionare.
<code>\$FILE</code>	<code>tail -n 4 \$FILE</code>	Indica il file <code>\$FILE</code> da cui selezionare le ultime 4 righe.
<code>-c \$NUM</code>	<code>cat ./myfile.txt \   tail -c 10</code>	Stampa gli ultimi <code>\$NUM</code> caratteri del testo da manipolare.
<code>-r</code>	<code>cat ./myfile.txt \   tail -r -n 5</code>	" <i>Reversed</i> ", inverte l'ordine dell'output.

## tee

Inoltra il contenuto di `stdin` su più file **contemporaneamente**.

```
cat ./myfile.txt | tee out1.txt out2.txt
```

*Nel seguente esempio, inoltra il messaggio `Hello, World!` sia a `stdout` che al file `greetings.txt`*

```
echo "Hello, World!" | tee greetings.txt
```

## cut

Seleziona solo una **porzione** del file da manipolare.

ARGS UTILI	ESEMPIO	DESCRIZIONE
<code>-b \</code>   <code>-c</code> <code>from-</code> <code>to,from-</code> <code>to,...</code>	<code>cat ./myfile.txt</code> <code>\   cut -c 2-</code> <code>10,15-20</code>	Indica il range di caratteri (o, equivalentemente, <i>bytes</i> ) da selezionare dal testo ricevuto ( <i>nell'esempio, seleziono i caratteri compresi tra il secondo e il decimo e tra il quindicesimo e il ventesimo</i> )

Esempio estratto dall'output del comando `man cut`:

*Extract users' login names and shells from the system passwd(5) file as "name:shell" pairs:*

```
cut -d : -f 1,7 /etc/passwd
```

## grep

Seleziona solo le righe che rispettano il criterio specificato.

```
cat ./dizionario.txt | grep APPLE
```

*Nell'esempio: seleziono la riga dal file `dizionario.txt` che contiene la parola `APPLE`*

ARGS UTILI	ESEMPIO	DESCRIZIONE
<code>-f</code> <code>\$FILE</code>	<code>grep APPLE -f</code> <code>\$DIZIONARIO</code>	Indica il file da cui leggere le righe. ( <i>Se non funziona, mettere il nome del file subito dopo l'espressione da cercare nel file.</i> )
<code>-v</code>	<code>cat ./dizionario.txt</code> <code>\   grep APPLE -v</code>	Inverte la selezione: seleziona tutte le righe che <b>NON</b> contengono la parola <i>APPLE</i> .
<code>-c</code>	<code>cat ./dizionario.txt</code> <code>\   grep APPLE -c</code>	Mostra il <b>numero di righe</b> che rispettano il criterio specificato.
<code>-i</code>	<code>cat ./dizionario.txt</code> <code>\   grep ApPlE -i</code>	Effettua una ricerca <b>case-insensitive</b> .
<code>-m</code> <code>\$NUM</code>	<code>cat ./voti.txt \   grep</code> <code>18 -m 5</code>	Mostra solo le prime <code>\$NUM</code> righe che soddisfano il criterio specificato.
<code>-n</code>	<code>cat ./voti.txt \   grep</code> <code>18 -n</code>	Precede ogni riga selezionata con il corrispondente numero di riga nel file.



## sed — Stream Editor

Modifica un testo sulla base di una `regular expression` specificata.

Sintassi tipo:

```
sed 's/DA_TOGLIERE/DA_METTERE/[g | numero]'
```

Dove:

- `s` indica l'operazione di **sostituzione** di porzioni di testo;
- `DA_TOGLIERE` corrisponde alla **porzione di testo da rimuovere**;
- `DA_METTERE` corrisponde alla porzione di testo da inserire al posto di `DA_TOGLIERE` ;
- `g` — *opzionale*: indica che la sostituzione va effettuata su tutto il testo su cui `sed` sta lavorando ("*global*") (altrimenti fatta solo sulla prima occorrenza di `DA_TOGLIERE` in ciascuna riga);
  - `numero` — *opzionale*: indica per quante occorrenze di `DA_TOGLIERE` deve essere fatta la sostituzione.

Alcuni utilizzi di `sed` :

1. Sostituisce **la prima occorrenza** di `togli` con `metti` in ciascuna riga del file `nomefile` .

```
sed 's/togli/metti/' nomefile
```

2. Rimuovere il carattere in **prima posizione di ogni linea** che si riceve da `stdin` .

```
sed 's/^./'''
```

- `^` : *inizio linea*
- `.` : *carattere qualunque*

3. Rimuovere **l'ultimo carattere** di ogni linea ricevuta dallo `stdin` .

```
sed 's/.$//'
```

- `.` : *carattere qualunque*
- `$` : *fine linea*

4. Eseguo **due rimozioni insieme** (`;`) — rimuovere il primo e l'ultimo carattere in ogni linea.

```
sed 's/^./.;s/.$/.'
```

5. Rimuove i primi 3 caratteri ad inizio linea

```
sed 's/.../'
```

6. Rimuove i primi 4 caratteri ad inizio linea.

```
sed -r 's/.{4}/'
```

7. Rimuove gli ultimi 3 caratteri di ogni linea.

```
sed -r 's/.{3}$/'
```

8. Rimuove tutto eccetto i primi 3 caratteri di ogni linea.

```
sed -r 's/(.{3}).*/\1/'
```

9. Rimuove tutto eccetto gli ultimi 3 caratteri di ogni linea.

```
sed -r 's/.*(.{3})/\1/'
```

10. Rimuove tutti i caratteri alfanumerici in una linea.

```
sed 's/[a-zA-Z0-9]/g'
```

ARGS UTILI	DESCRIZIONE
<code>-r \</code>   <code>-E</code>	Interpreta la <code>regular expression</code> come regular expression moderna (o estesa).

## Gestione processi

ESEMPIO	SIGNIFICATO
<code>./myscript</code> &	Esegue <code>myscript.sh</code> <b>in background</b> . All'interno della variabile <code>\$!</code> è memorizzato il suo PID. Il <b>carattere &amp;</b> rappresenta un separatore tra

ESEMPIO	SIGNIFICATO
	<b>comandi — è necessario quindi omettere il ";"</b>
<code>^Z (Ctrl-Z)</code>	<b>Sospende</b> un processo in <i>foreground</i> .
<code>^C (Ctrl-C)</code>	<b>Termina</b> un processo in <i>foreground</i> .
<code>bg</code>	Riprende l'esecuzione in background di un processo sospeso.
<code>fg %n</code>	Porta in foreground un processo sospeso — <code>%n</code> : <i>indice del job di riferimento. (vedi jobs)</i>
<code>kill 6152</code>	Elimina il processo con il PID <code>6152</code> .
<code>kill %2</code>	Elimina il processo con l' <b>indice del job</b> = 2 ( <i>vedi jobs</i> ).
<code>jobs</code>	Produce una lista numerata dei processi in background o sospesi nella shell corrente — il numero tra parentesi quadre che indica ciascun processo <b>non è il PID</b> ma un <b>indice del job</b> che si usa per gestire il job, premettendo il carattere <code>%</code> . <b>N.B.: Questo comando mostra solo i processi in esecuzione   sospesi a partire dalla shell corrente.</b>
<code>ps -ax</code>	" <i>Process Status</i> ": restituisce una <b>visione statica</b> dello stato di tutti i processi di tutti gli utenti ( <code>-a</code> ) che non necessariamente hanno controllato il terminale ( <code>-x</code> ).
<code>top</code>	Restituisce una <b>visione dinamica</b> dello stato di tutti i processi in esecuzione, aggiornata periodicamente ad intervalli di tempo regolari.
<code>wait \$JOBS \   \$PIDs</code>	Attende la terminazione del comando con Job ID / PID corrispondente, restituendone il risultato. Si possono specificare un elenco di <code>Job</code> o di <code>PID</code> di processi che si vogliono aspettare. Il comando <code>wait</code> termina la sua esecuzione solo quando tutti i processi specificati terminano la loro. <b>Può essere chiamato solo dalla shell che ha lanciato l'esecuzione dei comandi specificati</b> , altrimenti termina subito restituendo come risultato <code>127</code> . Se non riceve parametri, <b>attende che tutti i processi figli del processo da cui viene chiamato terminino la loro esecuzione.</b>

### Ricorda!

La variabile `$!` memorizza sempre il PID dell'ultimo processo lanciato in background. Fino allo spostamento di un nuovo processo in background, il suo valore rimane **immutato**.

## disown

ARGS UTILI	ESEMPIO	DESCRIZIONE
<code>-r</code>	<code>disown -r</code>	Sgancia dalla shell <b>tutti i job running</b> .

<code>-a</code>	<code>disown -a</code>	<b>DESCRIZIONE</b> Sgancia dalla shell tutti i job running e sospesi.
<code>-</code>	<code>disown</code>	Sgancia dalla shell l' <b>ultimo job messo in background</b> .
<code>%jobid</code>	<code>disown %jobid</code>	Sgancia dalla shell <b>il job specificato</b> .

## nohup

Esempio:

```
nohup ./myscript arg1 arg2 &
```

Lancia uno script in background e lo sgancia dalla shell di partenza. Equivalente di:

```
./myscript.sh arg1 arg2 &
disown
```