APPUNTI SCRIPT BASH

INTRODUZIONE

La shell Bash è un linguaggio di programmazione che si utilizza all'interno di un file di testo, interpretato dal motore Bash, con una certa sua sintassi e che ci permette di ordinare al nostro computer di fare le operazioni di cui abbiamo bisogno.

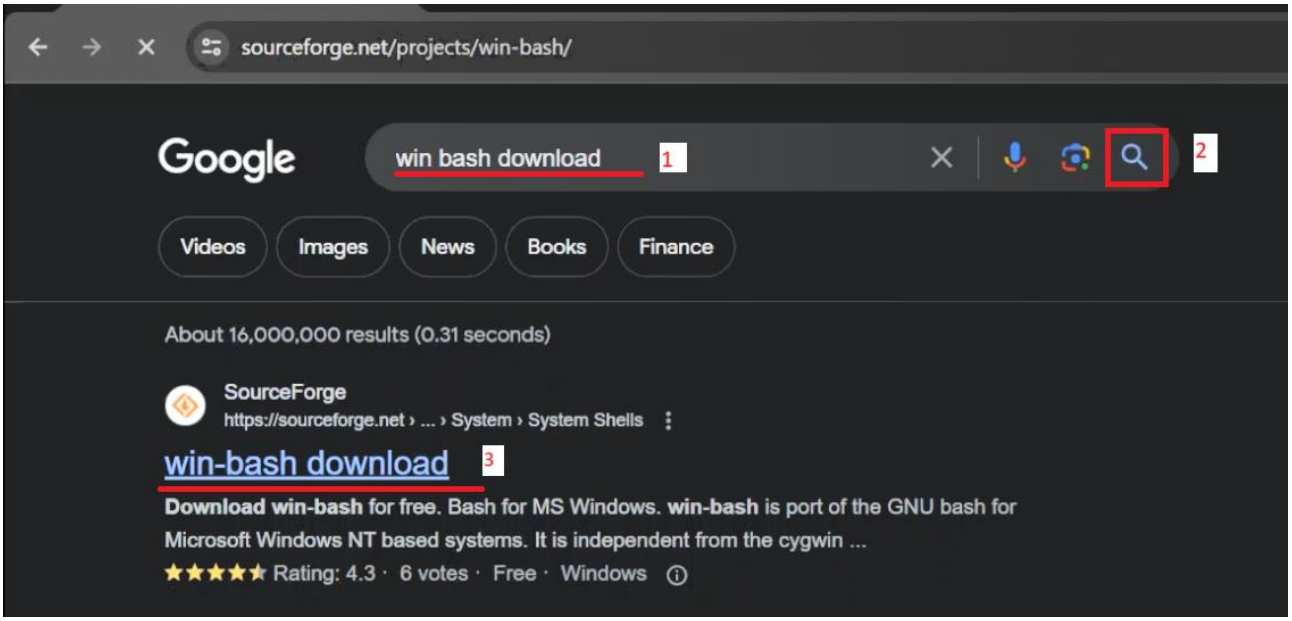
È un linguaggio di programmazione che è paragonabile, per le potenzialità che ha, a qualsiasi linguaggio di programmazione, con il vantaggio che è già tutto preinstallato e pronto all'uso.

Con la shell Bash abbiamo a disposizione tutta la potenza del terminale Linux, dove per potenza si intende il fatto di poter inserire tutti i comandi, che normalmente utilizzeremmo singolarmente da terminale, in uno script Bash per eseguirli automaticamente.

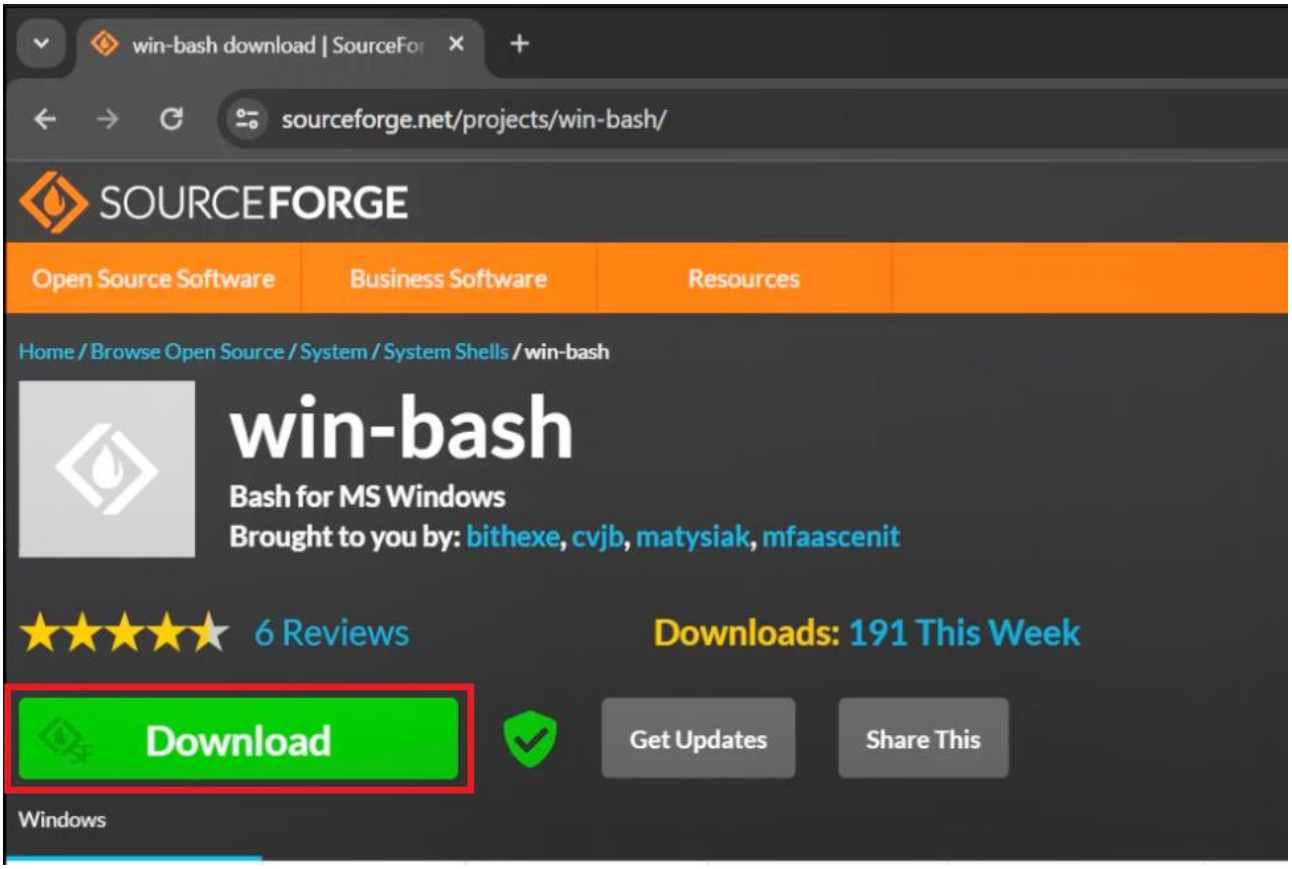
Il formato sh è l'estensione del file che indica al nostro sistema che si tratta di file con codice shell Bash.

INSTALLARE E AVVIARE WIN BASH

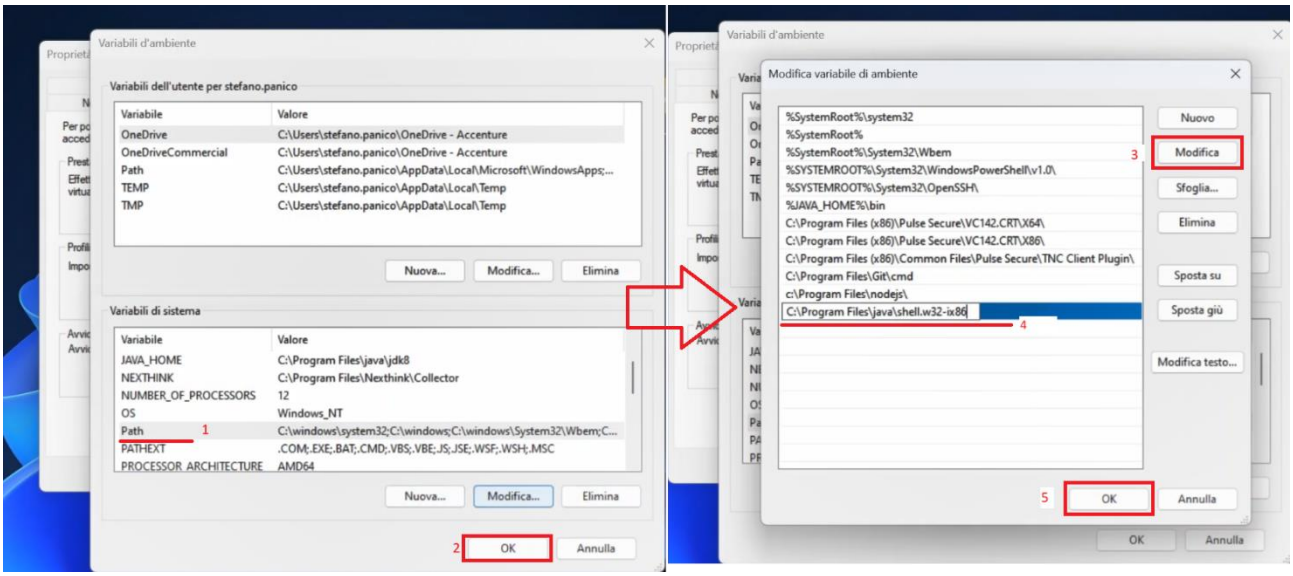
- **Step1**: Cercare su Google "**win bash download**" e cliccare sul primo link.



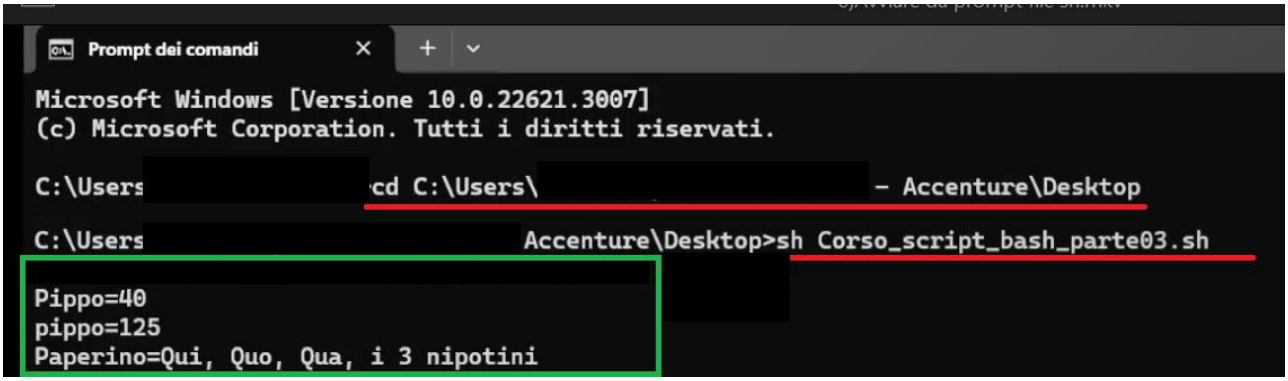
- **Step2**: Cliccare su "Download" per scaricare lo zip del win-bash.



- **Step3:** Una volta scaricato lo zip, eseguire l'estrazione e inserire il tutto in una cartella a piacere. Copiare il percorso delle cartelle in cui abbiamo inserito il win-bash e copiarlo all'interno della variabile d'ambiente Path.



- **Step4**: Avviare il prompt dei comandi, andare tramite il comando **cd** nella cartella dove è presente lo script sh da avviare e infine usare il comando **sh nome_file_sh** per avviare lo script.



COMMENTI

Il commento è importante perché, se ci creiamo il nostro script Besh in un determinato periodo e poi lo abbandoniamo per un altro periodo di tempo, non è per niente scontato che ciò che era chiaro quando lo abbiamo creato lo sia ancora successivamente. Quindi ha un valore aggiunto commentare tutto quello che ci viene in mente nel momento in cui stiamo creando lo script.

Il commento è importante anche per gli altri perché, se abbiamo intenzione di distribuire il nostro script che può servire anche ad altri, diventa importante che tutti possano leggere il motivo della scrittura del codice in un determinato modo.

Il carattere **#** si usa per commentare una singola riga.

La prima riga di uno script è sempre **#!/bin/bash** e sta ad indicare quale motore di shell usare per interpretare il codice che poi segue. Questo perché **bash** è solo una delle shell che esistono, che sono svariate.

Per commentare un blocco di più righe, si deve usare la forma seguente:

:<<commento

Questa modalità diventa molto utile se vogliamo concentrarci su una parte dello script, evitando di doverlo sempre eseguire per intero.

Corso_script_bash_parte02.sh

```
#!/bin/bash

################################################################################
# Questa shell ha solo scopo didattico ed è stata costruita durante il corso
# di programmazione sugli script bash pubblicato sul mio canale Youtube
# (http://www.youtube.com/user/VosaxAlo).
#
# autore : Nicola Jelmorini
# anno   : 2012-2013
# licenza: questo script è software libero: puoi ridistribuirlo e/o modificarlo
#          rispettando i termini della licenza GNU General Public License
#          stabiliti dalla Free Software Foundation, versione 3.
#          Questo script è distribuito nella speranza che possa essere utile,
#          ma SENZA ALCUNA GARANZIA.
#          Consulta la licenza GNU General Public License per avere ulteriori
#          dettagli.
#          Dovresti aver ricevuto una copia della GNU General Public License
#          insieme a questo programma.
#          Se non è così, visita <http://www.gnu.org/licenses/>.
#
#
# Uso dello script: ???? si apre un menu dal quale scegliere ????
#
################################################################################

:<<test_if
# comandi della shell
echo -n "Immetti parametro PIPPA: "
read PIPPA
```

```
echo Hai impostato PIPPA=$PIPPA

if [ "$PIPPA" -eq 5 ] || [ "$PIPPA" = "Pippa" ]
then
        echo Hai fornito il valore 5 oppure Pippa
else
        echo Hai fornito un valore diverso da 5 e da Pippa
fi

echo -n "Digita il valore ab= "
read ab
echo Hai impostato ab=$ab

if [ "$ab" -gt 0 ] && [ "$ab" -lt 5 ]
then
        echo valore fra zero e 4
else
        echo valore minore di zero oppure maggiore di 4
fi
test_if

:<<loop_for
echo;echo;echo "Elenco dei pianeti del nostro sistema solare:"
for pianeta in Mercurio Venere "Pianeta Terra" Marte Giove Saturno Uranio Nettuno
do
        echo $pianeta # Ogni pianeta visualizato su una riga separata (gli spazi sepa-
rano una voce dall'altra)
done
loop_for
```

VARIABILI

Le variabili danno vita al codice proprio perché variano nel tempo, e quindi interrogabili e mutabili nel corso dell'esecuzione del nostro script bash.

Una variabile è una cella di memoria del nostro computer che ci riserviamo per depositarvi un valore.

Negli script bash le variabili non hanno una definizione specifica, cioè non bisogna dire specificatamente se una variabile è numerica, o alfanumerica (in gergo chiamata stringa di caratteri).

Una cosa importante da sapere è che tutto il mondo Linux è case sensitive, ovvero c'è distinzione tra maiuscole e minuscole. Quindi la variabile **Pippo** è diversa dalla variabile **pippo**.

Per visualizzare il contenuto di una variabile si deve usare il comando **echo $variabile**.

Ci sono anche delle variabili predefinite che il sistema valorizza automaticamente e che sono a disposizione per darci dei valori che possono essere molto comodi per il nostro script. Come qualsiasi variabile creata da noi, anche queste possono essere interrogate con il comando echo. Di seguito alcuni esempi:

- echo $PWD → directory corrente
- echo $HOME → percorso della propria home
- echo $USER → nome del proprio user
- echo $RANDOM → ogni volta che viene letta, ritorna un valore casuale
- echo $LANG → lingua attiva sul sistema
- echo $HOSTTYPE → architettura del sistema (es. x86_64)

Corso_script_bash_parte03.sh

```
#!/bin/bash

clear

# creare una variabile numerica
Pippo=40
pippo=125

# creare una variabile alfanumerica o testuale (stringa)
Paperino="Qui, Quo, Qua, i 3 nipotini"

# visualizzare il contenuto delle variabili
echo "Pippo="$Pippo
echo "pippo="$pippo
echo "Paperino="$Paperino

# cambiamo valore alla variabile pippo
pippo=37
echo "nuova pippo="$pippo
```

ARRAY O TUPLE

Le tuple sono delle variabili multiple che lavorano con un indice per scrivere o leggere una specifica posizione nella memoria del nostro PC, che per convenzione chiameremo cella.

Le chiamiamo cella perché queste tuple hanno appunto un indice, quindi una variabile può contenere, con quello stesso nome e l'indice, più valori.

Per esempio, questa è la definizione della tupla dei giorni della settimana (sette celle, una per ogni giorno della settimana):

Settinana[1]=Lunedì
Settinana[2]=Martedì
Settinana[3]=Mercoledì
Settinana[4]=Giovedì
Settinana[5]=Venerdì
Settinana[6]=Sabato
Settinana[7]=Domenica

Ci sono altri modi di definire una tupla:

- Mesi=([1]=gennaio [2]=febbraio …)
- Valori=(zero uno due tre quattro) → se non si specifica l'indice, il medesimo parte da zero e non da uno

La tupla si può interrogare con il suo indice:

- echo ${Settimana[1]} → Lunedì
- echo ${Settimana[*]} → Lunedì Martedì Mercoledì Giovedì Venerdì Sabato Domenica → (al posto dell'asterisco si può mettere il "@" che ha lo stesso effetto)
- echo ${#Settimana[*]} → 7 (numero di elementi contenuti nella tupla)

Corso_script_bash_parte04.sh

```
#!/bin/bash

clear

# le tuple - i differenti modi di definirle
Settimana[1]=Lunedì
Settimana[2]=Martedì
Settimana[3]=Mercoledì
Settimana[4]=Giovedì
Settimana[5]=Venerdì
Settimana[6]=Sabato
Settimana[7]="Domenica - il meritato riposo"

Mesi=([1]=gennaio [2]=febbraio [3]=marzo [4]=aprile [5]=maggio [6]=giugno [7]=luglio
[8]=agosto [9]=settembre [10]=ottobre [11]=novembre [12]=dicembre)
Valori=( zero uno due tre quattro )

# la tupla si può interrogare con il suo indice:
echo ${Settimana[1]}
echo ${Settimana[5]}
echo "numero valori nella tupla Settimana="${#Settimana[*]}
echo ${Mesi[*]}
echo ${Valori[0]}
echo ${Valori[1]}
```

REINDIRIZZAMENTO E SCRITTURA DELL'OUTPUT

L'uso di comandi speciali per inviare l'output di un comando ad un file, o per impostare l'output di un comando come input di un altro è una modalità molto pratica e potente del bash.

Il primo di questi comandi speciali è il simbolo >, che permette di scrivere l'output di un comando in un nuovo file di testo. Per esempio:

- **ls > out.txt** scrive l'elenco dei file presenti nella cartella corrente nel file out.txt. Se out.txt esiste già, viene sovrascritto.

Il comando >> permette di continuare a scrivere l'output di un comando, accumulando però l'output in un file di testo già esistente. Quindi, il primo comando > scrive da 0 un file, con la differenza che se lo trova già, lo sovrascrive. Se vogliamo però andare avanti a riempire questo file senza perdere quanto abbiamo già scritto, allora dobbiamo mettere >>.

Il comando pipe **|** permette di prendere l'output di un comando e di farlo proseguire, trasformandolo come input per un ulteriore successivo comando. C'è una forma sintattica equivalente al pipe, che è il comando **<(comando)**, dove prima del < ci sarà il comando che si aspetta come input, l'output del comando tra parentesi tonde. Per esempio:

- **ls -l | cat** e **cat <(ls -l)** sono equivalenti e danno lo stesso risultato, ovvero l'elenco dei file contenuti nella cartella corrente stampato a video. In particolare, è il comando **cat** che permette la visualizzazione a video dell'output di un determinato comando, mentre **ls -l** ci permette di ottenere anch'esso l'elenco dei file presenti nella cartella corrente, ma con più dettagli.

Non siamo limitati ad un solo reindirizzamento, ma in realtà possiamo cumulare i reindirizzamenti degli output verso gli input di altri comandi. Per esempio:

- **cat <(ls -l) <(ls -al)** restituisce l'output di due comandi che contemporaneamente viene indirizzato verso il comando cat.

- **cat out.txt | sort -r | uniq,** dove il comando cat crea come output le righe contenute nel file out.txt, poi questo output passa come input nel comando sort -r che ordina le righe in ordine inverso, poi ancora l'output viene passato al comando uniq per eliminare eventuali righe doppie e, infine, il risultato viene visualizzato a video.

Il carattere speciale apice rovesciato ` (da tastiera non si può scrivere) permette di usare l'output di un comando come parametro (o argomento) di un altro comando. Per esempio:

- **echo "Il nome dell'utente connesso è `whoami`"** visualizza la stringa "il nome dell'utente connesso è nicola". Il comando whoami inserito fra apici rovesciati è stato eseguito e incluso come argomento del comando echo.

- **echo -e "Ecco il contenuto del file out.txt:\r\n`cat out.txt`"** visualizza il contenuto del file out.txt. L'opzione -e del comando echo permette di interpretare le sequenze speciali, come la riga a capo "\r\n".

La forma **$(comando)** è equivalente alla precedente, ovvero permette di usare l'output di un comando come argomento di un altro comando, ma ha il vantaggio di poter facilmente combinare più output di comandi innestati.

Corso_script_bash_parte05.sh

```
#!/bin/bash

clear

# il comando ">" scrive l'output di un comando in un nuovo file di testo
ls > out.txt
echo "ho scritto l'output del comando nel file out.txt"
read

# il comando ">>" permette di continuare a scrivere l'output di
# un comando accumulandolo in un file di testo già esistente
ls >> out.txt
echo "ho aggiunto l'output del comando nel file out.txt"
read

# il comando pipe "|" oppure l'equivalente comando "<(comando)"
# permettono di prendere l'output di un comando e di farlo proseguire
# come input di un comando successivo
ls -l | cat
read
echo "************************************************"
cat <(ls -l)
read
echo "************************************************"
cat <(ls -l) <(ls -al)
read
echo "************************************************"
cat out.txt | sort -r | uniq
read

# il carattere speciale apice rovesciato (backquote) "`" permette di
# usare l'output di un comando come argomento (parametro) di un altro
echo;echo "Il nome dell'utente connesso è `whoami`"
read
echo "************************************************"
echo -e "Ecco il contenuto del file out.txt:\r\n`cat out.txt`"
read

# la forma "$(comando)" è equivalente alla precedente, ma permette
# di facilmente combinare più output di comandi innestati
echo "************************************************"
echo "Il nome dell'utente connesso è $(whoami)"
read
echo -e "Contenuto di un file script\r\n$(cat $(ls *03.sh))"
```

GESTIONE DELLE STRINGHE

Nel mondo dello sviluppo software, una stringa è una sequenza di caratteri alfanumerici.

Se **A=132** è una variabile di tipo numerico, **B=”Pippo2”** è una variabile di tipo alfanumerico.

Con comando **echo** è possibile visualizzare a video, nel terminale che stiamo utilizzando, una stringa o il contenuto di una variabile alfanumerica.

Si usa il carattere **$** per espandere le variabili, ossia recuperare il contenuto di una variabile, per esempio:

- **echo $USER** ritorna l’utente corrente.
- **echo Pippa =\> $(ls)** ritorna Pippa = > elenco orizzontale di file o cartelle della cartella corrente, sempre separati di uno spazio.

Le parentesi graffe **{}** si usano per visualizzare il contenuto di una sequenza, una tupla, o applicare una trasformazione ad una variabile. Per esempio:

echo {a..e} ritorna “a b c d e”

echo ${Tabe[3]} ritorna il valore contenuto nella tupla “Tabe” in corrispondenza dell’indice 3

Pippa=$(echo {15..19}) carica la variabile Pippa con “15 16 17 18 19”

echo n{1,2,3}=xx ritorna “n1-xx n2-xx n3-xx”

x=”Ciao Mondo”; echo ${x^^} ritorna la stringa tutta maiuscola “CIAO MONDO”

x=”Ciao Mondo”; echo ${x,,} ritorna la stringa tutta minuscola “ciao mondo”

x=”Ciao Mondo”; echo ${x^} ritorna la stringa con solo la prima lettera maiuscola “Ciao mondo”

x=”ciao mondo”; echo ${x:0:3} ritorna i primi 3 caratteri della stringa “cia”

x=”ciao mondo”; echo ${x:3:5} ritorna 5 caratteri a partire dal quarto “o mon”

x=”oggi è.il 02.03.2013”; echo ${#x} ritorna la lunghezza della stringa → 20

x=”oggi è.il 02.03.2013”; echo ${x#*.} elimina tutta la stringa dall’inizio fino alla prima occorrenza di “.” → il 02.03.2013

x=”oggi è.il 02.03.2013”; echo ${x##*.} elimina tutta la stringa dall’inizio fino all’ultima occorrenza di “.” → 2013

x=”oggi è.il 02.03.2013”; echo ${x%.*} elimina tutta la stringa dalla fine fino alla prima occorrenza di “.” → oggi è.il 02.03

x=”oggi è.il 02.03.2013”; echo ${x%%.*} elimina tutta la stringa dalla fine fino all’ultima occorrenza di “.” → oggi è

Si usa la barra rovesciata (backslash) \ per non far interpretare i caratteri speciali (speciali per il motore della bash naturalmente), che possono essere **$** o >, e visualizzarli come gli altri caretteri. Per esempio:

> **echo \$USER** ritorna $USER

Le virgolette semplici **‘’** permettono di non far interpretare né i comandi, né espansioni con i caratteri speciali. Per esempio:

> **echo file ~/Musica/*.txt Pippa =\> $(whoami)** ritorna file /home/nicola/Musica/in_the_Road_Again(Linux).txt => Nicola
>
> **echo ‘file ~/Musica/*.txt Pippa =\> $(whoami)’** ritorna file ~/Musica/*.txt Pippa =\> $(whoami)

Le virgolette doppie **“ “** hanno l’effetto dimezzato rispetto gli apici semplici, ossia permettono di interpretare solo le espansioni con i caratteri speciali, ma non i comandi. Per esempio:

> **echo “file ~/Musica/*.txt Pippa =\> $(whoami)”** ritorna file ~/Musica/*.txt Pippa =\> nicola

Con il comando **tr** si può modificare la forma e il contenuto di una stringa. Per esempio:

> **echo "Questo è un messaggio stringa" | tr '[a-z]' ['A-Z']** ⇒ QUESTO è UN MESSAGGIO STRINGA
>
> **echo "Questo è un messaggio stringa" | tr '[a-z,è]' ['A-Z',È]** ⇒ QUESTO È UN MESSAGGIO STRINGA
>
> **echo "Questo è un messaggio stringa" | tr 's' '$' ⇒** Que$to è un me$$aggio $tringa
>
> **echo "Questo è un messaggio stringa" | tr -d 's'** ⇒ Queto è un meaggio tringa
>
> **echo "Questo è un messaggio stringa" | tr -s 'g' '*' ⇒** Questo è un messa*io strin*a

Corso_script_bash_parte06

```
#!/bin/bash

clear

# il carattere $
echo $USER
echo Pippa =\> $(ls)
read

# le parentesi graffe {}
echo {a..e}
Tabe[3]="Corso Bash"
echo "Tabe(3)="${Tabe[3]}
Pippa=$(echo {15..19})
echo "Pippa="$Pippa
echo n{1,2,3}-xx
# ottenere una stringa tutta maiuscola
x="Ciao Mondo";echo ${x^^}
# ottenere una stringa tutta minuscola
x="Ciao Mondo";echo ${x,,}
# ottenere solo la prima lettera maiuscola
x="ciao mondo";echo ${x^}
x="ciao mondo";echo ${x:0:3}
x="ciao mondo";echo ${x:3:5}
x="oggi è.il 02.03.2013"; echo ${#x}
x="oggi è.il 02.03.2013"; echo ${x#*.}
x="oggi è.il 02.03.2013"; echo ${x##*.}
x="oggi è.il 02.03.2013"; echo ${x%.*}
x="oggi è.il 02.03.2013"; echo ${x%%.*}

read

# la barra rovesciata (backslash) \
echo \$USER
read

# le virgolette semplici ''
echo "Senza virgolette:" file ~/Musica/*.txt Pippa =\> $(whoami)
echo "Con virgolette:" 'file ~/Musica/*.txt Pippa =\> $(whoami)'
read

# le virgolette doppie
echo "Senza virgolette:" file ~/Musica/*.txt Pippa =\> $(whoami)
echo "Con virgolette:" "file ~/Musica/*.txt Pippa =\> $(whoami)"
read

# modifica della forma e del contenuto di una stringa
echo "Questo è un messaggio stringa" | tr '[a-z]' ['A-Z']
echo "Questo è un messaggio stringa" | tr '[a-z,è]' ['A-Z',È]
echo "Questo è un messaggio stringa" | tr 'è' "e'"
echo "Questo è un messaggio stringa" | tr 's' '$'
echo "Questo è un messaggio stringa" | tr -d 's'
echo "Questo è un messaggio stringa" | tr -s 'g' '*'
```

ARITMETICA SEMPLICE

Negli script bash si possono effettuare delle semplici operazioni aritmetiche. Per esempio, la moltiplicazione si effettua con **echo $((2*3))**

L'aritmetica disponibile negli script bash è solo per numeri interi, quindi se si effettua ad esempio una divisione, non si vedranno i decimali.

Però ci vengono in soccorso dei comandi aggiuntivi se vogliamo ampliare i nostri orizzonti, in particolare il comando **bc**, che può fare dei calcoli anche più complessi. Richiamando il comando **bc** con l'opzione **l** otterremo anche i decimali. Per esempio:

echo "scale=5"; | bc -l ritorna → 0.66666

A=2, B=3 → var1=$(echo "scale=10"; $A/$B | bc -l) → 0.6666666666

Anche con le tuple si possono fare operazioni aritmetiche. Per esempio:

Numeri[5]='expr ${Numeri[1]} + ${Numeri[3]}'

sintassi equivalente a

Numeri[5]=$((${Numeri[1]} + ${Numeri[3]}))

Corso_script_bash_parte07

```
#!/bin/bash

clear

# operazioni base: somma, sottrazione, moltiplicazione, divisione
A=20
B=35
SOMMA=$((A+B))
echo Somma=$SOMMA
SOTTR=$((A-B))
echo Sottrazione=$SOTTR
MOLT=$((A*B))
echo Moltiplicazione=$MOLT
DIVI=$((A/B))
echo "Divisione con aritmetica bash"=$DIVI
DIVI=$(echo "scale=5; $A/$B" | bc -l)
echo "Divisione con comando bc"=$DIVI

# Somma di due valori della tupla e salvataggio del risultato in un altro indice
Numeri=( 25 43 60 2 15 )
# sintassi di calcolo 1
Numeri[5]=`expr ${Numeri[1]} + ${Numeri[3]}`
echo -n "Numeri[5] = "
echo ${Numeri[5]}
# sintassi di calcolo 2
Numeri[6]=$((${Numeri[0]} * ${Numeri[2]}))
echo -n "Numeri[6] = "
echo ${Numeri[6]}
```

DECISIONI

Le decisioni sono un costrutto che permettono di dare "intelligenza" al codice, che riesce a eseguire compiti diversi a seconda della situazione che si può presentare in un determinato momento.

Per decisione di intende dire **se una certa condizione si verifica, faccio qualcosa, altrimenti faccio qualcos'altro**. La sintassi è la seguente:

if […] then … else … fi

Se abbiamo a che fare con valori numerici, la sintassi prevede queste keyword:

-eq → uguale a

-ne → non uguale a

-gt → maggiore di

-ge → maggiore o uguale a

-lt → minore di

-le → minore o uguale a

Se abbiamo a che fare con le stringhe, la sintassi prevede queste keyword:

= → uguale a

!= → non uguale a

< → più corta di Esempio: **["$a" \< "$b"]**

> → più lunga di Esempio: **["$a" \> "$b"]**

-z → stringa vuota o a lunghezza zero Esempio: **[-z "$a"]**

-n → stringa non vuota o a lunghezza diversa da zero Esempio: **[-n "$a"]**

La verifica di una condizione può essere più complessa, perché può richiedere un confronto di più di due valori. Per cui, abbiamo i seguenti operatori logici:

&& → AND, ovvero entrambe le condizioni devono essere vere
Esempio: **[-z "$a"] && [-n "$b"]**

|| → OR, ovvero almeno una condizione deve essere vera
Esempio: **[-z "$a"] || [-n "$b"]**

Il comando if può essere innestato, ovvero possono essere inseriti più comandi if uno dentro l'altro. In questo caso c'è una variante della sintassi che facilita la lettura del codice:

if [...] then ... elif [...] then ... else ... fi

Corso_script_bash_parte08

```
#!/bin/bash

clear
# verifica se il valore nella variabile D è pari o dispari
# confronto fra valori numerici
D=113
N=$((D/2))
M=$((N*2))
echo D=$D
echo N=$N
echo M=$M

if [ "$D" -ne "$M" ]
then
       echo "Il valore della variabile D è DISPARI"
else
    echo "Il valore della variabile D è PARI"
fi

D=1124
N=$((D/2))
M=$((N*2))
echo; echo D=$D
echo N=$N
echo M=$M

if [ "$D" -eq "$M" ]
then
       echo "Il valore della variabile D è PARI"
else
    echo "Il valore della variabile D è DISPARI"
fi

read


# confronto fra stringhe
S1="Paperino"
S2="zio Paperone"
S3="Paperino"

echo "======================================================="
echo "S1="$S1;echo "S2="$S2;echo "S3="$S3;echo

if [ "$S1" = "$S3" ]
then
       echo "Le stringhe S1 e S3 sono uguali"
fi

if [ "$S1" != "$S2" ]
then
       echo "Le stringhe S1 e S2 non sono uguali"
fi

if [ "${S1:0:2}" = "${S2:4:2}" ]
then
       echo "Le stringhe S1 e S2 hanno la parte 'Pa' in comune"
fi

if [ -n "$S1" ]
```

```
then
      echo "La stringa S1 non è vuota"
fi

if [ -z "$S4" ]
then
      echo "La stringa S4 è vuota o inesistente"
fi

if [ "$S1" \< "$S2" ]
then
      echo "La stringa S1 è più corta della stringa S2"
fi

if [ "$S2" \> "$S1" ] && [ -n "$S3" ]
then
      echo "La stringa S2 è più lunga della stringa S1 e la stringa S3 non è vuota"
fi
```

SMISTAMENTO

Il commando CASE, a fronte di una serie di condizioni valutate al suo interno, smista l'esecuzione del codice.

Può essere paragonato ad un insieme di decisioni if, oppure può essere addirittura sostituito con una serie di decisioni if innestati, che però renderebbero il codice più difficile da leggere.

Tipicamente questo comando si usa per interpretare la risposta data alle opzioni di un menù con più scelte. La sintassi base del comando CASE è la seguente:

case … in cond1) …;; cond2) …;; condn) …;; *) (altrimenti);; esac

L'ultima condizione con l'asterisco sta per tutto il resto, ovvero per tutte le condizioni non previste precedentemente.

Appena una condizione è soddisfatta, il blocco CASE termina e non prosegue a verificare le condizioni successive.

Corso_script_bash_parte09

```
#!/bin/bash

clear

# Esempio di comando case
M=15
N=3
case "$M" in
      0) ;;       # non fa niente
      "$N")       echo "M e N sono uguali";;
      2 | 4 | 15) echo "il valore di M è 2 o 4 o 15";;
      [0-9])      echo "il valore di M è una cifra tra 0 e 9";;
      [0-9]*)     echo "il valore di M inizia con un numero";;
      [a-zA-Z])   echo "il valore di M è una lettera dell'alfabeto";;
      [a-zA-Z]*)  echo "il valore di M inizia con una lettera";;
      [!0-9]*)    echo "il valore di M non inizia con un numero";;
      *)          echo  "Errore - parametro non previsto";;
esac
```

CICLI O LOOP

I cicli, o loop, sono un altro elemento fondamentale della programmazione.

Il ciclo permette di ripetere una serie di istruzioni, finché una condizione viene soddisfatta.

I comandi che permettono di implementare questi cicli nello script bash sono i seguenti:

for arg in (lista) do … done

for … do … done

while … do … done

until … do … done

Si può influire sull'andamento di un ciclo con due comandi:

- **break** → permette di uscire immediatamente da un ciclo.
- **continue** → permette di passare al prossimo giro del ciclo evitando di eseguire i comandi che seguono.

Corso_script_bash_parte10

```
#!/bin/bash

clear

# Esempi di comando for con lista argomenti
echo "******************** FOR CON LISTA ARGOMENTI SINGOLI ********************"
for pianeta in "Mercurio" "Venere" "Terra" "Marte" "Giove"
do
	echo "$pianeta"
done
echo
read

echo "******************** FOR CON LISTA FILE IN CARTELLA ********************"
for docu in $(ls *.sh)
do
	echo "File contenuto nella cartella: $docu"
done
echo
read

echo "******************** FOR CON LISTA ARGOMENTI MULTIPLI ********************"
for pianeta in "Mercurio 36 circa" "Venere 67 circa" "Terra 93 circa" "Marte 142 circa"
"Giove 483 circa"
do
	set -- $pianeta
	echo "$1 è distante $2,000,000 di miglia dal sole $3"
done
echo
read

# Esempio di comando for con limite
```

```
echo "******************** FOR CON LIMITE ********************"
LIMITE=10
for ((a=1; a <= LIMITE ; a++))
do
       echo -n "$a "
done
echo
read

# esempio di comando while (fino a che)
echo "******************** WHILE ********************"
var0=0
LIMITE=10
while [ "$var0" -lt "$LIMITE" ]  #equivalente: while (( var0 < LIMITE ))
do
       echo -n "$var0 "
       var0=`expr $var0 + 1` #equivalente: var0=$((var0+1))
done

echo
read

# esempio di comando until (fino al momento in cui)
echo "******************** UNTIL ********************"
LIMITE=10
var=0
until (( var > LIMITE ))
do
       echo -n "$var "
       (( var++ ))
done
echo
```

FUNZIONI

Nel caso in cui lo script diventa particolarmente complesso, deve ripetere alcune operazioni a determinate condizioni e ha una parte di codice che può venirci utile in più occasioni, allora può essere utile suddividere il codice in parti distinte mediante le funzioni.

Le funzioni sono un costrutto, appunto, che permette di richiamare una parte di codice da altre parti dello script e possono avere dei parametri in ingresso.

La dichiarazione delle funzioni deve essere fatta prima del loro uso perché uno script bash è eseguito dall'inizio verso la fine sequenzialmente.

Le funzioni possono essere ricorsive, ovvero possono richiamare sé stesse. Può essere utile se si vuole forzare all'utente ad impostare un determinato valore prima di proseguire. Finché non è dato quel valore, la funzione può continuare a richiamare sé stessa.

La sintassi per dichiarare una funzione è la seguente:

nome_funzione( ) {……}

Corso_script_bash_parte11

```
#!/bin/bash

clear

glo_var="" #variabile globale vista dentro e fuori le funzioni
ouptut=0   #variabile globale vista dentro e fuori le funzioni

# Esempio di funzione senza parametri
mia_fun()
{
        local loc_var=1255
        echo "Questa è la funzione 'mia_fun'"
        echo "################################"
        glo_var="mia_fun"
        output=$loc_var
        return 5  #al massimo un valore fino a 255
}

# Esempio di funzione con un paio di parametri
mia_fun_par()
{
        local loc_var=455
        echo "Questa è la funzione 'mia_fun_par' con parametro1=$1 e parametro2=$2"
        echo "############################################################"
        glo_var="mia_fun_par"
        output=$loc_var

        #se il primo parametro è "Pippo" lo script viene interrotto
        if [ "$1" = "Pippo" ]
        then
                echo "L'elaborazione termina qui";read
```

```
                exit
        else
                return 10  #al massimo un valore fino a 255
        fi
}

# Esempio di funzione ricorsiva
# Esegue se stessa finché la variabile "conta" è minore di 10
# È importante che la variabile che funge da contatore sia globale
# perché se fosse locale sarebbe impostata al valore iniziale
# ad ogni esecuzione.
conta=1
mia_fun_ric()
{
        if [ "$conta" -lt 10 ]
        then
                echo "Ciclo nr.$conta per questa funzione ricorsiva"
                echo "-----------------------------------------------"
                (( conta++ ))
                mia_fun_ric
        fi
}

# codice principale dello script
echo "RICHIAMO mia_fun";read
mia_fun
echo "Codice ritorno di mia_fun=$?"
echo "glo_var=$glo_var"
echo "loc_var=$loc_var e output=$output"
echo

echo "RICHIAMO mia_fun_par";read
par1="Pippa"
par2="Paperina"
mia_fun_par $par1 $par2
echo "Codice ritorno di mia_fun_par=$?"
echo "glo_var=$glo_var"
echo "loc_var=$loc_var e output=$output"
echo

echo "RICHIAMO mia_fun_ric";read
mia_fun_ric
echo
echo "Elaborazione terminata"
```

RICHIESTA DI INPUT

Finora abbiamo visto e usato degli script su cui non è possibile interagire in alcun modo durante la loro esecuzione. Si tratta di script che devono essere pronti già dall'inizio, con tutti i parametri, per svolgere il loro compito e, una volta partiti, devono essere in grado di arrivare infondo, senza nessun intervento da parte nostra.

Però questo non sempre è utile, anzi è più frequente la necessità di poter interagire con il nostro script.

Il comando **read** permette di interrompere l'esecuzione di uno script in attesa dell'inserimento da parte dell'utente di un input, ovvero di un'informazione necessaria al prosieguo dell'elaborazione. Con questo comando possiamo aggiungere l'interattività al nostro script, che finora era un programma eseguito dall'inizio alla fine senza intervento alcuno, a meno di modificare il codice prima di rieseguirlo.

Il comando **read** non richiede per forza l'utilizzo di parametri, ma può anche essere usato semplicemente per introdurre una pausa nell'esecuzione dello script. Basta un enter per proseguire.

In altre parole, se finora lo script doveva avere tutte le impostazioni corrette prima di essere eseguito, perché una volta partito non potevamo più influire sull'esito in alcun modo, ora possiamo far partire l'esecuzione che ci chiederà cammin facendo le informazioni che gli servono.

Ad ogni esecuzione possiamo fornire informazioni diverse allo script, il quale può reagire in modo diverso a seconda dell'informazione che gli diamo.

Andiamo a vedere come si usa questo comando **read**:

> **read + ENTER** per gestire le pause nello script;
>
> **read variable1, variable2 … variableN** per inserire da prompt dei comandi i valori all'interno di una, o più variabili;
>
> **read -a tupla** per inserire da prompt dei comandi i valori all'interno di una tupla;
>
> **read -r rigatesto** legge un file di testo esterno, riga per riga, ed esegue delle operazioni sulla riga corrente che sta leggendo;

Corso_script_bash_parte12.sh

```
#!/bin/bash

clear
# verifica se il valore nelle variabili D1 e D2 è pari o dispari
echo "Immettere due numeri interi: "
read D1 D2
N=$((D1/2))
M=$((N*2))
echo D1=$D1
echo N=$N
echo M=$M
```

```
if [ "$D1" -ne "$M" ]
then
      echo "Il valore della variabile D1 è DISPARI"
else
    echo "Il valore della variabile D1 è PARI"
fi

N=$((D2/2))
M=$((N*2))
echo D2=$D2
echo N=$N
echo M=$M

if [ "$D2" -ne "$M" ]
then
      echo "Il valore della variabile D2 è DISPARI"
else
    echo "Il valore della variabile D2 è PARI"
fi

# prende quanto immesso dall'utente e lo archivia in una tupla
echo
echo "Immetti i tuoi colori preferiti separati da uno spazio"
read -a colori
echo
echo "Numero di colori immesso=${#colori[*]}"
echo "Colori specificati: ${colori[*]}"
echo
read

# legge le righe da un file ad una ad una e le visualizza in maiuscolo
i=0
while read -r testofile
do
      (( i++ ))
      echo -e "riga letta nr.$i: ${testofile^^}\r\n"
done < "/home/nicola/Documenti/Tecnica/Tutorial/Miei/Corso Bash/out.txt"
read
```