

Bash Scripting

Università di Modena e Reggio Emilia

Prof. Nicola Bicocchi (nicola.bicocchi@unimore.it)



Bash Script

- Editare lo script. La prima linea (shebang) specifica l'interprete da utilizzare per i comandi successivi (`#!/bin/bash`). Si tratta di un linguaggio interpretato (non compilato)!
- Tutte le altre linee che iniziano con # sono commenti nel codice.

```
$ vim script.sh
```

- Rendere lo script eseguibile

```
$ chmod u+x script.sh # oppure  
$ chmod 755 script.sh
```

- Eseguire lo script

```
$ ./script.sh
```



Bash Script

```
$ vim script.sh
#!/bin/bash
echo Total number of inputs: $#
echo First input: "$1"
echo Second input: "$2"
exit 0

$ chmod 755 script.sh

$ ./script.sh AAPL GOOGL MSFT
Total number of inputs: 3
First input: AAPL
Second input: GOOGL
```



Bash Script

`$./script.sh oppure $ script.sh ?`

- Se non specifichiamo un percorso - ma solo un nome - Bash cerca un programma eseguibile nell'elenco di cartelle rappresentato dalla variabile PATH.
- Se PATH non contiene la cartella punto (.) che indica la cartella corrente, i programmi non vengono trovati anche se si trovano nella cartella corrente. Due opzioni:
 - Invocazione tramite percorso esplicito relativo o assoluto (da preferire)
 - `$./script.sh`
 - Modifica alla variabile PATH (uso didattico)
 - `$ export PATH=$PATH: .`
 - `$ script.sh`

Variabili speciali

- All'interno di uno script Bash è possibile accedere ad un gruppo di variabili speciali che rendono possibile lo sviluppo
- **\$0** Il nome dello script in esecuzione
- **\$1, \$2, \$n** n-esimo parametro passato da linea di comando
- **\$*** tutti i parametri passati a linea di comando
- **\$@** come **\$*** ma in forma di lista
- **\$#** numero di parametri da linea di comando
- **\$\$** PID della shell
- **\$?** valore di ritorno dell'ultimo comando (exit)
- **shift** elimina il primo parametro dalla lista **\$1...\$n**, tutti gli altri scorrono indietro di una posizione

exit

- **exit** termina l'esecuzione di una shell (e di conseguenza anche di uno script) e ritorna al chiamante un valore [0, 255]

```
$ bash      # (avvio sotto-shell)
$ exit 15   # (terminazione sotto-shell con valore 15)
$ echo $?
15
$
```

expr

- **expr** è utilizzato per eseguire operazioni matematiche. Spesso utile negli script in abbinamento a costrutti iterativi.
 - Op. aritmetiche: +,-,* ,/,%
 - Op. di confronto: <, <=, ==, !=, >=, >
 - Op. logiche: &, |

```
$ expr 2 \* 6
```

```
12
```

```
$ A=12
```

```
$ A=$(expr $A - 1)
```

```
$ echo $A
```

```
$ 11
```



Costrutti di controllo

test

- test è un comando per eseguire verifiche di varia natura sulle stringhe (interpretandole in base ai casi come stringhe, numeri, o file). Un controllo avvenuto con successo ritorna 0, altrimenti 1.

```
$ test 5 -gt 3; echo $? # 0
$ test 5 -gt 3; echo $? # 1
$ test "nicola" == "nicola"; echo $? # 0
$ test "nicola" == "mario"; echo $? # 1
$ test -f /etc/passwd; echo $? # 0
$ test -d /etc/passwd; echo $? # 1
```



test (strings)

- `string1 = string2` True if strings are identical
- `string1 == string2` True if strings are identical
- `string1 != string2` True if strings are not identical
- `string` Return 0 exit status (=true) if string is not null
- `-n string` Return 0 exit status (=true) if string is not null
- `-z string` Return 0 exit status (=true) if string is null

test (numbers)

- int1 –eq int2 Test identity
- int1 –ne int2 Test inequality
- int1 –lt int2 Less than
- int1 –gt int2 Greater than
- int1 –le int2 Less than or equal
- int1 –ge int2 Greater than or equal

test (files)

- -d file Test if file is a directory
- -f file Test if file is not a directory
- -s file Test if the file has non zero length
- -r file Test if the file is readable
- -w file Test if the file is writable
- -x file Test if the file is executable
- -o file Test if the file is owned by the user
- -e file Test if the file exists
- -z file Test if the file has zero length

test (logic)

- -o logical or
- -a logical and
- ! logical not

[

- Il comando test ha un alter ego che si comporta nello stesso modo ma ha nome diverso
- Il commando è stato introdotto principalmente per aumentare la leggibilità degli script
- Gli spazi che vedete nell'esempio sotto, dopo [e prima di] sono obbligatori! La loro assenza produce errori!

```
$ which test  
/usr/bin/test  
$ which [  
/usr/bin/[  
$ [ -r /etc/passwd ]; echo $?  
$ 0
```



if

```
if test condizione; then
    comando
else
    comando
fi

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

```
# Esempio
if [ -f /etc/passwd -a -r
/etc/passwd ]; then
    echo “/etc/passwd leggibile!”
fi

# Esempio
if [ $# -ne 3 ]; then
    echo “numero parametri errato”
else
    echo “numero parametri corretto”
fi
```



if

```
if test condizione; then          # Esempio
    comando
elif test condizione; then
    comando
else
    comando
fi

if [ condizione ]; then
    comando
elif [ condizione ]; then
    comando
else
    comando
fi
```

```
#!/bin/bash

if [ $# -lt 2 ]; then
    echo "params < 2"
elif [ $# -lt 4 ]; then
    echo "2 <= params < 4"
else
    echo "params >= 4"
fi

exit 0
```



if (forma sintetica)

- In caso un *blocco if* determini l'esecuzione di poche istruzioni è possibile utilizzare una forma sintetica (`&&`, `||`)

```
$ [ 1 -eq 0 ] && echo "pass"
$ [ 1 -eq 1 ] && echo "pass"
$ [ 1 -eq 0 ] || echo "fail"
$ [ 1 -eq 1 ] || echo "fail"

$ [ 1 -eq 1 ] && (echo "pass"; pwd)
```

case

- Non è possibile effettuare operazioni di pattern matching all'interno di un costrutto if-test. Per ovviare al problema, si utilizza il costrutto switch-case che abbina il pattern matching alla possibilità di eseguire più confronti in modo sintetico (evitando else if).

```
# Esempio
if [ "$1" == "n?co*" ]; then
    echo "success"
fi

if [ "$1" != "[0-9]*" ]; then
    echo "success"
fi
```

case

```
case espressione in
  PATTERN_1)
    comando/i
    ;;
  PATTERN_2)
    comando/i
    ;;
  PATTERN_N)
    comando/i
    ;;
  *)
    comando/i
    ;;
esac
```

```
#!/bin/bash
if [ $# -ne 1 ]; then
  echo "usage: $0 arg"
  exit 1
fi

case "$1" in
  /*) echo "Percorso assoluto"
      ;;
  /*/*) echo "Percorso relativo"
      ;;
  *) echo "Nome semplice"
      ;;
esac
exit 0
```

Costrutti iterativi

for

```
for argomento in lista; do  
    comando/i  
    ...  
done
```

```
# Esempio: tabellina del 5  
for i in 1 2 3 4 5; do  
    echo "5 * $i = $(expr 5 \* $i )"  
done
```

```
# Esempio: mostra i nomi file in home directory  
for fname in "$HOME"/*; do  
    echo "$fname"  
done
```

while

```
while comando_esegue_con_successo; do  
    comando/i  
    ...  
done
```

```
# Esempio: contatore  
i=10  
while [ "$i" -gt 0 ]; do  
    echo $i  
    i=$(expr $i - 1)  
done
```

Espansione variabili

- Filesystem supportano nomi contenenti spazi. Di conseguenza, per evitare problemi, l'espansione di variabili all'interno di script (soprattutto se si tratta di nomi di file!) va effettuata con "".

```
$ vim script.sh
#!/bin/bash
for fname in "$HOME"/*; do
    if [ -f "$fname" ]; then
        echo F "$fname"
    elif [ -d "$fname" ]; then
        echo D "$fname"
    fi
done

$ touch "$HOME"/"Mario Rossi"
$ ./script.sh
./script.sh: line 4: [: /home/nicola/Mario: binary operator expected
```

Funzioni

Funzioni

- Definite con sintassi

```
nomefunzione() {  
    ...  
}
```

- Accedono a parametri di invocazione con sintassi **\$1 ... \$n** (come gli script)
- Ritornano al chiamante con istruzione **return** (script usano **exit**)
- Valori di ritorno possono essere letti dal chiamante con sintassi **\$?** (come gli script)

```
#!/bin/bash  
  
process() {  
    echo -n "$1"  
    [ -d "$1" -a -x "$1" ] && return 0  
    return 1  
}  
  
for f in $*; do  
    process "$f"  
    if [ $? -eq 0 ]; then  
        echo " [pass]"  
    else  
        echo " [fail]"  
    fi  
done  
  
exit 0
```

Script multi-file

```
$ vim lib.sh
```

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

```
process() {  
    echo -n "$1"  
    [ -d "$1" -a -x "$1" ] && return 0  
    return 1  
}
```

```
$ vim script.sh
```

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

```
source lib.sh  
# oppure  
# . lib.sh  
  
for f in $*; do  
    process "$f"  
    if [ $? -eq 0 ]; then  
        echo " [pass]"  
    else  
        echo " [fail]"  
    fi  
done  
  
exit 0
```



Best Practices



Struttura script

- Nello sviluppo di script è buona norma (*best practice*) aderire ad un canovaccio noto e consolidato
- Definizione interprete
- Definizione variabili globali
- Definizione funzioni
- Controllo parametri
- Corpo principale
- Terminazione

```
#!/bin/bash

USAGE="usage: $0 dirname"

if [ $# -ne 1 ]; then
    echo "$USAGE"
    exit 1
fi

if [ ! -d "$1" -o ! -x "$1" ]; then
    echo "$USAGE"
    exit 1
fi

F=0; D=0
for fname in "$1"/*; do
    if [ -f "$fname" ]; then
        F=$(expr $F + 1)
    fi
    if [ -d "$fname" ]; then
        D=$(expr $D + 1)
    fi
done

echo "#files=$F, #directories=$D"
exit 0
```

Struttura script (Best Practices)

- Trattandosi di un linguaggio antico, l'indentazione è ancora facoltativa (in Python, recente, è obbligatoria!). **Indentazione è comunque di fondamentale importanza!**
- Variabili globali sono MAIUSCOLE (ad es. USAGE="\$0 usage: ...")
- **Controllo dei parametri avviene in via negativa.** Si controllano le condizioni di fallimento e, se verificate, si termina lo script ritornando un codice errore (exit 1). Questa pratica evita indentazione eccessiva
- I valori di uscita (exit) utilizzano valori diversi per distinguere successo (exit 0) da fallimento (exit 1). Per differenziare fra diversi tipi di fallimento si possono utilizzare numeri positivi > 1 (exit 2)
- I filesystem moderni supportano la presenza di spazi. Per questo motivo, tutte le variabili fuori dal controllo del programmatore (ad es. nomi di file) vanno espanso fra doppie virgolette (echo "\$filename")