Shell scripting

Laboratorio Sistemi Operativi

Antonino Staiano
Email: antonino.staiano@uniparthenope.it

Lo scripting shell

- Lo **scripting Shell** consiste nella scrittura di script (file testuali contenenti un insieme di istruzioni) eseguibili in ambiente Shell
  - esecuzione di compiti, più o meno complessi, sulla base della combinazione dei comandi tipici offerti dal sistema operativo
- I file sono caratterizzati dall'**eseguibilità**
  - Hanno un particolare attributo che consiste nella possibilità di essere eseguiti dal sistema operativo
- Nei sistemi GNU/Linux è possibile "programmare la shell" , cioè creare dei programmi per eseguire automaticamente delle operazioni che altrimenti si dovrebbero eseguire manualmente
- Tali programmi sono noti a tutti come **script della shell**

Creazione di uno script

- Tutti i file di script hanno estensione "**.sh**" e vengono creati con un qualsiasi editor
- Una volta redatto lo script, per renderlo eseguibile occorre impostare i permessi di esecuzione mediante il comando chmod
  - Ad esempio, supponendo di aver scritto uno script chiamato prova.sh, i permessi di esecuzione si impostano aggiungendo la possibilità di eseguirlo, da parte dell'utente corrente, nel seguente modo

    chmod +x prova.sh

- Si potrà poi procedere ad eseguire lo script con il comando:

  ./prova.sh

Caratteristiche di uno script

- Uno script della shell può contenere
  - righe di commento (identificate da #)
  - una prima riga di commento speciale che indica quale shell utilizzare per interpretare lo script
  - comuni comandi Linux
- Quando un file di script viene mandato in esecuzione, la shell legge tale file riga per riga ed esegue tutte le istruzioni che incontra: **interpretazione** dei comandi
- Tutte le righe che iniziano con il carattere cancelletto **'#'** sono ignorate dalla shell
- La prima riga però ha una funzione precisa: indicare il tipo di shell che potrà eseguire lo script ed eventualmente il path per richiamare tale shell
  - Es. **#!/bin/bash**
- Nel caso questa prima riga di commento venga omessa verrà chiamata la shell di default, cioè la shell BASH

Le variabili ed il loro utilizzo

- Nella shell esistono variabili predefinite, dette built-in e le variabili definite dall'utente, dette di ambiente
- Il contenuto delle variabili viene mostrato con il comando echo seguito dal carattere **$**, che ne indica il valore
- Ad esempio, un utente potrà creare una variabile del tipo:
  - numero=234 e stamparla a video con il comando echo $numero
- Es. si supponga di voler conoscere il nome dell'utente corrente. E' possibile considerare la variabile built-in USER e stamparla a video nel modo che segue:
  - echo $USER ottenendo il nome dell'utente corrente

Le variabili BUILT-IN

- Un esempio di variabili built-in predefinite nella shell è:
  - **#** il numero di argomenti dati allo script o ad una funzione
  - **?** exit status del comando precedente (0 = ok, >0 errore)
  - **$** Process ID del processo di shell
  - **!** Process ID dell'ultimo comando eseguito in background
  - **0** Nome della shell o dello script di shell
  - **BASH** il completo pathname utilizzato per invocare la bash
  - **BASH_VERSION** la versione dell'istanza della bash utilizzata
  - **USER** nome dell'utente corrente
  - **HOME** nome della directory home dell'utente corrente

Assegnazione di un stringa ad una variabile

- Oltre a contenere valori numerici, una variabile può contenere anche stringhe
  - Ad esempio l'istruzione stringa=scatola crea una variabile di nome stringa all'interno della quale viene memorizzato il valore scatola
- Se occorre memorizzare più parole, si utilizzano gli apici singoli od anche i doppi apici
  - stringa='Centro Direzionale Isola C4'

Quoting

- I caratteri <,>,| e & od anche $ sono esempi di caratteri speciali che hanno significati particolari per la shell
- Talvolta è necessario usare i caratteri speciali in modo letterale ovvero senza far riferimento al loro significato speciale
  - Il *quoting*
- Racchiudendo una stringa di caratteri in singoli apici (quotes) priviamo i caratteri dei significati speciali che potrebbero avere
  - Da notare che esiste anche il quoting debole mediante i doppi apici nei quali una stringa è comunque sottoposta ad alcune elaborazioni da parte della shell (sostituzione di parametri, sostituzione di comando, valutazione di espressioni aritmetiche)
  - Un carattere speciale interpretato è $
    - Il contenuto della variabile a cui fa riferimento viene valutata

Esempio di script

```
#!/bin/bash
echo " il primo script"
# si crei la variabile oggi e si stampi il suo
# contenuto a video
oggi=sabato
echo  $oggi
echo "fine dello script"
```


Il comando echo

- Il comando echo visualizza una stringa sullo standard output. Vediamo un classico esempio di programma che utilizza il comando echo, con cui è possibile scrivere informazioni aggiuntive a video utili all'utente
- E' possibile utilizzarlo per scrivere un piccolo HELP per un utente a cui si vogliono dare informazioni

```
#!/bin/bash
# questo e' un semplice script
echo "  "
echo " per utilizzarlo correttamente eseguire lo script"
echo " come di seguito riportato …"
```

- Oppure si può utilizzare echo per stampare il contenuto di variabili

```
echo  $num
echo " il valore del totale  è pari a $num"
```


Il comando read

- Uno script generalmente accetta dei dati in input e genera dei dati in output: il comando read permette di leggere l'input ed il comando echo permette di scrivere in output
  - Ad esempio, lo script mioscript contiene:

```
#!/bin/bash
echo "digita qualcosa"
read qualcosa
echo "ok hai digitato: $qualcosa"
```

ottenendo, dopo la sua esecuzione:

```
digita qualcosa
ciao a tutti
ok hai digitato: ciao a tutti
```


Redirezione tra canali standard

- Per redirigere un canale standard in un altro canale standard occorre usare **l'operatore &**
- Infatti
  - **&0** fa riferimento al canale standard input
  - **&1** fa riferimento al canale standard output
  - **&2** fa riferimento al canale standard error

- La stringa '2>&1' significa: redirigere lo standard error (il 2) nel canale standard output (&1)
- Vediamo, ad esempio, il comando:

```
cat non_esiste 1> dati 2>&1
```

tale comando visualizza il contenuto del file denominato non_esiste inviandolo, non a video, ma all'interno del file dati. Inoltre invia eventuali errori all'interno dello stesso file perché scrivere 2>&1 significa redirigere lo standard error nello standard output che precedentemente era stato rediretto nel file dati

Processi e Job: PID

- Ogni processo sotto GNU/Linux viene identificato da un numero univoco: il PID
- La shell definisce i comandi eseguiti dagli utenti **job**
- Si tratta sempre di processi, ma vengono definiti job per l'implementazione del controllo dei job e per distinguerli dai processi di sistema

Processi e Job: PID (2)

- Un qualsiasi comando viene eseguito di norma in foreground, ma se si tratta di un processo che richiede un'elaborazione molto lunga è possibile eseguirlo in background (letteralmente in "sottofondo")
  - mentre il comando è in esecuzione l'utente può eseguire altre attività
- Si pensi ad esempio alla ricerca di un file all'interno di un grosso disco scorrendo tutte le directory
- Per eseguire un comando in background è sufficiente apporre il carattere & alla fine del comando
- Per conoscere i job in background basta digitare il comando:

  jobs
- E per conoscere anche il PID occorre digitare il comando:

  jobs -l

Esempio di background

- Si consideri il comando sleep, che lascia in attesa per un numero di secondi indicati come argomento e lo si esegua in background: sleep 60 &

  [8] 10896
- Se si digita il comando: jobs –l si ottiene:

[2] 10787 Stopped man cat
[3] 10861 Stopped vim
[4] 10863 Stopped vim
[7]- 10869 Stopped man sleep
[8]+10896 Running sleep 60 &
(il comando numero 8 con PID 10896 è in esecuzione)

Fermare un processo in background

- Per fermare il comando eseguito in background, basta digitare il comando kill, seguito dal numero di job, che termina il processo utente

  kill %8
- Ottenendo:

  [8] Terminated sleep 60

che indica che il comando sleep 60 è ora stato fermato

Da background a foreground

- Supponiamo di voler portare il comando eseguito in background in foreground ed attendere che venga completato
- Occorre digitare fg (foreground) seguito dall'identificativo del comando
  - In tal caso si ha: **fg %8** ottenendo una shell in attesa per 60 secondi

Background e foreground: il punto

- E' possibile portare un comando da modalità background a modalità foreground (la normale modalità di esecuzione dei comandi) utilizzando il comando **fg** seguito dall'identificativo del job.
- E' anche possibile l'operazione inversa, cioè da foreground a background mediante il comando **bg identificativo** (sospendendolo prima)
  - E' possibile anche sospendere temporaneamente un comando, senza terminarlo definitivamente, premendo la combinazione di tasti CTRL-Z. L'utente può riattivare il comando in un secondo tempo in modalità background o foreground mediante i comandi bg e fg

Background e foreground: il punto (2)

- Tutto ciò può essere utile nei casi in cui l'esecuzione di un comando appena digitato richieda più tempo del previsto
- In questo caso è possibile mettere in background il comando, ma occorre prima di tutto sospenderne l'esecuzione premendo i tasti CTR-Z.
  - Non è possibile, infatti, mettere il comando in background direttamente senza averne prima sospeso l'esecuzione
  - Una volta sospesa l'esecuzione del comando, lo si può mettere in background usando il comando bg:

    lpr divinacommedia.txt

    (stampa tutto il file divina commedia.txt)
  - CTRL-Z (sospende l'esecuzione del comando non appena si accorge che il comando richiede molto tempo). Ottengo

    jobs

    [1] 345 + stopped lpr divinacommedia.txt

    e digito: bg %1 (metto in background il comando di stampa)

Ulteriore esempio di background

- Per stampare un file molto grande si può eseguire il comando lpr in questo modo:

  lpr dati &

  [1] 534
- permettendoci di eseguire altre attività evitando di dover aspettare che il comando termini la sua esecuzione. La riga presente sotto quella che contiene il comando rappresenta il numero del job (1) ed il PID del processo (534)
- Ciò significa che il comando lpr appena lanciato è identificato dal sistema come processo numero 534 e job numero 1. E' possibile eseguire in background più di un comando contemporaneamente

Considerazioni sui jobs

- Digitando jobs potremmo ottenere:

  [1] - running lpr dati
  [2] + running cat *.txt > testo

- Il comando jobs visualizza i comandi in esecuzione in background o meno e per ogni comando viene indicato il numero del job, lo stato del comando (running o stopped) ed il comando stesso. Un comando in stato "running" è un comando in esecuzione mentre un comando in stato "stopped" è un comando fermato.
- Il carattere + indica l'ultimo job posto in background (ultimo job fermato mentre era in foreground o avviato in background), mentre il carattere – il successivo job più recente

Notifica di terminazione comando in background

- Si è detto che è possibile eseguire più comandi in background digitandoli sulla stessa riga e terminandoli con il carattere &:
- lpr dati & cat *.txt > testo &
  [1] 534
  [2] 567
- GNU/Linux avverte della conclusione di job solo quando si termina il comando corrente (ad esempio una sessione di editing in vi). Per essere avvertiti della conclusione di un job si può utilizzare il comando set -b
  - Non appena un job termina GNU/Linux interrompe il comando corrente (indipendentemente dall'operazione che si sta svolgendo) avvertendo l'utente della conclusione del job

Esempio di script n.1

- Si è detto che il comando read legge dallo standard input e il risultato della lettura lo assegna alla variabile definita dall'utente. Inoltre, il comando echo visualizza il contenuto della variabile
- Un esempio di script è il seguente:

```
#!/bin/sh
echo "quali file vuoi visualizzare?"
read nome_file
ls -l $nome_file
```

- L'utente digita il nome file da visualizzare e il comando ls lo visualizza a video

- NOTA: Per eseguire correttamente questo script lo si scriva in un file denominato prova.sh, si cambino i permessi mediante chmod (chmod +x prova.sh) e lo si esegua nel modo che segue : **./prova.sh**

Esempio di script n. 2

- Si consideri la possibilità di eseguire comandi che saranno assegnati ad una variabile. Per fare ciò occorre considerare l'accento grave (backtick) che si ottiene premendo ALTGR +apostrofo
- Ad esempio creiamo lo script "primo.sh" aiutandoci con l'editor VI.
- Si digiti:
  - touch primo.sh
  - chmod +x primo.sh
  - vi primo.sh
  - entrati nell'editor si prema la "i" per inserire del testo
  - alla fine della digitazione del testo si esca premendo " : " seguito da "wq" con w= write e q=quit

Esempio di script n.2

- Scriviamo il seguente codice:

```
#!/bin/sh
OGGI=`date +%D`
echo "la data di oggi è $OGGI"
```

- Si stamperà sullo standard output, eseguendo lo script con ./primo.sh, la data formattata:

  05/29/15

I comandi: more

- Supponiamo ad esempio di voler visualizzare il contenuto di un file e che esso sia molto lungo tanto che non appare tutto nella singola finestra della shell
- E' possibile in questo caso utilizzare il comando **more** che accetta in input dei dati (dal file) e li visualizza a video sotto forma di pagine formattate, consentendo la visualizzazione di una pagina per volta. Ad es.

  more nome_file
- Ottenendo così di visualizzare il file potendolo scorrere di riga in riga fino alla fine premendo il tasto INVIO

I comandi: less

- E' possibile usare il comando less al posto del comando more, in quanto tale comando permette di scorrere le pagine avanti ed indietro (less è in sostanza una versione migliorata del comando more)
- Difatti consente anche di poter scorrere all'indietro il file appena visualizzato
- Il suo uso è il seguente:

  less nome.file

Il comando wc

- Il comando wc conta il numero di parole/caratteri/righe in un file, in base all'opzione specificata
  - wc -w file1 *conta le parole in file1*
  - wc -c file1 *conta i caratteri in file1*
  - wc -l file1 *conta le linee in file1*

I comandi head e tail

- Il comando head mostra le prime 10 linee di un file. Se si usa l'opzione –n, head mostra le prime n linee

  head [-n] file

- Il comando tail mostra le ultime 10 linee di un file. Se si usa l'opzione –n, tail mostra le ultime n linee

  tail [-n] file

- Se sono utilizzati senza alcun argomento, visualizzano ciò che si scrive da standard input

Head e tail

- Sia head che tail hanno una serie di opzioni:
  - -nc
    - head mostra i primi n byte del file passato come argomento
    - tail mostra gli ultimi n byte
  - -v
    - Sia head che tail mostrano un header contenente il nome del file passato come argomento

Esempi

- Se vogliamo visualizzare le ultime 3 linee del file scriveremo:
  - tail --lines=3 nome_file, oppure
  - tail -3 nome_file, o anche
  - tail –n3 nome_file
- Per estrarre le prime 85 righe di un file
  - head -85 file.txt
- Per estrarre le righe da 40 a 50 da un file
  - head -50 file.txt | tail -10

Costrutti

COSTRUTTI DI PROGRAMMAZIONE DELLA SHELL BASH

Argomenti trattati

- Costrutto condizionale if..then…else, elif
- Costrutto condizionale case
- Cicli: while
- Cicli: for
- Le funzioni ed il loro utilizzo negli script

Variabili built-in

- Quando è avviato uno script di shell, sono inizializzate alcune variabili dette built-in

Variabile	Descrizione
$HOME	Home directory dell'utente corrente
$PATH	Lista di directory separate dai due punti (:) per la ricerca di comandi
$PS1	Prompt dei comandi. Solitamente $. E' possibile usare valori più complessi. Ad esempio, [\u@\h \W] è il default per indicare utente, nome della macchina, directory corrente e $
$PS2	Prompt dei comandi secondario. Usato quando è necessario fornire input aggiuntivi. Solitamente è il simbolo >
$IFS	Separatore campi di input
$0	Nome dello script
$#	Numero di parametri passato
$$	PID del processo dello script

Variabili associate ai parametri

- Se lo script è invocato con dei parametri, sono create alcune variabili aggiuntive

Variabile	Descrizione
$1, $2, …	Parametri forniti allo script
$*	Lista di tutti i parametri, mostrata in una singola variabile, separati dal primo carattere della variabile di ambiente IFS (spazio, tab, …)
$@	Sottile variante di $* che non usa il separatore IFS

Parametri e variabili built-in

```
#!/bin/bash
salutation="Hello"
echo $salutation
echo "Ora è in esecuzione il programma $0"
echo "Il secondo parametro era $2"
echo "Il primo parametro era $1"
echo "La lista dei parametri era $*"
echo "La home directory dell'utente è $HOME"

echo "Inserire un nuovo saluto"
read salutation
echo $salutation
echo "Script completato"
exit 0
```

Parametri e variabili built-in (cont.)

```
$ ./prova_var foo bar baz
Hello
Ora è in esecuzione il programma ./prova_var
Il secondo parametro era bar
Il primo parametro era foo
La lista dei parametri era foo bar baz
La home directory dell'utente è /home/staiano
Inserire un nuovo saluto
Ciao
Ciao
Script completato
```


Esempio

- Verifichiamo la differenza tra $@ e $*

```
$ IFS = ''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

- Come si può vedere, nelle virgolette, $@ espande i parametri posizionali come campi separati, indipendentemente dal valore di IFS

Condizioni

- In tutti i linguaggi di programmazione è fondamentale la possibilità di testare delle condizioni ed eseguire azioni differenti in seguito all'esito dei test
- Vediamo alcuni costrutti condizionali e successivamente alcune strutture di controllo che li usano
- Uno script di shell può testare il codice di uscita di un qualsiasi comando, inclusi gli script scritti dall'utente
  - Ragion per cui è sempre meglio includere un comando exit alla fine dei propri script

Comando test (oppure [ )

- Molti script fanno un uso estensivo di **[** o del comando test (il controllo Booleano della shell)
- I comandi [e test sono sinonimi
  - Quando, però, si utilizza [è necessario includere anche] per questioni di leggibilità
- Esempio: verifica esistenza di un file

Comando test	Comando [

`if test -f fred.c`
`then`
`…`
`fi`
	`if [ -f fred.c ]`
	`then`
	`…`
	`fi`

Tipi di condizione: confronto stringhe

Confronto	Risultato
stringa1 = stringa2	Vero se le stringhe sono uguali
stringa1 != stringa2	Vero se le stringhe non sono uguali
-n stringa	Vero se la stringa non è nulla
-z stringa	Vero se la stringa è nulla (stringa vuota)

Tipi di condizione: confronto aritmetico

Confronto	Risultato
espressione1 **–eq** espressione2	Vero se le espressioni sono uguali
espressione1 **–ne** espressione2	Vero se le espressioni non sono uguali
espressione1 **–gt** espressione2	Vero se espressione1 è maggiore di espressione2
espressione1 **–ge** espressione2	Vero se espressione1 è maggiore o uguale di espressione2
espressione1 **–lt** espressione2	Vero se espressione1 è minore di espressione2
espressione1 **–le** espressione2	Vero se espressione1 è minore o uguale di espressione2
!espressione	Vero se espressione è falsa

Tipi di condizione: condizioni sui file

Condizione	Risultato
-d file	Vero se il file è una directory
-e file	Vero se il file esiste
-f file	Vero se il file è regolare
-g file	Vero se set-group-id è 1 per file
-r file	Vero se il file è leggibile
-s file	Vero se il file ha una dimensione non zero
-u file	Vero se set-user-id è 1 per file
-w file	Vero se il file è scrivibile
-x file	Vero se il file è eseguibile

Strutture di controllo

- Note le condizioni, possiamo passare alle strutture di controllo che le usano
- La shell ha un insieme di strutture di controllo che sono molto simili a quelle di altri linguaggi di programmazione

Costrutto condizionale if… then…else

- Nel caso di scelta tra due alternative è possibile utilizzare il costrutto if…then…else, che deve essere scritto secondo la sintassi nell'esempio che segue:

```
#domanda
echo –n "Qual è il nome di Hendrix? "
# recupero la risposta
read RISPOSTA
# stampo il risultato
if [ $RISPOSTA = "jimi" ]
then
  echo "risposta corretta"
else
  echo "risposta errata"
fi
```


Un piccolo problema con le variabili

- Nell'esempio precedente, supponiamo di premere invio piuttosto che inserire il nome di Hendrix
  - Otteniamo l'errore: [: =: unary operator expected
  - Il problema sta nella clausola if
    - Quando la variabile RISPOSTA viene testata, essa ha assunto il valore di stringa nulla. Ciò equivale ad una clausola if del tipo: `if [ = "jimi" ]` che non è valida
  - Per evitare questo tipo di problemi, dobbiamo usare le virgolette con la variabile
    - `if [ "$RISPOSTA" = "jimi" ]`
    - In questo caso, premendo enter otterremo una clausola if corretta del tipo
      - `if [ "" = "jimi" ]`

Costrutto condizionale if… then…else

- Creiamo il file "costrutto_if.sh", scriviamo il codice prima presentato e cambiamo il permesso

- I comandi da digitare in sequenza sono:
  - vi costrutto_if.sh + scrittura del codice dall'editor VI
  - chmod +x costrutto_if.sh
  - ./costrutto_if.sh

Esempio

```
#!/bin/bash

if [ -f /bin/bash ]
then
  echo "il file /bin/bash esiste"
fi

if [ -d /bin/bash ]
then
  echo "/bin/bash è una directory"
else
  echo "/bin/bash non è una directory"
fi
```

Costrutti if innestati…elif

```
#!/bin/bash
#domanda
echo -n "quanti anni hai?"
#risposta
read ANNI
#costrutti if annidati
if [ "$ANNI" -lt 10 ]
then
 echo "sei un bambino"
elif [ "$ANNI" -lt 18 ]
then
echo "sei minorenne"
else
 echo "sei maggiorenne"
fi
```


Costrutto condizionale CASE

- Si consideri la scelta tra una serie di possibilità che devono essere elencate per consentire l'esecuzione di un differente codice a seconda della risposta fornita
- Per ottenere ciò si utilizza il costrutto CASE con la seguente sintassi:

```
case variabile in
  pattern [ | pattern] …) comandi;;
  pattern [ | pattern] …) comandi;;
  …
esac
```


Esempio

```
#!/bin/bash

echo "Supererete la prova intercorso? Prego rispondere si o no"

read esito

case "$esito" in
  si) echo "Bene, sosterrete anche la prossima";;
  no) echo "Ok, ci vediamo al prossimo appello";;
   s) echo "Bene, sosterrete anche la prossima";;
   n) echo "Ok, ci vediamo al prossimo appello";;
   *) echo "Spiacente, risposta non riconosciuta";;
esac
exit 0
```


Esempio: seconda versione

```
#!/bin/bash

echo "Supererete la prova intercorso? Prego rispondere si o no"
read esito

case "$esito" in
  si | s | Si | SI) echo "Bene, sosterrete anche la prossima";;
  n* | N*)          echo "Ok, ci vediamo al prossimo appello";;
  *)              echo "Spiacente, risposta non riconosciuta"
                  exit 1;;
esac
exit 0
```

- Attenzione con l'uso di *: digitando nino, ad esempio, verrà visualizzato `Ok, ci vediamo al prossimo appello`

Cicli: while

- Consideriamo la possibilità di ripetere un blocco di codice mentre è verificata una condizione
  - è il caso del ciclo WHILE che presenta una sintassi del tipo:

```
while condizione do
      comandi
done
```


Ciclo while: esempio

```
#!/bin/bash
var0=0
while [ "$var0" -lt 10 ]
do
  echo -n "$var0"
  var0=`expr $var0 + 1`
done
```

- E' da osservare che l'incremento è preferibile eseguirlo nella forma (expr è più lento)
- `var0=$(($var0+1))`

Espansione aritmetica

- Il comando expr consente l'esecuzione di comandi aritmetici
  - L'esecuzione, però, è lenta poiché è invocata una nuova shell per eseguire expr
- Un'alternativa più recente, e migliore, è l'espansione $((…))
  - Racchiudendo l'espressione che vogliamo valutare in $((…)), siamo in grado di eseguire operazioni aritmetiche semplici in modo efficiente

Cicli: for

- Il costrutto for consente di ciclare attraverso un range di valori rappresentato da un qualsiasi insieme di stringhe

```
for variabile in valori
do
      comandi
done
```

Esempio

```
for X in 1 2 3 4 5
do
  echo $X
done
```

- Eseguendo lo script si ha:

1
2
3
4
5

Cicli: for

- I valori normalmente sono stringhe, per cui possiamo scrivere

```
#!/bin/bash
for foo in bar fud 43
do
  echo $foo
done
exit 0
```


Altro esempio

```
#!/bin/bash
for file in $(ls f*.sh); do
  lpr $file
done
exit 0
```

- Notare l'uso della sintassi $(comando) che fornisce la lista dei parametri al for mediante l'output del comando racchiuso nella sequenza $()

Esecuzione di comandi

- Quando si scrivono degli script, spesso è necessario catturare il risultato dell'esecuzione di un comando
  - Si vuole, cioè, eseguire un comando e porre il risultato in una variabile
- Ciò è possibile usando la sintassi $(comando) oppure usando il backtick (`) (una vecchia forma ancora in uso)
- Il risultato di $(comando) è semplicemente l'output di comando

Cicli: for

- Il ciclo for presenta anche un'altra sintassi molto simile ai linguaggi di alto livello:

```
for (( X=0; X<10; X++ ))
do
echo $X
done
```


Esempio

- Eseguendo lo script si ha un analogo risultato:

```
0
1
2
3
4
5
6
7
8
9
```


Funzioni

- Una funzione è una sorta di script nello script
  - Si usa per assegnare un nome a porzioni di codice e conservarlo in memoria. Successivamente è possibile invocarla in qualsiasi momento
- Le funzioni migliorano la programmabilità della shell per due ragioni
  - Quando si invoca una funzione la si trova già in memoria
  - Ideali per organizzare lunghi script in pezzi di codice modulare più semplici da sviluppare e mantenere

Funzioni

- Per definire una funzione, scriviamo il suo nome seguito da parentesi vuote e racchiudendo le istruzioni tra {}

```
nome_funzione () {
      istruzioni
}
```

- o, alternativamente

```
function nome_funzione{
      istruzioni}
```

Funzioni

- Una volta definita, una funzione può essere richiamata semplicemente invocandone il nome seguito (eventualmente) da uno o più argomenti, come avviene per gli script
- E' da osservare
  - Le funzioni non sono eseguite in processi separati. Invece gli script si
  - Se una funzione ha lo stesso nome di uno script o di un eseguibile, la funzione ha la precedenza
- L'ordine di precedenza delle varie sorgenti di comandi della shell è
  - Alias
  - Parole chiavi
  - Funzioni
  - Comandi built-in
  - Script ed eseguibili

Esempio

```
#!/bin/bash
foo() {
  echo "La funzione foo è in esecuzione"
}
echo "inizio script"
foo
echo "fine script"
exit 0
L'esecuzione ci fornisce
inizio script
La funzione foo è in esecuzione
fine script
```


Funzioni e parametri

- Le funzioni usano i parametri posizionali e le variabili speciali come * e # alla stessa maniera degli script

```
function alice
{
  echo "alice: $@"
  echo "$0: $1 $2 $3 $4"
  echo "$# argomenti"
}

alice nel paese delle meraviglie
```


Scrittura di una funzione e richiamo

- Si consideri, ad esempio, una funzione che accetta in ingresso un prezzo e ne restituisce il corrispondente con aggiunta di IVA al 20%

```
tot_iva() {
IVA=$(($1 * 20))
IVA=$(($IVA / 100))
TOT=$(($1 + $IVA))
echo $TOT
}
```

- Ove $1 si riferisce al parametro passato dallo script che eseguirà la funzione

Scrittura di una funzione e richiamo

- Scriviamo ora un semplice script per richiamare la funzione

```
#!/bin/bash

#definiamo due variabili
DVD=7
BRAY=12
#richiamo la funzione
echo "prezzo di un DVD con IVA è: $(tot_iva $DVD) euro"
echo "prezzo di un BRAY con IVA è: $(tot_iva $BRAY) euro"
```


Scrittura di una funzione e richiamo

- Eseguiamo lo script e otteniamo:

prezzo di un DVD con IVA è: 8 euro
prezzo di un BRAY con IVA è: 14 euro

Funzioni e parametri

- E' sempre necessario definire una funzione prima di invocarla
  - Gli script iniziano l'esecuzione a partire dall'inizio del file. Basta porre la dichiarazione di tutte le funzioni all'inizio dello script
- Quando viene invocata una funzione, i parametri posizionali dello script $*, $@, $#, $1, $2 e così via sono sostituiti dai parametri per la funzione
  - Quando la funzione termina, sono ripristinati i parametri precedenti
- Le funzioni possono ritornare valori numerici mediante il comando return. Se non è specificato return, la funzione restituisce lo stato di uscita dell'ultimo comando eseguito
  - Il modo, invece, per tornare una stringa è memorizzare la stringa in una variabile che verrà usata anche dopo che la funzione termina
  - Alternativamente è possibile fare l'echo di una stringa e "catturare" il risultato:

```
foo () {echo NAPOLI;}
…
result = "$(foo)"
```


Esempio

```
#!/bin/bash
# lo script inizia con la definizione della
  funzione
si_o_no() {
  echo "$* è il tuo nome?"
  while true
  do
    echo -n "Immetti si o no: "
    read x
    case "$x" in
      s | si) return 0;;
      n | no) return 1;;
      *)     echo "Inserisci si o no"
    esac
  done
}
```

Esempio (cont.)

```
# inizia la parte principale del programma
echo "I parametri originali sono $*"
if si_o_no "$1"
then
  echo "Ciao $1, bel nome"
else
  echo "a buon rendere, ciao"
fi
exit 0
```

```
$ ./my_name Marek Dries
I parametri originali sono Marek Dries
Marek è il tuo nome?
Immetti si o no: si
Ciao Marek, bel nome
$
```


Esercizio

- Scrivere una funzione che ritorni il numero di parametri (massimo due) con cui è stata "chiamata"


```
#!/bin/bash
func2 () {
   if [ -z "$1" ]
   # vediamo se ci sono parametri
   then
     echo "nessun parametro passato alla funzione"
     return 0
 else
     echo "il primo  parametro passato è ... $1."
   fi
if [ -n "$2" ]
   then
     echo "il secondo parametro passato è ... $2."
   fi
}
#richiamo la funzione con nessun parametro
func2
echo " altro caso"
#richiamo la funzione con un parametro
func2 Primo
echo "altro caso"
#richiamo la funzione con due parametri
func2 Primo Secondo
echo "stop"
exit 0
```