



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Introduzione allo Scripting Bash

Un'introduzione allo scripting della shell Bash

by

Wolf



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Struttura della mia mezzora

In questa mezzora (e in quelle future) ci saranno molti esempi, e alcuni esercizi per verificare e approfondire quanto imparato. Una nota: trattandosi di una introduzione, un livello base, cerco di usare (nei limiti del possibile!) un linguaggio che sia il più possibile comprensibile a chiunque.

Negli esempi userò “\$” per indicare il prompt primario e “>” all'inizio della riga (non è la redirectione) per indicare il secondario, se non compariranno questi simboli vuol dire che il codice sarà il contenuto di uno script.

Il *pseudocodice* sarà in corsivo, **il codice in blu**, **i commenti in blu più scuro**, **gli esercizi seguiranno la parola “esercizi” in verde**, alcune parole particolarmente importanti verranno sottolineate, **e le parti che potrebbero portare dei danni a propri file o al sistema se usati incautamente in rosso**.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Introduzione a Bash

Bash (Bourne Again Shell, Shell rinata...), è la shell più usata sotto Linux anche se non è la sola...
Una Shell è un'interfaccia testuale per interagire col sistema.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Cos'è lo scripting Bash

Lo scripting Bash è un vero e proprio linguaggio di programmazione.

Esistono principalmente due tipi di linguaggi, uno compilato, come ad esempio può essere il C, e uno interpretato, come ad esempio Python, lo scripting Bash è un linguaggio interpretato.

I primi sono più veloci nell'esecuzione in quanto i file eseguibili vengono eseguiti direttamente dal sistema, dalla macchina, mentre i secondi sono più lenti perché il programma viene sostanzialmente scritto in un file di testo e passato alla macchina da un interprete al momento dell'esecuzione, questi permettono però un uso più immediato (non devono essere ogni volta compilati) e più flessibile, tanto è vero che spesso si possono usare anche da linea di comando e non soltanto inclusi in un file eseguibile.

C'è da dire che il divario in velocità di esecuzione tra le due metodologie con il moderno hardware si minimizza sempre di più, nonostante per certi usi i linguaggi compilati, come il C o il C++, sono insuperabili.

Esercizi: citate oltre a quelli elencati 3 linguaggi interpretati e 3 linguaggi compilati.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Perchè usare lo scripting Bash

Uno dei motivi è quello citato nel paragrafo precedente, ovvero il fatto di poterlo usare anche da linea di comando, lanciando delle istruzioni "al volo" che ci semplificano molti compiti complessi o ripetitivi, se le istruzioni sono un po' lunghe o si pensa di riutilizzarle altre volte conviene includerle in un file, creando uno script Bash vero e proprio.

Il secondo motivo è l'interazione con tutti i numerosissimi comandi di sistema tipici di Unix e di Linux, cosa che si ottiene anche con altri linguaggi, ma con questo la cosa è particolarmente efficace.

Il terzo motivo è la maggior comprensione del sistema e la possibilità di personalizzarlo una volta che si conosce questo linguaggio, Linux è basato su moltissimi script Bash, dall'avvio a molti compiti che svolge durante il suo funzionamento.

Il quarto motivo è che Linux è molto... loquace, nel senso che produce molto output, nei moltissimi log, ma anche i device sono file spesso in lettura e scrittura, la shell fornisce moltissimi strumenti per la lettura, scrittura e manipolazione delle stringhe di testo, questo si traduce in una potentissima interazione col sistema tramite comandi da linea di comando o script, nei prossimi paragrafi imparerete a scrivere in un file, da quello sarà possibile settare addirittura un dispositivo tramite la scrittura di un file, ad esempio attualmente nel mio portatile gira un programma in C che mi regola la temperatura della CPU tramite l'attivazione e la disattivazione della ventola, prima di fare il programma in C era uno script Bash che regolava tale funzionamento, e la base era una redirectione su di un file che attiva o disattiva la ventola del processore.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Cosa bisogna già sapere per usare lo scripting Bash

Nonostante l'introduzione già citata alla Shell Bash e alla sua utilità, va precisato che questo non è un corso introduttivo alla Bash, ma allo scripting Bash, perciò si presuppone una conoscenza almeno di base alla stessa Bash, un uso dei comandi di base, l'uso del pipe (no Gianluca!!! :D [n.d.r. "pipex" è il nick di Gianluca]) e della redirectione (anche se di seguito darò una rinfrescatina su questi argomenti), sarebbe gradita (ma non indispensabile) un'infarinatura di programmazione. Un'altra cosa che si richiede è il sapere cosa si intende per input e per output (comunque anche su questo aggiungerò qualcosa).

Per non allungare troppo questo mini-corso sarebbe utile conoscere il quoting, l'escaping e i caratteri speciali.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Pipe

Pipe (|), serve per canalizzare l'output di un comando in un comando successivo, esempio:

```
$ ls | grep pippo
```

In questo esempio faccio un “ls” della directory corrente, l'output anzichè essere visualizzato a schermo viene "canalizzato" al comando “grep” che verificherà se ci sarà qualche file o qualche directory che ha il nome "pippo" all'interno del suo nome (esempio: pippo, superpippo, pippociao, ecc).

Esercizi: trovate almeno 3 situazioni dove è utile concatenare dei comandi e fate delle prove.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Redirezione

La redirezione sostanzialmente viene usata per scrivere l'output di un comando in un file, normalmente si usano due metodi, il primo è con il segno di maggiore (o parentesi angolata chiusa) singolo, questo scrive l'output nel file, eventualmente sovrascrivendolo se già esiste, esempio:

```
$ ls > contenuto-directory.txt
```

scrivo il contenuto della directory corrente visualizzato con "ls" in un file anzichè visualizzarlo a schermo, se "contenuto-directory.txt" esiste già verrà sovrascritto.

Se uso lo stesso segno doppio (>>) e il file esiste non verrà sovrascritto, l'output verrà aggiunto in coda al file (append), se il file non esiste verrà creato, esempio:

```
$ find . -iname '*.jpg' >> tutti-i-jpg.txt
```

scrivo la lista di tutti i file con estensione “.jpg” nella directory corrente e in tutte le sotto-directory nel file "tutti-i-jpg.txt", se non esiste lo crea, se esiste ne aggiunge in coda il contenuto.

C'è anche un terzo modo di usare la redirezione, con “<”, redirezionando lo standard input, un esempio può essere:

```
$ wc -l < tutti-i-jpg.txt
```

in questo caso il comando “wc” con l'opzione “-l” prenderà lo standard input del file “tutti-i-jpg.txt” e ne conterà le linee, però al momento focalizzatevi più sui primi due usi della redirezione, che sarà la maggioranza dei casi che incontrerete nei vari script e che userete.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>



Introduzione allo scripting Bash

La redirectione è spesso usata negli script anche per eseguire un comando senza visualizzare o solo lo standard output o solo lo standard error o tutti e due.

In Linux/Unix si dice che “tutto è un file”, in questo senso tutti i dispositivi sono rappresentati nella directory /dev, per questo lo standard input è rappresentato dal file “0” (ad esempio l'input da tastiera), lo standard output è rappresentato da “1” (tipicamente il risultato a schermo di un comando), e lo standard error da “2” (i messaggi di errore), il riferimento a questi file verrà usato con la redirectione per ottenere alcuni effetti molto utili, di seguito alcuni esempi:

```
$ ls 1>/dev/null
```

(“/dev/null” è una specie di buco nero dove sparisce ciò che vi viene rediretto), in questo caso non si vedrà la lista di file e directory che “ls” dovrebbe visualizzare, ma si vedranno eventuali messaggi di errore (ad esempio può essere utile per stabilire se la directory è vuota senza produrre un eventuale lungo output, se ne vedrà l'uso nella sezione dedicata agli exit code).

```
$ ls 2>/dev/null
```

si vedranno file e directory, ma non si visualizzeranno eventuali messaggi di errore.

```
$ ls &>/dev/null
```

non verranno visualizzati file e directory e nemmeno eventuali messaggi di errore.

Una nota: preciso che ho usato l'estensione .txt nei file in cui si scrive l'output con le redirectioni per far meglio capire che sono file di testo, però l'estensione non è obbligatoria in quanto a Linux non servono le estensioni per riconoscere i file!

Un'altra nota: se volete usare la redirectione visualizzando contemporaneamente l'output a schermo potete usare il comando "tee", rimando alla (semplice) pagina di manuale per questo comando...

Esercizi: lanciate alcuni comandi scrivendo l'output in alcuni file, differenziate la cosa a seconda che vi serva sovrascrivere il file o aggiungere l'output alla fine, almeno 3 situazioni per ogni caso.

ATTENZIONE: verificate di non sovrascrivere file esistenti!

Fate delle prove a linea di comando, magari con “ls”, con i vari “1>/dev/null”, “2>/dev/null” e “&>/dev/null” per visualizzare o non visualizzare o solo i messaggi di errore o solo l'output, o nessuno dei due (suggerimento: se volete produrre un messaggio di errore cercate con “ls” un file inesistente).



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



“Ciao mondo” e il comando “echo”

Come in tutti i linguaggi di programmazione la prima frase che si andrà a scrivere sarà “ciao mondo” (ma cambiare frase no????!! :D), beh, aprendo una shell e digitando:

```
$ echo ciao mondo
```

otterremo “ciao mondo” a schermo, semplice, no?

Il comando “echo” mostra a schermo una linea di testo, si spiega meglio con degli esempi (negli esempi non c'è il simbolo del prompt, ma potete provarli direttamente da linea di comando):

```
echo ciao
```

produrrà a schermo “ciao”

```
echo che palle sta mezzora!
```

produrrà a schermo “che palle sta mezzora!”

Con “echo” si potrà non usare gli apici, usare gli apici doppi, o usare gli apici singoli, nel caso degli apici singoli si definisce come “quoting forte”, perchè eventuali caratteri speciali o espansioni delle variabili (si veda il capitolo sulle variabili) non verranno interpretate e la stringa verrà scritta così com'è in origine.

“echo” ha anche delle opzioni, vi rimando al semplice man.

Esercizi: provate a fare un po di scritte dalla linea di comando con “echo”, fate anche un po di esperimenti con le opzioni dopo aver letto il man.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



I caratteri jolly, i caratteri speciali e il globbing

I caratteri jolly e i caratteri speciali sono alcuni caratteri che nella shell sono molto utili per vari scopi, ad esempio dovremmo sapere più o meno tutti che “~” corrisponde alla home dell'utente, così come “*” indica qualsiasi stringa o carattere, una breve lista dei più interessanti:

* sostituisce qualsiasi stringa o carattere (esempio “*.jpg” indica tutti i file che finiscono in “.jpg”)

? sostituisce un solo carattere

~ la home directory

~+ la directory corrente (come la variabile d'ambiente “PWD”)

; il punto e virgola equivale ad andare a capo

Ce ne sono altri, vi invito a fare una ricerca per ampliare la vostra conoscenza in tal senso in quanto sarebbe un argomento molto lungo da trattare e la cosa è decisamente utile sia da riga di comando (tipicamente per cercare file o testo all'interno di essi) che dagli script.

Esercizi: se non conoscete già i caratteri jolly e il globbing, dopo aver trovato documentazione al riguardo esercitatevi nel fare delle ricerche nel vostro sistema, tramite il comando “ls” e “find”.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Lo scripting dalla linea di comando

Come già detto lo scripting dalla linea di comando permette di creare delle istruzioni complesse al volo per eseguire compiti ripetitivi che possono servire solo quella volta (altrimenti è più conveniente includerli in uno script vero e proprio).

Per spiegare meglio il concetto faccio un esempio pratico.

A volte, molto raramente a dire il vero, mi è capitato di avere la batteria del portatile quasi a zero e voler farla scaricare del tutto, magari lasciandolo acceso quando vado a letto, però in alcuni casi mi sarebbe piaciuto sapere per quanto tempo è rimasto acceso, perciò da linea di comando ho lanciato i seguenti comandi (al momento non sforzatevi di capirli, cogliete solo l'utilità della cosa):

```
$ while :
```

```
> do
```

```
> uptime > uptime.txt
```

```
> sleep 120
```

```
> done
```

in pratica questo costrutto è un ciclo infinito, ogni volta il comando “uptime” scrive da quanto tempo è acceso il PC nel file “uptime.txt”, sovrascrivendolo, poi aspetta 120 secondi prima di ripetere il ciclo, quando riaccenderò il computer leggerò nel file quanto tempo sarà rimasto acceso il computer, con uno scarto massimo di 120 secondi (che nel caso mi sta più che bene!).

Esercizi: elencare almeno 3 casi in cui vi verrebbe utile questa caratteristica.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Gli script

Arriviamo ai veri e propri script...

Esercizi: elencate almeno 3 casi in cui vi sarebbe utile uno script per delle operazioni che fate ciclicamente e vi risultano noiose o vi portano via molto tempo.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



La sha-bang

La sha-bang è un doppio carattere che dovete mettere all'inizio di ogni script affinché questo, una volta reso eseguibile, venga eseguito dalla Bash.

I caratteri sono “#!”, seguiti dal percorso dell'eseguibile “bash”, o “sh”, che in Linux è un collegamento a “bash”, la linea completa per convenzione è solitamente:

```
#!/bin/sh
```

oppure direttamente

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

Una nota: la sha-bang serve anche per altri linguaggi di scripting, un esempio può essere Python, in cui la prima linea sarà:

```
#!/usr/bin/python
```

Un'altra nota: se volete sapere il percorso degli eseguibili vi rimando al comando “which”, con l'opzione “-a” potete verificare se ce n'è più di uno in “PATH” (se non sapete cos'è vi rimando alla descrizione delle variabili di ambiente).



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



I commenti

I commenti negli script si ottengono con “#” (in effetti anche il primo carattere della sha-bang sarebbe un commento, ma... su questo mi fermo qua!!!), servono generalmente per non far eseguire parti di codice senza cancellarlo, e per aggiungere commenti al codice che state scrivendo, cosa utile se qualcun'altro metterà mano ai vostri script, ma anche a voi stessi se ci mettete mano dopo un po di tempo!

Esempi:

```
# Questo è un commento
```

```
# Il comando qua sotto mi saluta
```

```
echo ciao
```

```
# il commento qua sotto fa sì che il codice non venga eseguito
```

```
#echo ciao
```



Mezzora
d'amicizia

<NOME DELLA MEZZORA>



Introduzione allo scripting Bash

Le variabili

Una variabile possiamo immaginarla come un contenitore in cui inserire dei dati, generalmente ne esistono due tipi, una per le stringhe e una per i numeri, in molti linguaggi queste vengono tipizzate, ovvero una variabile numerica non può includere stringhe di caratteri e viceversa, nel nostro scripting però questo non avviene, perciò una variabile potrà includere indifferentemente stringhe o numeri.

La dichiarazione di una variabile è molto semplice, basta aggiungere l'operatore di assegnazione “=” e il valore da assegnare, il tutto senza spazi, esempio (da linea di comando):

```
$ a=ciao
```

per riavere il valore incluso nella variabile dovremo anteporre l'operatore “\$”, perciò nel caso dell'esempio per visualizzare “ciao” a schermo basterà lanciare:

```
$ echo $a
```

Se vogliamo aggiungere una stringa con degli spazi dovremo usare gli apici, esempio:

```
$ a="ciao mondo"
```

lanciamo

```
$ echo $a
```

per vedere il risultato, la differenza tra apici doppi e singoli è stata spiegata prima.

Se vogliamo aggiungere un numero la procedura è la stessa:

```
$ n=2
```

A volte può capitare di vedere le parentesi graffe “{}” nella sintassi, esempio:

```
$ echo ${a}
```

sostanzialmente non c'è differenza, si potrebbe dire che non usarle è una forma più veloce e abbreviata, la forma completa però ci può essere utile in alcuni casi, faccio un esempio, proviamo ad aggiungere dei caratteri alla nostra variabile “a”:

```
$ echo $acane
```

non avremo nessun output perchè Bash cercherà una variabile “acane”, se però usiamo:



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



```
$ echo ${a}cane
```

avremo... “ciao mondocane” (:D), capito il principio?

Ci sono anche delle operazioni di “pattern matching” utilizzabili con le parentesi graffe, ma non è oggetto di questo corso, verrà riproposto in un corso più avanzato...

Per assegnare una variabile ad un'altra variabile la cosa è molto semplice, un esempio lo spiega bene:

```
$ b=$a
```

il valore di “a” è stato assegnato a “b”, se vorremo integrare la stessa variabile potremo usare un sistema come il seguente:

```
$ a=${a}cane
```

ora se digiteremo

```
$ echo $a
```

l'output non sarà più “ciao mondo” ma “ciao mondocane”, capito il principio?

Le variabili possono poi avere una visibilità limitata alla funzione o allo script (e perciò alla shell in cui viene eseguito), perciò saranno variabili locali, o possono essere visibili alle shell generate dalla stessa tramite il comando “export”, a parte quanto vedete in file come “/etc/profile”, un esempio può chiarire più di mille parole, se in una shell io digito:



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



```
$ a=ciao
```

dando

```
$ echo $a
```

visualizzerò “ciao”, e sino a qui nulla di nuovo, però provate ad aprire una sub-shell dando:

```
$ su nomeutente
```

“nomeutente” sarà ovviamente il vostro nome utente, poi da questa seconda shell lanciate ancora

```
$ echo $a
```

non visualizzerete nulla, ritornate nella prima shell con “exit”, ora digitate

```
$ export a
```

poi di nuovo

```
$ su nomeutente
```

successivamente

```
$ echo $a
```

e vedrete che adesso la variabile è stata esportata alla seconda shell.

Un sistema per fare questa cosa più rapidamente è esportarla contemporaneamente a quando gli si assegna il valore, nel caso della nostra variabile “a”:

```
$ export a=ciao
```

spero sia chiaro...



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Variabili d'ambiente

In alcune variabili sono memorizzati alcuni dati riguardanti il sistema, sono appunto le “variabili d'ambiente”, per convenzione queste sono formate da lettere maiuscole, ad esempio “PATH” memorizza tutti i percorsi dove sono collocati i file eseguibili, “HOME” la home dell'utente corrente, “PWD” la directory corrente, ce ne sono molte altre, per visualizzarle usate il comando “env”.

Queste variabili sono modificabili in ambito locale negli script, ad esempio:

```
PATH=$PATH:/home/nomeutente/bin
```

aggiungerà ai vari percorsi in “PATH” il percorso “/home/nomeutente/bin”, solo per lo script corrente.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Variabili posizionali

Le variabili posizionali sono variabili che contengono gli argomenti che vengono passati ad uno script o ad una funzione, sono:

\$0 contiene il nome dello script

\$1 contiene il primo argomento passato allo script

\$2 contiene il secondo argomento passato allo script

...e così via

\$# contiene il numero di argomenti passati allo script

\$@ e \$* contengono tutti gli argomenti

\$? contiene il codice di uscita del comando precedente (lo vedremo più avanti)

Per “svuotare” una variabile potrete usare il comando “unset” seguito dal nome della variabile.

Esercizi: provate a creare alcune variabili, visualizzate il contenuto di alcune variabili d'ambiente, aggiungete un percorso inventato a “PATH”, fate delle prove con “export” come quella citata nell'esempio.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Read

Questo comando legge gli input da tastiera, tipicamente è usato per scrivere una variabile direttamente da tastiera da un prompt, per questo “read” ha un'opzione molto utile, “-p”, ad esempio:

```
read -p "Come ti chiami? " nome
```

darà come risultato il prompt “Come ti chiami? “ e una volta digitato il nome e dato invio immagazzinerà il nome immesso nella variabile “nome”.

Un altro uso di “read” può essere quando delle istruzioni si devono fermare e per farle continuare lo script aspetta un input da tastiera.

Esercizi: da linea di comando sperimentate quanto avete letto riguardo a “read”.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Istruzioni condizionali, if, else, elif e il comando test

Ci avviciniamo alla parte vera e propria della programmazione, iniziamo con le istruzioni condizionali, la più usata è sicuramente “if”, tradotta “se”.

Usando dello pseudocodice (un codice che non è codice ma spiega in linguaggio più umano quello che poi dovrà fare il programma), possiamo scrivere il seguente programma:

se esiste il file miofile.txt allora

scriverò “il file esiste”

altrimenti

creerò il file miofile.txt

fine istruzioni

tradotto in scripting:

```
if [ -f /home/utente/miofile.txt ]; then
```

```
    echo Il file esiste
```

```
else
```

```
    touch /home/utente/miofile.txt
```

```
fi
```

in questo caso vediamo delle parentesi quadre, ma a che servono?

Sono un alias (un equivalente) del comando “test”, normalmente si usa la parentesi quadra, anche se volendo si potrebbe usare direttamente “test” o la doppia parentesi quadra “[[condizione...]]”, quest'ultima modalità ha dei particolari che non sono argomento di questo corso, limitiamoci perciò alla parentesi quadra singola.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>



Introduzione allo scripting Bash

“test” può fare un confronto tra due variabili numeriche o tra due stringhe, tramite gli operatori logici che vedremo nel prossimo capitolo, oppure fa dei controlli sui file, come nell'esempio, in cui verifica se il file esiste, ma si potrebbe verificare se è una directory, se è un eseguibile, se è vuoto o se non è vuoto, e tante altre cose, perciò date una bella occhiata al man, perchè “test” è fondamentale nello scripting!

“else” significa “se il risultato del codice precedente da risultati negativi allora esegui il seguente codice”.

“fi” (il contrario di “if”) deve esserci alla fine di un costrutto “if”.

C'è anche una terza possibilità oltre a “if” e “else”, “elif”, nel caso si vogliano fare dei test concatenati (attenzione: non però annidati), esempio:

```
miofile="/home/utente/miofile.txt"
```

```
tuofile="/home/utente/tuofile.txt"
```

```
if [ -f $miofile ]; then  
    echo Il mio file esiste  
elif [ -f $tuofile ]; then  
    echo Il tuo file esiste  
else  
    echo non esiste nessun file
```

```
fi
```

in questo caso abbiamo incluso i percorsi dei file nelle variabili “miofile” e “tuofile” (ricordiamoci che abbiamo incluso non il vero percorso ma delle stringhe che lo descrivono, c'è una differenza! ;)), dopodichè abbiamo verificato se esiste “/home/utente/miofile.txt”, se esiste esegue l'istruzione “echo Il mio file esiste” ed esce dal costrutto, in caso contrario passa al test successivo “elif [-f \$tuofile]; then” e se da esito positivo eseguirà “echo Il tuo file esiste”, nel caso però che nemmeno questo test dia esito positivo eseguirà l'istruzione dopo “else”.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Preciso che non è obbligatorio usare “else” in coda ad un'istruzione “if” o “if/elif”.

Gli “if” e i suoi derivati possono essere annidati, seguendo il precedente esempio potremmo scrivere:

```
miofile="/home/utente/miofile.txt"
tuofile="/home/utente/tuofile.txt"
if [ -d /home/utente ]; then
    if [ -f $miofile ]; then
        echo Il mio file esiste
    elif [ -f $tuofile ]; then
        echo Il tuo file esiste
    else
        echo non esiste nessun file
    fi
else
    echo non esiste la directory /home/utente
fi
```

Penso sia abbastanza chiaro...



Mezzora
d'amicizia

<NOME DELLA MEZZORA>



Introduzione allo scripting Bash

Case

Un costrutto “case” possiamo immaginarlo come una lunga serie di “if/elif”, un esempio molto semplice per visualizzarne la funzione è visualizzare uno dei tanti script di avvio del nostro sistema, ad esempio se io ne prendo uno a caso nella mia Slackware, “/etc/rc.d/rc.hald”, in fondo al file leggerò:

```
# See how we were called.  
case "$1" in  
  start)  
    hal_start  
    ::  
  stop)  
    hal_stop  
    ::  
  restart)  
    hal_stop  
    sleep 1  
    hal_start  
    ::  
  *)  
    echo $"Usage: $0 {start|stop|restart}"  
    ::  
esac
```



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Analizzando il codice possiamo vedere che il primo è un commento per spiegare cosa fa il costrutto, nella seconda linea troviamo:

case "\$1" in

possiamo notare che "\$1" rappresenta il primo argomento passato allo script, perciò "case" va a valutare il contenuto di questa variabile, ovvero

case + il valore da analizzare + in

Successivamente ci sono vari blocchi che iniziano con una stringa delimitata da una parentesi tonda chiusa, di seguito c'è un blocco di istruzioni, infine un doppio punto e virgola a chiudere tutto il blocco, tutto questo è molto simile ai blocchi "if/elif" visti prima, la stringa prima della parentesi tonda viene confrontata con il valore da analizzare, nel caso coincida esegue il blocco di istruzioni e, incontrando il doppio punto e virgola, esce, nel caso non ci fosse stato il doppio punto e virgola avrebbe continuato nei confronti con la parte successiva del costrutto (in questo caso una cosa inutile!!!).

Notare che nei confronti spesso vengono usati caratteri jolly e globbing.

Nella parte finale possiamo notare che prima della parentesi tonda c'è un asterisco, questo è l'equivalente di "else" nei costrutti "if", esauriti i confronti precedenti senza successo verranno eseguite le istruzioni dopo "*").

Come possiamo vedere il costrutto finisce con la parola "case" al contrario", "esac", come "if" finiva con "fi" (che fantasia, eh?! :D).



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Esercizi: analizzate alcuni file di sistema (**senza modificarli, mi raccomando!!!**) e cercate di interpretare quello che viene fatto con i costrutti “if” e “case”, successivamente create almeno 6 script, 3 per “if” e 3 per “case”, possibilmente usando in “case” caratteri jolly e globbing.

Decifrate il codice che segue “*)” nell'esempio e spiegate meglio perchè l'asterisco...



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Eeguire gli script

Tra un po cominceremo a creare i primi script (se non avete già iniziato), le strade per eseguire uno script sono principalmente due, renderlo eseguibile, con un “`chmod +x nomescript`”, o eseguirlo con:

```
$ sh ./nomescript
```

o

```
$ bash ./nomescript
```

se invece vogliamo renderlo permanente nel sistema dovremo installarlo in una delle directory incluse nei percorsi in “PATH” (usare “`echo $PATH`”), tipicamente però sono “`/usr/bin`” o “`/usr/local/bin`”.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Operatori logici

Spesso dobbiamo fare più verifiche o confronti contemporaneamente, immaginatevi di dover verificare l'esistenza di due file, se esistono tutti e due ne copiamo il contenuto in un terzo file:

```
primofile=/percorso/primofile
secondofile=/percorso/secondofile
filedestinazione=/percorso/filedestinazione
if [ -f $primofile ] && [ -f $secondofile ]; then
    cat $primofile $secondofile >> $filedestinazione
fi
```

in questo caso tutte e due le condizioni devono essere vere, questo è l'operatore "AND".

Il prossimo è invece l'operatore "OR", un esempio:

```
mionome="gigi"
miaetà=34 # ops, forse ho invertito i numeri...
if [ "$mionome" == "gigi" ] || [ $miaetà -ne 34 ]; then
    echo "Potrei essere io, almeno o il nome o l'età corrispondono!!!"
fi
```

in questo caso basta che sia vera solo una delle due per eseguire le istruzioni (infatti è vera solo la prima per lo script! Beh, a dire il vero anche la seconda, non per lo script ma per la realtà!!! :D).



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Esiste poi un operatore “NOT”, si scrive col punto esclamativo, serve per ribaltare la condizione, esempio:

```
if [ ! -f /percorso/miofile ]; then
```

.....

in questo caso esegue la e le istruzioni soltanto se “/percorso/miofile” NON esiste, capito il funzionamento?

Esercizi: provate a integrare i 3 script con “if“ che avete creato prima con istruzioni correlate da questi operatori.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Sostituzione di comandi

Negli script potete usare i comandi normalmente, semplicemente scrivendoli come fate alla linea di comando, però in molti casi sarà utile passare l'output ad una variabile per poi lavorarci sopra, in questo caso si usano gli apici inversi “`” (si nota tra le virgolette?), ottenibili normalmente sulla tastiera con il layout italiano con AltGr+', in poche parole, se ad esempio devo passare alla variabile “a” l'output di un “ls -lah” della home scriverò:

```
a=`ls -lha ~`
```

semplice, no?

Esercizi: integrate gli script che avete creato con variabili che prendono i valori da output di comandi, se volete rendere la cosa più difficile racchiudete comandi concatenati da dei pipe (|).



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Operatori aritmetici

Per eseguire operazioni aritmetiche nello scripting Bash gli operatori sono le doppie parentesi tonde, se ad esempio vogliamo includere nella variabile “n” il risultato di un'operazione, scriveremo:

```
n=$((2 + 2))
```

gli operatori sono “+” per l'addizione, “-” per la sottrazione, “/” per la divisione, “*” per la moltiplicazione, “%” per il resto di una divisione, se invece vogliamo ad esempio incrementare una variabile numerica già esistente possiamo usare il seguente metodo:

```
n=10
```

```
echo $n # darà 10
```

```
n=$((n + 5))
```

```
echo $n # darà 15
```

chiaro?

Esercizi: create degli script o usate la linea di comando per fare alcuni esperimenti di calcoli fatti con questi operatori e per stampare il risultato a schermo, negli script descrivete con brevi commenti ciò che fate.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



I cicli, for, while e until

Iniziamo da “for, con del pseudocodice e traducendo “for in “per”:

per argomento nella lista

esegui

istruzione sul primo argomento della lista

istruzione sul secondo argomento della lista

.....

fatto

esempio di codice:

```
for i in a b c d e
```

```
do
```

```
    echo $i
```

```
done
```

“i” è la variabile che conterrà l'argomento della lista, “a b c d e” è la lista, l'istruzione visualizzerà l'argomento.

Si può anche includere la lista in una variabile, oppure usare direttamente una sostituzione di comando, un paio di esempi:

```
lista="mele pere banane cetrioli"
```

```
for i in "$lista"
```

```
do
```

```
    echo "Oggi mangio $i"
```

```
done
```

oppure



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



```
for i in `ls *.jpg ~`  
do  
    echo $i  
done
```

stamperà tutti i file eventualmente trovati che finiscono in “.jpg” presenti nella nostra home, in quest'ultimo esempio il ciclo “for” servirebbe relativamente a poco, bastava “ls”, però era solo per far capire le potenzialità di tale comando.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>



Introduzione allo scripting Bash

Il ciclo “while”, questo ciclo in pseudocodice potrebbe essere riassunto nel seguente modo:

fino a che sussiste la condizione

esegui

istruzione 1

istruzione 2

fatto

un esempio:

```
n=1
```

```
while [ $n -le 10 ]
```

```
do
```

```
    echo $n
```

```
    n=$((n + 1))
```

```
done
```

stamperà a schermo

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

sino a che la condizione era vera, ovvero “n” era inferiore o uguale a 10, ha eseguito il ciclo, poi è uscito.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Una nota: nell'esempio dell'uso dello scripting dalla linea di comando uso ":" dopo "while" per ottenere un ciclo infinito, tale operatore equivale a "true", e da una condizione sempre vera, per la cronaca esiste anche "false", si possono poi ribaltare le condizioni con l'operatore "!".

"until" è il contrario di "while", esegue il ciclo se la condizione NON è vera.

Per uscire da un ciclo si può usare l'istruzione "break", mentre per saltare un ciclo facendo però continuare i cicli successivi si può usare "continue", potete cercare nel manuale di Bash l'uso specifico, che poi è comune a molti altri linguaggi, C compreso.

Esercizi: integrate ulteriormente i vostri script (o createne di nuovi) con dei cicli "for" e "while", in più provate ad usare da linea di comando questi cicli per abbreviare eventuali compiti lunghi e ripetitivi che svolgete sul vostro PC.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Exit code

Gli “exit code” costituiscono un numero restituito all'uscita di un programma o di una funzione, per uscire da una shell normalmente usiamo “exit”, in uno script possiamo aggiungere un numero che identifica uno stato di uscita, senza complicare troppo le cose i più usati sono “0” per “successo” e “1” per “fallito” (ma attenzione: qualsiasi risultato diverso da “0” è un fallimento), questo codice è contenuto in una variabile, “?”, che si visualizza appunto con un “echo \$?” subito dopo l'uscita del comando.

Se ad esempio digito un “ls” in una directory vuota o su di un file che non esiste, mi restituirà “1” o un numero diverso da “0”, mentre se non è vuota o il file esiste restituirà “0”.

Questo ci da modo di far interagire in modo interessante quanto sino ad ora imparato, un esempio:

```
ls *.jpg ~ &>/dev/null
if [ $? -ne 0 ]; then
    echo "Nella mia directory home non ci sono file jpg"
    exit 1
else
    echo "Nella mia directory home ci sono file jpg"
    exit 0
fi
```



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



spiegando il codice, un “ls” per verificare se ci sono file con estensione “.jpg” nella mia home non stamperà ne output e ne eventuali messaggi di errore a schermo in quanto redirigo tutti e due questi flussi su “/dev/null”, questo perchè non mi interessano, in realtà mi interessa soltanto sapere se ci sono o non ci sono quei file, perciò l'unica cosa che mi interessa è l'exit code del comando, exit code valutato poi nel costrutto “if”! (che a sua volta restituirà degli exit code)

Una nota: l'equivalente di “exit” per una funzione è “return”, si vedrà nel prossimo capitolo...

Esercizi: fate un po di esperimenti a linea di comando con comandi che falliscono o hanno successo, visualizzate subito dopo l'exit code con un “echo \$?”, in più vedete se riuscite a integrare ulteriormente i vostri script con queste nuove nozioni.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Le funzioni

In tutti i linguaggi che si rispettano si usano le funzioni, questi sono come dei piccoli programmi all'interno del programma principale, in alcuni casi esse potranno addirittura essere scritte in un file a parte e incluse nel nostro script tramite l'operatore “.” o “source” (che è la stessa cosa), un esempio:

`./percorso/file-con-le-funzioni`

comunque, le funzioni contribuiscono a tenere più ordinato il codice, a riusare parti di codice più volte, e a fare modifiche con più facilità.

La sintassi è:

```
function nome_funzione() {  
    istruzioni  
    return 0  
}
```

oppure senza “function”, ovvero

```
nome_funzione() {  
    istruzioni  
    return 0  
}
```



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



l'utilizzo poi della funzione avviene riportandone il nome senza le parentesi tonde in una parte successiva a dove è stata scritta (se invertite questo ordine per la shell la funzione non esisterà), ad esempio:

`nome_funzione`

Avrete notato il comando “return” alla fine della funzione seguito da un numero, questo non è ne più e ne meno dell'equivalente del comando “exit” con l'exit code che si usa nello script con “exit”, anche perchè se usato “exit” nella funzione provocherebbe l'uscita non dalla sola funzione, ma da tutto lo script!

Un esempio pratico:



Mezzora
d'amicizia

<NOME DELLA MEZZORA>



Introduzione allo scripting Bash

```
funzione_avvertimento() {  
    read -p "Hai sbagliato, te ne sei reso conto(si/no)? " risp  
    if [ "$risp" == "no" ]; then  
        echo "Bravo, stai più attento!"  
    else  
        echo "Beh, almeno lo hai riconosciuto..."  
    fi  
    return 0  
}
```

```
read -p "Quanto fa 2 + 2? " risp  
if [ $risp -ne 4 ]; then  
    funzione_avvertimento  
else  
    echo "Bravo"  
fi
```

```
read -p "Qual'è la nostra capitale? " risp  
if [ "$risp" != "Roma" ]; then  
    funzione_avvertimento  
else  
    echo "Bravo"  
fi  
exit 0
```



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



in realtà questo è un esempio un po' stupido, come le domande poste (!!!), però da un esempio dell'uso di una funzione, in realtà anche il corpo del programma si prestava a essere incluso in una funzione, potreste spiegarne il perchè come **esercizio!** ;)

Un'altra cosa che si nota è che la variabile "risp" viene sovrascritta ogni volta e con tipi di dati differenti, in più quella che troviamo all'interno della funzione non è la stessa presente fuori in quanto la prima è locale alla stessa funzione.

Se dovessimo passare dei parametri alla funzione varrebbero i parametri posizionali già spiegati sopra, come se fosse uno script a se stante (e da un certo punto di vista è così!).

Esercizi: a questo punto, se avete evoluto mano a mano i vostri script degli esercizi, dovrete avere degli script con un minimo di complessità, spero abbiate aggiunto anche molti commenti per comprendere meglio il loro funzionamento, a questo punto provate a vedere se riuscite a ordinare parte del codice usando delle funzioni.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Debug

Come chiunque programmi difficilmente avremo il dono di produrre immediatamente programmi senza errori, anche in questo Bash ci viene incontro, con un'opzione, “-x”, un esempio:

```
$ sh -x ./nomescript
```

o

```
$ bash -x ./nomescript
```

vedremo tutte le istruzioni eseguite nello script una dopo l'altra, precedute dal segno “+”, verificando dove eventualmente lo script si è bloccato o ha avuto problemi.

C'è anche il modo di isolare alcune parti e includerne altre in questa visualizzazione, ma al momento direi che quanto scritto può bastare!

Esercizi: se avete creato gli script degli argomenti precedenti, visualizzatene il funzionamento con l'opzione “-x” descritta in questo capitolo.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Conclusioni

Molti argomenti introdotti con questo mini-corso sarebbero da approfondire, in più ce ne sarebbero molti altri, ad esempio volutamente non ho citato gli array, ho rimandato alcuni argomenti alla lettura della documentazione, come ad esempio i comandi “break” e “continue”, o “test”, o altri, non ho spiegato la differenza tra i comandi builtin e quelli esterni, questo anche perchè già non so se sto dentro nella mezzora canonica concessa per questi tutorial, però un buon manuale su Bash che ho davanti comprende circa 180 pagine, contro la ventina che occorrerebbero per includere questo (ne vedrete più di 40 nella presentazione, ma le pagine sono state divise per farle stare correttamente nelle pagine di Impress).

Gli esercizi sono abbastanza approssimativi, più che altro servono per stimolare l'apprendimento di argomenti essenziali per quelli successivi, se richiesto integrerò con esercizi più specifici.

A parte tutto, dopo questo corso dovrete essere già in grado di usare lo scripting dalla linea di comando, fare script per molte delle vostre esigenze e capire quelli presenti nel sistema, cose che si proponevano come obiettivi di questo corso.



Mezzora
d'amicizia

<NOME DELLA MEZZORA>

Introduzione allo scripting Bash



Rimando alla lettura di documenti molto interessanti, a parte i man citati (“bash” tra tutti), in italiano potete trovare un'ottima guida al seguente indirizzo:

<http://www.pluto.it/files/ildp/guide/abs/index.html>

traduzione di:

<http://tldp.org/LDP/abs/html/>

oppure cercare nei sempre validi Appunti di Informatica Libera di Daniele Giacomini, al seguente indirizzo:

<http://a2.pluto.it/>

Se poi volete qualcosa sul cartaceo posso consigliare una guida che a me è piaciuta molto:

“Bash e la Shell di Linux”, di Giorgio Zarelli, edito da J. Book (<http://www.jbook.it>).

Buona programmazione e salutonì da

Gigi, conosciuto nel Linuxvar anche come “wolf”, o wolf, conosciuto nel Linuxvar anche come Gigi... :P :D :)