

# Indice

<b>1</b>	<b>Intro</b>	<b>3</b>
1.1	Setup . . . . .	3
1.2	Utilities . . . . .	4
1.3	Caratteristiche base . . . . .	5
1.4	Stream e redirectione . . . . .	5
1.4.1	Pipe . . . . .	6
1.4.2	Redirezione . . . . .	6
1.5	Esecuzione di comandi . . . . .	7
1.5.1	Lista di comandi . . . . .	7
1.5.2	Terminatori di riga . . . . .	7
1.5.3	Esecuzione in background . . . . .	7
1.6	Here documents . . . . .	8
1.7	Scripting . . . . .	8
1.7.1	Incipit . . . . .	8
1.7.2	Parametri di posizione . . . . .	8
1.7.3	Valore di ritorno . . . . .	8
<b>2</b>	<b>Variabili</b>	<b>9</b>
2.1	Creazione . . . . .	9
2.2	Listing . . . . .	10
2.3	Utilizzo . . . . .	10
2.4	Rimozione . . . . .	11
<b>3</b>	<b>Control flow</b>	<b>13</b>
3.1	if/test . . . . .	13
3.1.1	test . . . . .	14
3.1.2	[ vs [[ . . . . .	15
3.2	case (switch) . . . . .	16
3.3	for . . . . .	16
3.4	while . . . . .	18
3.5	break e continue . . . . .	18
<b>4</b>	<b>Funzioni</b>	<b>19</b>
4.1	Definizione . . . . .	19
4.2	Chiamata . . . . .	19



# Capitolo 1

## Intro

`man bash` non fa schifo, davvero.

### 1.1 Setup

Alcuni file generali sulle shell:

- `/etc/shells` lista tutte le shell conosciute dal sistema
- la shell di login di ogni utente è memorizzata in `/etc/passwd`
- per impostar la propria shell utilizzare `chsh` (anche se non si è root)

```
chsh -s /usr/bin/zsh
```

**Login shell** La shell di login è il *primo processo che viene eseguito dall'utente*; si ha quando

- ci si logga per la prima volta in locale (o mediante ssh)
- ci si logga con `su -` (o `sudo -`)

La shell di login imposta variabili che saranno disponibili ai processi figli. Esegue nell'ordine (se esistono)

```
/etc/profile
~/.bash_profile
~/.bash_login
~/.profile
```

All'uscita, se esiste, esegue `~/.bash_logout`

**Non-login shell** Quella che tipicamente si ottiene aprendo un terminale: esegue nell'ordine le configurazioni contenute in

```
/etc/bash.bashrc
~/.bashrc
```

**Uniformità** Spesso però al fine di avere **uniformità** tra login shell e non-login shell in `.profile` si fa il source di `.bashrc` e si modifica quest'ultimo. Questo anche perché non è detto che il `.profile` sia valutato (es login/display manager grafico)

**Template di configurazione** Si trovano in `/etc/skel/`

**Bash setup** Si impostano opzioni con

- `set` (vedere `help set` per le opzioni disponibili). Listing delle opzioni dove `+` indica l'impostazione *disattivata*, `-` l'impostazione attiva

```
set +o
```

Ad una rapida occhiata non vi sono cose che mi hanno fatto cadere dalla sedia

- `shopt` (vedere `man bash` e cercare `assoc_expand_once` la prima opzione)

```
shopt
```

Qui invece ci sono cose interessanti, da impostare con `-s` o togliere con `-u`,

## 1.2 Utilities

**History** Si ha che per

- interrogarla, usare `history`
- rieseguire un comando `!numero`
- rilanciare l'ultimo comando `!!`
- rilanciare l'ultimo comando che inizia con una stringa `!apt`
- riutilizzare un parametro dell'ultimo comando

```
mkdir /tmp/test
cd !$
```

- per effettuare una ricerca
  - `Ctrl+R`,
  - idare `Ctrl+R` per scorrere le altre che matchano fino a trovare la desiderata
  - invio per eseguire il comando scelto oppure `Esc` per modificare la ricerca usandola come punto di partenza

**Alias** Per

```
alias dataset="echo dataset_$(date +%Y_%m_%d)" # definizione
unalias dataset                                # rimozione
```

## 1.3 Caratteristiche base

### Caratteri speciali e quoting

```
#          commento
`comando`  sostituisce l'output dato comando nella linea di bash
$(comando) sostituisce l'output dato comando nella linea di bash
$variabile sostituisce la variabile nella linea di bash
\          andare a capo senza interrompere il comando
\          il carattere che lo segue viene preso alla lettera
*          uno o più caratteri qualsiasi in un nome di un file/percorso
?          un carattere qualsiasi
[]         uno qualsiasi dei caratteri inclusi tra parentesi
```

Il quoting

- con apici singoli serve per stringhe che si vogliono tenere assieme in presenza di spazi (non effettua sostituzioni di variabili ed è il massimo del safe)

```
echo 'ciao $USER'
echo "ciao $USER"

## ciao $USER
## ciao l
```

- con apici doppi effettua la sostituzione

**Caricare altri file** Ipotezzando di aver definito variabili o funzioni in un file, per caricarne i valori e poterle utilizzare si comanda alternativamente:

```
source file_path
. file_path
```

## 1.4 Stream e redirezione

Lo standard

- input **stdin** è l'input fornito dall'utente (tastiera)
- output **stdout** è l'output di un certo programma
- error **stderr** è l'output di errore eventualmente derivante dall'esecuzione di un comando che non va a buon fine. Stampato a video come lo standard output non è bufferizzato.

### 1.4.1 Pipe

Il pipe reindirige lo stdout di ‘comando1’ allo stdin di ‘comando2’:

```
comando1 | comando2
```

Alcuni comandi comandi utili nei pipe seguono

**tee** Manda a standard output e salva su file video il risultato ricevuto da stdin

```
ls -l | tee unsorted.txt | sort -r -k9 | tee sorted-rev.txt
```

Il comando sopra salverà in unsorted e sorted i contenuti ordinati o meno per nome del file (stamperne il contenuto a video dell’ultimo, non ulteriormente processato)

**xargs** Prende dallo standard input i valori passati come se fossero parametri di un comando ed esegue un comando specificato (di default echo) con i parametri passati.

Di default effettua l’echo

```
echo {1..9} | xargs          # di default fa l'echo di quello che è passato
echo {1..9} | xargs -n 3     # gestisci 3 comandi alla volta
ls /home/l/cs/*.pdf | xargs zip /tmp/pdf.zip # zippa i pdf

## 1 2 3 4 5 6 7 8 9
## 1 2 3
## 4 5 6
## 7 8 9
## updating: home/l/cs/bash.pdf (deflated 3%)
## updating: home/l/cs/c.pdf (deflated 3%)
## updating: home/l/cs/python.pdf (deflated 8%)
## updating: home/l/cs/r.pdf (deflated 7%)
## updating: home/l/cs/shell.pdf (deflated 3%)
## updating: home/l/cs/test_py.pdf (deflated 4%)
```

### 1.4.2 Redirezione

È possibile gestire input e output in modo da farli provenire e finire in diverse direzioni

```
<      prende stdin da file
<<<   prende stdin da stringa
>      stdout su file, sovrascrivendolo
>>    stdout su file, appending
2>     stderr su file, sovrascrivendo
2>>    stderr su file, appending
&>     sia stdout che stderr su file, sovrascrivendolo
&>>    sia stdout che stderr su file, appending
```

Esempi

```
comando < file_input > file_risultati 2> file_errori
comando > file_risultati 2> /dev/null
```

Per eseguire un comando alla volta prendendo da input si può fare (invece di `echo testo1 testo2 testo3 | xargs -n 1`)

```
xargs -n1 <<< "testo1 testo2 testo3"

## testo1
## testo2
## testo3
```

## 1.5 Esecuzione di comandi

### 1.5.1 Lista di comandi

```
comando1 ; comando2 ; comando3
comando1 && comando2 && comando3
comando1 || comando2 || comando3
```

Il

- primo esegue tutti i comandi in sequenza
- secondo esegue ogni comando in sequenza se il precedente non è fallito (l'ultimo ad essere eseguito è il comando che restituisce non zero);
- terzo esegue ogni comando a turno indipendentemente dall'esito del precedente; il primo che termina correttamente (ritorna un valore pari a 0) termina l'esecuzione della catena.

L'exit status in entrambi i casi è l'exit status dell'ultimo comando eseguito.

### 1.5.2 Terminatori di riga

```
\      spezza un comando e prosegue sulla linea successiva
;      separa dal comando successivo sulla stessa linea
&      esegue il comando in background in una sottoshell
```

### 1.5.3 Esecuzione in background

Se un processo sta occupando la shell e ne abbiamo bisogno momentaneamente:

- dare `Ctrl+Z` per porre il processo in background
- utilizzare la shell per altro
- comandare `bg 1` per rimettere a schermo il processo posto in background

Per la lista dei processi in esecuzione in background comandare `'jobs'` (dal quale si ricava l'id numerico diverso da 1, nel caso di molteplici processi in background)

## 1.6 Here documents

Sono un modo di programmare la shell per ottenere un effetto simile a

```
programma < file_di_comandi
```

dove a un programma (es ftp) viene passato in stdin una serie di comandi da effettuare che sono contenuti in un file.

Il fatto è che prima bisogna definire tali comandi in apposito file e il funzionamento dello script dipende dall'esistenza e la correttezza di tale file. Mediante l'here document si specificano i comandi all'interno della sintassi di bash e poi si passano gli stessi al comando specificato. La sintassi è

```
programma << EOF
comando1
comando2
...
EOF
```

Al posto di EOF ci può essere qualsiasi cosa: i due delimitatori (EOF in questo caso) conterranno i comandi che saranno passati al programma.

Come nelle altre parti degli script vi è sostituzione delle variabili con il loro contenuto.

## 1.7 Scripting

### 1.7.1 Incipit

Uno script bash inizia con

```
#!/bin/bash
#!/usr/bin/env bash  # piu portabile
```

### 1.7.2 Parametri di posizione

Sono variabili speciali creati/associati all'esecuzione di uno script

- \$0 è il nome con cui è stato chiamato lo script
- \$1 è il primo parametro, \$2 il secondo

### 1.7.3 Valore di ritorno

Similmente alle funzioni gli script possono ritornare al chiamante mediante `exit`

- tipicamente 0 se comando/funzione eseguiti con successo
- 1 o maggiori in caso di errore



# Capitolo 2

## Variabili

### 2.1 Creazione

Sono

- definite con l'uguale senza spazi (in presenza di spazi l'istruzione viene interpretata come un comando)

```
identificatore=valore
```

- tutte stringhe. Non vi è differenza, a livello di tipo tra le seguenti assegnazioni

```
my_message="Hello World"  
my_short_message=hi  
my_number=1  
my_pi=3.142  
my_other_PI="3.142"  
my_mixed=123abc
```

routine che aspettano numeri tratteranno le cose fornite come tali

- categorizzabili in *variabili locali* e *d'ambiente*:
  - le *variabili locali* sono disponibili solo nella shell attuale e si creano semplicemente mediante
  - le *variabili d'ambiente* vengono rese disponibili anche alle shell figlie di quella dove sono definite e vengono definite alternativamente come

```
identificatore=contenuto  
export identificatore  
## oppure  
export identificatore=contenuto
```

Come nomi è buona norma creare identificatori minuscoli (i maiuscoli sono utilizzati dalle variabili d'ambiente impostate dal sistema)

**Prendere input dall'utente** Utilizzare `read`

```
echo Please, enter your name
read NAME
echo "Hi $NAME!"
```

per prendere più input nella stessa linea

```
echo Please, enter your firstname and lastname
read FN LN
echo "Hi! $LN, $FN !"
```

## 2.2 Listing

Dando due TAB dopo

```
echo $
```

vengono listate tutte le variabili attualmente disponibili.  
Con *printenv* le variabili d'ambiente

## 2.3 Utilizzo

La sostituzione viene effettuata mediante `$identificatore` nel comando desiderato

```
var="foo" # creazione senza spazi se no sembra un comando
echo "$var" # sostituzione
echo '$var' # niente sostituzione
echo ${#var} # lunghezza della stringa
```

```
## foo
## $var
## 3
```

```
# esecuzione di comandi
cmd=ls
eval "$cmd"
```

```
## control_flow.Rnw
## funzioni.Rnw
## intro.Rnw
## TODO
## variabili.Rnw
```

```
var1=asd
var2="asd"
if [[ $var1 == $var2 ]] ; then echo "si sono uguali"; fi

## si sono uguali
```

### stringhe

```
# numeri
# -----
var1=1
var2=2
var3=$(( var2 * 3 + var1 )) # interpreta come numerico
echo $var3

## 7
```

### numeri

## 2.4 Rimozione

Per eliminare una variabile

```
unset identificatore
```



## Capitolo 3

# Control flow

### 3.1 if/test

Formalmente la struttura di un generico `**if**`:

```
if COMANDI; then COMANDI; [ elif COMANDI; then COMANDI; ]... [ else COMANDI; ] fi
```

Spesso si trova riadattato come

```
if [[ condizione ]]
then
    comandi
elif [[ condizione2 ]]
then
    comandi
else
    comandi
fi
```

dove:

- è importante che dentro parentesi quadre ci sia uno spazio bianco all’inizio e uno alla fine;
- di fatto bash sostituisce `[[ condizione ]]` al comando `test condizione`; spesso è quindi la builtin `test` che effettua le comparazioni quindi rifarsi al suo help per vedere i test disponibili;
- In generale `[[` è la versione migliorata di bash del generico `[` ed è più sicuro ma meno portabile. Entrambi si basano sull’utility `test` (`man test`)
- non si pone punto e virgola dopo comandi posti sulla stessa riga devono essere separati dal punto e virgola.
- non si pone punto e virgola dopo comandi dato che l’`elif`, l’`else` e il `fi` sono su un’altra
- occhio che il `then` segue solo `if` ed `elif`

Ad esempio

```
if [[ `hostname` == "ambrogio" ]]
then
    echo "siamo su ambrogio"
    echo "seconda istruzione su ambrogio"
elif [[ `hostname` == "anacleto" ]]
then
    echo "siamo su anacleto"
    echo "seconda su anacleto"
else
    echo "non so dove siamo"
fi

## siamo su ambrogio
## seconda istruzione su ambrogio
```

È possibile nidificare gli if, ad esempio

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

### 3.1.1 test

# booleani	
# -----	
( EXPRESSION )	EXPRESSION is true
! EXPRESSION	EXPRESSION is false
EXPRESSION1 -a EXPRESSION2	both EXPRESSION1 and EXPRESSION2 are true
EXPRESSION1 -o EXPRESSION2	either EXPRESSION1 or EXPRESSION2 is true
# operatori per stringhe	
# -----	
-n STRING	the length of STRING is nonzero
STRING	the length of STRING is nonzero
-z STRING	the length of STRING is zero
STRING1 = STRING2	stringhe uguali
STRING1 != STRING2	stringhe diverse
STRING1 < STRING2	stringa1 è inferiore nel locale corrente
STRING1 > STRING2	stringa1 è superiore nel locale corrente

```
# operatori per interi
# -----
INTEGER1 -eq INTEGER2      INTEGER1 is equal to INTEGER2
INTEGER1 -ge INTEGER2      INTEGER1 is greater than or equal to INTEGER2
INTEGER1 -gt INTEGER2      INTEGER1 is greater than INTEGER2
INTEGER1 -le INTEGER2      INTEGER1 is less than or equal to INTEGER2
INTEGER1 -lt INTEGER2      INTEGER1 is less than INTEGER2
INTEGER1 -ne INTEGER2      INTEGER1 is not equal to INTEGER2

# operatori per file (alcuni)
# -----
-d FILE                     FILE exists and is a directory
-f FILE                     FILE exists and is a regular file
-L FILE                     FILE exists and is a symbolic link (same as -h)
-x FILE
```

In base all'operatore di fatto le variabili sono considerate numeri o stringhe (ad esempio nel seguito usiamo eq che si attende

```
if [[ 1 -eq 1 ]]; then echo "sono uguali"; fi
if [[ "1" -eq 1 ]]; then echo "sono uguali"; fi
if [[ "1" -eq "1" ]]; then echo "sono uguali"; fi

## sono uguali
## sono uguali
## sono uguali
```

### 3.1.2 [ vs [[

[[ è la versione pimpata da bash di [: in generale se si ha meno esperienza con la shell è spesso l'opzione più sicura. Alcune feature

- si può evitare di porre le stringhe tra indici per avere la sostituzione

```
if [ -f "$file" ]    # sh
if [[ -f $file ]]    # bash
```

- si possono utilizzare gli operatori booleani && o || per i test
- ha un operatore per matchare espressioni regolari
- ha il globbing incorporato ad esempio si può scrivere

```
if [[ $ANSWER = y* ]]
```

**Esecuzione di comandi condizionale** Tra parentesi si pone il test oppure un comando (di cui viene utilizzato il risultato)

```
# esecuzione di un comando se il test fallisce (usando ||)
# [[ `cat /tmp/failing_command` ]] || notify-send -t 1500 "command failed"

# esecuzione di un comando se il test ha successo (usando &&)
[[ -f ~/.bashrc ]] && echo "il file ~/.bashrc esiste"

# esecuzione di un comando in un caso o nell'altro
[[ -f /tmp/asd ]] && echo "file esiste" || echo "file non esiste"

## il file ~/.bashrc esiste
## file non esiste
```

## 3.2 case (switch)

Il `case` è simile allo `switch` del C; formalmente

```
case PAROLA in [MODELLO [| MODELLO]...) COMANDI ;;]... esac
```

Esegue in modo selettivo COMANDI basati sulla PAROLA corrispondente al MODELLO e restituisce lo stato dell'ultimo comando eseguito. Spesso viene riparafrasato come segue

```
case "$variable" in
    "$var1") command... ;;
    "$var2") command... ;;
    *) comando default ;;
esac
```

dove se si inseriscono variabili come nel caso è necessario quotare perché non viene effettuata la sostituzione di default.

Un esempio

```
site=asd

case $site in
    google) exec echo "si è scelto google" ;;
    amazon) exec echo "si è scelto amazon" ;;
    *) exec echo "si è scelto altro?" ;;
esac

## si è scelto altro?
```

## 3.3 for

Il `for` è costruito così

```
for elem in elenco; do comando con elem; done
```



spesso riarrangiato come

```
for elem in lista
do
    comandi con $elem
done
```

Alcuni esempi seguono:

- spesso si usa seq per generare indici numerici

```
sum=0
for i in $(seq 1 100)
do
    sum=$((sum + i))
done

echo "Sum is: $sum"

## Sum is: 5050
```

- Elementi possono essere la qualunque (qui hello, 1, nomi file in directory corrente etc)

```
for i in hello 1 * 2 goodbye
do
    echo "Looping: i is set to $i"
done

## Looping: i is set to hello
## Looping: i is set to 1
## Looping: i is set to control_flow.Rnw
## Looping: i is set to funzioni.Rnw
## Looping: i is set to intro.Rnw
## Looping: i is set to TODO
## Looping: i is set to variabili.Rnw
## Looping: i is set to 2
## Looping: i is set to goodbye
```

```
for f in ~/.bashrc ~/.profile ~/.conky
do
    [[ -f $f ]] && echo "file $f esiste" || echo "file $f non esiste"
done

## file /home/l/.bashrc esiste
## file /home/l/.profile esiste
## file /home/l/.conky non esiste
```

### 3.4 while

La sintassi del while è

```
while COMMANDS; do COMMANDS-2; done
```

riarrangiabile a

```
while COMMANDS # eg [[ condizione ]]  
do  
    comandi  
done
```

ed esegue comandi fino a che COMMANDS ha successo

```
countdown=10  
while [[ $countdown > 0 ]]  
do  
    echo $countdown  
    countdown=$((countdown - 1))  
done
```

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

### 3.5 break e continue

**break** e **continue** funzionano come di consueto: il primo fa uscire dal loop (**for** o **while**), il secondo fa saltare all'iterazione successiva.

## Capitolo 4

# Funzioni

### 4.1 Definizione

Vi sono due sintassi

```
# sintassi 1
function function_name {
    comandi
}
```

```
# sintassi 2 - più portabile
function_name () {
    comandi
}
```

In entrambi i casi la seconda graffa deve esser posta su una riga a se:

- eventuali argomenti sono disponibili all'interno attraverso i parametri di posizione `$1`, `$2`
- la restituzione di valori avviene mediante `exit` (equivalente di `return` di altri linguaggi)
  - 0 se comando/funzione eseguiti con successo
  - 1 o maggiori in caso di errore

### 4.2 Chiamata

Si fa semplicemente

```
function_name
function_name param1 param2
```