



- 1 Programmazione Bash
  - Caratteristiche principali
  - Comandi della shell
  
- 2 Bash script
  - Primi rudimenti di shell scripting
  - Variabili e parametri

## Programmazione Bash

# Caratteristiche principali

# La shell

- Finora, uso molto limitato della Bash
  - Solo **comandi singoli** e poi invio (in *foreground* o in *background* usando `&`)
  - L'**input** da tastiera quando i file non sono specificati, oppure appunto da file,
  - L'**output** quasi sempre su schermo, tranne che per i comandi che permettono di specificare un file dove mettere l'output
    - per esempio `patch...`
  - Semplificando possiamo dire che la shell è un programma che esegue iterativamente sempre la stessa operazione:
    - attende che gli venga fornito in input un comando da eseguire, lo valuta per verificare che il comando sia sintatticamente corretto e lo esegue
    - quindi torna ad attendere che sia fornito in input il comando successivo

# La shell

- Tutte e tre le caratteristiche sopra possono essere modificate
  - si possono specificare sequenze di comandi
  - si possono specificare varie condizioni in dipendenza delle quali eseguire un comando oppure un altro, anche organizzando i comandi in cicli
  - se l'input è da tastiera, si può dire di prenderlo da un file
  - se l'output è su schermo, si può dire di scriverlo su un file
  - si può far sì che l'input di un comando sia l'output di un altro



# La shell

- Quando si lancia una shell in un terminale in una sessione esistente, si ottiene una interactive non-login shell
- Per verificare se si è in una login shell, si può fare `echo $0`. Se si ottiene:
  - `-bash` con `"-"` come primo carattere, allora si tratta di una login shell
  - `bash` il primo carattere NON è `"-"`, allora NON si tratta di una login shell
- Digitando `shopt login_shell` si può ottenere:
  - `login_shell on` se si tratta di un a login shell, oppure
  - `login_shell off` in caso contrario

# La shell

- *Configurazione della bash*
  - *system-wide*, scelta dall'amministratore del sistema e che si applica ad ogni utente; basata sui files `/etc/profile` e `/etc/bash.bashrc`
  - una configurazione definibile dall'utente (molti utenti vogliono avere lo stesso ambiente in qualsiasi shell, interattiva, di login o no), che può anche sovrascrivere alcune impostazioni della configurazione system-wide
    - basata sui file `~/.bash_profile`, `~/.bash_login`, `~/.profile`, `~/.bashrc` (si ricorda che `~` è la directory home dell'utente attualmente loggato)
    - una configurazione di uscita può essere scritta su `~/.bash_logout`
- La situazione è riassunta nella seguente Tabella (dovrebbe fare un po' di chiarezza)

## La shell

Tipo	Invocazione	echo \$0
login senza autenticazione	bash -l	bash
	bash --login	
	(exec -a "-bash" bash)	-bash
	(exec -l bash)	
login con autenticazione	CTRL+ALT+Fn	
	ssh utente@nomemacchina	
	ssh utente@localhost	
	su - utente	-su
	su -l utente	
interactive	bash [-i] o invocazione terminale	bash
non-interactive	bash nomefile	nomefile
	bash -l nomefile	
	bash --login nomefile	
sottoshell		uguale shell padre

# La shell

- Un po' di storia delle (principali!) shell:
  - `sh`, detta *Bourne Shell*, dal nome del ricercatore che la ideò, nel 1977 ai Bell Labs
    - Le shell che si ispirano ad essa hanno il prompt che termina in `$`
  - `csh`, detta *C Shell*, ideata nel 1978 da Joy a Berkeley per la BSD. Oggi la si usa come `tcsh`
    - Le shell che si ispirano ad essa hanno il prompt che termina in `%`
  - `bash`, detta *Bourne Again Shell*, `sh` reimplementata, e migliorata, per GNU (Fox, 1989). Come la `sh`, ma con le caratteristiche interattive (ad es., la history) della `csh`

# La shell

- Le shell bash vengono aperte a richiesta dell'utente
  - alcune in modo "automatico": se si apre un terminale grafico (come l'`LXTerminal` della macchina virtuale dei laboratori, o il `gno-terminal` standard di Ubuntu), il processo che gestisce tale terminale crea anche una bash con il comando `bash -i` (vedere Tabella)
  - altre in modo esplicito: o perché si preme `CTRL+ALT+Fn`, o perché da dentro una shell si esegue uno dei comandi della seconda colonna di Tabella

# La shell

- Le shell bash vengono chiuse a richiesta dell'utente o per necessità di sistema
  - richiesta utente: basta usare il comando `exit [n]`
  - richiesta utente: basta usare il comando `logout`, ma solo se è una shell di login
  - necessità di sistema: una bash che cerca di fare qualche operazione proibita grave può essere terminata dal sistema operativo
  - necessità di sistema: in caso di bash in esecuzione remota (ad es., `ssh`), la bash può essere terminata se cade la connessione, o se l'utente supera la quota consentita di utilizzo

## Shell Programming

# Comandi della shell

# Generalità sui comandi

- Ogni comando viene dato immettendo da tastiera una serie di parole, separate da spazi
  - la prima è il comando (da cercare nel filesystem, a meno che non sia interno alla bash), poi seguono opzioni ed argomenti
  - per alcuni comandi, opzioni ed argomenti possono essere mischiati (ad es. `ls`)
  - per altri, prima le opzioni, poi gli argomenti (ad es. `find`)

# Generalità sui comandi

- Un comando viene considerato completato:
  - quando si trova un ; (esecuzione in *foreground*)
  - quando si trova un & (esecuzione in *background*)
  - per le shell interattive, quando viene premuto invio, il che fa partire l'esecuzione del comando (o dei comandi)
    - questo però vale solo per comandi in cui non ci sono parentesi o apici aperti ma non chiusi, nel qual caso il prompt cambia (diventa >) e occorre immettere la conclusione della riga
- Il ; può anche essere usato per separare comandi
  - Ad esempio digitando:  
`pwd ; echo $SHELL ; hostname`  
si ottiene:  
`/home/osboxes`  
`/bin/bash`  
`osboxes`
- È possibile spezzare un comando usando il carattere \

# Generalità sui comandi

- Qualsiasi comando può essere inserito tra **parentesi tonde**
- Quel comando non viene eseguito dal processo corrispondente alla bash corrente, ma viene lanciato un nuovo processo bash (*sottoshell*), all'interno del quale viene eseguito quel comando
  - se il comando è unico, allora il nuovo processo è costituito da quel comando
  - se è una lista di comandi (separati ad esempio da ;), allora il nuovo processo è una bash che esegue quei comandi
  - utilità:
    - raggruppare tutte le redirezioni in una volta sola (vedere sotto)
    - raggruppare tutto l'input/output da mandare in pipelining in una volta sola (vedere sotto)
    - raggruppare più comandi in esecuzione condizionale (vedere sotto), facendo sì che non ci sia effetto sulla bash corrente

# Generalità sui comandi

- Qualsiasi comando può essere inserito tra **parentesi graffe** (*group command*)
- Quel comando va eseguito dal processo corrispondente alla bash corrente
  - se il comando è unico, mettere le parentesi graffe è inutile (anzi, complica le cose, vedere sotto)
  - notare che serve *spazio* dopo la parentesi graffa aperta, ; dopo l'ultimo comando, *spazio* prima della parentesi graffa chiusa
  - utilità :
    - raggruppare tutte le redirezioni in una volta sola (vedere sotto)
    - raggruppare tutto l'input/output da mandare in pipelining in una volta sola (vedere sotto)
    - raggruppare più comandi in esecuzione condizionale (vedere sotto), facendo sì che l'effetto sia sulla bash corrente
- **Esercizio** provare a dare i comandi `(cd ..)` e `{ cd ..; }`

# Generalità sui comandi

- Ogni comando genera un processo
- Il processo generato, quando termina, restituisce un **exit code** alla bash: storicamente, 0 indica *tutto ok*, mentre un valore tra 1 e 255 indica un errore
  - se c'è errore, ci possono essere molte cause (non più di 255...)
  - per vedere il codice dell'errore: `echo $?`
  - se un comando non è riconosciuto, viene restituito 127
  - se un comando è costituito da una sequenza di comandi separati da `;`, allora l'exit code è quello dell'ultimo comando eseguito
  - se un comando viene eseguito in background, l'exit code è 0; per prendere il suo vero exit code, occorre usare il comando builtin `wait`

# Generalità sui comandi

- La bash permette l'esecuzione *condizionale* dei comandi
  - un comando viene eseguito solo se una certa condizione è vera
  - Il modo più semplice per farlo è condizionare un comando alla corretta (o sbagliata) esecuzione di un comando precedente
  - dove *corretta* equivale a *exit code è 0* e *sbagliata* equivale a *exit code è diverso da 0*
  - sintassi:
    - `comando1 && comando2`: `comando2` viene eseguito solo se `comando1` è corretto
    - `comando1 || comando2`: `comando2` viene eseguito solo se `comando1` è sbagliato
    - si possono concatenare più di 2 comandi, e anche usare le parentesi, sia tonde che graffe

# Esercizi

- **Esercizio:** capire se l'exit code di `ls` è 0 o diverso da 0 nei seguenti casi:
  - nessun argomento
  - un file esistente come argomento
  - un file esistente ma non accessibile in lettura come argomento
  - una directory esistente ma non accessibile in lettura come argomento
  - un file non esistente come argomento
  - un file esistente e uno non esistente come argomento

# Esercizi

- **Esercizio:** capire se l'exit code di `find` è 0 o diverso da 0 nei seguenti casi:
  - le opzioni non sono corrette (ad esempio: `-name file1 file2`)
  - non trova nessun file
  - trova 1 file
  - trova più di un file

# Generalità sui comandi

- Ogni comando genera un processo cui vengono subito associati 3 *stream*:
  - *standard input* o `stdin`, con *file descriptor* 0 (di default: la tastiera)
  - *standard output* o `stdout`, con *file descriptor* 1 (di default: lo schermo)
  - *standard error* o `stderr`, con *file descriptor* 2 (di default: lo schermo)
  - un file descriptor è un intero non negativo associato o ad uno stream (come sopra) o ad un file vero e proprio
- Ciascuno degli stream dati sopra può essere *rediretto*

# Tabella delle redirezioni

Il numero  $n$  in Tabella deve essere un file descriptor

Operatore	Significato	Commento
$n < \text{file}$	Apertura in lettura	Aprire in lettura il file con nome <i>file</i> sul file descriptor $n$ . Ovvero, dopo tale redirezione, si può usare $n$ nelle redirezioni di lettura, e l'effetto sarà quello di leggere da <i>file</i> (finché $n$ non viene chiuso). Inoltre, se il file descriptor $n$ era già aperto, tutte le letture fatte su $n$ vengono fatte su <i>file</i> (a meno di sovrascrizioni). Il default per $n$ è 0 (stdin).
$n <> \text{file}$	Apertura in lettura e scrittura	Aprire in lettura e in scrittura il file con nome <i>file</i> sul file descriptor $n$ . Dopo tale redirezione, si può usare $n$ nelle redirezioni sia di lettura che di scrittura, e l'effetto sarà quello di leggere e/o scrivere su <i>file</i> (finché $n$ non viene chiuso). Inoltre, se il file descriptor $n$ era già aperto, tutte le operazioni fatte su $n$ vengono fatte su <i>file</i> . Il default per $n$ è 0 (stdin).

# Tabella delle redirezioni

Il numero  $n$  in Tabella deve essere un file descriptor

Operatore	Significato	Commento
$n > \text{file}$	Apertura in scrittura	Apre in scrittura il file con nome <code>file</code> sul file descriptor $n$ . Se il file esiste, come prima cosa viene troncato a dimensione 0. Dopo tale redirezione, si può usare $n$ nelle redirezioni di scrittura, e l'effetto sarà quello di scrivere su <code>file</code> (finché $n$ non viene chiuso). Inoltre, se il file descriptor $n$ era già aperto, tutte le scritture fatte su $n$ vengono fatte su <code>file</code> . Il default per $n$ è 1 (stdout).
$n >> \text{file}$	Apertura in scrittura (append)	Apre in scrittura il file con nome <code>file</code> sul file descriptor $n$ . Se il file esiste, sposta il suo offset alla fine del file. Dopo tale redirezione, si può usare $n$ nelle redirezioni di scrittura, e l'effetto sarà quello di scrivere a partire dalla fine di <code>file</code> (finché $n$ non viene chiuso). Inoltre, se il file descriptor $n$ era già aperto, tutte le scritture fatte su $n$ vengono fatte alla fine di <code>file</code> . Il default per $n$ è 1 (stdout).

# Tabella delle redirezioni

Il numero  $n$  in Tabella deve essere un file descriptor

Operatore	Significato	Commento
<code>&amp;&gt; file</code>	Redirezione in scrittura	Redirige sia lo stdout che lo stderr.
<code>&gt;&amp; file</code>	Redirezione in scrittura	Redirige sia lo stdout che lo stderr.
<code>&amp;&gt;&gt; file</code>	Redirezione in scrittura	Redirige sia lo stdout che lo stderr, ma anziché sovrascrivere <code>file</code> , appende alla fine di esso.
<code>n &lt;&amp; m</code>	Duplicazione in lettura	Duplica il file descriptor <code>m</code> in <code>n</code> . Dopo tale redirezione, l'effetto sarà quello che letture chieste al file descriptor <code>n</code> verranno invece fatte dal file descriptor <code>m</code> . Il default per <code>n</code> è 0 (stdin)
<code>n &gt;&amp; m</code>	Duplicazione in scrittura	Duplica il file descriptor <code>m</code> in <code>n</code> . Dopo tale redirezione, l'effetto sarà quello che scritture effettuate sul file descriptor <code>n</code> verranno invece fatte sul file descriptor <code>m</code> . Il default per <code>n</code> è 1 (stdout)

# Tabella delle redirezioni

Il numero  $n$  in Tabella deve essere un file descriptor

Operatore	Significato	Commento
$n <&-$	Chiusura in lettura	Chiude il file descriptor $n$ : non potrà più essere usato (ovvero, l'input non verrà più rediretto al file o al device collegato in precedenza ad $n$ ). Il default per $n$ è 0 (stdin).
$n >&-$	Chiusura in scrittura	Chiude il file descriptor $n$ : non potrà più essere usato (ovvero, l'output non verrà più rediretto al file o al device collegato in precedenza ad $n$ ). Il default per $n$ è 1 (stdout).
$n <& m-$	Spostamento in lettura	Combinazione di duplicazione in lettura seguita da chiusura di $m$ .
$n >& m-$	Spostamento in scrittura	Combinazione di duplicazione in scrittura seguita da chiusura di $m$ .

# Generalità sui comandi

- Le redirection possono trovarsi in *qualsiasi punto* di un comando (anche prima del comando stesso)
- Le redirezioni avvengono sempre *prima* che il comando sia eseguito
- Supponendo che un file di nome `file` sia non vuoto, provare a dare il comando `awk '{print}' < file > file`: dopo, `file` sarà vuoto
- Infatti `>`, come prima cosa, tronca il file (è come aprire un file in scrittura)
- Le redirezioni, se applicate ad un group command (comandi tra graffe) o ad una subshell (comandi tra tonde) hanno effetto su tutti i comandi del gruppo
- Ad esempio, anziché scrivere `cmd1 > out; cmd2 >> out`, si può scrivere `{ cmd1; cmd2; } > out`



# Generalità sui comandi

- Il **pipelining** offre un'opportunità un più e si realizza usando il simbolo |
- Il **pipelining** serve a collegare gli stdout agli stdin per le sequenze non condizionali (quindi, equivalenti alla separazione con ; o con l'andata a capo)
- Cioè, anziché scrivere `cmd1 >file1; cmd2 <file1; rm -f file1`, si può scrivere `cmd1 | cmd2`
- Così lo stdout del processo a sinistra viene rediretto nello stdin del processo a destra
- Scrivendo `|&` anziché `|`, allora anche lo stderr viene rediretto nello stdin

# Generalità sui comandi

- Se viene mandato in background, con `jobs -l` si vede cosa succede: tutti i processi corrispondenti ai comandi in pipelining sono stati lanciati
- Il pipelining, se applicato ad un group command (comandi tra graffe) o ad una subshell (comandi tra tonde) ha effetto sull'input/output combinato di tutti i comandi del gruppo
- Ad esempio, anziché scrivere `{ cmd1; cmd2; } > out; cmd3 < out; rm out`, si può scrivere `{ cmd1; cmd2; } | cmd3`



# Cominciare

- Prendere un comando qualsiasi (ad esempio, `ls`), e scriverlo su un file di testo; sia `filename` il nome di tale file
- È possibile eseguire tale file in 4 modi:
  - 1 `chmod u+x filename; ./filename` (non proprio ortodosso: ci vorrebbe una prima riga, detta *shabang*, fatta in un certo modo; ci ritorneremo)
  - 2 `bash filename`
  - 3 `source filename`
  - 4 `. filename`

# Cominciare

- `filename` è un *bash script* (spesso, si usa l'estensione `.sh` o `.script`)
- Eseguirlo nei primi 2 modi equivale a lanciare una sottoshell (sempre, anche se c'è dentro un solo comando) che esegue uno alla volta i comandi contenuti nello script
- Invece, nei secondi 2 modi *non* si lancia una sottoshell, e l'esecuzione avviene nel contesto della shell corrente
- In realtà, la `bash` permette di avere un vero e proprio linguaggio di programmazione *Turing-completo*, i cui comandi base sono, essenzialmente, i comandi della shell visti finora più gli assegnamenti e i controllori di flusso (vedere sotto)

# Cominciare

- Quindi, è possibile compiere decisioni (ad esempio, con l'`if`) e cicli (ad esempio, con il `for` e il `while`)
- Quindi, alcuni comandi potrebbero non essere eseguiti, o eseguiti più volte
- Se ci sono errori di sintassi: la parte che precede l'errore viene sempre eseguita
- La parte che segue l'errore potrebbe essere eseguita o no, a seconda della gravità dell'errore

# Cominciare

- Tutto quello che si scrive sulla bash interattiva può essere messo in uno script; per separare i comandi, si può usare l'andata a capo al posto del ; ,
- Il viceversa non è vero, perché le andate a capo possono essere solo negli script, infatti, nella shell interattiva, premere invio vuol dire “esegui il comando” ...
- C'è però l'eccezione già menzionata sopra (se ci sono parentesi o apici aperti)
- Altra eccezione: mettendo un backslash alla fine del comando (ma proprio alla fine, senza spazi successivi), l'effetto è lo stesso di quando ci sono apici o parentesi aperte e non chiuse

Bash script

## Variabili e parametri

# Variabili

- Le variabili sono definite come nei linguaggi di programmazione: un *identificatore* cui si assegna, e dal quale si può recuperare, un valore
- niente tipi: di default tutte stringhe, ma possono essere interpretate come interi in opportune circostanze; con opportuna sintassi, possono essere anche degli array
- Formalmente, in bash ci sono le variabili (definite come sopra), i *parametri posizionali* e i *parametri speciali*
- Inoltre, sempre formalmente, tutte e 3 le categorie appena definite sono indicate genericamente come **parametri**

# Variabili e parametri

- Sia i parametri che le variabili sono identificati nello stesso modo: per leggere il loro contenuto: occorre il simbolo \$ davanti al nome (o al simbolo)
- Il termine parametro viene utilizzato per definire una variabile speciale che può essere solo letta e rappresenta alcuni elementi particolari dell'attività della shell
- Per fare riferimento al contenuto di un parametro occorre utilizzare il prefisso \$ che non è parte del nome del parametro: quindi, dire che il primo parametro posizionale si chiama \$1 non è esatto: è semplicemente 1, ma per leggerne il contenuto si deve aggiungere \$ davanti
- Un parametro è definito, cioè esiste, quando contiene un valore, compresa la stringa vuota

# Variabili

- Solo le variabili possono essere soggette ad *assegnamento*; per i parametri posizionali, si può usare il comando builtin `set`
- Tutti i parametri possono essere soggetti ad *espansione* (ovvero, si può usare il valore che contengono)
- Le variabili si assegnano con `identificatore=espressione`, **senza** spazi prima e dopo l'uguale
- Si espandono con `$identificatore`, oppure con `${identificatore}`
- Quindi, per vedere quanto valgono, è sufficiente dare un comando come `echo ${identificatore}` (il meccanismo che c'è sotto si chiama *espansione*)

# Variabili

- La versione con le graffe è importante quando ci sono 2 variabili, i cui nomi sono l'uno il prefisso dell'altro
- Senza graffe, il nome della variabile finisce non appena non ci sono né caratteri alfanumerici né underscore; altrimenti, il nome della variabile finisce con la }
- Nel caso delle stringhe (default), si possono concatenare a piacimento stringhe costanti e stringhe variabili, senza usare nessun operatore di concatenazione
- Espandere una variabile mai definita prima dà luogo alla stringa vuota (o al valore 0, nelle espansioni aritmetiche)
- Dentro alle parentesi graffe si possono fare molte cose...
- Tutte le variabili assegnate come descritto sopra sono **locali**, quindi un processo lanciato da uno script, dopo che una variabile è stata settata, non può vederne il valore

# Parametri speciali

- Alcuni **parametri speciali** sono i seguenti:
  - `$0`: il nome dello script, come è stato avviato
  - `$*`: lista di tutti i parametri posizionali (da 1)
  - `$@`: lista di tutti i parametri posizionali (da 1); (con una la differenza per il word splitting)
  - `$#`: numero di parametri posizionali (da 1), separati da uno spazio
  - `$?`: exit status dell'ultimo processo terminato
  - `$!`: pid dell'ultimo processo avviato in background (terminato o no)
  - `$$`: pid della bash (o del parent della bash, se si tratta di una sottoshell)

# Parametri posizionali

- Alcune informazioni sui **parametri posizionali**
  - Si usano solo negli script o nei corpi delle funzioni
  - Un parametro posizionale è definito da una o più cifre numeriche (a eccezione di \$0 che ha significato speciale)
  - I parametri posizionali rappresentano gli argomenti forniti al comando: \$1 è il primo, \$2 è il secondo, e così via
  - \$n sta per l'n-esimo argomento dato allo script o alla funzione
  - Se n necessita più di una cifra decimale per essere scritto, allora l'espansione va fatta con le parentesi graffe: \${n}
  - Con set {valore} si possono cambiare i valori dei parametri: se vengono specificati n valori, allora il primo viene assegnato a \$1, il secondo a \$2, ..., l'n-esimo a \$n
  - Con shift [n] si possono cambiare i valori dei parametri: quelli da \$1 a \$n avranno come valore la stringa vuota (sono non assegnati), mentre \$(n+1) avrà il vecchio valore di \$1, \$(n+2) avrà il vecchio valore di \$2 e così via. L'argomento di default di shift è 1.