Modulo 2 - Sezione A
Programmazione in linguaggio script

Laboratorio di Sistemi Operativi I
Anno Accademico 2007-2008

Francesco Pedullà
(Tecnologie Informatiche)

Massimo Verola
(Informatica)

Script - I

- **Cos'è uno script?**
  - Qualunque sequenza di comandi può essere registrata in un file di testo (*script*) per poi essere eseguita
  - Uno script è utile per eseguire sequenze di comandi ripetitive e procedure automatiche
  - Gli script sono molto usati nella gestione dei sistemi e di applicazioni complesse (per es., costituite da piu' moduli indipendenti)
  - Si possono anche scrivere programmi strutturati di una certa complessità
  - A volte si programma in linguaggio script per eseguire uno sviluppo prototipale di un'applicazione

Script - II

- **Per eseguire uno script:**
  - Scrivere il codice dello script in un file
  - Renderlo eseguibile tramite il comando **`chmod +x <file>`**
  - Digitate il nome del file (previa raggiungibilità via **`$PATH`**)
- **Azioni svolte dalla shell per eseguire uno script**
  - Determina quale tipo di shell deve essere utilizzate per eseguire lo script (di norma leggendone il path all'inizio dello script stesso)
  - Lancia una sottoshell del tipo appropriato
  - Il file dello script viene passato come argomento alla sottoshell
  - Il contenuto del file viene utilizzato come input della shell

Requisiti di uno script

- Lo script deve girare senza errori
- Deve svolgere le funzioni per cui è stato progettato
- La logica del programma dovrebbe essere chiara ed evidente
- Non deve contenere codice non necessario
- Dovrebbe poter essere riutilizzabile
- Le condizioni di errore devono essere gestite opportunamente, considerando eventualmente anche la possibilità che lo script venga interrotto
- Il codice deve essere commentato in modo chiaro ed esaustivo

Comandi built-in: script execution

`. filename [arguments]`

`source filename [arguments]`

- Esegue uno script nel contesto della shell corrente
- Paragonabile alla direttiva **`#include`** del preprocessore C
- Esempio:

```
#!/bin/bash
# cr.sh (change dir to root)
cd /
```

- Dalla linea di comando:

```
% pwd
/home/penguin
% ./cr.sh
% pwd
/home/penguin
```

```
% pwd
/home/penguin
% source cr.sh
% pwd
/
```

Selezione della shell

- **La selezione della shell che eseguirà lo script è basata sulla prima riga dello script stesso:**
  - se contiene solo il simbolo **#**, viene utilizzata la shell da cui lo script è stato lanciato
  - se contiene la stringa **#!*pathname***, viene utilizzata la shell identificata da pathname (è la forma raccomandata perché non ambigua)
    - Esempio: per eseguire uno script con la Korn shell
      #!/bin/ksh
  - altrimenti, viene interpretato da una Bourne shell
- **Alcuni comportamenti interessanti (cosa succede? perchè?):**

```
#!/bin/rm

#!/bin/cat

#!/usr/bin/sed -f
```

Variabili built-in

Identificativo	Descrizione	Disponibile:
`$$`	identificatore di processo (PID) della shell	*sh*
`$0`	nome dello shell script	*sh*, *csh*
`$1-$9`	`$`***n*** è l'n-esimo argomento della linea di comando	*sh*, *csh*
`${n}`	`${`***n***`}` è l'n-esimo argomento della linea di comando	*sh*, *csh*
`$*`	lista tutti gli argomenti di command line	*sh*, *csh*
`$#`	numero di argomenti sulla command line	*sh*
`$?`	valore di uscita dell'ultimo comando eseguito	*sh*

Esempio di script

```
#!/bin/bash
a=23              # Simple case
echo $a
b=$a
echo $b

# Now, getting a little bit fancier...

a=`echo Hello!` # Assigns result of 'echo' command to 'a'
echo $a

a=`ls -l`       # Assigns result of 'ls -l' command to 'a'
echo $a

exit 0
```

Esercizi

- **Scrivere uno script che:**
  - Conta il numero di processi che girano sul sistema e ne stampa il numero
  - Suggerimento: usare ps e wc con le opportune opzioni – attenzione alla prima riga di ps.
- **Scrivere uno script che:**
  - Trova tutti gli utenti connessi
  - Ne stampa i nomi a terminale
  - Suggerimento: usare ps, sort e uniq con le opportune opzioni

Espressioni: valutazione

- **Valutazione di espressioni**
  - La shell non supporta direttamente la valutazione di espressioni

```
% a=5-3
% echo $a
5-3
```

```
% a=`expr 5 - 3`
% echo $a
2
```

  - Esiste l'utility *built-in* **`expr expression`**
    - Valuta l'espressione e scrive il risultato sullo standard output
    - Tutti i componenti dell'espressione **devono essere separati da spazi**
    - Tutti i metacaratteri devono essere prefissi da un backslash (ad es. *)
    - Il risultato può essere numerico oppure stringa a seconda dell' operatore
    - Il risultato può essere assegnato ad una variabile facendo uso opportuno della *command substitution*

Espressioni: operatori

- **Operatori**
  - Aritmetici: `+ - * / %`
  - Confronto: `= != > < >= <=`
  - Logici: `& |`
  - Parentesi: `( EXPRESSION )`
    - Nota: devono essere prefissate da `\`, come gli operatori `* & |`
  - Stringa:
    - `match string regularExpression`
    - `substr string start length`
    - `length string`
    - `index string charList`
  - Nota: vedi ***espressioni regolari*** nei lucidi successivi

Espressioni: esempi

```
% x=1
% x=`expr $x + 1`
% echo $x
2
% expr $x \* $x
4
% expr length "cat"
3
% expr index "donkey" "key"
4
% expr substr "donkey" 4 3
key
```

Exit status

- **Quando un processo termina, ritorna un exit status**
- **Convenzioni UNIX/Linux:**
  - 0 *success* (TRUE)
  - non-zero *failure* (FALSE)
- **Comando `exit [n]`**
  - Lo script termina con exit status ***n***
- **Built-in variable $?**
  - Ritorna l'exit status dell'ultimo comando eseguito

Exit status: esempio

```
% cat script.sh
#!/bin/bash
echo hello
echo $?     # Exit status 0 returned (success).
lskdf       # Unrecognized command.
echo $?     # Non-zero exit status returned.
exit 113    # Will return 113 to shell.

% ./script.sh
hello
0
./script.sh: lskdf: command not found
127
% echo $?
113
```

Condizioni

- **Condizione = exit status**
  - Nelle strutture di controllo delle shell, gli exit status di un comando vengono utilizzati come condizioni nei test
- **Esempio:**

```
if cmp filea fileb > /dev/null  # Suppress output.
then echo "Files filea and fileb are identical."
else echo "Files a and b differ."
fi
```

Condizioni

- **Utility `test <expression>`**
  - ritorna un exit code 0 se l'espressione viene valutata vera
  - ritorna un exit code 1 se l'espressione viene valutata falsa
- **Espressioni valutate da `test`**
  - **`int1 -eq int2`** true if integer **`int1`** == integer **`int2`**
  - **`int1 -ne int2`** true if integer **`int1`** != integer **`int2`**
  - **`int1 -ge int2`** true if integer **`int1`** >= integer **`int2`**
  - **`int1 -gt int2`** true if integer **`int1`** > integer **`int2`**
  - **`int1 -le int2`** true if integer **`int1`** <= integer **`int2`**
  - **`int1 -lt int2`** true if integer **`int1`** < integer **`int2`**

Condizioni

- **Espressioni booleane**
  - `!expr` true if `expr` is false
  - `expr1 -a expr2` true if `expr1` and `expr2`
  - `expr1 -o expr2` true if `expr1` or `expr2`
  - `\( \)` grouping expressions
- **Confronti fra stringhe**
  - `str1 = str2` true if strings `str1`, `str2` are equal
  - `str1 != str2` true if strings `str1`, `str2` are not equal
  - `-z string` true if string `string` is empty
  - `-n string` true if string `string` contains at least one character

Condizioni

- **Espressioni su file**
  - `-d filename` true if `filename` exists as directory
  - `-f filename` true if `filename` exists as regular file
  - `-r filename` true if `filename` exists and is readable
  - `-w filename` true if `filename` exists and is writable
  - `-x filename` true if `filename` exists and is executable
  - `-s filename` true if `filename` exists and is not empty
  - etc.

Valutazione di condizioni

- **Sinonimo per test: [ ]**
  - Esegue la condizione contenuta tra parentesi quadre
  - [è un comando built-in
- **Sinonimo per test: [[ ]]**
  - Extended test command
  - Esegue la condizione contenuta tra doppie parentesi quadre
  - Non viene effettuato filename expansion tra [[e]]
  - Preferibile a [ ], in quanto operatori come && || > < vengono interpretati correttamente in [[ ]] e non in []
  - [[ ]] sono keyword

Esempi

- Controlla l'esistenza di un argomento

```
if [ -n "$1" ]
then
  lines=$1
fi
```

- Cambia directory e verifica la directory corrente

```
cd $LOG_DIR
if [ `pwd` != "$LOG_DIR" ]
# or: if ! cd $LOG_DIR >& /dev/null
then
  echo "Can't change to $LOG_DIR."
  exit $ERROR_XCD
fi
```

Strutture di controllo: `if – then – elif – else`

- **Struttura**

```
if list1
then
   list2
elif list3
then
   list4
else
   list5
fi
```

- **Significato**
  - I comandi in **`list1`** vengono eseguiti
  - Se l'ultimo comando in **`list1`** ha successo, **`list2`** viene eseguito
  - Altrimenti, **`list3`** viene eseguito
  - Se l'ultimo comando in **`list3`** ha successo, **`list4`** viene eseguito
  - Altrimenti viene eseguito **`list5`**

- **Nota:**
  - Un costrutto **`if`** può contenere zero o più sezioni **`elif`**
  - La sezione **`else`** è opzionale

Esempio: if – then – elif – else

```
echo "Your choice?"
read reply
if [[ "$reply" = "1" ]]; then
    echo "administrator"
elif [[ "$reply" = "2" ]]; then
    echo "student"
elif [[ "$reply" = "3" ]]; then
   echo "teacher"
else
    echo "Error"
fi
```

Strutture di controllo: `case – in – esac`

- **Struttura**

```
case expression in
pattern1 )
list1
;;
pattern2 )
list2
;;
esac
```

- **Significato:**
  - *espression* viene valutata e confrontata con ognuno dei pattern (dal primo all'ultimo)
  - quando il primo matching *pattern* viene trovato, la lista di comandi associati viene eseguita
  - dopo di che si salta al **esac** corrispondente

- **Formato:**
  - *expression* è una espressione di stringa
  - *pattern* può contenere wildcard

Esempio: case – in – esac

```
echo “Your choice?”

read reply

case "$reply" in

   "1") echo “administrator”

   ;;

   “2”) echo “student”

   ;;

   “3”) echo “teacher”

   ;;

   *[!1-3]*) echo "Usage: `basename $0` (=1,2,3)";
   exit 1

   ;;

esac
```

Esercizi

- **Scrivere un codice che:**
  - Accetta un parametro con il nome di una directory
  - Prova ad entrare in quella directory e ritorna dopo aver stampato un messaggio di conferma o di errore
- **Scrivere un codice che:**
  - Verifica la correttezza di tutti i parametri passati allo script
  - Assegna valori di default ai parametri non inizializzati
- **Scrivere un codice che:**
  - Accetta in input un'espressione aritmetica e ne calcola il valore
  - Gestisce gli errori nelle espressioni in ingresso

Strutture di controllo: `while – do – done`

- **Struttura**

```
while list1
do
  list2
done
```

- **Significato:**
  - Il comando `while` esegue i comandi in *`list1`* e termina se l'ultimo comando in *`list1`* fallisce
  - altrimenti, i comandi in *`list2`* sono eseguiti e il processo viene ripetuto
  - un comando `break` forza l'uscita istantanea dal loop
  - un comando `continue` forza il loop a riprendere dalla prossima iterazione

- **Formato:**
  - *`list1`* e *`list2`* sono liste di comandi

Esempio: while – do - done

```
#!/bin/bash
# tabelline
if [ -z $1 ]; then
   echo "Usage: $0 max"
   exit
fi
x=1
while [ $x -le $1 ]
do
   y=1
   while [ $y -le $1 ]
   do
      echo -en `expr $x \* $y` " "
      y=`expr $y + 1`
   done
   echo
   x=`expr $x + 1`                                    `
done
```

Strutture di controllo: `until – do – done`

- **Struttura**

```
until list1
do
  list2
done
```

- **Significato:**
  - Il comando `until` esegue i comandi in *`list1`* e termina se l'ultimo comando in *`list1`* ha successo
  - altrimenti, i comandi in *`list2`* sono eseguiti e il processo viene ripetuto
  - un comando `break` forza l'uscita istantanea dal loop
  - un comando `continue` forza il loop a riprendere dalla prossima iterazione

- **Formato:**
  - *`list1`* e *`list2`* sono liste di comandi

Esempio: until – do – done

```
#!/bin/bash
# tabelline
if [ -z $1 ]; then
  echo "Usage: $0 max"
  exit
fi
x=1
until [ $x -gt $1 ]
do
  y=1
  until [ $y -gt $1 ]
  do
    echo `expr $x \* $y` " "
    y=`expr $y + 1`
  done
  echo
  x=`expr $x + 1`
done
```

Strutture di controllo: `for – in – do – done`

- **Struttura**

```
for name [in words]
do
  list
done
```

- **Significato:**
  - Il comando `for` esegue un ciclo variando il valore della variabile ***name*** per tutte le parole nella lista ***words***
  - I comandi in ***list*** vengono valutati ad ogni iterazione.
  - Se la clausola `in` è omessa, `$*` viene utilizzato al suo posto.
  - è possibile utilizzare `break` e `continue`

- **Formato:**
  - ***name*** è un identificatore di variabile
  - ***words*** è una lista di parole
  - ***list*** è una lista di comandi

Esempio: for – in – do – done

```
#!/bin/bash
i=1
for color in red yellow green blue
do
  echo color n. $i is $color
  i=`expr $i + 1`
done
```

Output:

```
color n. 1 is red
color n. 2 is yellow
color n. 3 is green
color n. 4 is blue
```

Esempio: for - in - do - done

```
#!/bin/bash
# Cancella i file di backup di emacs, previo consenso
for file in $(ls *~)
do
  echo -n "Sei sicuro di voler rimuovere $file ?"
  read reply
  if [ $reply = "Y" -o $reply = "y" ]; then
    rm -f $file && echo File $file removed
  fi
done
```

Forma alternativa di command substitution; equivalente a 'ls *~ '

Esempio: for - in - do - done

```
#!/bin/bash
# bin-grep.sh: Locates matching strings in a binary file.
# A "grep" replacement for binary files.
# Similar to grep -a
EBADARGS=65
ENOFILE=66
if [ $# -ne 2 ]; then
  echo "Usage: `basename $0` string filename"
  exit $EBADARGS
fi
if [ ! -f "$2" ]; then
  echo "File \"$2\" does not exist."
  exit $ENOFILE
fi
```

Esempio: for - in - do - done

```
# The "strings" command lists strings
# in binary files.
for word in $( strings "$2" | grep "$1" )
# Output then piped to "grep",
# which tests for desired string.
  do
    echo $word
  done
exit 0
```

Esercizi

- Progettare uno script che prende in input come parametri i nomi di due directory e copia tutti i file della prima nella seconda, trasformando tutte le occorrenze della stringa SP in SU in ogni file.
- Progettare uno script drawsquare che prende in input un parametro intero con valore da 2 a 15 e disegna sullo standard output un quadrato (utilizzando i caratteri +, - e |) come nel seguente esempio:
  - **`> drawsquare 4`**

```
+--+
|  |
|  |
+--+
```

- Progettare uno script che prende in input come parametro il nome di una directory e cancella tutti i file con nome core dall'albero di directory con radice la directory parametro.

File di configurazione: inizializzazione

- **`/etc/profile`**
  - system wide defaults, mostly setting the environment (all Bourne-type shells, not just Bash)
- **`/etc/bashrc`**
  - system wide functions and aliases for Bash
- **`$HOME/.bash_profile`**
  - user-specific Bash environmental default settings, found in each user's home directory (the local counterpart to **`/etc/profile`**)
- **`$HOME/.bashrc`**
  - user-specific Bash init file, found in each user's home directory (the local counterpart to **`/etc/bashrc`**). Only interactive shells and user scripts read this file.

File di configurazione: terminazione

- `$HOME/.bash_logout`
  - user-specific instruction file, found in each user's home directory. Upon exit from a login (Bash) shell, the commands in this file execute.
- **Esempio: file di configurazione (`.bashrc`)**

```
alias rm="rm -i"
alias cp="cp -i"
alias mv="mv -i"
alias mnt="mount /mnt/zip"
```

Esempio: file di configurazione (/etc/profile)

```
if ! echo $PATH | /bin/grep -q "/usr/X11R6/bin" ; then
  PATH="$PATH:/usr/X11R6/bin"
fi
USER=`id -un`
HOSTNAME=`/bin/hostname`
for i in /etc/profile.d/*.sh; do
  if [ -x $i ]; then
    . $i
  fi
done
```

Comandi built-in: I/O (I)

- **`echo`**
  - Stampa i suoi argomenti su stdout
  - **`echo -n`** evita che sia aggiunto un newline
  - **`echo`**, utilizzato con command substitution, può essere utilizzato per settare una variabile
    - Es: **`a=`echo $myvar | tr A-Z a-z`**
- **`printf`**
  - Versione migliorata di **`echo`**, con sintassi simile a quella di **`printf`** in C
  - Per dettagli, vedere manuale (**`info printf`**)
  - Esempio:

```
PI=3.14159265358979
PI5=$(printf "%1.5f" $PI)
```

Comandi built-in: I/O (II)

- **`read`**
  - **`read var`** legge l'input da una tastiera e lo copia nella variabile ***`var`***
  - Esempio:

```
read -s -n1 -p "Hit a key " keypress
echo; echo "Keypress was \"$keypress\"."
```

    - opzione **`-s`** significa no echo input
    - opzione **`-nN`** significa accetta solo N caratteri di input
    - opzione **`-p`** significa stampa il prompt seguente

Comandi built-in: variabili

- **`declare` o `typeset` (sinonimi)**
  - Permettono di restringere le proprietà di una variabile
  - Una forma "debole" di gestione dei tipi
  - Opzioni:
    - **`declare -r var`** Dichiara una variabile come read-only
    - **`declare -i var`** Dichiara la variabile come intera
    - **`declare -a var`** Dichiara la variabile come array
    - **`declare -x var`** Esporta la variabile
    - **`declare -p var`** Mostra informazioni sulla variabile

    Per disabilitare la dichiarazione usare +[opzione]:
    ad es. **`declare +i var`**
- **Sinonimi**
  - **`export`** è equivalente a **`declare -x`**
  - **`readonly`** è equivalente a **`declare -r`**

Comandi built-in: variabili

- **let**
  - Il comando **let** esegue operazioni aritmetiche sulle variabili. In molti casi, funziona come una versione più semplice di **expr**
  - Esempi:

```
let a=11          # Assegna 11 ad a
let "a <<= 3"     # Shift a sinistra di tre posizioni
let "a /= 4"      # Divisione per 4 di a
let "a -= 5"      # Sottrazione per 5
```

- **unset**
  - Cancella il contenuto di una variabile

Comandi built-in: comandi vari

- `true`
  - Ritorna sempre un exit status 0 (successo)
- `false`
  - ritorna sempre un exit status 1 (fallimento)
- `type [cmd]`
  - stampa un messaggio che descrive se il comando *cmd* è una keyword, è un comando built-in oppure un comando esterno
- `shopt [options]`
  - setta alcune opzioni della shell
    - Es: **`shopt -s cdspell`** permette mispelling in **`cd`**

Comandi built-in: comandi vari

- `alias, unalias`
  - Un alias Bash non è altro che una scorciatoia per abbreviare lunghe sequenze di comandi
  - Esempio:
    - `alias dir="ls -l"`
    - `alias rd="rm -r"`
    - `unalias dir`
    - `alias h="history -r 10"`
- `history`
  - permette di visualizzare l'elenco degli ultimi comandi eseguiti

Esercizio: `saferm` (I)

- **Descrizione**
  - Scrivere uno script che abbia le funzionalità di **`rm`**, ma che invece di cancellare definitivamente i file li sposti in una directory **`.trash`** nella vostra home
  - Usage:
    - **`saferm -l`** elenca il contenuto del cestino
    - **`saferm -p`** svuota ("purge") il cestino
    - **`saferm -r files`** ripristina il file *file*
    - **`saferm files`** rimuove i file spostandoli nel cestino
  - Nota: le varie opzioni sono esclusive; ovvero, non si può lanciare un comando **`saferm -l -p`** ; generate un errore e stampate lo *usage* dello script nel caso

Esercizio: `saferm` (II)

- **Gestione di file con lo stesso nome:**
  - se un nome di file da inserire nel cestino esiste già, rinominare il file esistente concatenando la sua data
  - suggerimento: utilizzate **`date -r` *`nomefile`* `+%s`**
  - Esempio:
    - Se nel cestino c'è un file **`prova.sh`**, e volete aggiungere un altro file prova.sh, rinominate il primo come "**`prova.sh.1030606290`**" e poi spostate il secondo nel cestino
- **Estensioni**
  - tenete conto della possibilità di ripristinare file precedenti