

## 3 – Script di Shell

56127 Pisa



# Tem

- Ambiente di Lavoro (2h)
- Comandi di UNIX (2h)
- Script di Shell (4h)
- ( Emacs Beginner's HOWTO )

# Overview

1 AWK scripting

2 Shell scripting

## 1 AWK scripting

## 2 Shell scripting

## awk - Aho, Weinberger, Kernighan

- **awk** è un linguaggio di scripting
- strumento ideato per **processare file di testo strutturati in campo** (definibili dall'utente), farne report
- legge riga per riga i file ed esegue una o più **azioni** su tutte le linee che soddisfano certe **condizioni**.
- Il programma è definito da un insieme di azioni e condizioni.
- sintassi simil C
- molto usato per scriptini di SHELL
- Riferimenti

<https://www.gnu.org/software/gawk/manual/>

# Sintassi

- Un programma **awk** è costituito da una sequenza di regole

pattern { action }

- dove `pattern` è uno tra

**BEGIN** {istruzioni-iniziali} prima di processare l'input;

**boolexpr** fa match se l'espressione binaria in C è vera (es. `$1 == "root"`)

`/regexp/` fa match se l'**espressione regolare** è vera (es. `/^January/`)

**pat1,pat2** dal record che fa match con `pat1` a quello che fa match con `pat2`

pattern vuoto: la corrispondente azione viene sempre eseguita

**END** {istruzioni-finali} dopo aver processato l'input

- default action è `print $0` (print all)

# Record

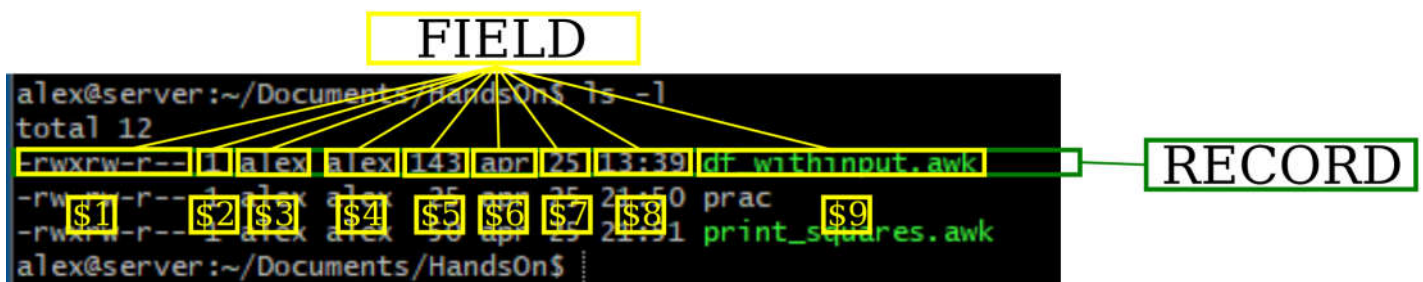
- L'input viene letto in blocchi chiamati record e viene processato un record alla volta in base alle regole del programma
- I record all'interno di un file sono separati da un carattere (o un'espressione regolare) chiamato record separator
- Il carattere utilizzato come record separator di default è il carattere di new line (a capo)

```
alex@server:~/Documents/HandsOn$ ls -l
total 12
-rwxrw-r-- 1 alex alex 143 apr 25 13:39 df_withinput.awk
-rw-rw-r-- 1 alex alex 25 apr 25 21:50 prac
-rwxrw-r-- 1 alex alex 50 apr 25 21:51 print_squares.awk
alex@server:~/Documents/HandsOn$
```

RECORD

# Field

- Ogni record viene suddiviso in field (campi)
- I field all'interno dei file sono separati da un carattere (o un'espressione regolare) chiamato field separator
- Di default i campi sono separati da whitespace costituiti da uno o più spazi, tab o newline
- La suddivisione in field rende possibile accedere alle singole componenti di un record





## awk Variabili (1)

- **Variabili predefinite:**
  - `$1, $2, . . .` contengono il 1<sup>o</sup>, 2<sup>o</sup>,... campo della linea corrente
  - `$0` contiene l'intera linea corrente;
  - `NF` contiene il numero dei campi della linea corrente;
  - `NR` contiene il numero di linea corrente
  - `FILENAME` contiene il nome del file corrente;
  - ...
- **Variabili definiti dall'utente:** non occorre dichiararle; sono automaticamente inizializzate a 0 oppure alla stringa vuota.

## Variabili (2) - DEMO

Esempi di base

Stampare ciascuna riga di input (come cat)

```
$ df -k | awk '{ print }'
```

Estrarre colonne di input

```
$ df -k | awk '{print $1, $5, $NF}'
```

Estrarre righe di input

```
$ awk '/HandsOn/ { print }' /etc/passwd
```

Variabili definiti dall'utente

```
$ echo | awk -v myname="Kocian" 'BEGIN {printf  
"Buongiorno. Sono \%s. \n", myname}'  
Buongiorno. Sono Kocian.
```

## awk: i separatori

**FS** Field Separator: Separatore di campi in input (default: spazi/tab)

```
awk -F: '{print $1, $3, $NF}' /etc/passwd
```

**RS** Record Separator: Separatore di record (default: \n)

```
awk 'BEGIN {RS = ",,"} {print $1 }'  
/etc/passwd
```

**OFS** Output Field Separator: Separatore di campi in output (default: \n)

```
awk 'BEGIN{FS=":"; OFS=";"} {print $1,$3,$NF}'  
/etc/passwd
```

# awk: Operatori e Predicati

- **Operatori aritmetici** (sia per interi che floating point):  
+ somma, - sottrazione, \* prodotto, \*\* esponente, /  
divisione, % resto.

```
awk '{a = $1*$1; print "a =", a; exit}'
```

- **Operatori booleani:**  
&& congiunzione, || disgiunzione, ! negazione.
- **Predicati** (sia per numeri che stringhe):  
<, <=, >, >= ordinamento, == uguaglianza, != diversità.

```
ls -l | awk '$5 >= 1e7 {print $5, $9}'
```

```
~ match con regex (ls -l | awk '$9 ~ /foo/ {print}')
```

```
!~ match negativo (ls -l | awk '$9 !~ /foo/ {print}')
```

## awk: Espressioni regolari

## awk per abbinare i campi

```
ls -l | awk ' $9 ~ /^[a-d][0-9])/ {print}'
```

stampa tutti gli file che iniziano con una lettera (a-d) seguita da un numero.

## Esempio

```
awk '/\\/*/, /* \\/* { print $0 }' myfile.c
```

Le due espressioni servono a selezionare le righe che contengono commenti nella forma `/ * . . . */`

N.B.

Senza la barra obliqua inversa (escape) il carattere / viene interpretato come l'inizio di un regexp!

# awk

## Operatori aritmetici

```
$ echo | awk -v PI=3.1415 'BEGIN {printf "The
cosine of %f is %f\n", PI, cos(PI)}'
The cosine of 3.141500 is -1.000000
```

## pretty print

```
df | awk 'BEGIN {print "Elenco partizioni"}
/dev\|sda2/ {print "La partizione:" $1 "\t è usata
al " $ 5 } END {print "Fine Report\n" }'
Elenco partizioni
La partizione :/dev/sda2 è usata al 85%
Fine Report
```

## rimuover i duplicati

```
awk '!visited[$0]++' myfile
es. ls -l | awk '!visited[$1]++'
```

## awk script (1)

Vogliamo automatizzare l'avvio dell'interprete per l'esecuzione di uno script `myfirst.awk`:

```
#!/usr/bin/awk -f
{ a = $1*$1; print "a =", a }
```

**#!** : il **Shebang** indica al sistema che un interprete venga utilizzato per eseguire lo script stesso.

`/usr/bin/awk` : l'interprete

`-f` : opzione, utilizzata per leggere un file di programma

### Esempio con input dall'utente

```
$ chmod u+x myfirst.awk
$ ./myfirst.awk
2
a = 4
```

## awk script (2)

L'esempio precedente dal file `mydata.txt` contenente un elenco di numeri:

```
2  
3  
4
```

### Esempio con input dal file

```
$ ./myfirst.awk mydata.txt  
a = 4  
a = 9  
a = 16
```



## raggruppare ed elaborare

Supponiamo di voler estrarre il dato e l'indirizzo IP di ogni hacker del mio server di posta

- cerca i record "postfix" nel file syslog
- raggrupparli in field
- trova le righe con "smtp" nel risultato

## Esempio [non implementato nell'emulatore]

```
$ grep postfix /var/log/syslog | awk '{ print  
$1" "$2" "$3 " "$12}' | sed -n '/smtp/p'
```

May 21 01:11:37 (smtp:192.241.211.5)

May 21 03:28:31 (smtp:113.69.207.113)

May 21 06:21:24 (smtps:167.248.133.54)

1 AWK scripting

2 Shell scripting

## Obiettivi

- Abbiamo già imparato
  - Interazione e comandi principali
  - Alias, history
  - Wild cards (\*, ?)
  - Pipelining e ridirezione (|,>,<,>>)
- Approfondiremo la struttura interna della shell:
  - variabili, espansione della riga di comando, comandi composti (liste, pipe, sequenze condizionali)
- Daremo le basi di programmazione di shell (*shell scripting*)
  - Funzioni, costrutti di controllo, debugging

## Un primo esempio di script

```
localhost:~$ /bin/bash
localhost:~$ nano mioprimo.sh
echo "Hello World !"
```

C-O, enter, C-X

```
localhost:~$ cat mioprimo.sh
echo "Hello World !"
localhost:~$
```

## Un primo esempio di script

- Come procedere per l'esecuzione di uno script:
  - salvare i comandi sopra in un file (**mioprimo.sh**)
    - attenti al separatore (newline)
  - lanciare **la bash** con lo script (ed i suoi eventuali argomenti) come argomento

## Un primo esempio di script

```
localhost:~$ bash mioprimo.sh  
Hello Wold!  
localhost:~$
```

## Uno script con argomenti

```
localhost:~$ nano mioargo.sh  
echo "Script $0"  
echo "Primo Parametro $1"  
echo "Secondo Parametro $2"
```

## Uno script con argomenti

```
localhost:~$ bash mioargo.sh ciccio pippo  
Script mioargo.sh  
Primo Parametro ciccio  
Secondo Parametro pippo  
localhost:~$
```



## #!/bin/bash

- In realtà possiamo specificare la shell direttamente nello script

```
localhost:~$ nano mioargo.sh
```

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

```
echo "Script $0"
```

```
echo "Primo Parametro $1"
```

```
echo "Secondo Parametro $2"
```

- assicurarsi che su mioargo.sh sia permessa l'esecuzione (u+x)

```
localhost:~$ ls -l mioargo.sh
```

```
-rwxr-xr-x 1 HandsOn user2 Apr 29 2021 mioargo.sh
```

```
localhost:~$
```

## #!/bin/bash

- Il risultato è lo stesso di prima, ma non è necessario invocare la bash esplicitamente

```
localhost:~$ ./mioargo.sh gg ff dd
```

```
Script ./mioargo.sh
```

```
Primo Parametro gg
```

```
Secondo Parametro ff
```

```
localhost:~$
```

- Questo è quello che faremo in tutti i nostri script

## Variabili di shell

- Le variabili della shell:
  - una variabile è un *nome* cui è associato un *valore*
  - nome: stringa alfanumerica che comincia per lettera  
valore: stringa di caratteri
  - per dichiarare/assegnare un valore ad una variabile  
**<varname>=[<value>]**
    - se **varname** non esiste viene creata, altrimenti il valore precedente viene sovrascritto
    - attenzione: prima e dopo il segno '=' non devono comparire spazi

## Variabili di shell

- Una variabile si dice *definita* quando contiene un valore
  - anche la stringa vuota!
- Può essere cancellata con  
`unset varname`
- Per riferire il valore si usa la notazione  
`<varname>` oppure `${<varname>}`

## Variabili di shell

- Script con variabili:

```
localhost:~$ nano vari.sh
```

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

```
HELLO="Hello"
```

```
touch $HELLO
```

```
localhost:~$ ./vari.sh
```

```
localhost:~$ ls -l Hello
```

```
-rw-rw-r-- 1 HandsOn user2 0 apr 29 12:20 Hello
```

```
localhost:~$
```

## Variabili di shell predefinite

- Alcuni variabili sono assegnate da Bash, es:

<b>SHELL</b>	-- <i>shell di login</i>
<b>HOSTTYPE</b>	-- <i>tipo di host, es i386-linux</i>
<b>HISTSIZE</b>	-- <i>numero cmd nella history</i>
<b>HISTFILE</b>	-- <i>file dove salvare la history</i>

– Per vederle tutte : **set**

- esempi:

```
localhost:~$ echo $HISTSIZE    500
localhost:~$ echo $HISTFILE
/home/HandsOn/.bash_history
localhost:~$
```

## Variabili di shell: **PS1**

- Controllare il prompt:

- **PS1** controlla il *prompt primario*, quello della shell interattiva. Alcune stringhe hanno un significato particolare

- **\u** nome dell'utente
    - **\s** nome della shell
    - **\v** versione della shell
    - **\w** working directory
    - **\h** hostname

- esempio:

```
localhost:~$ PS1='\u@\h:\w$'
```

```
HandsOn@localhost:~$ PS1='\s:~$'
```

```
bash:~$
```

## Variabili di ambiente (1)

- Le variabili di shell fanno parte dell'ambiente *locale* della shell stessa
  - quindi non sono visibili a processi o sottoshell attivate
  - una classe speciale di variabili, dette variabili di ambiente, sono invece visibili anche ai sottoprocessi
  - una qualsiasi variabile può essere resa una **variabile d'ambiente** esportandola:

```
export <varnames>          --esporta
export <varname>=<value>    --defin e esporta
export                    --lista variabili esportate
```



## Variabili di ambiente (2)

- Esempio

```
bash:~$ nano vari2.sh
#!/bin/bash
HELLO="Hello World!"
export $HELLO
```

```
bash:~$ ./vari2.sh
echo $HELLO
```

```
bash:~$
```

Perche la variabile HELLO non è impostata ?

## Builtin . / e source

- Lo script genera un nuovo processo di shell come figlio della shell corrente.
- Qualsiasi modifica apportata all'ambiente nel processo figlio non può influire sul padre.
- Con **source** <script> i comandi sono eseguiti nella shell corrente.

```
bash:~$ source vari2.sh
echo $HELLO
Hello World!
```

## Variabili di ambiente (4)

- Alcune variabili locali sono esportate di default:
  - es **HOME**, **PATH**, **PWD**
  - le definizioni in **.bashrc** sono valide in ogni shell interattiva

- Esempi:

```
bash:~$ PATH=$PATH:.
```

```
bash:~$ bash      -- creo una sottoshell
```

```
bash:~$ echo $PATH
```

```
/local/bin:/usr/local/bin/X11:/bin:/usr/bin:/usr/bin/X11:.
```

```
bash:~$          -- PATH è stata ereditata
```

## Parametri speciali (alcuni)

- \$0** Nome dello script
- \$\*** Insieme di tutti i parametri posizionali a partire dal primo. Tra apici doppi rappresenta un'unica parola composta dal contenuto dei parametri posizionali .
- \$#** contiene il numero di argomenti passati da terminale al programma (\$0 escluso) .
- \$@** Contiene la lista dei parametri passati allo script corrente. Quindi "\$@" equivale a "{\$1,\$2,\$3, ...}"
- \$\$** PID (*process identifier*) della shell

## Parametri speciali (alcuni) (2)

- Esempio

```
bash:~$ nano scriptArg.sh
#!/bin/bash
echo Sono lo script $0
echo Mi sono stati passati $# argomenti
echo Eccoli: $*

bash:~$ ./scriptArg.sh ll kk
Sono lo script ./scriptArg
Mi sono stati passati 2 argomenti
Eccoli: ll kk
bash:~$
```

## Alcuni operatori

<code>\$ (comando)</code>	output del comando
<code>\$ ( (operazione) )</code>	risultato dell'operazione
<code>{parametri}</code>	sostituzione dei parametri
<code>var1=1 echo \${var1} #1</code>	
<code>[condizione]</code>	per testare una condizione,
<code>[[condizione]]</code>	per testare + stato dell'esito