



# Shell Script

---

*Roberto De Virgilio*

*Sistemi operativi - 23 Ottobre 2017*

# Script BASH

- ◆ Non c'è differenza tra quello che si può scrivere utilizzando la shell interattivamente o mediante uno script.
- ◆ Uno script di shell BASH è un file di testo che:
  - contiene comandi di shell;
  - inizia con la stringa “#!/bin/bash”;
  - ha permesso di esecuzione.
- ◆ I primi due caratteri #! indicano che il file è uno script o più in generale un programma interpretato. La stringa successiva è il pathname dell'interprete per il programma.
- ◆ La shell esegue l'interprete specificato nella prima linea passandogli come argomento il nome dello script.

# Comando awk

- ◆ AWK è un filtro generico per file di testo che permette di trovare sequenze di caratteri in file di testo e di effettuare una serie di azioni sulle linee corrispondenti usando comandi espressi con un linguaggio molto simile al linguaggio C.
- ◆ La sintassi della riga di comando è la seguente:

`awk 'script' nomefile`

`awk -f fileprogramma nomefile`

# Comando awk

- ◆ AWK elabora il file *nomefile* secondo le istruzioni contenute in '*script*' oppure nel file *programma*.
- ◆ Si può usare AWK anche come filtro:

*comando* | *awk 'script'*

*comando* | *awk -f fileprogramma*

# Comando awk

- ◆ AWK elabora i file di testo una riga alla volta, eseguendo azioni diverse a seconda del contenuto della riga.

- ◆ Il primo fondamento su cui si basa AWK è:

*pattern { azione }*

*pattern { azione }*

...

- ◆ Quando il pattern è soddisfatto viene eseguita l'azione.

# Comando awk

- ◆ *Il secondo fondamento su cui si basa AWK è la suddivisione dei file di testo in campi (fields) e linee (records).*
- ◆ *Ciascun record rappresenta una linea del file e ciascun campo una "parola" (i campi sono divisi tra loro dal carattere contenuto nella variabile **FS** che di default è lo spazio).*

# Comando awk: pattern

- ◆ *I pattern possono essere*
  - *semplici espressioni regolari racchiuse tra "/" (es: / pippo/)*
  - *un'espressione logica o ancora le espressioni BEGIN ed END (vengono ritenute vere rispettivamente prima di incominciare a leggere il file in input e dopo averlo esaminato tutto).*

# Comando awk: pattern

- ◆ AWK *legge una ad una le righe del file e se una riga contiene il pattern specificato viene eseguita l'azione associata.*
- ◆ *Le azioni non sono altro che dei piccoli programmi C-like. È importante notare come le variabili non debbano essere dichiarate (a differenza del C) e vengano automaticamente inizializzate a zero oppure alla stringa nulla.*

# Comando awk: campi e righe

- ◆ Per indicare i campi (parole) della riga corrente si usano le variabili  $\$0, \$1, \$2, \dots;$
- ◆ la variabile  $\$0$  contiene l'intera riga (record) mentre  $\$1$  contiene il primo campo,  $\$2$  il secondo e così via.

# Comando awk: istruzioni

- ✿ Una istruzione AWK appartiene ai seguenti tipi:
  - *Assegnazione:* var = exp dove exp calcola il valore di un'espressione e lo assegna alla variabile var (es: doppio = pluto \* 2).
  - *Statement if:* if (exp) statement1 [else statement2] dove se exp è diverso da zero viene eseguito statement1, altrimenti statement2.
  - *Ciclo while:* while (exp) statement dove statement viene eseguito finchè exp continua ad avere un valore diverso da zero.

# Comando awk: istruzioni

- ◆ Una istruzione AWK appartiene ai seguenti tipi:
  - **Ciclo for:** `for(exp1;exp2;exp3)` statement dove `exp1` è eseguita al momento dell'inizializzazione del ciclo, `exp3` viene eseguita all'inizio di ogni ciclo e `exp2` fa sì che si esca dal ciclo quando diventa falsa.
  - **Ciclo for in:** `for(var in arrayname)` statement simile al ciclo `for` della shell, fa sì che alla variabile `var` vengano assegnati ad uno ad uno i valori contenuti nel vettore (unidimensionale) `arrayname`.
  - **Stampa:** `print exp,[exp,...,exp]` in cui ogni espressione `exp` viene calcolata e stampata nello standard output. I valori delle varie `exp` saranno distanziati dal carattere contenuto nella variabile `OFS` che di default è lo spazio. Se `print` viene usata senza `exp` viene eseguita la `print $0`.

# Comando awk: istruzioni

- ◆ *Notare che è possibile redirigere l'output su file nel seguente modo:*
  - `print "Ciao", "a", "tutti">> ciao.txt.`
- ◆ *scrive nel file ciao.txt la frase "Ciao a tutti" seguita dal newline (ammesso che OFS contenga uno spazio).*
  - `print "Ciao", "a", "tutti">>> ciao.txt`
- ◆ *scrive invece in modo append (come nella shell); se il file non esiste viene creato.*

# Comando awk: istruzioni

- ✿ *Una istruzione AWK appartiene ai seguenti tipi:*
  - *Stampa: printf( formato, [exp,...exp] ) che è praticamente uguale alla printf del C*

# Comando awk: istruzioni

- ◆ esistono poi le istruzioni:
  - *break*: esce dal ciclo `while` o `for` attivo.
  - *continue*: fa partire l'iterazione seguente del ciclo `while` o `for` ignorando le istruzioni rimanenti del ciclo
  - *next*: salta le istruzioni rimanenti del programma AWK
  - *exit*: fa terminare immediatamente AWK

# Comando awk: istruzioni

- ◆ *Oltre alle variabili FS e OFS, in AWK esistono altre variabili che vengono aggiornate automaticamente durante l'elaborazione del file in input:*
  - **NF**: *Numero dei campi della riga correntemente elaborata.*
  - **NR**: *Numero della riga correntemente elaborata*
  - **FILENAME**: *Nome del file correntemente elaborato.*  
*Questa variabile è indefinita all'interno del blocco BEGIN e contiene “-“ se non sono specificati file nella linea di comando*

# Comando awk: qualche esempio

- ◆ `cat elenco.txt | awk '/Luca/ {print $3}'`
  - *stampa il terzo campo di tutte le righe del file elenco.txt che contengono la parola Luca (stampa una riga vuota se il terzo campo della riga è vuoto)*
- ◆ `cat elenco.txt | awk '/Luca/ {print}'`
  - *stampa tutte le righe del file elenco.txt che contengono la parola Luca (print equivale a print \$0).*
- ◆ `awk 'BEGIN{FS=":"} ($2 == "OFF") {print $3,$1}' /etc/passwd`
  - *stampa lo username e l'UID di tutti gli utenti del sistema che sono senza password.*
- ◆ `awk '/main()/{print FILENAME}' *.c`
  - *Stampa il nome di tutti i file con estensione .c che contengono la funzione main().*

# Comando awk: simulazione di altri comandi

- ◆ **cat**
  - `awk '{print}'` stampa l'intero file (*si ricorda che print e print \$0 sono identiche*)
- ◆ **cat -n**
  - `awk '{print NR,$0}'` stampa l'intero file includendo i numeri di riga
- ◆ **wc -l**
  - `awk 'END {print NR}'` stampa il numero di righe del file

# Comando awk: qualche esempio

- ◆ *Si possono anche creare comandi interessanti come:*
  - ◆ `awk '{if ($1>max){max = $1}} END {print max}'`
    - *Cercare il valore massimo contenuto di una determinata colonna in un file di input*
  - ◆ `awk '(NR % 2) {print}'`
    - *Estrarre le righe dispari di un file: notare che il pattern è NR % 2; se il risultato dell'espressione in esso contenuta è diverso da zero viene eseguita l'azione print.*
  - ◆ `awk '{for(i = 1;i <= NF;i+=2){printf("%s ", $i)}} printf("\n")'`
    - *Stampare solo i campi dispari di ciascuna riga mediante la scansione delle variabili \$1, ..., \$NF*

# Comando awk: programmi scritti su file

- ◆ Un programma awk può essere scritto su file proprio come gli shell script:

```
#!/bin/awk -f

{
    for ( i = 1; i <= 100; i++ ) {
        if ( i != 50 ) {
            print i
        }
    }
}
```

# Comando awk: programmi scritti su file

- ◆ Si può notare l'uso dell'opzione **-f** che consente di leggere il programma da file.
- ◆ Dopo **#!/bin/awk -f** è necessario lasciare una riga vuota.
- ◆ Non bisogna dimenticare di usare il comando **chmod** per rendere eseguibile il file.

# Comando awk: i vettori

- ◆ *La loro dichiarazione avviene nel momento stesso in cui si fa riferimento ad una variabile vettore, ad esempio: `saluti[2] = "ciao";` l'elemento 2 dell'array `saluti` contiene ora la stringa `ciao`.*
- ◆ *Se l'array non esiste viene creato: i suoi elementi sono creati nel momento in cui assegnano all'array.*
- ◆ *Un elemento non inizializzato conterrà la stringa nulla (zero se si considera come un numero).*
- ◆ *Gli elementi di un array non sono ordinati sequenzialmente come in un array lineare (sono stringhe e non numeri).*

# Comando awk: i vettori

- ◆ *Se interessa scandire tutti gli elementi inseriti nella hash table corrispondente ad un array si usa il ciclo for (var in arrayname)*
- ◆ *Esempio*

```
for (i in vettore) {  
  
    print vettore[i]  
  
}
```

# Comando awk: i vettori

- ◆ *Tuttavia nella maggior parte delle applicazioni si utilizzano array indicizzati mediante interi e quindi è possibile scandire tutti gli elementi di un array vettore di max elementi in maniera "ordinata" mediante l'usuale ciclo*

```
for (i=0; i<max; i++) {  
    print vettore[i]  
}
```

# Esercizi (svolti)

- ◆ *Implementare un programma AWK che dato un file con il seguente formato*

1

2 3

4 5 6

7 8 9 10

.....

- ◆ *produca in uscita la somma dei valori di ciascuna colonna*

# Esercizi (svolti)

- ◆ *Creiamo il programma AWK che risolva il problema:*

```
#!/bin/awk -f

{
    for (i=1; i<=NF; i++) somma[i] += $i;
}

END {
    for (i=1; i<=NF; i++) printf("%d ",somma[i]);
    printf("\n");
}
```

# Esercizi (svolti)

- ◆ *Notare che il vettore somma non viene dichiarato, ma è creato automaticamente durante l'utilizzo (i suoi elementi sono automaticamente inizializzati a zero).*
- ◆ *Dopo aver sommato gli elementi delle colonne si visualizzerà il risultato.*
- ◆ *END è un pattern che risulta vero solo dopo che il file in input è stato scandito fino ad EOF, viene quindi spesso utilizzato per stampare i risultati della elaborazione di tutte le linee di un file.*

# Esercizi (svolti)

- ◆ *Dato un file con il seguente formato:*

A: 10 100 b c

B: 101 a b 200 c

C: x y z

A: 102 x 100 b c

.....

.....

- ◆ *scrivere un programma AWK che crei un file doppio.txt con lo stesso formato del precedente, ma con i numeri che compaiono nelle righe etichettate da "A:" moltiplicati per 2.*

# Esercizi (svolti)

## ◆ soluzione:

```
#!/bin/awk -f

/^A:/ {

    for(i=1; i<=NF; i++) {

        if ($i !~ /^[0-9]+/) printf("%s ", $i) >> "doppio.txt"

        else printf("%d ",$i*2) >> "doppio.txt";

    }

    printf("\n") >> "doppio.txt";

}

/^B-Z:/ { print >> "doppio.txt" }
```

# Esercizi (svolti)

- ◆ *Si sono usate le espressioni regolari per trattare in modo diverso le linee del file.*
- ◆ *Quando AWK elabora una linea che inizia per "A:" il pattern /<sup>A:</sup>/ risulta vero viene quindi eseguita l'azione.*
- ◆ *L'operatore !~ è usato per verificare se \$i (i è la variabile usata per scandire \$1, \$2, ..., \$NF) non corrisponde all'espressione regolare ^ [0-9] + (verifica se \$i è un numero).*
- ◆ *Più precisamente, l'operatore ~ è usato per verificare se la stringa \$i contiene una stringa di caratteri conforme all'espressione regolare.*

# Esercizi (svolti)

## ◆ N.B.

- *Gli operatori ~ (contiene) e !~ (non contiene) possono essere usati solo all'interno delle azioni.*
- *In questo programma si nota anche l'uso della redirezione delle printf e printf sul file doppio.txt*

# Esercizi (svolti)

- ◆ Scrivere un programma che dato l'output generato dal comando `ls -l` sia in grado di calcolare la distribuzione delle lunghezze dei vari file in intervalli di 1 Kb e di tracciare l'istogramma corrispondente.

[ 0 - 1 ] K = 15 | \*\*\*\*\*

[ 1 - 2 ] K = 0 |

[ 2 - 3 ] K = 0 |

[ 3 - 4 ] K = 0 |

[ 4 - 5 ] K = 6 | \*\*\*\*\*

[ 5 - 6 ] K = 0 |

[ 6 - 7 ] K = 0 |

[ 7 - 8 ] K = 1 | \*

# Esercizi (svolti)

## ◆ *soluzione:*

```
#!/bin/awk -f

{
    num = int($5/1024);

    distrib[num]++;
    if (num > max) max = num;

    # max e' automaticamente inizializzata a 0
}

}
```

segue =>

# Esercizi (svolti)

## ◆ *soluzione:*

```
END {  
  
    # Disegna l'istogramma  
  
    print "Istogramma\n" # Aggiunge un newline a quello di default di print  
  
    for (i=0; i<=max; i++) {  
  
        printf ("[%6d -%6d] K = %6d | " , i, i+1, distrib[i]);  
  
        for (j=0; j < distrib[i]; j++) printf("*");  
  
        printf("\n")  
  
    }  
  
}
```

segue =>

# Esercizi (svolti)

- ◆ *Si è usata la funzione `int()` che converte un numero reale in uno intero tramite troncamento; AWK dispone di molte funzioni utili che possono essere esaminate con il comando di Unix `man awk`.*
- ◆ *In questo programma si nota inoltre la somiglianza della funzione `printf` di AWK con la corrispondente funzione del C.*

# Esercizi (svolti)

- ◆ *Si sarebbe potuto utilizzare il costrutto for (i in distrib)*

```
#!/bin/awk -f

{
    distrib[int($5/1024)]++; # $5 è il campo che contiene la lunghezza del file
}

END {
    # Disegna l'istogramma

    print "Istogramma\n" # Meglio avere due newline

    for(i in distrib) {

        printf("[%6d -%6d] K = %6d | ",i,i+1,distrib[i]);
        for(j=0;j<distrib[i];j++) printf("*");

        printf("\n")
    }
}
```

# Esercizi (svolti)

- ◆ *ma occorre ricordare che questo uso è appropriato :*
  - *quando si vuole fare riferimento solo agli elementi del vettore che sono stati modificati*
  - *l'ordine degli elementi nel vettore non ha importanza.*

# Esercizi (svolti)

- ◆ *Questo è un esempio di output di questa soluzione:*

[ 4 - 5 ] K = 6 | \*\*\*\*\*

[ 26 - 27 ] K = 1 | \*

[ 7 - 8 ] K = 1 | \*

[ 47 - 48 ] K = 1 | \*

[ 9 - 10 ] K = 1 | \*

[ 49 - 50 ] K = 1 | \*

[ 20 - 21 ] K = 1 | \*

[ 22 - 23 ] K = 1 | \*

[ 0 - 1 ] K = 14 | \*\*\*\*\*

[ 50 - 51 ] K = 1 | \*

# Esercizi (svolti)

- ◆ Come si può notare l'output non è ordinato - il costrutto `for(i in distrib)` non scandisce ordinatamente il vettore - e inoltre mancano le righe  $[1 - 2] \ K = 0, [2 - 3] \ K = 0$ , ecc.

# Esercizi (svolti)

- ◆ *Realizzare una semplice calcolatrice (si assume che l'utente non commetta errori di digitazione) con AWK*

```
#!/bin/awk -f  
  
($2=="*") { print $1*$3 }  
  
($2=="/") { print $1/$3 }  
  
($2=="+") { print $1+$3 }  
  
($2=="-") { print $1-$3 }  
  
($1=="q") { exit(0) }
```

# Esercizi (svolti)

- ◆ *Possiamo utilizzare una versione che usi le espressioni regolari*

```
#!/bin/awk -f

/[0-9\.\.]+ [\+\-\*\*\// [0-9\.\.]+\{/

if($2=="*") { print $1*$3; next }

if($2=="/") { print $1/$3; next }

if($2=="+") { print $1+$3; next }

if($2=="-") { print $1-$3; next }

}

($1=="q") { exit(0) }
```

# Comando awk: passare variabili esterne

- ◆ *Talvolta è utile passare delle variabili esterne ad una procedura AWK.*
- ◆ *Per fare questo si usa l'opzione -v*

```
awk -v inizio=3 -v fine=6 '{for(i=inizio;i<=fine;i++) print $i}'
```

- ◆ *oppure, nel caso in cui si voglia passare gli argomenti all'interno di uno script di shell:*

```
awk -v inizio=$1 -v fine=$2 '{for(i=inizio;i<=fine;i++) print $i}'
```

- ◆ *Questa opzione permette spesso di evitare l'uso del pattern BEGIN che generalmente viene utilizzato solamente per l'inizializzazione delle variabili a valori diversi da zero.*

# Comando awk: operatori

- ◆ *Gli operatori di AWK, in ordine decrescente di precedenza, sono*

Operatore	Descrizione
( ... )	Raggruppamento
\$	Riferimento a campi
++ --	Incremento e decremento, sia prefisso che postfisso
^	Elevamento a potenza
+ - !	Più e meno unari, e negazione logica
* / %	Moltiplicazione, divisione e resto
spazio	Concatenazione di stringhe.

# Comando awk: operatori

- ◆ *Gli operatori di AWK, in ordine decrescente di precedenza, sono*

Operatore	Descrizione
< > <= >= != ==	I ben noti operatori di relazione
~ !~	Controllo di conformità ("match") tra regular expression, e controllo di non conformità.
in	Controllo di appartenenza ad un vettore
\$\$	AND e OR logici
+ - !	Più e meno unari, e negazione logica
+= -- *= /= %= ^=	Assegnamento con operatore

# Comando awk: funzioni numeriche

- ◆ *AWK ha le seguenti funzioni aritmetiche predefinite:*

funzione	Descrizione
<code>atan2(y, x)</code>	l'arcotangente di $y/x$ in radianti
<code>sin(expr)</code> <code>cos(expr)</code>	seno e coseno di expr (si aspetta radianti).
<code>exp(expr)</code>	esponenziale
<code>int(expr)</code>	troncamento ad intero
<code>log(expr)</code>	logaritmo naturale
<code>rand()</code>	fornisce un numero casuale tra 0 ed 1.
<code>sqrt(expr)</code>	radice quadrata

# Comando gawk: funzioni su stringhe

- ◆ *GAWK offre le seguenti funzioni di stringa predefinite:*

funzione	Descrizione
getline	Setta \$0 leggendo la linea successiva; setta anche NF, NR, FNR
getline <file	Come sopra, ma legge da file
getline var	Setta var leggendo la linea successiva; setta NR, FNR.
getline var <file	Come sopra, ma legge da file
gensub( <b>r</b> , <b>s</b> , <b>h</b> [, <b>t</b> ])	cerca nella stringa obiettivo <b>t</b> corrispondenze con la regular expression <b>r</b> . Se <b>h</b> è una stringa che inizia con <b>g</b> o <b>G</b> , tutte le corrispondenze con <b>r</b> sono sostituite con <b>s</b> ; altrimenti, <b>h</b> è un numero che indica la particolare corrispondenza con <b>r</b> che si vuole sostituire. Se <b>t</b> non è specificata, al suo posto è usato <b>\$0</b> .

# Comando gawk: funzioni su stringhe

- ◆ *GAWK offre le seguenti funzioni di stringa predefinite:*

funzione	Descrizione
gsub( <b>r</b> , <b>s</b> [, <b>t</b> ])	per ogni sottostringa conforme alla regular expression <b>r</b> nella stringa <b>t</b> , sostituisce la stringa <b>s</b> , e restituisce il numero di sostituzioni. Se <b>t</b> non è specificata, al suo posto è usato <b>\$0</b> .
index( <b>s</b> , <b>t</b> )	trova l'indice posizionale della stringa <b>t</b> nella stringa <b>s</b> , o restituisce <b>0</b> se <b>t</b> non è presente.
length([ <b>s</b> ])	la lunghezza della stringa <b>s</b> , oppure la lunghezza di <b>\$0</b> se <b>s</b> non è specificata.
match( <b>s</b> , <b>r</b> )	trova la posizione in <b>s</b> del tratto che si conforma alla regular expression <b>r</b> , oppure <b>0</b> se non ci sono conformità.
split( <b>s</b> , <b>a</b> [, <b>r</b> ])	spezza la stringa <b>s</b> nel vettore <b>a</b> utilizzando il metodo di separazione descritto dalla regular expression <b>r</b> , e restituisce il numero di campi. Se <b>r</b> è omessa, il separatore utilizzato è <b>FS</b> .

# Comando gawk: funzioni su stringhe

- ◆ *GAWK offre le seguenti funzioni di stringa predefinite:*

funzione	Descrizione
<code>sprintf(fmt, expr-list)</code>	stampa (in modo fittizio) expr-list secondo il formato fmt , e restituisce la stringa risultante
<code>sub(<b>r</b>, <b>s</b> [, <b>t</b>])</code>	come gsub(), ma è sostituita solo la prima sottostringa trovata.
<code>substr(<b>s</b>, <b>i</b> [, <b>n</b>])</code>	restituisce la sottostringa di <b>s</b> di <b>n</b> caratteri al più che inizia nella posizione <b>i</b> . Se <b>n</b> è omesso, è usato il resto di <b>s</b> .
<code>tolower(str)</code>	restituisce una copia della stringa str, con tutti i caratteri maiuscoli tradotti nei minuscoli corrispondenti. I caratteri non alfabetici restano invariati.
<code>toupper(str)</code>	restituisce una copia della stringa str, con tutti i caratteri minuscoli tradotti nei maiuscoli corrispondenti. I caratteri non alfabetici restano invariati.

# Comando gawk: funzioni di tempo

- ◆ *GAWK mette a disposizione le seguenti due funzioni per ottenere marche temporale e per manipolarle:*

funzione	Descrizione
<code>systime()</code>	restituisce la data e l'ora correnti, espresse come numero di secondi trascorsi da una certa data convenzionale (la mezzanotte del 1/1/1970 sui sistemi POSIX).
<code>strftime( [format [, timestamp] ] )</code>	Applica il formato format a timestamp.

