Programmare in Bash guida sintetica

di Marcello Zaniboni

versione 1.2 - agosto 2024

eventuali versioni più aggiornate sono scaricabili dalla pagina web
www.marcellozaniboni.net

per commenti, suggerimenti e correzioni:
marcello.zaniboni@gmail.com

Storia del documento

versione	*elenco delle modifiche*
1.2	recepiti 4 anni di piccoli ritocchi, aggiunte e precisazioni, in tutti capitoli
1.1	moltissimi ritocchi, piccole precisazioni, correzioni e aggiunte, un po' ovunque
1.0	aggiunta la verifica UTF-8 in "Un controllo rapido all'ambiente di sviluppo"; aggiunto il capitolo "Debugging o quasi"; aggiunto il capitolo "Come elaborare xml"; piccoli aggiornamenti e correzioni
0.9	aggiunti due script di esempio in "Lavorare con le stringhe"; apportate correzioni allo script in "E ora un esempio completo quasi utile"; spiegazione migliorata nei commenti degli script in "Tipizzazione avanzata e array (cioè liste)"; apportate piccole correzioni nel testo
0.2	aggiunto il capitolo "Un controllo rapido all'ambiente di sviluppo"; aggiunto il capitolo "Costruire script suddivisi in più file"; piccoli aggiornamenti e correzioni
0.1	prima versione pubblica

Breve ma indispensabile introduzione

Questa guida nasce per essere uno strumento rapido e pratico per approcciare allo scripting su ambiente Linux/Unix con la shell Bash. Tutto in una trentina di pagine!
L'approccio della guida è pragmatico, asciutto e poco teorico, cioè si vanno ad esplorare gradualmente i concetti più importanti del linguaggio, ma corredando il tutto con diversi esempi. Non vengono illustrate tutte le possibilità del linguaggio (per quelle ci sono guide online molto più complete e dettagliate), ma solo una selezione utile per conoscere le caratteristiche più utilizzate e rendere il lettore indipendente.
Ogni script di esempio è fondamentale: prestare poca attenzione agli esempi è male! Ogni script va ben compreso, salvato (il copia-incolla da file pdf funziona abbastanza bene), eseguito e possibilmente modificato a piacere, in quanto per esigenze di sintesi lo stesso concetto non viene ripetuto tante volte.

Nel testo si dà per scontato che il lettore abbia già programmato in altri linguaggi: infatti, sebbene non sia richiesta alcuna conoscenza specifica e specialistica, è essenziale sapere cos'è una variabile, una stringa, una costante, una funzione, ecc... Per sottolineare taluni concetti si farà occasionalmente riferimento ad altri linguaggi, come C e Java, ma non è richiesta la loro conoscenza.

Un'ultima raccomandazione: per leggere questa guida è indispensabile avere i fondamenti di base su come muoversi in ambienti Linux/Unix da terminale; ciò significa che questa guida non vi insegnerà affatto a

- cambiare directory, capire i concetti di path relativo, assoluto e directory corrente,
- distinguere la differenza tra slash e backslash
- consultare con curiosità il man dei comandi del sistema operativo che mano a mano vengono introdotti negli esempi,
- capire cosa sono standard output, standard input e standard error,
- usare la redirezione di standard output, standard input e il pipe,
- capire cosa sono encoding e escaping,
- assegnare i permessi ai file capendo ciò che si sta facendo,
- scegliere e usare l'editor di testo più opportuno secondo i vostri gusti.

Davvero, se vi mancano alcune di queste nozioni, sospendete per un po' la lettura e fatevi un giro sul web prima di riprendere.

Tutti gli esempi funzioneranno alla perfezione se userete sempre l'encoding UTF-8. Se non avete capito questa ultima frase, se ogni tanto andate in panico per quelli che taluni chiamano "caratteri speciali" o se "file di testo", "file ASCII" e "codifica UTF-8" sono concetti poco chiari, approfonditeli prima di continuare... altrimenti siete pronti per iniziare!

L'inevitabile Hello world

Evitiamo da subito il capitolo introduttivo, che spiega "perché Bash", la potenza del linguaggio, quando sì e quando no, la sua larga diffusione, e altre parti che, in testi come questo, quasi tutti scelgono di non leggere.
Partiamo quindi con il più classico degli inizi: un fantastico "Hello world", che non può mai mancare in ogni testo che parla di programmazione.

```
#!/usr/bin/env bash
# Primo script in Bash
echo Hello world  # commento in linea
```

Salviamo tutto in un file di testo hello.sh e impostiamo i permessi di esecuzione (cosa che bisognerà fare ogni volta che si crea un nuovo script) , dando il comando

```
$ chmod 755 hello.sh
```

Infine eseguiamo direttamente il file (trattandosi di un linguaggio di scripting, in Bash non c'è nessun passaggio per la compilazione)

```
$ ./hello.sh
Hello world
```

Ok, sono soddisfazioni.
Tuttavia questo script banale insegna già alcune cose.
Innanzitutto la prima linea. Tutti gli script Bash devono iniziare con quella prima linea di testo. Si tratta del cosiddetto "shebang"[1]. È un'istruzione che serve al sistema operativo per identificare l'interprete che eseguirà lo script (ad esempio, se scrivessimo uno script in linguaggio Perl, la prima linea sarebbe probabilmente qualcosa come "#!/usr/bin/perl").
La seconda linea è un commento. Regola generale: tutto ciò che segue un cancelletto è un commento e questo vale anche per l'ultima parte della terza linea.

Ma andiamo alla terza linea: la prima cosa importante è che "echo" non è uno statement del linguaggio Bash, bensì un normalissimo comando, cioè un eseguibile incluso nel sistema operativo, con tanto di man, che non fa altro che scrivere sullo schermo ogni parametro che riceve, cioè ogni argomento della linea di comando. Nel nostro esempio, il comando echo riceve due argomenti: "Hello" e "world".

Una conseguenza importantissima, ma spesso sottovalutata dai principianti, è che se la seconda linea viene modificata mettendo cinque spazi tra i due argomenti anziché uno, si ottiene lo stesso risultato:

```
#!/usr/bin/env bash
# Primo script in Bash
echo Hello     world  # commento in linea
```

...eseguiamo lo script:

```
$ ./hello.sh
Hello world
```

Il motivo è semplice: echo scrive a video un argomento alla volta e in Bash, gli argomenti dei comandi possono essere separati da un numero qualsiasi di spazi. Se vogliamo ottenere l'effetto desiderato, dobbiamo passare un argomento unico, usando le virgolette o l'escaping degli spazi, cioè

[1] Negli script più vecchi e anche nelle versioni precedenti di questo documento, la prima linea di uno script è differente: "#!/bin/bash". Se utilizzate questa versione, molto probabilmente i vostri script funzionano ugualmente, ma per evitare di fare assunzioni sulla posizione dell'eseguibile "bash" all'interno del proprio sistema operativo, negli ultimi anni sempre più persone consigliano di utilizzare "#!/usr/bin/env bash".

```
echo "Hello     world"
```

oppure

```
echo Hello\ \ \ \ \ world
```

A proposito di escaping, se il concetto non vi è chiaro, vi ho beccato: non avete letto l'introduzione o avete barato. In questi casi, approfondite la questione sul web fino a capire ogni singolo byte del seguente comando:

```
echo "ecco le \"virgolette\", seguite da cancelletto #" #commento
```

Se è tutto chiaro, si può proseguire con il capitolo seguente.

Un controllo rapido all'ambiente di sviluppo

Sebbene Bash sia diffusissimo, non si tratta dell'unica shell disponibile nel mondo Unix/Linux. In particolare capita che alcune distribuzioni, per ottimizzare i propri script di sistema, includano anche altre shell, meno potenti ma più leggere. Alcune distribuzioni hanno addirittura modificato il link simbolico della shell di default. È proprio il caso di Debian e molte sue derivate (Ubuntu incluso):

```
$ ls -l /bin/sh
lrwxrwxrwx 1 root root 4 giu  4  2015 /bin/sh -> dash
```

Dash non è Bash... Ed è molto importante capire quale shell viene eseguita quando si apre un terminale. Utilizzando il comando ps si può avere conferma o meno del fatto che si stia eseguendo un'istanza di Bash piuttosto che di un'altra shell.

```
$ ps
  PID TTY          TIME CMD
15401 pts/0    00:00:00 bash
29296 pts/0    00:00:00 ps
```

Questa verifica è particolarmente importante se si vuole usufruire di una grande possibilità: testare le singole righe presenti negli script di questa guida da linea di comando, eseguendole manualmente una alla volta. Se la shell dei comandi è Bash, si tratta di un'opportunità molto comoda.
Infine è opportuno accertarsi che l'encoding in uso sia UTF-8, utilizzando il comando locale:

```
$ locale
LANG=it_IT.UTF-8
LC_CTYPE="it_IT.UTF-8"
LC_NUMERIC="it_IT.UTF-8"
LC_TIME="it_IT.UTF-8"
LC_COLLATE="it_IT.UTF-8"
LC_MONETARY="it_IT.UTF-8"
LC_MESSAGES="it_IT.UTF-8"
LC_PAPER="it_IT.UTF-8"
LC_NAME="it_IT.UTF-8"
LC_ADDRESS="it_IT.UTF-8"
LC_TELEPHONE="it_IT.UTF-8"
LC_MEASUREMENT="it_IT.UTF-8"
LC_IDENTIFICATION="it_IT.UTF-8"
```

Se si dovesse scoprire che la shell in esecuzione non è Bash, o che non si sta utilizzando UTF-8, si consiglia di prendere in considerazione l'opportunità di modificare queste impostazioni, consultando le sezioni di aiuto online ed i forum relativi al proprio sistema operativo.

Variabili, costanti, tipizzazione

Iniziamo ora con l'argomento variabili, introducendo subito due operatori: l'uguale per l'assegnazione di un valore e il dollaro per sostituire il nome di una variabile con il suo valore **prima** che l'interprete esegua una linea di codice. Nell'esempio che segue viene stampato a video per due volte lo stesso testo, ottenuto però in modo molto diverso

```
#!/usr/bin/env bash
via="via dei Test"
n=0
indirizzo="$via, $n"
echo "$indirizzo"
echo $via, $n
```

Durante l'esecuzione dello script, la quarta riga sarà interpretata come

```
indirizzo="via dei Test, 0"
```

Il primo echo scrive il contenuto della variabile *indirizzo* (si noti l'utilizzo delle virgolette per fare in modo che echo riceva un solo argomento, formattato esattamente come vogliamo). Grazie all'operatore dollaro, il valore di *indirizzo* è espanso "al volo"; il secondo echo invece non presenta le virgolette e pertanto diventa

```
echo via dei Test, 0
```

cioè echo elabora quattro argomenti:

- "via"
- "dei"
- "Test,"
- "0"

In altre parole, l'operatore $ estrae il valore di una variabile e lo sostituisce nella linea che sta per essere eseguita dall'interprete, appena prima della sua esecuzione.

Un'altra particolarità di Bash è la non tipizzazione delle variabili. Chi è abituato a linguaggi con tipizzazione forte come C e Java può rimanere sorpreso dalla natura promiscua delle variabili in Bash. In questo linguaggio infatti apparentemente non c'è alcuna tipizzazione (anche se, come vedremo più avanti, se proprio lo si desidera, questo comportamento in alcuni casi può essere modificato con direttive esplicite).
Lo statement readonly serve a definire una costante. Vediamo un esempio per chiarire l'uso delle costanti e della tipizzazione dinamica (l'esempio anticipa anche una possibile la sintassi per effettuare calcoli con numeri interi, che vedremo meglio più avanti):

```
#!/usr/bin/env bash
readonly UNO=1   # dichiarazione di una costante numerica che vale 1
a=42             # a è un numero intero
b=$(($a + $UNO)) # sintassi particolare per usare l'aritmetica intera
echo "$a $b"     # scrive: 42 43
a="testo"        # a da intero diventa la stringa "testo"
b=$(($a + $UNO)) # attenzione: "testo" nei calcoli aritmetici vale 0
echo "$a $b"     # scrive: testo 1
```

La cosa interessante è che, poiché non c'è tipizzazione e poiché l'operatore $ estrae il valore delle variabili prima di eseguire ogni linea, lo script funziona alla perfezione anche se sostituiamo la linea

```
a=42             # a è un numero intero
```

con

```
a="42"           # stringa con la rappresentazione di un numero intero
```

Nell'esempio non si presti particolare attenzione alla sintassi particolare che è stata usata per effettuare le operazioni aritmetiche: più avanti nella guida c'è un apposito capitolo dedicato all'argomento.

Debugging o quasi

Dagli esempi precedenti, si intuiscono due aspetti inquietanti del linguaggio Bash. Il primo è la sintassi particolare e, come si vedrà proseguendo con la lettura di questo documento, quanto si è visto finora è niente: parentesi tonde doppie, parentesi quadre, graffe, punteggiatura, spazi da evitare e spazi obbligatori.... Il secondo è che, poiché prima dell'esecuzione di ogni riga avviene una sorta di preprocessamento che ne sostituisce delle porzioni (ad esempio estraendo i valori delle variabili tramite l'operatore "$"), in caso di problemi può diventare difficile capire in quale zona di uno script risieda un errore.
Tuttavia esiste una soluzione parziale al problema e come al solito partiamo con un esempio: uno script contenente un errore (da salvare in test.sh):

```
#!/usr/bin/env bash
i=0
a="i = $i"
echo "$a"
b = "zero = $i"
echo "$b"
```

Alla riga 5 c'è un errore ed in realtà è pure banale: uno spazio di troppo cambia una assegnazione in una invocazione del comando "b", che non esiste (aggiungiamo fortunatamente... pensate ai danni collaterali di chiamare una variabile "rm"!)

```
$ ./test.sh
i = 0
./test.sh: riga 5: b: comando non trovato
```

In caso di errori meno banali, esiste un meccanismo che ci permette di indagare meglio e capire quello che è successo in dettaglio, eseguendo lo script in maniera da visualizzare ciò che sta per essere eseguito, linea per linea, usando esplicitamente il comando "bash" con l'argomento "-x":

```
$ bash -x test.sh
+ i=0
+ a='i = 0'
+ echo 'i = 0'
i = 0
+ b = 'zero = 0'
test.sh: riga 5: b: comando non trovato
+ echo ''
```

Da qui si capiscono due cose:

1. nella riga 5, prima della sua esecuzione, la variabile $i è stata correttamente sostituita dal suo valore e l'interprete Bash si appresta ad eseguire il comando "b", che tuttavia non esiste, passandogli due argomenti: "=" e la stringa "zero = 0";
2. l'errore della riga 5 non ha provocato la terminazione dello script, ma solo la non corretta inizializzazione della variabile "b", che risulta quindi mai utilizzata e quindi vuota. Infatti la riga 6 è stata eseguita come un echo di una stringa vuota.

In conclusione "bash -x" non è per nulla equiparabile ad un debugger evoluto come quelli che troviamo in linguaggi di programmazione più evoluti, ma spesso aiuta moltissimo.

Quando l'operatore $ sembra non bastare

Supponiamo di avere una variabile *secondi*, contenente un numero di secondi, ad esempio 15. Se volessimo scrivere con echo il suo valore seguito dall'unità di misura "s", senza spazi intermedi, come potremmo fare? Fino a questo punto abbiamo estratto il valore della variabile con l'operatore $, ma se scriviamo "$secondis", questo viene interpretato male da Bash.

```
#!/usr/bin/env bash
secondi=15
echo "tempo misurato: $secondis" # la variabile secondis non esiste!
echo "tempo misurato: ${secondi}s" # la variabile secondi esiste
```

In questi casi l'ambiguità viene risolta racchiudendo il nome della variabile tra parentesi graffe.

Valorizzare una variabile con lo stdout di un comando

La programmazione Bash fa largo uso di piccoli comandi eseguibili del sistema operativo, come echo, che nei sistemi Linux/Unix sono centinaia. In questo linguaggio la conoscenza dei singoli comandi è il vero e proprio patrimonio a disposizione del programmatore. Il meccanismo essenziale che permette di costruire script veramente utili è costituito dal catturare lo standard output di un comando e inserirlo in una variabile. Bash ci offre due sintassi assolutamente equivalenti per ottenere questo risultato:

```
#!/usr/bin/env bash
# il comando pwd scrive in standard output la directory corrente
a=`pwd`
echo "directory corrente = \"$a\""
b=$(pwd)
echo "directory corrente = \"$b\""
```

La prima sintassi prevede l'uso dell'apice rovesciato, che non è un carattere solitamente presente nelle tastiere italiane (l'apice rovesciato non è da confondere con il carattere apostrofo, che in queste tastiere è in genere posizionato tra "0" e "ì"). Spesso su queste tastiere è possibile ottenere l'apice rovesciato con la combinazione di tasti AltGr + apostrofo, ma in generale è opportuno preferire la seconda sintassi, che racchiude il comando da eseguire (che può essere a sua volta un altro script) tra parentesi tonde precedute da un carattere dollaro.

Questo meccanismo è potentissimo; vediamo un altro esempio

```
#!/usr/bin/env bash
# Questo script scrive in fondo a un file di log il numero
# di file e directory contenuti nella home directory
# dell'utente che esegue lo script. In testa ad ogni riga
# è scritto il relativo timestamp.

readonly LOGFILE="numero_file.log"
timestamp=$(date +"%Y-%m-%d %H:%M:%S")

cd  # si posiziona nella home directory

# "find ." scrive in standard output tutti i file contenuti
# dalla directory corrente in giù.
# "wc -l" riceve in standard input un testo e ne conta le
# righe; il numero viene stampato in standard output
# Il pipe (carattere "|") prende lo standard output di un
# primo comando e lo dà in pasto come standard input al
# secondo comando:

n_file=$(find . | wc -l)

# l'operatore ">>" aggiunge (append) lo standard output ad un file
echo "$timestamp - num. file nella home: $n_file" >> $LOGFILE
cd -  # torna nella directory precedente
```

Un consiglio importante: dare un'occhiata al man dei comandi wc, date e find, perché negli script vengono spesso utili.

Per concludere, come esercizio, è molto utile tentare di indovinare la differenza tra l'output dei due echo nello script che segue (e infine eseguirlo veramente):

```
#!/usr/bin/env bash
readonly stringa="ci     sono molti      spazi da tagliere"
echo "$stringa"
echo $(echo "$stringa")
```

Aritmetica e lettura di input da tastiera

È giunto il momento di vedere come fare semplici calcoli con i numeri interi. Ci sono due sintassi quasi equivalenti: dollaro seguito da doppie parentesi tonde e statement let, che ricorda alcuni antichi dialetti del linguaggio BASIC:

```
#!/usr/bin/env bash
a=3
b=4
let d=a*b
let x=b/d
c=$((a + b))
echo "c=$c"
echo "d=$d"
echo "x=$x"
```

(Per i pignoli: in realtà c'è una piccola differenza tra let e $(()): la prima istruzione comporta necessariamente un'assegnazione a una variabile, mentre la seconda fa il calcolo del valore di un'espressione aritmetica ed esegue subito la sostituzione dell'espressione con il valore appena calcolato prima dell'esecuzione della riga che la contiene.)

I calcoli espressi in questa maniera riguardano solo i numeri interi; infatti l'ultimo echo scrive x=0. Si tratta quindi di un modo comodo e rapido per contare file, linee, record su un DB o iterazioni all'interno di un ciclo, ma non appropriato per compiere calcoli più complicati.

Per superare il limite dei numeri interi, ci si appoggia ad un eseguibile che di solito è presente o è facilmente installabile su un sistema operativo serio: bc. Si tratta di una potentissima calcolatrice che può essere usata in maniera interattiva dal terminale, ma anche da script, se riceve istruzioni adeguate in standard input.
Supporta numeri a virgola mobile con un numero arbitrario di decimali, funzioni matematiche avanzate e tanto altro. Ad esempio, se da terminale si dà il comando

```
echo "scale=1000; 4*a(1)" | bc -l
```

vengono scritte le prime 1000 cifre del pi greco (e, per esercizio, con poco di più è possibile scrivere la rappresentazione del pi greco in base 16).

Ecco un esempio di come utilizzare bc da script (che usa lo statement read, per leggere il valore di una variabile da tastiera, o meglio, dallo standard input):

```
#!/usr/bin/env bash
# Teorema di Pitagora
# calcolo dell'ipotenusa con precisione di 10 decimali
echo -n "base? "
# lo statement read valorizza una variabile da standard input
read base
echo -n "altezza? "
read altezza
ipotenusa=$(echo "scale=10; sqrt($base^2 + $altezza^2)" | bc)
echo "ipotenusa=$ipotenusa"
```

Uso di if per l'esecuzione condizionata

Con l'ultimo esempio è stato introdotto lo statement read. Prima di vedere come utilizzare read per leggere ed elaborare un file di testo una linea alla volta, è necessario introdurre altri statement fondamentali del linguaggio. Partiamo da if, che permette di eseguire dei blocchi di codice in maniera condizionata.
Nota importantissima: nell'uso di questo statement è essenziale inserire gli spazi esattamente come specificato negli esempi (se provate ad esempio ad omettere lo spazio prima della chiusura di una parentesi quadra, sperimenterete un errore di sintassi).

Lo statement if si usa così:

```
if [ condizione1 ]; then
      echo "eseguito se la condizione1 è vera"
elif [ condizione2 ]; then # significa "else if"
      echo "eseguito se condizione1 è falsa ma condizione2 è vera"
else
      echo "eseguito se condizione1 e condizione2 sono false"
fi
```

dove i blocchi elif ed else non sono obbligatori.

Riguardo le possibili condizioni, Bash offre una grande varietà di strumenti, sia per valutare i valori delle variabili stringhe e numeriche, sia per lavorare su file e directory. Di seguito sono riportate le condizioni di uso più frequente per effettuare controlli su file e directory (per un elenco completo si rimanda alle numerose guide sul web):

```
[ -f FILE ] vale true se FILE esiste ed è un file regolare
[ -s FILE ] vale true se FILE esiste ed ha dimensione maggiore di zero
[ -d FILE ] vale true se FILE esiste ed è una directory
[ -r FILE ] vale true se FILE esiste ed ha il permesso di lettura
[ -w FILE ] vale true se FILE esiste ed ha il permesso di scrittura
[ -x FILE ] vale true se FILE esiste ed ha il permesso di esecuzione
```

La negazione, analogamente al linguaggio C, si esprime con un punto esclamativo, ma è necessario separare con uno spazio il punto esclamativo dal resto della condizione.
Per fare un esempio di uso di if, possiamo affinare lo script visto in precedenza che logga il numero di file nella home directory:

```
#!/usr/bin/env bash
readonly LOGFILE="numero_file.log"
timestamp=$(date +"%Y-%m-%d %H:%M:%S")
cd  # si posiziona nella home directory
n_file=$(find . | wc -l)
if [ ! -f "$LOGFILE" ]; then
       # crea per la prima volta il file di log, inserendo
       # una prima riga di intestazione
       echo "Numero di file nella home directory" > $LOGFILE
fi
echo "$timestamp - num. file: $n_file" >> $LOGFILE
cd -  # torna nella directory precedente
```

> Nota molto, molto importante: in questa guida viene fatto un uso ossessivo delle doppie virgolette per racchiudere i valori delle variabili. Ciò non significa effettuare una conversione a stringa, ma serve solo a fare l'escaping dei valori: in generale è sempre una buona prassi utilizzarle, specialmente con le variabili stringhe, in quanto il valore di una stringa potrebbe contenere spazi o tab o interruzioni di riga e in questi casi l'assenza di virgolette può produrre errori di sintassi quando l'interprete sostituisce $variabile con il suo valore. Tuttavia l'utilizzo delle virgolette anche attorno a variabili numeriche è una buona abitudine, in quanto permette di gestire meglio eventuali errori in caso di valori inattesi all'interno delle variabili.

Tornando all'if, esistono altri operatori che permettono di comparare due argomenti, stringhe o numerici. Supponiamo che a e b siano due interi e che s1 e s2 siano due stringhe.

```
if [ "$a" -eq "$b" ]; then
       echo "a e b sono uguali"
fi
if [ "$a" -ne "$b" ]; then
       echo "a e b sono diversi"
fi
# altri operatori per gli interi sono
# -gt per maggiore
# -ge per maggiore o uguale
# -lt per minore
# -le per minore o uguale
# (probabilmente i veterani del linguaggio Fortran
# avranno provato un brivido lungo la schiena)
```

ma torniamo alle stringhe:

```
if [ -z "$s1" ]; then
       echo "s1 è null, cioè la lunghezza è zero"
fi
if [ "$s1" == "$s2" ]; then
       echo "s1 e s2 hanno lo stesso contenuto"
fi
if [ "$s1" = "$s2" ]; then
       echo "s1 e s2 hanno lo stesso contenuto"
       # Sì, programmatori Java e C/C++, avete letto
       # bene: il risultato è il medesimo rispetto
       # allo statement if precedente!!! Brutto eh?
fi
# altri operatori per le stringhe sono
#
# != per verificare che s1 sia diversa da s2
# \< per verificare che s1 preceda alfabeticamente s2
#
# notare che "<" necessita di escaping: altrimenti,
# dentro il costrutto [ ], viene confuso con un redirect
# dello standard input.
```

Infine, ecco gli operatori and e or:

```
if [ "$a" -eq "0" -o "$b" -eq "0" ]; then
       echo "a è uguale a zero o b è uguale a zero"
fi
if [ "$a" -eq "0" -a "$b" -eq "0" ]; then
       echo "a e b sono uguali a zero"
fi
```

Realizzare loop con while, until e for

Come in tutti i linguaggi, anche in Bash è possibile effettuare dei loop. Ci sono vari tipi di cicli: quelli che si basano su una condizione e quelli che processano una lista di elementi. Nella prima categoria ci sono i cicli while e until.

Ciclo while

```
#!/usr/bin/env bash
i=0
# ripeti finché la condizione è vera
while [ "$i" -lt "10" ]; do
       echo "i=$i"
       let i=i+1
done
```

Ciclo until

```
#!/usr/bin/env bash
i=20
# ripeti finché la condizione è falsa
until [ "$i" -lt "10" ]; do
	echo "i=$i"
	let i-=1  # forma abbreviata di let i=i-1
done
```

In Bash c'è però un altro tipo di ciclo molto utile, che non ragiona per intervalli di interi o per condizioni come in C, ma funziona diversamente: si tratta del ciclo for, che può assumere varie forme. Il for si limita a esaminare uno dopo l'altro gli elementi presenti in una lista.
Spesso in Bash è conveniente inizializzare una lista direttamente dallo standard output di un altro comando. Esiste ad esempio il comando seq, che, usato nella maniera più semplice, scrive in standard output una serie di righe corrispondenti a tutti i numeri interi compresi nell'intervallo specificato dai due argomenti (estremi inclusi).
Dopo avere un po' sperimentato seq da linea di comando, si provi lo script seguente:

```
#!/usr/bin/env bash
for i in $(seq 1 10); do
	echo "i=$i"
done
```

Questo script fa capire esattamente come ragiona for. Viene preso ogni elemento dallo standard input, che accidentalmente in questo caso, avendo usato seq, è una lista di numeri interi. Per ogni elemento della lista viene eseguito il ciclo, valorizzando la variabile i. Si tratta di un meccanismo potentissimo che può essere usato con qualunque altro comando diverso da seq (compresi altri script!).
Lo script precedente è istruttivo, ma in realtà esiste un modo più comodo di realizzare cicli for su valori interi contigui, senza utilizzare lo standard output di un comando esterno:

```
#!/usr/bin/env bash
for i in {19..43}; do
	echo "i=$i"
done
```

Tornando alla possibilità di elaborare lo standard output di comandi, proviamo ad esempio ad utilizzare ls, cioè i nomi dei file nella directory corrente

```
#!/usr/bin/env bash
for nome_file in $(ls); do
	echo "nome_file=\"$nome_file\""
done
```

Provate ora a utilizzare questo ultimo script in una directory contenente almeno un file nel cui nome compare uno spazio. Il giochino si rompe, perché gli elementi da processare sono interpretati come separati da spazi, ma gli spazi sono presenti anche all'interno del nome di un file! Avremmo un risultato analogo se avessimo scritto *for nome_file in $(echo "pippo pluto")*. Ci sono due modi per superare il problema. Il primo è agire sul comando "ls": una possibilità è a separare tra loro i nomi dei singoli file con una interruzione di linea, usando "ls -1", per poi redirigere lo standard output su un file. A questo punto bisogna leggere il file una riga alla volta... operazione che vedremo nel prossimo capitolo. Complicato e non efficiente, ma funziona. Il secondo modo, più diretto e preferibile, è un uso differente di for, che non

abbiamo ancora visto: questo statement può infatti lavorare direttamente con i nomi dei file, cioè senza passare necessariamente per l'output di un comando come "ls":

```
#!/usr/bin/env bash
for nome_file in *; do
	echo "nome_file=\"$nome_file\""
done
```

Abbiamo usato "*" per indicare ogni file, ma avremmo potuto usare "*txt" o qualunque altra espressione per indicare un insieme di file. Molto potente e semplice. Più avanti nel documento, alla fine del capitolo "Lavorare con le stringhe", è riportato un esempio analogo concreto.

Lettura da standard input e lettura di file

Nelle sezioni precedenti è già stato introdotto l'uso di read, che permette di valorizzare una variabile con quanto inserito su una riga di standard input, cioè generalmente la tastiera.
Uniamo ora l'uso di read ad un'altra caratteristica di Bash che riguarda una condizione particolare che può essere utilizzata all'interno di if, while e until: in maniera analoga a quanto accade per altri linguaggi (chi ha programmato in C non ne rimarrà sorpreso), il fatto che una variabile sia valorizzata o meno può essere di per sé usato come espressione rispettivamente vera o falsa. Per rendersene conto, basta creare il file file_da_elaborare.txt, riempirlo con qualche riga di testo ed eseguire questo script:

```
#!/usr/bin/env bash
readonly NOME_FILE="file_da_elaborare.txt"
a=0
while read linea; do
	let a+=1
	echo "linea $a: \"$linea\""
done < "$NOME_FILE"
```

Questo è il modo più semplice per leggere un file una riga alla volta: un ciclo while con lo standard input rediretto da file e che ha come condizione di non uscita il fatto che la variabile linea sia valorizzata, cioè che ci sia ancora una linea da leggere nel file di testo. (Notare che la condizione logica così semplice nel while, permette di omettere le parentesi quadre.)

Supponiamo ora di essere in presenza di un file le cui linee sono formattate con regole certe, ad esempio i cui campi sono chiaramente delimitati da un separatore. Con l'aiuto del comando cut, possiamo elaborare le singole linee per estrarne una parte significativa. Eccone una dimostrazione banale: il comando

```
echo "Ciao cari programmatori" | cut -d' ' -f2
```

scrive "cari", cioè prende il secondo elemento (parametro -f2) delimitato da spazi (parametro -d).

Se i dati fossero stati separati da un carattere di tabulazione, che, come in molti altri linguaggi di programmazione, si esprime con la sintassi '\t', avremmo dovuto usare il parametro -d'\t'.

Passiamo adesso ad un esempio pratico. Supponiamo di volere monitorare per motivi di sicurezza l'IP di accesso ad una applicazione web. Siamo interessati ad analizzare il comportamento dell'utente applicativo "admin", l'amministratore. Supponiamo che l'applicativo web logghi ogni accesso in una tabella, comprensiva della data di accesso, l'IP da cui viene effettuato l'accesso e lo useragent del browser, cioè quella stringa che identifica il browser, la sua versione e il sistema operativo.

I dati di monitoraggio possono poi essere estratti dal DB con uno script Bash simile a questo

```
#!/usr/bin/env bash
readonly DB_IP="192.168.1.18"  # indirizzo del server di DB
readonly DB_USERNAME="utente"
readonly DB_PASSWORD="password"
readonly DB_SCHEMA="myschema"
readonly NOME_FILE="file_da_elaborare.txt"
oggi="$(date +"%Y-%m-%d")"

# in Bash, se si ha la necessità di interrompere una
# linea di testo, è sufficiente farla terminare con un
# carattere backslash "\"
sqlquery=" \
    use $DB_SCHEMA; \
    select \
        last_ip as 'ip', \
        useragent as 'useragent'
    from log_utente \
    where \
        data_login='$oggi'
        and
        username='admin';”

mysql --default-character-set=utf8 \
    -h "$DB_IP" -u "$DB_USERNAME" \
    -p"$DB_PASSWORD" --column-names \
    -e "$sqlquery" > "$NOME_FILE"
```

Si otterrà un file di testo contenente come prima riga la linea con i nomi delle due colonne, l'IP e lo useragent, e successivamente, se ci sono stati dei login, una serie di linee che hanno due valori, separati dal carattere di tabulazione.
Un monitoraggio di sicurezza molto elementare potrebbe essere questo: io, unico essere umano amministratore dell'applicativo web, che utilizzo sempre l'utente applicativo admin, voglio controllare che non ci siano anomalie riguardo chi si logga: chi entra come admin deve farlo da Linux, con Firefox aggiornato ad una precisa versione, cioè la mia. Sono tutti dati che si trovano nello usaragent del browser, che è costituito da una stringa simile a questa

```
Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:54.0) Gecko/20100101 Firefox/54.0
```

se l'account di admin venisse violato, ad esempio da un giovane hacker ignorante (che magari usa pure un sistema operativo proprietario!), uno script di controllo potrebbe trovare dei valori differenti di useragent e a questo punto potrebbe mandarmi automaticamente una email di allarme:

```
#!/usr/bin/env bash
readonly NOME_FILE="file_da_elaborare.txt"
readonly MY_UA="da_sostituire_con_lo_useragent_atteso"
numerolinea=0
while read linea; do
    # la prima linea contiene i nomi delle colonne
    if [ "$numerolinea" -gt "0" ]; then
        useragent=$(echo "$linea" | cut -f2 -d$'\t')
        if [ "$useragent" != "$MY_UA" ]; then
              ip=$(echo "$linea" | cut -f1 -d$'\t')
              messaggio="Accesso sospetto da ip $ip"
              # La parte che segue non è implementata per sintesi:
              # 1. loggare il messaggio in un posto sicuro, o
              # 2. inviarlo via http ad una API tipo REST, o
              # 3. inviarlo al proprio account di email...
        fi
    fi
    let numerolinea+=1
done < "$NOME_FILE"
```

Argomenti e valori di exit

Nel linguaggio Java e C gli argomenti passati ad un programma da linea di comando al momento dell'esecuzione sono leggibili come parametri rispettivamente del metodo pubblico main e della funzione main. In Bash lo stesso risultato si ottiene leggendo il valore delle seguenti variabili speciali:

$# contiene il numero di argomenti passati (0 se non ce ne sono)
$1 primo argomento
$2 secondo argomento
...eccetera.

Sono molto comuni gli script che elaborano un file il cui nome è passato come argomento:

```
#!/usr/bin/env bash
if [ $# -eq 0 ]; then
      echo "Errore: specificare il file da elaborare"
      exit 1
fi
nomefile="$1"
echo "inizio dell'elaborazione del file $nomefile"
# eccetera...
```

Il comando exit all'interno del blocco if permette di interrompere immediatamente l'esecuzione dello script e di uscire con uno stato di errore, che può assumere valori da 0 a 255. Per convenzione, zero significa nessun errore e un numero maggiore di zero significa errore.
Il fatto di terminare l'esecuzione con un valore / stato, avviene anche per la maggior parte dei comandi del sistema operativo (provare a consultare velocemente il man di alcuni comandi scelti a caso), gli stessi comandi che possono essere eseguiti all'interno di uno script. Per questo motivo, Bash offre un pratico meccanismo per recuperare il valore di stato dell'ultimo comando eseguito all'interno di uno script: la variabile $?
Usando questa variabile speciale si possono gestire eventuali errori:

```
# Supponiamo di essere all'interno di uno script che recupera
# periodicamente un file da remoto tramite il comando curl.
# Testare l'esito del download del file ha senso in quanto ci
# permette di terminare subito lo script, loggare l'errore ed
# eventualmente lanciare degli allarmi.
# Ovviamente MY_URL e DATA_FILE sono stati valorizzati in
# precedenza.
# ...
curl -s "$MY_URL" > "$DATA_FILE"
status=$?
if [ "$status" -ne "0" ]; then
      # logga il valore di $status,
      # scatena l'inferno, eccetera...
      exit 1
fi
# ...
```

Uso delle funzioni

Ecco come realizzare delle funzioni in Bash.

```
#!/usr/bin/env bash
# dichiarazione:
function funzioneciao() {
      echo "Ciao a tutti"
}
# invocazione:
funzioneciao
```

C'è una sintassi alternativa: Bash accetta la dichiarazione di funzioni anche senza la keyword "function": si tratta di una notazione più moderna. Tuttavia all'interno di questa guida sarà utilizzata sempre la notazione tradizionale.

L'ultima riga dell'esempio è la chiamata alla funzione. Le funzioni sono quindi richiamate esattamente come se fossero altri comandi del sistema operativo. Questo ha tre conseguenze interessanti:

1. le funzioni possono ricevere valori come parametri e dentro una funzione i parametri possono essere letti guarda a caso utilizzando le variabili $1, $2, $3, eccetera... proprio come uno script legge gli argomenti passati da linea di comando;
2. le funzioni hanno un valore di uscita (lo vedremo nel prossimo esempio), quasi come se fossero script eseguibili a sé e il loro valore di ritorno è contenuto nella variabile $?; quindi per chi le invoca non c'è differenza rispetto all'esecuzione di un altro programma eseguibile o script
3. esattamente come accade per i comandi del sistema operativo, lo standard output delle funzioni è assegnabile a variabili, se le si invocano dentro il costrutto $(nomefunzione).

Come esempio, scriviamo una funzione che estragga un numero casuale intero da x a y compresi:

```
#!/usr/bin/env bash
# La funzione deve essere invocata con due argomenti:
#  - estremo inferiore dell'intervallo
#  - estremo superiore dell'intervallo
# Poiché scrive il risultato, è consigliabile leggere
# tale valore utilizzando la sintassi $(random_number),
# come se fosse un comando del sistema operativo.
function random_number() {
	local -r min=$1
	local -r max=$2
	local range
	local risultato
	let range=max-min+1
	let risultato=$RANDOM%range+min
	echo $risultato
	return 0
}
for i in {1..10}; do
	n=$(random_number 10 20)
	echo "numero $i: $n"
done
```

Alcune puntualizzazioni (magari sono intuitive, ma è meglio essere precisi):

1. la variabile $RANDOM contiene un numero intero casuale, è un po' come invocare la funzione rand() in linguaggio C; con il solito trucchetto dell'operatore "modulo", rappresentato dal carattere percento, si ottiene il resto di una divisione per N (e di conseguenza il valore ottenuto ha N valori possibili);
2. lo statement local serve a definire lo scope locale di una variabile e il parametro "-r" serve a specificare che si tratta di una costante: se lo omettiamo, possiamo usare e anche sovrascrivere i valori di variabili fuori dalla funzione - un meccanismo a volte anche utile, ma da usare con parsimonia per evitare problemi;
3. per una funzione non è obbligatorio ritornare un valore (come neppure lo è in generale per uno script Bash); ad ogni modo, lo statement return serve proprio a questo scopo ed è stato inserito nell'esempio a mero scopo didattico, ma in questo caso poteva essere tranquillamente omesso.

Una conseguenza del punto 2: ecco un altro modo, estremamente deplorevole, per restituire un risultato da parte di una funzione:

```
#!/usr/bin/env bash
function miafunzione() {
	# valorizzo una variabile non locale
	risultato="un valore interessante"
}
miafunzione
echo $risultato
```

Infine, se abbiamo la certezza assoluta di essere in un caso in cui ha senso ritornare solo valori da 0 a 255, il risultato di una funzione può essere ritornato con return (da recuperare leggendo la variabile $?). Parere personale dell'autore: generalmente si tratta di una pratica sconsigliabile, poco lungimirante e neppure così elegante.

Costruire script suddivisi in più file

A volte può essere utile strutturare uno script o un'insieme di script in più file. Infatti può nascere la necessità di memorizzare alcune costanti utilizzate da più script in un unico file o di isolare per pulizia le funzioni che si occupano di un determinato aspetto (logging, accesso al DB, ecc...); l'operazione ha forti analogie a quanto si fa ad esempio nel linguaggio C con la direttiva di precompilatore include.
Creiamo dunque un normalissimo file, chiamato "costanti.src", con questo contenuto (una sola riga di testo):

```
readonly prova="variabile definita altrove"
```

Nota importantissima: il file appena creato non è di per sé uno script autonomo, infatti

1. non contiene il shebang iniziale;
2. non importa che abbia i permessi di esecuzione, anzi, non essendo autonomo è proprio meglio che non li abbia, così non potrà essere confuso con uno script vero e proprio.

Bene, ora creiamo uno script che lo utilizza. Quello di cui abbiamo bisogno è solamente uno statement: source. Questa direttiva permette di includere nello script corrente il codice scritto in altri file:

```
#!/usr/bin/env bash
source costanti.src
echo "$prova"
```

Sebbene non sia il caso di abusare di questa tecnica, è opportuno tenere presente che essa può in realtà essere molto utile non solo per memorizzare costanti, ma anche per definire librerie di funzioni comuni a più script. In questo modo è possibile incapsulare porzioni di codice in file dedicati e mantenere un certo ordine all'interno di un progetto di dimensioni rilevanti.

Un altro modo di spezzare il codice in più file è costruire più script indipendenti tra loro. Nulla vieta ad uno script di eseguirne un altro, magari intercettando pure il suo standard output con l'uso della sintassi $(script_esterno.sh) o redirigendolo su un file di testo che può poi essere elaborato dallo script chiamante.

Lavorare con le stringhe

In questa sezione vediamo come lavorare con le stringhe; le possibilità del linguaggio sono piuttosto vaste, ma in questa guida ci limitiamo alle operazioni di base: la determinazione della lunghezza, l'estrazione di sottostringhe e poco altro.
La lunghezza di una variabile stringa si ottiene con l'espressione ${#stringa}, mentre la sottostringa con le espressioni ${stringa:p1} e ${stringa:p1:p2}, dove p1 e p2 sono numeri interi e identificano la posizione da cui partire (il primo carattere è alla posizione numero 0) e la lunghezza della sottostringa che si vuole estrarre. La presenza nel parametro p1 di un valore negativo significa "dalla fine". Il secondo parametro è opzionale.

```
#!/usr/bin/env bash
stringa="abcdefghijklmnopqrstuvwxyz"
echo ${#stringa}         # scrive 26
echo ${stringa:0:7}      # scrive abcdefg
echo ${stringa:20}       # scrive uvwxyz
echo ${stringa:20:2}     # scrive uv
echo ${stringa:(-8)}     # scrive stuvwxyz
echo ${stringa:(-8):3}   # scrive stu
# notare: eventuali eccessi non vengono considerati:
echo ${stringa:100}      # scrive nulla
echo ${stringa:(-8):90}  # scrive stuvwxyz
```

Nota 1: l'uso delle parentesi davanti ai numeri negativi è importantissimo.

Nota 2: utilizzando le espressioni regolari e gli operatori # e % è possibile fare dei giochini molto interessanti, ma l'argomento è complesso e non viene trattato in questa guida sintetica. Per stuzzicare la fantasia del lettore, che può approfondire in rete l'argomento, ecco un esempio che può essere utile per trattare l'estensione di un file:

```
#!/usr/bin/env bash
nomefile="ilmiofile.txt.bak"
echo ${nomefile#*.}  # scrive txt.bak
echo ${nomefile%.*}  # scrive ilmiofile.txt
```

Un ultimo elemento essenziale: la conversione in caratteri maiuscoli e minuscoli di una stringa:

```
#!/usr/bin/env bash
s="Ciao a TUTTI, blablabla..."
s_up="${s^^}"
s_down="${s,,}"
echo "tutto maiuscolo: $s_up"
echo "tutto minuscolo: $s_down"
```

Concludiamo con un utilizzo pratico di queste ultime nozioni: un paio di piccoli script che mettono assieme anche quanto mostrato nei capitoli precedenti. Il primo degrada un po' delle immagini jpeg per ridurne la dimensione (utile ad esempio nei casi in cui si vogliano inviare diverse immagini per email). Il secondo converte una serie di immagini png in formato jpg, rinominandone anche l'estensione:

```
#!/usr/bin/env bash
if [ "$(which magick)" == "" ]; then
       echo "Errore: comando magick non trovato"
       exit 1
fi
for f in ../*jpg; do
       magick -quality 40% "$f" "${f:3}"
done
```

```
#!/usr/bin/env bash
if [ "$(which magick)" == "" ]; then
       echo "Errore: comando magick non trovato"
       exit 1
fi
for f in ../*png; do
       ofile=${f%.*}
       magick -quality 50% "$f" "${ofile:3}.jpg"
done
```

Entrambi gli script vanno posizionati ed eseguiti da una directory inferiore rispetto a quella che contiene le immagini da trattare e fanno uso del comando “magick”, che è parte di una serie di strumenti distribuiti sotto il nome di ImageMagick (nelle versioni più vecchio di ImageMagick il comando era “convert”). Nota importante: il funzionamento è garantito anche se i nomi dei file contengono spazi, in quanto si è utilizzata la sintassi di for che processa direttamente i nomi di file (la cosa era stata anticipata precedentemente alla fine del capitolo “Realizzare loop con while, until e for”). Se non si conosce il comando which, il man è sempre disponibile[2].

Un esempio quasi utile: generazione di galleria fotografica

È arrivato il momento di scrivere uno script che serva veramente a qualcosa e che metta insieme un po’ di cose studiate finora. Si tratta di uno script che esplora una sottocartella “immagini” della directory corrente. Per come è costruito questo script, tale cartella deve contenere solo dei file immagine. Questi file vengono utilizzati per creare una cosiddetta “galleria web”, cioè un file html nella directory corrente che contenga delle piccole miniature di anteprima cliccabili, che aprono i file originali. Lo script è molto rudimentale e, a seconda della codifica delle foto, potrebbe non riuscire a gestire il formato verticale. Ma con un po' di fantasia questo scheletro di script può diventare molto più completo e molto più utile e comunque rende l’idea della produttività di Bash, specialmente considerando che la metà delle linee di codice sono commenti e controlli per prevenire un uso scriteriato.
Questo script non contiene concetti nuovi, quindi se non si capisce qualche sua parte, è consigliabile rileggere le poche pagine che costituiscono questa guida fino a questo punto.

[2] considerare questa frase come una versione educata del famosissimo RTFM (https://it.wikipedia.org/wiki/RTFM)

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

readonly ORIGINALI="./immagini"
readonly HTMLPAGE="./index.html"

function html_write() {
	echo "$1" >> $HTMLPAGE
}

# scrive il pezzo di html per un'immagine
# parametro 1: nome del file di immagine originale
# parametro 2: nome del file della miniatura
function html_per_file() {
	local -r big_img="$1"
	local -r small_img="$2"
	html_write "<a href=\"$big_img\" target=\"_blank\">"
	html_write "<img src=\"$small_img\"/>"
	html_write "</a>  "
}

# controlli iniziali
if [ ! -d "$ORIGINALI" ]; then
	echo "Errore directory $ORIGINALI inesistente"
	exit 1
fi
if [ "$(which magick)" == "" ]; then
	echo "Errore: comando magick non disponibile."
	echo "Si consiglia l'installazione di imagemagick."
	exit 2
fi
n_immagini=$(ls -1 2>/dev/null "$ORIGINALI" | wc -l)
if [ "$n_immagini" -eq 0 ]; then
	echo "Errore: non c'è nulla da elaborare."
	exit 3
fi
n_miniature=$(ls -1 *.gif 2>/dev/null | wc -l)
if [ "$n_miniature" -gt "0" ]; then
	echo "Attenzione: sono presenti elaborazioni precedenti"
	read -p "Continuare ed eliminare ogni file gif? (s/n) " risposta
	if [ "$risposta" == "s" ]; then
		rm -f *.gif
		rm -f "$HTMLPAGE"
	else
		exit 4
	fi
fi

# elaborazione vera e propria
echo "$n_immagini elementi da elaborare"
i=1  # per il nome dei file di anteprima
html_write "<html><head><title>Indice di immagini</title>"
html_write "<style type=\"text/css\">"
html_write "A:link, A:visited { text-decoration: none }"
html_write "</style>"
html_write "</head>"
```

```
html_write "<body><center>"
for inputfile in "$ORIGINALI"/*; do
      outputfile="./$i.gif"
      magick "$inputfile" -resize 16384@ "$outputfile"
      html_per_file "$inputfile" "$outputfile"
      echo -n . # un puntino dietro l'altro, progressbar dei poveri
      let i+=1
done
echo # chiude la linea dei puntini
html_write "</center></body></html>"
echo "FINITO"
```

Come elaborare xml

Nei sistemi Linux/Unix esiste una grande varietà di comandi che sono utilizzabili per gli scopi più vari e utilizzandoli un semplice script può fare grandi cose. Ad esempio, in questo breve capitolo, grazie al comando xmllint, realizziamo un semplice lettore di news RSS dal sito dell'ANSA; in venti righe di codice. Le informazioni contenute in un feed RSS sono in formato xml, scaricabile dal web; pertanto abbiamo bisogno di un comando che scarichi le informazioni (qui viene usato curl, ma lo script può essere riadattato per utilizzare wget) e di un altro comando in grado di elaborare xml, cioè xmllint.

```
#!/usr/bin/env bash
readonly URL="https://www.ansa.it/sito/ansait_rss.xml"
readonly DATA_FILE="test.xml"


curl "$URL" > $DATA_FILE
echo
n_news=$(xmllint --xpath 'count(//rss/channel/item/title/text())' "$DATA_FILE")
echo "numero notizie: $n_news"
for i in $(seq 1 $n_news); do
    echo -n "NOTIZIA $i - "
    xpath_title="//rss/channel/item[$i]/title/text()"
    xpath_desc="//rss/channel/item[$i]/description/text()"
    xpath_link="//rss/channel/item[$i]/link/text()"
    titolo=$(xmllint --nocdata --xpath "$xpath_title" "$DATA_FILE")
    desc=$(xmllint --nocdata --xpath "$xpath_desc" "$DATA_FILE")
    link=$(xmllint --nocdata --xpath "$xpath_link" "$DATA_FILE")
    echo "Titolo: $titolo"
    echo "Descrizione: $desc"
    echo "URL: $link"
    echo
done
rm -f "$DATA_FILE"
```

Tipizzazione avanzata e array (cioè liste)

Abbiamo visto che le variabili in Bash non sono tipizzate, cioè è perfettamente lecito che il tipo di una variabile cambi nel tempo; prima un intero, poi una stringa.... Una delle conseguenze di questa caratteristica del linguaggio è che, per stare sempre sul sicuro, nella scrittura di espressioni logiche per if,

while, eccetera è buona pratica racchiudere i valori delle variabili tra doppie virgolette (una stringa può sempre contenere la rappresentazione di un numero mentre l'opposto non è vero), tanto alla fine ci pensa Bash a interpretare tutto nel modo più opportuno: almeno vengono evitati errori di sintassi quando il valore viene estratto dalla variabile mediante l'operatore "$".
La tipizzazione dinamica può essere tuttavia eliminata, se proprio lo si desidera (direi di no, ma nella vita non si sa mai), con lo statement declare. Ecco un esempio:

```
#!/usr/bin/env bash
# forzatura a numero intero
declare -i numero
numero=10
echo "numero=$numero"
numero="maiale"  # non funziona come ci aspettiamo
echo "numero=$numero"  # scrive 0
```

In verità il declare può avere una certa varietà di usi avanzati, non trattati in questa guida, come la definizione di puntatori a funzioni simili a quelli del linguaggio C; tuttavia nella pratica questo statement è più che altro utilizzato per un altro scopo: definire array o liste. Nel corso di questo testo le liste sono già state usate, anche se non esplicitamente e in maniera inconsapevole. Gli argomenti del ciclo for sono liste di valori, cioè array.
Per definire array con indici interi l'uso di declare non è obbligatorio, mentre lo è per gli array associativi; ma per uniformità e pulizia del codice, in questa guida se ne fa comunque uso, in entrambi i casi.
Vediamo prima un esempio di array con indicizzazione intera, cioè i cui elementi sono individuati da una chiave numerica.

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

# il "-a" serve a definire array con chiave numerica
declare -a MESE
MESE=( \
       "gennaio" "febbraio" "marzo" "aprile" "maggio" "giugno" \
       "luglio" "agosto" "settembre" "ottobre" "novembre" "dicembre" \
)

read -p "Inserire il numero del mese: " num
let num-=1  # gli elementi di un array iniziano da zero

# la sintassi ${variabile_array[$n]} estrae l'n-esimo valore da una lista
echo "il nome del mese corrispondente è ${MESE[$num]}"

# con questa sintassi ${#variabile_array[@]} si ottiene il
# numero di elementi presenti in un array
numero_di_mesi=${#MESE[@]}
echo "Numero totale di mesi: $numero_di_mesi"

echo "Ecco la lista completa dei mesi:"
# con la sintassi ${variabile_array[@]} si ottiene una lista utilizzabile
# all'interno di un ciclo for - attenzione però, tutto ciò funziona
# perfettamente solo perché gli elementi # nella lista sono composti da
# una sola parola
for mese in ${MESE[@]}; do
       echo -n "$mese "
done
echo  # andiamo a capo

# oppure avremmo potuto ottenere lo stesso risultato con # questo ciclo for
# (un po' meno performante perché passa per l'esecuzione del comando esterno
# seq per generare una lista di indici numerici da 0 a 11, ma immune dal
# problema di eventuali spazi presenti negli elementi in lista)
echo "Ecco una seconda lista (identica) dei mesi:"
for i in $(seq 0 $((numero_di_mesi - 1))); do
       mese=${MESE[i]}
       echo -n "$mese "
done
echo  # andiamo a capo

# oppure, se vogliamo solo scrivere la lista, senza avere un ciclo che può
# venire utile per elaborarne i valori, c'è una comoda scorciatoia inclusa
# nel linguaggio Bash
echo "Ecco una terza lista dei mesi:"
echo ${MESE[*]}
```

Passiamo ora agli array associativi: si tratta di array i cui elementi sono recuperabili non tramite un indice intero, ma usando altri valori arbitrari, definiti a piacere. Per definirli, bisogna obbligatoriamente utilizzare declare, ma con il parametro “-A”, in maiuscolo:

```
#!/usr/bin/env bash
# in questo esempio il declare è obbligatorio (provate a toglierlo)
declare -A DIZIONARIO
DIZIONARIO["gennaio"]="January"
DIZIONARIO["febbraio"]="February"
DIZIONARIO["marzo"]="March"
DIZIONARIO["aprile"]="April"
DIZIONARIO["maggio"]="May"
DIZIONARIO["giugno"]="June"
DIZIONARIO["luglio"]="July"
DIZIONARIO["agosto"]="August"
DIZIONARIO["settembre"]="September"
DIZIONARIO["ottobre"]="October"
DIZIONARIO["novembre"]="November"
DIZIONARIO["dicembre"]="December"
echo "Inserire il nome di un mese in italiano (lettere minuscole)"
read mese
echo "La traduzione in inglese è"
echo ${DIZIONARIO["$mese"]}
```

L'argomento array in Bash è piuttosto vasto e non è il caso di approfondirlo all'interno di questo documento. Per ulteriori approfondimenti avanzati sugli array, come ad esempio l'estrazione di sotto-array, rimozione di elementi, copia e concatenazione di array e molte altre perversioni che possono nascere dall'applicazione di espressioni regolari, fare riferimento alle seguenti pagine web:

- http://www.pluto.it/sites/default/files/ildp/guide/abs/arrays.html
- http://tldp.org/LDP/abs/html/arrays.html (versione inglese originale)

Blocchi case-esac e interfacce utente

Da un punto di vista teorico sono già stati coperti gli argomenti del linguaggio più frequentemente utilizzati. Rimangono da esplorare come si possono costruire semplici interfacce per l'interazione con gli utenti. Prima di vedere i relativi comandi, è utile apprendere l'uso di dei blocchi case-esac, che possono facilmente sostituire una lunga catena di if-elif-elif-...-elif-else-fi. La sintassi è questa:

```
case $espressione in
	valore1)
		# istruzioni eseguite se l'espressione vale valore1
		;;
	valore2)
		# istruzioni eseguite se l'espressione vale valore2
		;;
	valore3)
		# istruzioni eseguite se l'espressione vale valore3
		;;
	*)
		# eseguito se non vale nessuno dei precedenti
		# questo blocco è opzionale
		;;
esac
```

L'uso di un blocco case-esac può essere molto utile per elaborare una scelta multipla fatta dall'utente, magari letta con read.

Interfacce utente evolute

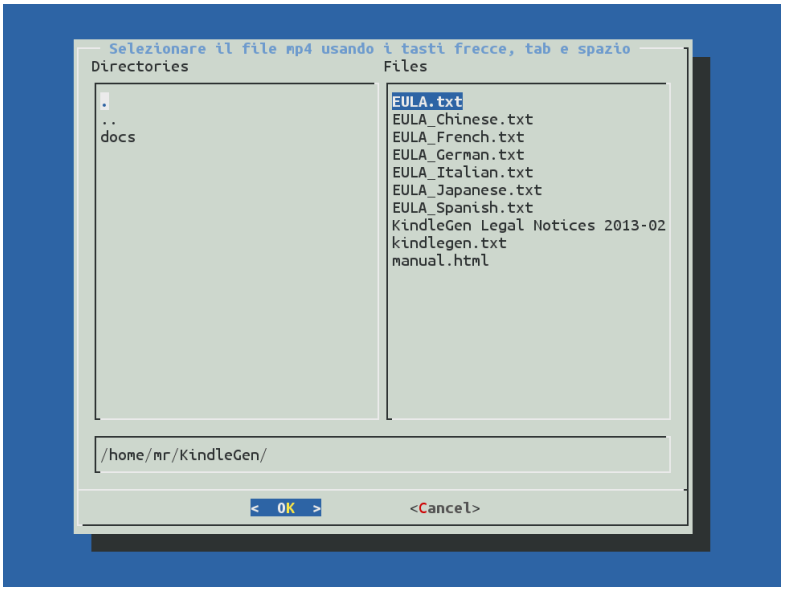
Ma passiamo all'interfaccia utente e vediamo tre tipologie di interfaccia dalla crescente complessità.
Ma prima tipologia di interfacce è molto semplice e permette di realizzare dei menu a scelta multipla, in modalità puramente testuale e senza l'uso dei colori; lo statement che consente tutto ciò è select e si basa sull'uso di array:

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

readonly DB="clienti.txt"
define -a MENU
MENU=( \
       "Ricerca cliente" \
       "Inserisci cliente" \
       "Cancella cliente" \
       "Uscita" \
)

if [ ! -f "$DB" ]; then
       touch "$DB"
       echo "Creato file database \"$DB\"."
       echo
fi

echo "GESTIONE ANAGRAFICA CLIENTI"
select scelta in "${MENU[@]}"; do
       case $scelta in
              "Ricerca cliente")
                     echo "Inserire una chiave di ricerca"
                     read query
                     echo "Risultati:"
                     grep -i "$query" "$DB"
                     ;;
              "Inserisci cliente")
                     echo "Funzione inserimento non ancora implementata"
                     ;;
              "Cancella cliente")
                     echo "Funzione cancellazione non ancora implementata"
                     ;;
              "Uscita")
                     echo "Cordiali saluti"
                     exit 0
                     ;;
       esac
done
```

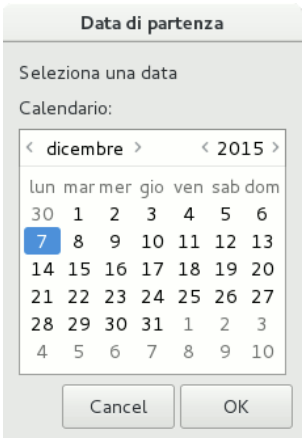
Si tratta di un'interfaccia davvero elementare e ha la controindicazione di funzionare solo se il contenuto di ogni elemento dell'array è perfettamente riportato nei blocchi all'interno di case-esac.

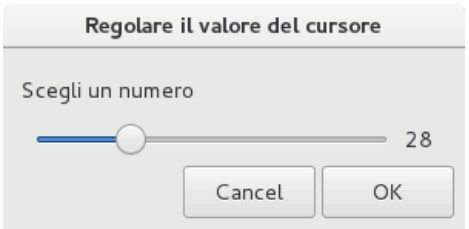
Esistono però altri modi più efficienti e più gradevoli per realizzare interfacce sia testuali sia grafiche in ambiente Wayland/X-Window.

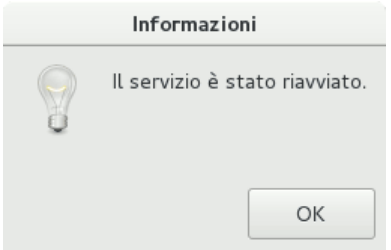
Per gli ambienti testuali, il consiglio è di studiare il comando dialog (spesso è già installato, altrimenti nelle distribuzioni più popolari esiste quasi sempre un pacchetto omonimo da installare). I risultati sono abbastanza soddisfacenti; come esempio, di seguito è riportata l'interfaccia per la selezione di un file:

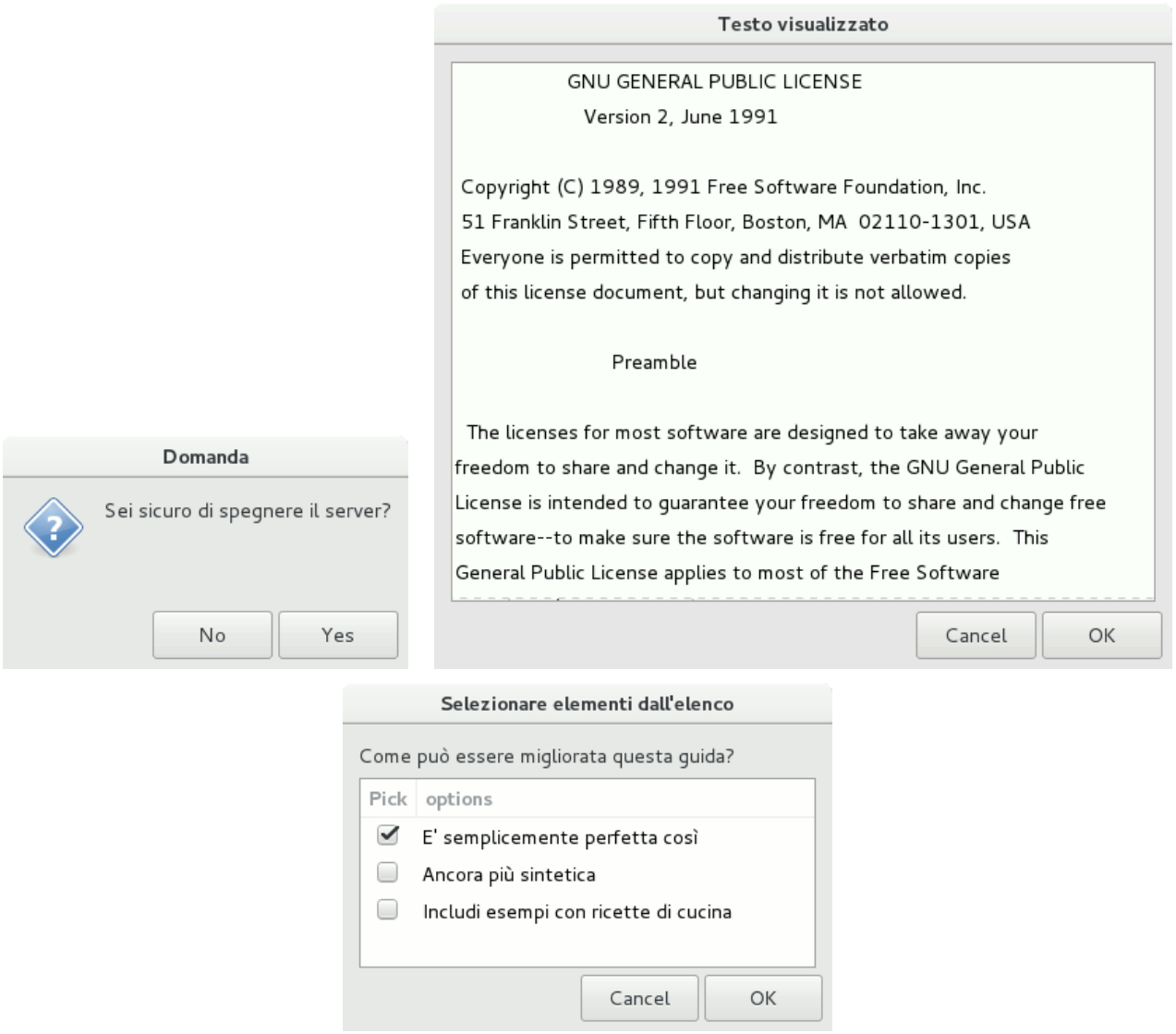


Se invece si ha la necessità di realizzare un'interfaccia grafica, si può scegliere zenity (anche questo è installato nella maggior parte delle distribuzioni o è facilmente installabile). Con zenity è molto facile ottenere delle finestre di dialogo standard, con cui costruire una rudimentale interfaccia utente basata su domande e richieste di inserimento dati. Ecco alcune finestre di dialogo facilmente ottenibili con zenity:



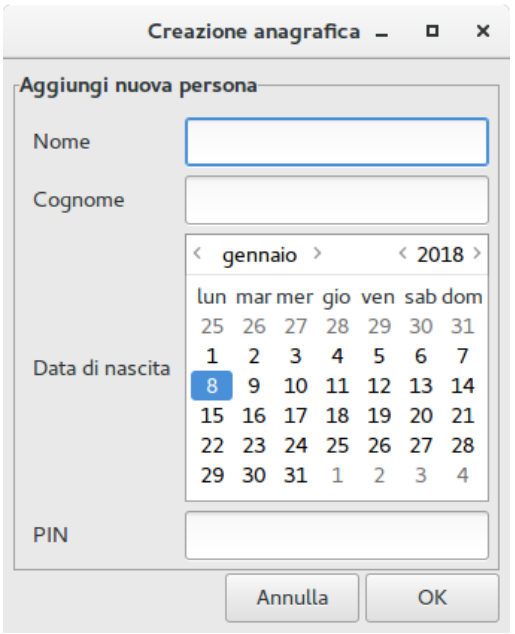






In generale, si tenga presente che zenity è uno strumento molto potente e le possibilità di utilizzo sono molto ampie.

Ad esempio è possibile definire nuovi dialog personalizzati: il comando che segue crea una finestra unica per l'inserimento in modo aggregato più dati riguardanti la stessa persona.




```
zenity --forms --title="Creazione anagrafica" \
       --text="Aggiungi nuova persona" \
       --add-entry="Nome" \
       --add-entry="Cognome" \
       --add-calendar="Data di nascita" \
       --add-password="PIN"
```

Un ultimo esempio

Solo un ultimo esempio contenente alcuni elementi nuovi che il lettore potrà approfondire autonomamente: uno script completo con interfaccia grafica fatta con zenity. Lo script processa un file contenente dati separati da tab. Si tratta di file molto diffusi, ad esempio per la pubblicazione degli open data da parte delle Pubbliche Amministrazioni, ma è anche il formato più comune dell'output dei comandi che eseguono direttamente query su DB (come il client testuale mysql). Lo script processa questi file e li converte in file CSV, che possono essere comodamente aperti con LibreOffice Calc (o se proprio preferite, Excel).
All'interno del codice viene creato un alias; in Bash gli alias permettono di definire velocemente un comando nuovo. In questo caso alias è utilizzato per redirezionare lo standard error ed eliminare i fastidiosi warning presenti in alcune versioni di zenity (ma se si verificano dei problemi, è consigliabile eliminare l'alias per capire cosa sta succedendo).
Nello script viene fatto un uso di read dentro un while abbastanza avanzato, che scompone una riga in un array in base alla variabile speciale IFS di Bash (IFS significa Internal Field Separator)... ovviamente in rete si può approfondire l'argomento a piacere.

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

readonly PROG_NAME="Convertitore CSV"

if [ "$(which zenity)" == "" ]; then
	echo "Errore: zenity non presente. Installare zenity."
	exit 10
fi
alias zenity="zenity 2>/dev/null"

istruzioni="$PROG_NAME\n\
Questo script elabora i file che contengono dati in formato tsv (Tab \
Separated Values) e li trasforma in file csv, facilmente consultabili \
con LibreOffice Calc o con MS Excel."
printf "$istruzioni" | zenity --text-info --title "$PROG_NAME" \
	--width 530 --height 400 2> /dev/null

input_filename="$(zenity --file-selection \
	--title="Seleziona un file con dati separati da tab" 2> /dev/null)"
if [ "$input_filename" == "" ]; then
	exit 0 # nessun file selezionato
fi
output_filename="$input_filename.csv"

sep=";"
sep_desc=""
while [ "$sep_desc" == "" ]; do
	sep_desc=$(zenity --title "$PROG_NAME" --list  --text \
		"Scegliere il separatore dei campi:" --radiolist  --column \
		"Pick" --column "Separatore" TRUE \
		"punto e virgola (italiano)" FALSE virgola 2> /dev/null)
done
if [ "$sep_desc" == "virgola" ]; then
	sep=","
fi

rm -f "$output_filename"
N_TAB=$(head -n 1 "$input_filename" | awk '{print gsub(/\t/,"")}')
count=0
while IFS=$'\t' read -r -a fieldarray; do
	echo -n . # scrive un puntino per ogni linea elaborata
	line=""
	for i in $(seq 0 $N_TAB); do
		line="$line\"${fieldarray[$i]}\""
		if [ "$i" -lt "$N_TAB" ]; then
			line="$line$sep"
		fi
	done
	echo $line >> "$output_filename"
	let count+=1
done < "$input_filename"
echo

zenity --info --text "generato il file \"$output_filename\"\n\
	$count linee elaborate." \
```

```
        --title "$PROG_NAME" 2> /dev/null
```

Ultimi trucchetti e conclusioni sintetiche finali

La guida finisce con questo capitoletto.
A questo punto dovreste essere sufficientemente dotati di conoscenza e malizia per studiarvi in rete quello che rimane del linguaggio e capire cose come il significato dell'operatore &&, che può sempre essere utile per comprendere ad esempio come mai alcuni script ricavino in questo modo il path della directory contenente lo script stesso:

```
readonly SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
```

E infine, due spunti decorativi.

Se eseguite script in ambiente grafico Wayland/X Window può essere utile fare uso delle notifiche desktop, ad esempio per comunicare all'utente che è terminato un lungo lavoro: a tale scopo si consideri l'uso del comando "notify-send".

Esistono modi per colorare nei terminali il testo e il suo sfondo e aggiungerci un minimo di formattazione: scritte in grassetto, lampeggianti, sottolineate,... basta cercare su internet cose tipo *ANSI/VT100 control sequences*: vi si aprirà un mondo di luccicanti opportunità decorative. Ecco un esempio di funzione che scrive un titolo utilizzando decorazioni colorate

```
#!/usr/bin/env bash
function title() {
        echo -en "\n\e[94m" # colore blu
        for i in $(seq 1 ${#1}); do
                echo -n "#"
        done
        echo -e "\e[0m"  # fine del colore blu
        echo -e "\e[92m\e[1m${1}\e[0m" # scritta in verde
        echo -ne "\e[94m" # colore blu
        for i in $(seq 1 ${#1}); do
                echo -n "#"
        done
        echo -e "\e[0m" # fine del colore blu
}

title "Il mio script"
echo Benvenuti nel mio script colorato...
```

Non è meraviglioso? Date un'occhiata qui per approfondire:
https://it.wikipedia.org/wiki/Codici_di_escape_ANSI

Ultimissima raccomandazione: la documentazione in rete è vostra amica, quindi mi raccomando, RTFM, senza offesa!
Grazie a tutti quelli che vorranno scrivermi commenti, suggerimenti e correzioni.