

Guida a Bash per Principianti

Machtelt Garrels

Garrels BVBA

<tille.wants.no.spam@garrels.be>

Version 1.11 Last updated 20081227 Edition

Tabella dei Contenuti

Introduzione

1. [Perché questa guida?](#)
2. [Chi dovrebbe leggere questo libro?](#)
3. [Nuove versioni, traduzioni e disponibilità](#)
4. [Storico delle Revisioni](#)
5. [Contributi](#)
6. [Feedback](#)
7. [Informazioni sul Copyright](#)
8. [Di cosa c'è bisogno?](#)
9. [Convenzioni usate in questo documento](#)
10. [Organizzazione di questo documento](#)
1. [Bash e script Bash](#)
 - 1.1. [Programmi shell comuni](#)
 - 1.2. [Vantaggi della Bourne Again SHell](#)
 - 1.3. [Esecuzione dei comandi](#)
 - 1.4. [Blocchi di costruzione](#)
 - 1.5. [Sviluppare degli ottimi script](#)
 - 1.6. [Sommario](#)
 - 1.7. [Esercizi](#)
2. [Scrittura e debug di script](#)
 - 2.1. [Creazione ed esecuzione di uno script](#)
 - 2.2. [Nozioni di base sugli script](#)
 - 2.3. [Debugging di script Bash](#)
 - 2.4. [Sommario](#)
 - 2.5. [Esercizi](#)
3. [L'ambiente Bash](#)
 - 3.1. [File di inizializzazione della shell](#)
 - 3.2. [Variabili](#)
 - 3.3. [Testo citato](#)
 - 3.4. [espansione della shell](#)
 - 3.5. [Gli alias](#)
 - 3.6. [Altre opzioni per Bash](#)
 - 3.7. [Sommario](#)
 - 3.8. [Esercizi](#)
4. [Espressioni regolari](#)
 - 4.1. [Espressioni regolari](#)
 - 4.2. [Esempi di utilizzo di grep](#)
 - 4.3. [Pattern matching con le funzioni Bash](#)
 - 4.4. [Sommario](#)
 - 4.5. [Esercizi](#)

5. [Sed, l'editor di stream di GNU](#)
 - 5.1. [Introduzione](#)
 - 5.2. [Modifica interattiva](#)
 - 5.3. [Modifica non interattiva](#)
 - 5.4. [Sommario](#)
 - 5.5. [Esercizi](#)
6. [Awk, il linguaggio di programmazione di GNU](#)
 - 6.1. [Iniziare con gawk](#)
 - 6.2. [Il programma di stampa](#)
 - 6.3. [Variabili gawk](#)
 - 6.4. [Sommario](#)
 - 6.5. [Esercizi](#)
7. [Istruzioni condizionali](#)
 - 7.1. [Introduzione a "if"](#)
 - 7.2. [Altri usi avanzati di "if"](#)
 - 7.3. [Uso di "case"](#)
 - 7.4. [Sommario](#)
 - 7.5. [Esercizi](#)
8. [Scrivere script interattivi](#)
 - 8.1. [Mostrare i messaggi utente](#)
 - 8.2. [Catturare l'input dell'utente](#)
 - 8.3. [Sommario](#)
 - 8.4. [Esercizi](#)
9. [Lavori ripetitivi](#)
 - 9.1. [Il ciclo "for"](#)
 - 9.2. [Il ciclo "while"](#)
 - 9.3. [Il ciclo "until"](#)
 - 9.4. [Re-indirizzamento dell'I/O e i cicli](#)
 - 9.5. ["Break" e "continue"](#)
 - 9.6. [Creare menù col "select" nativo](#)
 - 9.7. [Lo "shift" nativo](#)
 - 9.8. [Sommario](#)
 - 9.9. [Esercizi](#)
10. [Approfondimenti sulle variabili](#)
 - 10.1. [Tipi di variabili](#)
 - 10.2. [Array di variabili](#)
 - 10.3. [Operazioni sulle variabili](#)
 - 10.4. [Sommario](#)
 - 10.5. [Esercizi](#)
11. [Funzioni](#)
 - 11.1. [Introduzione](#)
 - 11.2. [Esempi di funzioni negli script](#)
 - 11.3. [Sommario](#)
 - 11.4. [Esercizi](#)
12. [Catturare i segnali](#)
 - 12.1. [I segnali](#)
 - 12.2. [I comandi "trap"](#)
 - 12.3. [Sommario](#)
 - 12.4. [Esercizi](#)
- A. [Funzionalità della shell](#)
 - A.1. [Funzionalità comuni](#)
 - A.2. [Funzionalità diverse](#)

[Glossario](#)

[Indice](#)

Elenco delle Tabelle

1. [Convenzioni tipografiche e stilistiche](#)
- 1-1. [Panoramica dei termini di programmazione](#)
- 2-1. [Panoramica del set di opzioni di debug](#)
- 3-1. [Variabili Bourne shell riservate](#)
- 3-2. [Variabili Bash riservate](#)
- 3-3. [Variabili bash speciali](#)
- 3-4. [Operatori aritmetici](#)
- 4-1. [Operatori per le espressioni regolari](#)
- 5-1. [Comandi di editing di "sed"](#)
- 5-2. [Opzioni per "sed"](#)
- 6-1. [Caratteri di formattazione per "gawk"](#)
- 7-1. [Espressioni primarie](#)
- 7-2. [Combinare le espressioni](#)
- 8-1. [Sequenze si escape utilizzate dal comando "echo"](#)
- 8-2. [Opzioni per il "read" nativo](#)
- 10-1. [Opzioni per il "declare" nativo](#)
- 12-1. [Segnali di controllo in Bash](#)
- 12-2. [Segnali comuni per il "kill"](#)
- A-1. [Caratteristiche Comuni della Shell](#)
- A-2. [Diverse funzionalità della Shell](#)

Elenco delle Figure

1. [Prima copertina di Guida Bash per Principianti](#)
 - 2-1. [script1.sh](#)
 - 3-1. [Prompt diversi per utenti diversi](#)
 - 6-1. [I campi in awk](#)
 - 7-1. [Test di un argomento della riga di comando con "if"](#)
 - 7-2. [Esempio con gli operatori Booleani](#)
-

Introduzione

1. Perché questa guida?

Il motivo principale per scrivere questo documento è che molti lettori ritengono che l'[HOWTO](#) esistente sia troppo breve e incompleto, mentre la guida [Bash Scripting](#) è troppo per essere un riferimento. Non c'è niente tra questi due estremi. Ho inoltre scritto questa guida sul principio generale che non sono disponibili abbastanza corsi gratuiti di base, anche se ci dovrebbero essere.

Questa è una guida pratica che, pur non essendo sempre troppo seria, cerca di fornire esempi reali piuttosto che teorici. L'ho scritto in parte perché non mi entusiasmano gli esempi ridotti e semplificati scritti da persone che sanno di cosa stanno parlando, mostrando alcune funzionalità davvero interessanti di Bash così fuori dal suo contesto che in pratica non si possono mai usarle. Questo genere di cose si potranno leggere dopo aver finito questo libro, che contiene esercizi ed esempi che aiuteranno a sopravvivere nel mondo reale.

Dalla mia esperienza di utente UNIX/Linux, amministratore di sistema e docente, so che le persone possono avere anni di interazione quotidiana con i loro sistemi, senza avere la minima conoscenza dell'automazione delle attività. Quindi spesso pensano che UNIX non sia facile da usare e, peggio ancora, hanno l'impressione che sia lento e antiquato. Questo è un altro problema che può essere risolto da questa guida.

2. Chi dovrebbe leggere questo libro?

Tutti coloro che lavorano su un sistema UNIX o simile a UNIX che vogliono semplificarsi la vita, sia gli utenti esperti che gli amministratori di sistema, possono trarre vantaggio dalla lettura di questo libro. I lettori che hanno già una conoscenza del funzionamento del sistema utilizzando la "riga di comando" impareranno i dettagli dello "shell scripting" che facilitano l'esecuzione delle attività quotidiane. L'amministrazione del sistema si basa molto sugli script di shell; le comuni attività sono spesso automatizzate utilizzando semplici script. Questo documento è pieno di esempi che incoraggeranno a scriverne di propri e che ispireranno a migliorare quelli esistenti.

Prerequisiti/non in questo corso:

- Si dovrebbe essere un utente UNIX o esperto Linux, avere familiarità con i comandi di base, le pagine "man" e la documentazione
- Essere in grado di utilizzare un editor di testo
- Comprendere i processi di avvio e arresto del sistema, init e initscript
- Creare utenti e gruppi, impostare password
- Permessi, modalità speciali
- Conoscere le convenzioni di denominazione per device, il partizionamento, mounting/unmounting di file system
- Aggiunta/rimozione di software sul sistema

Vedere [Introduction to Linux](#) (o il proprio [TLDP mirror](#) più vicino) se si ignora uno o più di questi argomenti. Ulteriori informazioni si possono trovare nella documentazione del sistema (pagine man e info) o su [the Linux Documentation Project](#).

3. Nuove versioni, traduzioni e disponibilità

L'edizione più recente si può trovare su <http://tille.garrels.be/training/bash/>. La stessa versione si dovrebbe trovare su <http://tldp.org/LDP/Bash-Beginners-Guide/html/index.html>.

Questa guida è disponibile in stampa da [Fultus.com](#).

Figura 1. Prima copertina di Guida Bash per Principianti



Questa guida è stata tradotta:

- La traduzione cinese su <http://xiaowang.net/bgb-cn/>, è di Wang Wei.
- La traduzione in ucraino su http://docs.linux.org.ua/index.php/LDP:Bash_beginners_guide, è di Yaroslav Fedevych e del suo team.
- La traduzione italiana su <https://github.com/BravoBaldo/Translations> è di Baldassarre Cesarano.

Una traduzione francese è in preparazione e sarà linkata appena sarà terminata.

4. Storico delle Revisioni

Storico delle Revisioni

Revision 1.11	2008-12-27	Revisionato da: MG
Input elaborato dai lettori.		
Revision 1.10	2008-06-06	Revisionato da: MG
cambio indirizzo		
Revision 1.9	2006-10-10	Revisionato da: MG
Inclusi i commenti dei lettori, aggiunto l'indice utilizzando i tag DocBook.		
Revision 1.8	2006-03-15	Revisionato da: MG
chiarito esempio nel Cap. 4, qui corretto doc nel cap. 9, controlli generali e correzione degli errori di battitura, aggiunto link alla traduzione cinese e ucraina, note e cose da sapere su awk nel cap. 6.		
Revision 1.7	2005-09-05	Revisionato da: MG
Correzioni nei capitoli 3, 6 e 7, inclusi i commenti dei lettori, aggiunta nota nel chap. 7.		
Revision 1.6	2005-03-01	Revisionato da: MG
Correzioni marginali, aggiunte altre keyword, info sul nuovo Bash 3.0, eliminata immagine vuota.		
Revision 1.0	2004-04-27	Revisionato da: TM
Versione iniziale per LDP; altri esercizi, più markup, meno errori e abusi; aggiunto glossario.		
Revision 1.0-beta	2003-04-20	Revisionato da: MG
Pre-release		

5. Contributi

Un grazie a tutti gli amici che ci hanno aiutato (o ci hanno provato) e a mio marito; le tue parole incoraggianti hanno reso possibile questo lavoro. Grazie a tutte le persone che hanno segnalato bug, esempi e commenti - tra molti, molti altri:

- Hans Bol, uno dei fan
- Mike Sim, note sullo stile
- Dan Richter, per gli esempi di array
- Gerg Ferguson, per le idee sul titolo
- Mendel Leo Cooper, per fare spazio
- #linux.be, per avermi tenuto con i piedi per terra
- Frank Wang, per i suoi dettagliati commenti su tutte le cose che ho sbagliato ;-)

Un ringraziamento speciale a Tabatha Marshall, che si è offerta volontaria per fare una revisione completa e un controllo ortografico e grammaticale. Formiamo una grande squadra: lei lavora quando io dormo. E viceversa ;-)

6. Feedback

Informazioni, link, caratteri, commenti mancanti? Inviare una mail a

[<tille_wants_no_spam_at_garrels_dot_be>](mailto:tille_wants_no_spam_at_garrels_dot_be)

il curatore di questo documento.

7. Informazioni sul Copyright

```
* Copyright (c) 2002-2007, Machtelt Garrels
* Tutti i diritti riservati.
* La redistribuzione e l'uso in forma di codice sorgente e in forma binaria,
  con o senza
* modifiche, è consentito purché siano rispettate le seguenti condizioni:
*
*   * Le redistribuzioni del codice sorgente devono conservare la nota di
  copyright sopra
*   riportata, questa lista di condizioni e la seguente limitazione di
  responsabilità.
*   * Le redistribuzioni in forma binarie devono riprodurre la nota di
  copyright
*   sopra riportata, questa lista di condizioni e la seguente limitazione
  di
*   responsabilità nella documentazione e/o altri materiali forniti con la
  distribuzione.
*   * Né il nome dell'autore, Machtelt Garrels, né i nomi dei suoi
  collaboratori possono
*   essere utilizzati per avallare o promuovere prodotti derivati da questo
  software
*   senza uno specifico permesso scritto..
*
* QUESTO SOFTWARE È FORNITO DALL'AUTORE E DAI CONTRIBUTORI "COSÌ COM'È" E
  QUALSIASI
* GARANZIA ESPRESSA O IMPLICITA, INCLUSIVA DI, MA NON LIMITATA A, GARANZIE
  IMPLICITE
* DI COMMERCIALIZZABILITÀ E IDONEITÀ AD UNO SCOPO PARTICOLARE, VIENE DISCONOSCIUTA.
  IN NESSUN CASO IL DETENTORE DEL COPYRIGHT SARÀ RITENUTO RESPONSABILE PER
  QUALSIASI DANNO DIRETTO,
* INDIRETTO, CONNESSO, PARTICOLARE, ESEMPLARE O CONSEGUENTE (INCLUSIVO DI, MA
  NON LIMITATO A,
* APPROVVIGIONAMENTO DI BENI O SERVIZI ALTERNATIVI; PERDITA DI UTILITÀ, DATI O
  PROFITTI; INTERRUZIONE DI AFFARI)
* COMUNQUE CAUSATI E SU QUALSIASI IPOTESI DI RESPONSABILITÀ, COME DA CONTRATTO,
  RESPONSABILITÀ
* OGGETTIVA, O TORTO (COMPRESA NEGLIGENZA O ALTRO) DERIVANTE IN QUALSIASI MODO
  DALL'UTILIZZO DI QUESTO
* SOFTWARE ANCHE SE AL CORRENTE DELLA POSSIBILITÀ DI TALE DANNO.
```

L'autore e l'editore hanno compiuto ogni sforzo nella preparazione di questo libro per garantire l'accuratezza delle informazioni. Tuttavia, le informazioni contenute in questo libro sono offerte senza garanzia, esplicita o implicita. Né l'autore, né l'editore, né alcun rivenditore o distributore saranno ritenuti responsabili per eventuali danni causati o presumibilmente causati direttamente o indirettamente da questo libro.

I loghi, i marchi e i simboli utilizzati in questo libro sono di proprietà dei rispettivi proprietari.

8. Di cosa c'è bisogno?

bash, disponibile su <http://www.gnu.org/directory/GNU/>. La shell Bash è disponibile su quasi tutti i sistemi Linux e oggi giorno si può trovare su un'ampia varietà di sistemi UNIX.

Si compila facilmente se c'è bisogno di crearne uno proprio, testato su un'ampia varietà di sistemi UNIX, Linux, MS Windows e altri.

9. Convenzioni usate in questo documento

In questo testo si seguono le seguenti convenzioni tipografiche e stilistiche:

Tabella 1. Convenzioni tipografiche e stilistiche

Tipo di testo	Significato
"Testo citato"	Citazioni di persone, output del computer in esame.
vista terminale	Input e output testuale del computer catturati dal terminale, di solito reso con uno sfondo grigio chiaro.
comando	Nome di un comando da immettere sulla riga di comando.
VARIABLE	Nome di una variabile o puntatore al contenuto di una variabile, come in \$VARIABLE.
opzione	Opzione per un comando, come in "l'opzione -a al comando ls ".
argomento	L'argomento di una comando, come in "leggere man ls ".
comandi opzioni argomenti	Sinossi del comando e uso generale, su una riga separata.
nomefile	Nome di un file o di una directory, per esempio "Spostarsi nella directory /usr/bin".
Tasto	Tasti da premere sulla tastiera, come "digitare Q per uscire".
Pulsante	Pulsante grafico da cliccare, come il pulsante OK.
Menù->Scelta	Scelta da selezionare da un menù grafico, per esempio: "Select Help->About Mozilla nel browser."
Terminologia	Termine o concetto importante: "Il <i>kernel</i> Linux è il cuore del sistema".
\	Il backslash nella schermata di un terminale o nella sinossi del comando indica una riga non finita. In altre parole, se si vede un lungo comando suddiviso in più righe, \ indica "Non premere ancora Enter !"
Vedere Capitolo 1	link al relativo argomento in questa guida.
L'autrice	Link cliccabile verso una risorsa web esterna.

10. Organizzazione di questo documento

Questa guida discute concetti utili nella vita quotidiana dell'utente serio di Bash. Sebbene sia richiesta una conoscenza di base sull'uso della shell, iniziamo con una discussione delle componenti e delle pratiche di base della shell nei primi tre capitoli.

I capitoli da quattro a sei trattano gli strumenti di base comunemente usati negli script di shell.

I capitoli da otto a dodici trattano i costrutti più comuni negli script di shell.

Tutti i capitoli sono accompagnati da esercizi per testare la preparazione per il capitolo successivo.

- [Capitolo 1](#): Basi di Bash: perché Bash è così buono, blocchi costitutivi, prime linee guida per lo sviluppo di buoni script.
- [Capitolo 2](#): Nozioni di base sugli script: scrittura e debug.
- [Capitolo 3](#): L'ambiente Bash: file di inizializzazione, variabili, virgolette, ordine di espansione della shell, alias, opzioni.
- [Capitolo 4](#): Espressioni regolari: un'introduzione.
- [Capitolo 5](#): Sed: una introduzione all'editor di riga sed.
- [Capitolo 6](#): Awk: introduzione al linguaggio di programmazione awk.
- [Capitolo 7](#): comandi condizionali: costrutti usati in Bash per testare le condizioni.
- [Capitolo 8](#): Script interattivi: creare script di facile utilizzo, intercettando l'input dell'utente.
- [Capitolo 9](#): Esecuzione di comandi in modo ripetitivo: costrutti utilizzati in Bash per automatizzare l'esecuzione dei comandi.
- [Capitolo 10](#): Variabili avanzate: specificare i tipi delle variabili, introduzione agli array di variabili, operazioni sulle variabili.
- [Capitolo 11](#): Funzioni: un'introduzione.
- [Capitolo 12](#): Cattura dei segnali: introduzione al processo di signalling, cattura dei segnali inviati dall'utente.

Capitolo 1. Bash e script Bash

In questo modulo introduttivo

- Vengono descritte alcune comuni shell
- Si indicano i vantaggi e le caratteristiche della GNU Bash
- Si descrivono i blocchi della shell
- Si parla dei file di inizializzazione di Bash
- Si esamina come la shell esegue i comandi
- Si esaminano alcuni semplici esempi di script

1.1. Programmi shell comuni

1.1.1. Funzioni shell generali

Il programma shell UNIX interpreta i comandi utente, che vengono immessi direttamente dall'utente o che possono essere letti da un file chiamato script shell o programma shell. Gli script shell vengono interpretati, non compilati. La shell legge i comandi dalla riga dello script per ciascuna riga e cerca quei comandi nel sistema (vedere la [Sezione 1.2](#)), mentre un compilatore converte un programma in un formato leggibile dalla macchina, un file eseguibile - che può quindi essere utilizzato in uno script di shell.

Oltre a passare comandi al kernel, il compito principale di una shell è fornire un ambiente utente, che possa essere configurato individualmente utilizzando i file di configurazione delle risorse della shell.

1.1.2. Tipi di shell

Proprio come le persone conoscono diverse lingue e dialetti, il sistema UNIX di solito offre una varietà di tipi di shell:

- **sh** o Bourne Shell: la shell originale è ancora usata nei sistemi UNIX e in ambienti ad esso correlati. Questa è la shell di base, un piccolo programma con poche funzionalità. Sebbene non sia la shell standard, è ancora disponibile su tutti i sistemi Linux per la compatibilità con i programmi UNIX.
- **bash** o Bourne Again shell: è la shell standard GNU, intuitiva e flessibile. Probabilmente quella più consigliabile per i principianti pur essendo allo stesso tempo uno strumento potente per l'utente avanzato e professionale. Su Linux, **bash** è la shell standard per i normali utilizzi. Questa shell è un cosiddetto *superset* della Bourne shell, un insieme di componenti aggiuntivi e plug-in. Ciò significa che la shell Bourne Again è compatibile con la shell Bourne: i comandi che funzionano in **sh**, funzionano anche in **bash**. Non è, tuttavia, sempre vero il contrario. Tutti gli esempi e gli esercizi in questo libro usano **bash**.
- **csh** o C shell: la sintassi di questa shell ricorda quella del linguaggio di programmazione C. A volte è richiesto dai programmatori.
- **tcsh** o TENEX C shell: un superset della comune C shell, che ne migliora la facilità d'uso e la velocità. Ecco perché alcuni lo chiamano anche Turbo C shell.
- **ksh** o la Korn shell: a volte apprezzata da persone con un background UNIX. Un superset della Bourne shell; con la configurazione standard è un incubo per i principianti.

Il file `/etc/shells` fornisce una panoramica delle shell note nel sistema Linux:

```
mia:~> cat /etc/shells
/bin/bash
/bin/sh
/bin/tcsh
/bin/csh
```

La propria shell di default shell è impostata nel file `/etc/passwd`, come questa riga per l'utente *mia*:

```
mia:L2NOFqdlPrHwE:504:504:Mia Maya:/home/mia:/bin/bash
```

Per passare da una shell all'altra basta inserire il nome della nuova shell nel terminale attivo. Il sistema trova la directory in cui si trova il nome leggendo la variabile `PATH` e poiché una shell è un file eseguibile (programma), la shell corrente lo attiva e la esegue. Di solito viene mostrato un nuovo prompt, perché ogni shell ha il suo aspetto tipico:

```
mia:~> tcsh
[mia@post21 ~]$
```

1.2. Vantaggi della Bourne Again SHell

1.2.1. Bash è la shell di GNU

Il progetto GNU (GNU's Not UNIX) fornisce strumenti per l'amministrazione di sistemi simili a UNIX come software libero e conformi agli standard UNIX.

Bash è una shell compatibile con sh che ingloba utili funzionalità della Korn shell (ksh) e della C shell (csh). È concepito per essere conforme allo standard per Shell e Tools IEEE POSIX P1003.2/ISO 9945.2. Offre migliorie funzionali rispetto a sh sia per la programmazione che per l'uso interattivo; tra cui la modifica della riga di comando, la cronologia dei comandi di dimensioni illimitate, il controllo degli job, le funzioni e gli alias della shell, gli array indicizzati di dimensioni illimitate e l'aritmetica degli interi in qualsiasi base da due a sessantaquattro. Bash può eseguire la maggior parte degli script sh senza modifiche.

Come gli altri progetti GNU, l'iniziativa bash è stata avviata per preservare, proteggere e promuovere la libertà di utilizzare, studiare, copiare, modificare e ridistribuire il software. È generalmente noto che tali condizioni stimolano la creatività. Questo è stato anche il caso del programma bash, che ha molte funzionalità extra che altre shell non offrono.

1.2.2. Funzionalità che si trovano solo in bash

1.2.2.1. Invocazione

Oltre alle opzioni a riga di comando della shell a carattere singolo che generalmente possono essere configurate utilizzando il comando integrato nella shell **set**, sono utilizzabili diverse opzioni a più caratteri. Ci impareremo in alcune delle opzioni più note in questo e nei prossimi capitoli; l'elenco completo si trova nelle pagine di informazioni di Bash, Funzionalità di Bash->Invocazione di Bash.

1.2.2.2. File di avvio di Bash

I file di avvio sono script che vengono letti ed eseguiti da Bash al suo avvio. Le sottosezioni seguenti descrivono diversi modi per avviare la shell e, di conseguenza, i file di avvio che vengono letti.

1.2.2.2.1. Richiamato come shell interattiva di login, o con '--login'

Interattivo significa che si possono inserire i comandi. La shell non è in esecuzione perché è stato attivato uno script. Shell di login significa che è stata ottenuta la shell dopo essersi stati autenticati dal sistema, di solito fornendo il nome utente e la password.

File letti:

- /etc/profile
- ~/.bash_profile, ~/.bash_login or ~/.profile: viene letto il primo file leggibile esistente
- ~/.bash_logout al logout.

I messaggi di errore vengono stampati se esistono i file di configurazione ma non sono leggibili. Se un file non esiste, bash cerca il successivo.

1.2.2.2.2. Richiamato come shell di non-login interattiva

Shell di non-login significa che non è necessario essere autenticati nel sistema. Ad esempio, quando si apre un terminale utilizzando un'icona o una voce di menù, si tratta di una shell di non-login.

File letti:

- `~/ .bashrc`

Di solito si fa riferimento a `~/ .bash_profile`:

```
if [ -f ~/ .bashrc ]; then . ~/ .bashrc; fi
```

Vedere [Capitolo 7](#) per ulteriori informazioni sul costrutto **if**.

1.2.2.2.3. Invocato non-interattivamente

Tutti gli script utilizzano shell non interattive. Sono programmati per svolgere determinati compiti e non possono essere istruiti per svolgere altri lavori oltre a quelli per cui sono programmati.

File letti:

- definito da `BASH_ENV`

`PATH` non viene utilizzato per cercare questo file, quindi se lo si vuole utilizzare, è meglio farvi riferimento fornendo il percorso completo e il nome del file.

1.2.2.2.4. Invocato col comando **sh**

Bash cerca di comportarsi come lo storico programma Bourne **sh** pur essendo conforme allo standard POSIX.

File letti:

- `/etc/profile`
- `~/ .profile`

Quando viene invocato in modo interattivo, la variabile `ENV` uò puntare a informazioni di avvio aggiuntive.

1.2.2.2.5. Modalità POSIX

Questa opzione è abilitata utilizzando il **set** nativo:

```
set -o posix
```

oppure chiamando il programma **bash** con l'opzione `--posix`. Bash cercherà quindi di comportarsi nel modo più conforme possibile allo standard POSIX per le shell. L'impostazione della variabile `POSIXLY_CORRECT` fa lo stesso.

File letti:

- definito dalla variabile `ENV`.

1.2.2.2.6. Invocato da remoto

File letti quando invocato da **rshd**:

- `~/ .bashrc`

Evitare l'uso degli r-tools

Si deve porre attenzione quando si usano tool come **rlogin**, **telnet**, **rsh** e **rcp**. Questi sono intrinsecamente insicuri perché i dati riservati vengono inviati sulla rete non crittografati. Se sono necessari strumenti per l'esecuzione remota, il trasferimento di file e così via, usare un'implementazione di Secure SHell, generalmente conosciuta come SSH, disponibile gratuitamente da <http://www.openssh.org>. Sono disponibili diversi programmi client anche per sistemi non UNIX, vedere il mirror locale del software.

1.2.2.2.7. Invocato quando UID non è uguale a EUID

In questo caso non vengono letti file di avvio.

1.2.2.3. Shell interattive

1.2.2.3.1. Che cos'è una shell interattiva?

Una shell interattiva generalmente legge e scrive sul terminale di un utente: input e output sono collegati a un terminale. Il comportamento interattivo di Bash viene avviato quando si richiama il comando **bash** senza argomenti diversi dalle opzioni, tranne quando l'opzione è una stringa da cui leggere o quando viene invocata la shell per leggere dallo standard input, il che consente parametri posizionali da impostare (vedere [Capitolo 3](#)).

1.2.2.3.2. Questa shell è interattiva?

Lo si può testare guardando il contenuto del parametro speciale `-`, contiene una `i` quando la shell è interattiva:

```
eddy:~> echo $-  
himBH
```

Nelle shell non interattive, il prompt `PS1` non è impostato.

1.2.2.3.3. Comportamento interattivo della shell

Differenze nella modalità interattiva:

- Bash legge i file di avvio.
- Il controllo del job è abilitato per default.
- I prompt sono impostati, `PS2` è abilitato per i comandi su più righe, di solito è impostato su `>>`. Questo è anche il prompt che si riceve quando la shell pensa che sia stato inserito un comando incompleto, ad esempio quando si dimenticano le virgolette, le strutture di comando che non possono essere tralasciate, ecc.
- I comandi vengono letti per default dalla riga di comando utilizzando **readline**.
- Bash interpreta l'opzione della shell `ignoreeof` invece di uscire immediatamente dopo aver ricevuto EOF (End Of File).
- La cronologia dei comandi e l'espansione della cronologia sono abilitate per default. La cronologia viene salvata nel file a cui punta `HISTFILE` quando la shell esce. Per default, `HISTFILE` punta a `~/.bash_history`.
- L'espansione degli alias è abilitata.
- In assenza di trap, il segnale `SIGTERM` viene ignorato.
- In assenza delle trap, `SIGINT` viene catturato e gestito. Pertanto, digitando **Ctrl+C**, ad esempio, non si esce dalla shell interattiva.
- L'invio di segnali `SIGHUP` a tutti i job all'uscita viene configurato con l'opzione `huponexit`.
- I comandi vengono eseguiti al momento della lettura.
- Bash controlla periodicamente la posta.
- Bash può essere configurato per uscire quando incontra variabili senza riferimenti. In modalità interattiva questo comportamento è disabilitato.
- Quando i comandi nativi della shell riscontrano errori di re-indirizzamento, ciò non causerà l'uscita dalla shell.
- Speciali errori nativi di ritorno, quando utilizzati in modalità POSIX non provocano l'uscita della shell. I comandi nativi sono elencati nella [Sezione 1.3.2](#).
- Il fallimento di `exec` non provoca l'uscita dalla shell.
- Gli errori di sintassi del parser non causano l'uscita dalla shell.
- Il semplice controllo ortografico per gli argomenti nativo `cd` è abilitato per default.
- Viene abilitata l'uscita automatica dopo che è trascorso il periodo di tempo specificato nella variabile `TMOU`.

Maggiori informazioni:

- [Sezione 3.2](#)
- [Sezione 3.6](#)
- Vedere il [Capitolo 12](#) per ulteriori informazioni sui segnali.
- La [Sezione 3.4](#) tratta le varie espansioni eseguite quando si immette un comando.

1.2.2.4. Espressioni condizionali

Le espressioni condizionali vengono utilizzate dal comando composto `[[` oltre che dal **test** e dai comandi nativi `[`.

Le espressioni possono essere unarie o binarie. Le espressioni unarie vengono spesso usate per esaminare lo stato di un file. C'è bisogno di un solo oggetto, ad esempio un file, su cui eseguire l'operazione.

Esistono anche operatori stringa e operatori di confronto numerico; questi sono operatori binari, che richiedono due oggetti su cui eseguire l'operazione. Se l'argomento `FILE` di uno dei primari è nella forma `/dev/fd/N`, allora viene controllato il descrittore `N`. Se l'argomento `FILE` di uno dei primari è

uno tra `/dev/stdin`, `/dev/stdout` o `/dev/stderr`, allora il viene controllato, rispettivamente, il descrittore 0, 1 o 2.

Le espressioni condizionali vengono discusse in dettaglio nel [Capitolo 7](#).

Maggiori informazioni sui descrittori di file si trovano nella [Sezione 8.2.3](#).

1.2.2.5. Aritmetica della shell

La shell consente di valutare le espressioni aritmetiche, come una delle espansioni della shell o tramite il nativo `let`.

La valutazione viene eseguita con interi a larghezza fissa senza alcun controllo per l'overflow, sebbene la divisione per 0 venga intercettata e contrassegnata come errore. Gli operatori e la loro precedenza e associatività sono gli stessi del linguaggio C, vedere [Capitolo 3](#).

1.2.2.6. Gli Alias

Gli alias consentono di sostituire una parola con una stringa quando viene utilizzata come prima parola di un comando semplice. La shell mantiene un elenco degli alias che possono essere impostati o rimossi coi comandi **alias** e **unalias**.

Bash legge sempre almeno una riga completa di input prima di eseguire qualsiasi comando su tale riga. Gli alias vengono espansi quando viene letto un comando, non quando viene eseguito. Pertanto, una definizione di alias che appare sulla stessa riga di un altro comando non ha effetto finché non viene letta la successiva riga di input. I comandi che seguono la definizione dell'alias su quella riga non sono interessati dal nuovo alias.

Gli alias vengono espansi quando viene letta una definizione di funzione, non quando viene eseguita la funzione, perché una definizione di funzione è essa stessa un comando composto. Di conseguenza, gli alias definiti in una funzione non sono disponibili fino a quando tale funzione non viene eseguita.

Discuteremo gli alias in dettaglio nella [Sezione 3.5](#).

1.2.2.7. Array

Bash fornisce variabili array mono-dimensionali. Qualsiasi variabile può essere utilizzata come array; il **declare** nativo dichiara esplicitamente un array. Non esiste un limite massimo alla dimensione di un array, né alcun requisito che i membri debbano essere indicizzati o assegnati in modo contiguo. Gli indici degli array partono da zero [zero-based]. Vedere il [Capitolo 10](#).

1.2.2.8. Stack delle Directory

Lo stack delle directory è un elenco delle directory visitate di recente. Il **pushd** nativo aggiunge delle directory allo stack mentre cambia la directory corrente e **popd** rimuove quelle specificate e cambia la directory corrente in quella rimossa.

Il contenuto è visualizzabile col comando **dirs** o controllando il contenuto della variabile `DIRSTACK`.

Maggiori informazioni sul funzionamento di questo meccanismo si possono trovare nelle pagine informative di Bash.

1.2.2.9. Il prompt

Bash rende ancora più divertente giocare col prompt. Vedere la sezione *Controlling the Prompt* nelle pagine info di Bash.

1.2.2.10. La shell limitata

Quando viene invocato come **rbash** o con una delle opzioni `--restricted` o `-r`, accade quanto segue:

- Il **cd** nativo è disabilitato.
- Non è possibile settare o re-settare `SHELL`, `PATH`, `ENV` e `BASH_ENV`.
- I nomi dei comandi non possono più contenere barre '/' [slash].
- I nomi di file contenenti una barra '/' non sono consentiti con il comando **.** (**source**).
- **hash** non accetta barre '/' con l'opzione `-p`.
- L'import delle funzioni all'avvio è disabilitato.
- `SHELLOPTS` viene ignorato all'avvio.
- Il re-indirizzamento dell'output utilizzando `>`, `>|`, `><`, `>&`, `&>` o `>>` è disabilitato.
- La funzione **exec** è disabilitata.
- Le opzioni `-f` e `-d` sono disabilite per la funzione **enable**.
- Non è possibile specificare un `PATH` di default col comando **command**.
- Non è possibile disattivare la modalità con restrizioni.

Quando viene eseguito un comando che risulta essere uno script di shell, **rbash** disattiva qualsiasi restrizione nella shell generata per eseguire lo script.

Maggiori informazioni:

- [Sezione 3.2](#)
 - [Sezione 3.6](#)
 - Info Bash->Basic Shell Features->Redirections
 - [Sezione 8.2.3](#): re-indirizzamento avanzato
-

1.3. Esecuzione dei comandi

1.3.1. In generale

Bash determina il tipo di programma che deve essere eseguito. I programmi normali sono comandi di sistema che esistono in forma compilata sul sistema. Quando un tale programma viene eseguito, viene creato un nuovo processo perché Bash fa una esatta copia di se stesso. Questo processo figlio ha lo stesso ambiente del genitore, solo l'ID del processo è diverso. Questa procedura è chiamata *forking*.

Dopo il fork, lo spazio degli indirizzi del processo figlio viene sovrascritto con i nuovi dati del processo. Questo viene fatto tramite una chiamata *exec* al sistema.

Il meccanismo *fork-and-exec* scambia quindi un vecchio comando con uno nuovo, mentre l'ambiente in cui viene eseguito il nuovo programma rimane lo stesso, inclusa la configurazione dei dispositivi di input e output, le variabili di ambiente e la priorità. Questo meccanismo viene utilizzato per creare tutti i processi UNIX, quindi si applica anche al sistema operativo Linux. Anche il primo processo, **init**, con ID 1, viene 'forkato' durante la procedura di avvio nella cosiddetta procedura di *bootstrapping*.

1.3.2. Comandi nativi della shell

I comandi nativi sono contenuti all'interno della shell stessa. Quando il nome di un tale comando viene utilizzato come prima parola di un comando semplice, la shell lo esegue direttamente, senza creare un nuovo processo. I comandi nativi sono necessari per implementare funzionalità impossibili o scomode da ottenere con utilità separate.

Bash supporta 3 tipi di comandi nativi:

- Integrazioni Bourne Shell:

:, ., break, cd, continue, eval, exec, exit, export, getopts, hash, pwd, readonly, return, set, shift, test, [, times, trap, umask e unset.

- Integrazioni Bash:

alias, bind, builtin, command, declare, echo, enable, help, let, local, logout, printf, read, shopt, type, typeset, ulimit e unalias.

- Comandi nativi speciali:

Quando Bash viene eseguito in modalità POSIX, i comandi speciali nativi differiscono dagli altri comandi nativi per tre aspetti:

1. I comandi speciali si trovano prima delle funzioni della shell durante la ricerca dei comandi.
2. Se un comando speciale restituisce uno stato di errore, termina una shell non interattiva.
3. Le istruzioni di assegnazione che precedono il comando rimangono attive nell'ambiente della shell dopo il completamento del comando.

I comandi speciali POSIX sono `:`, `.`, **break**, **continue**, **eval**, **exec**, **exit**, **export**, **readonly**, **return**, **set**, **shift**, **trap** e **unset**.

La maggior parte di questi sarà discussa nei prossimi capitoli. Per gli altri comandi, si rimanda alle pagine di Info.

1.3.3. Esecuzione di programmi da uno script

Quando il programma in esecuzione è uno script di shell, bash creerà un nuovo processo bash utilizzando *fork*. Questa sub-shell legge le righe dallo script della shell, una alla volta. I comandi su ogni riga vengono letti, interpretati ed eseguiti come se provenissero direttamente dalla tastiera.

Mentre la sub-shell elabora ogni riga dello script, la shell genitore attende che il processo figlio finisca. Quando non ci sono più righe da leggere nello script della shell, la sub-shell termina. La shell genitore si attiva e visualizza un nuovo prompt.

1.4. Blocchi di costruzione

1.4.1. Blocchi di costruzione della shell

1.4.1.1. Sintassi della shell

Se l'input non è commentato, la shell lo legge e lo divide in parole e operatori, utilizzando regole per gli apici/virgolette per definire il significato di ciascun carattere di input. Poi tali parole e operatori vengono tradotti in comandi e altri costrutti, che restituiscono uno stato di uscita disponibile per l'ispezione o l'elaborazione. Lo schema *fork-and-exec* di cui sopra viene applicato solo dopo che la shell ha analizzato l'input nel modo seguente:

- La shell legge il suo input da un file, da una stringa o dal terminale dell'utente.
- L'input è suddiviso in parole e operatori, obbedendo alle regole delle virgolette, vedere [Capitolo 3](#). Questi token vengono separati da *metacaratteri*. Viene eseguita l'espansione degli alias.
- La shell *parserizza* (analizza e sostituisce) i token in comandi semplici e composti.
- Bash esegue varie espansioni della shell, suddividendo i token espansi in elenchi di nomi di file, comandi e argomenti.
- Se necessario viene eseguito il re-indirizzamento, gli operatori di re-indirizzamento e i relativi operandi vengono rimossi dall'elenco degli argomenti.
- I comandi vengono eseguiti.
- Facoltativamente, la shell attende il completamento del comando e colleziona lo stato di uscita.

1.4.1.2. Comandi shell

Un semplice comando di shell come **touch file1 file2 file3** è costituito dal comando stesso seguito da argomenti, separati da spazi.

I comandi più complessi sono composti da semplici comandi disposti insieme in vari modi: in una pipeline in cui l'output di un comando diventa l'input di un secondo, in un ciclo o in un costrutto condizionale o in qualche altro raggruppamento. Un paio di esempi:

`ls | more`

`gunzip file.tar.gz | tar xvf -`

1.4.1.3. Funzioni della shell

Le funzioni della shell sono un modo per raggruppare i comandi per l'esecuzione successiva utilizzando un unico nome per il gruppo. Vengono eseguiti proprio come un comando "normale". Quando il nome di una funzione viene utilizzato come nome di comando semplice, viene eseguito l'elenco dei comandi associati a quel nome di funzione.

Le funzioni della shell vengono eseguite nel contesto della shell corrente; non viene creato alcun nuovo processo per interpretarli.

Le funzioni sono spiegate nel [Capitolo 11](#).

1.4.1.4. Parametri della shell

Un parametro è un'entità che memorizza i valori. Può essere un nome, un numero o un valore speciale. Ai fini della shell, una variabile è un parametro che memorizza un nome. Una variabile ha un valore e zero o più attributi. Le variabili vengono create col comando shell **declare**.

Se non viene fornito alcun valore, alla variabile viene assegnata la stringa nulla. Le variabili possono essere rimosse solo col comando **unset**.

L'assegnazione delle variabili è discussa nella [Sezione 3.2](#), l'uso avanzato delle variabili nel [Capitolo 10](#).

1.4.1.5. Espansioni della shell

L'espansione della shell viene eseguita dopo che ogni riga di comando è stata suddivisa in token. Le espansioni eseguite sono queste:

- Espansione delle parentesi graffe "Brace expansion"
- Espansione della tilde
- Espansione del parametro e della variabile
- Sostituzione del comando
- Espansione aritmetica
- Divisione della parola
- Espansione del nomefile

Discuteremo questi tipi di espansione in dettaglio nella [Sezione 3.4](#).

1.4.1.6. Reindirizzamenti

Prima che un comando venga eseguito, l'input e l'output può essere reindirizzato usando una notazione speciale interpretata dalla shell. Il reindirizzamento può essere utilizzato anche per aprire e chiudere i file per l'ambiente di esecuzione della shell corrente.

1.4.1.7. Esecuzione dei comandi

Quando si esegue un comando, le parole che il parser ha contrassegnato come assegnazioni di variabili (che precedono il nome del comando) e i reindirizzamenti vengono salvati per riferimento futuro. Le parole che non sono assegnazioni di variabili o reindirizzamenti vengono espanso; la prima parola rimanente dopo l'espansione è considerata il nome del comando e il resto sono argomenti di quel comando. Poi vengono eseguiti i reindirizzamenti, quindi le stringhe assegnate alle variabili vengono espanso. Se non risulta alcun nome di comando, le variabili modificheranno l'ambiente della shell corrente.

Una parte importante dei compiti della shell è la ricerca di comandi. Bash lo fa in questo modo:

- Controlla se il comando contiene barre '/'. In caso contrario, controlla prima con l'elenco delle funzioni per vedere se contiene un comando col nome da ricercare.
 - Se il comando non è una funzione, lo cerca nell'elenco dei comandi nativi.
 - Se il comando non è né una funzione né uno nativo, lo cerca analizzando le directory elencate in `PATH`. Bash usa una *tabella hash* (area di memorizzazione dei dati in memoria) per ricordare i nomi dei percorsi completi degli eseguibili in modo da evitare ulteriori ricerche in `PATH`.
 - Se la ricerca ha esito negativo, bash stampa un messaggio di errore e restituisce lo stato di uscita di 127.
 - Se la ricerca ha avuto esito positivo o se il comando contiene barre '/', la shell esegue il comando in un ambiente [environment] di esecuzione separato.
 - Se l'esecuzione non riesce perché il file non è eseguibile e non è una directory, si presume che sia uno script di shell.
 - Se il comando non è stato avviato in modo asincrono, la shell ne attende il completamento e ne raccoglie lo stato di uscita.
-

1.4.1.8. Script di shell

Quando un file, contenente comandi shell, viene utilizzato come primo argomento e "non di opzione", invocando Bash (senza `-c` o `-s`, verrà creata una shell non interattiva. Tale shell cerca prima il file dello script nella directory corrente, poi lo cerca in `PATH` se il file non si trova lì.

1.5. Sviluppare degli ottimi script

1.5.1. Le proprietà degli script ottimali

Questa guida si occupa principalmente dell'ultimo blocco di costruzione della shell, gli script. Alcune considerazioni generali prima di continuare:

1. Uno script dovrebbe girare senza errori.
 2. Dovrebbe svolgere il compito a cui è destinato.
 3. La logica del programma dovrebbe essere chiaramente definita ed evidente.
 4. Uno script non fa il lavoro non necessario.
 5. Gli script dovrebbero essere riutilizzabili.
-

1.5.2. Struttura

La struttura di uno script shell è molto flessibile. Anche se in Bash viene concessa molta libertà, è necessario garantire la logica corretta, il controllo del flusso e l'efficienza, in modo che gli utenti che eseguono lo script possano farlo facilmente e correttamente.

Prima di iniziare la scrittura di un nuovo script, ci si devono porre le seguenti domande:

- Avrò bisogno di informazioni dall'utente o dall'ambiente dell'utente?
 - Come memorizzerò tali informazioni?
 - Ci sono file che devono essere creati? Dove e con quali permessi e di chi sono [ownership]?
 - Quali comandi userò? Utilizzando lo script su sistemi diversi, tutti questi sistemi hanno i comandi nelle versioni giuste?
 - L'utente ha bisogno di notifiche? Quando e perché?
-

1.5.3. Terminologia

La tabella seguente fornisce una panoramica dei termini di programmazione con cui è necessario avere familiarità:

Tabella 1-1. Panoramica dei termini di programmazione

Termine	Che cos'è?
Comando di controllo	Testare lo stato di uscita di un comando per determinare se una parte del programma deve essere eseguita.
Ramo condizionale	Punto logico nel programma in cui una condizione determina cosa succede dopo.
Flusso logico	Il progetto generale del programma. Determina la sequenza logica delle attività in modo che il risultato sia positivo e controllato.
Ciclo	Parte del programma che viene eseguita zero o più volte.
Input dell'utente	Le informazioni fornite da una fonte esterna durante l'esecuzione del programma possono essere memorizzate e richiamate quando necessario.

1.5.4. A proposito della logica

Per accelerare il processo di sviluppo, l'ordine logico di un programma dovrebbe essere definito in anticipo. Questo è il primo passo nello sviluppo di uno script.

È possibile utilizzare diversi metodi; uno dei più comuni è lavorare con le liste. Elencare le attività coinvolte in un programma consente di descrivere ciascun processo. Le singole attività possono essere referenziate dal loro numero di sequenza.

Utilizzando a propria lingua per definire le attività da eseguire aiuterà a creare una forma comprensibile del programma. Successivamente, è possibile sostituire le istruzioni del linguaggio comune con parole e costrutti del linguaggio della shell.

L'esempio seguente mostra un tale flusso logico di progetto. Descrive la rotazione dei file di log. Questo esempio mostra un possibile ciclo ripetitivo, controllato dal numero di file di log di base che si desiderano ruotare:

1. Vuoi ruotare i log?
 - a. Se sì:
 - i. Immettere il nome della directory contenente i log da ruotare.
 - ii. Immettere il nome-base del file di log.
 - iii. Immettere il numero di giorni per cui i log devono essere conservati.
 - iv. Rendi permanenti le impostazioni nel crontab dell'utente.
 - b. Se no, vai al passo 3.
2. Vuoi ruotare un altro set di log?
 - a. Se sì, ripeti il passo 1.
 - b. Se no, vai al passo 3.
3. Esci

L'utente deve fornire informazioni affinché il programma possa fare qualcosa. L'input dell'utente deve essere acquisito e memorizzato. L'utente dovrebbe essere informato che il suo crontab cambierà.

1.5.5. Un esempio di script Bash: `mysystem.sh`

Il seguente script `mysystem.sh` esegue alcuni noti comandi (`date`, `w`, `uname`, `uptime`) per visualizzare informazioni sull'utente e sulla macchina.

```
tom:~> cat -n mysystem.sh
 1  #!/bin/bash
 2  clear
 3  echo "Questa è un'informazione fornita da mysystem.sh.  Il programma
comincia adesso."
 4
 5  echo "Ciao, $USER"
 6  echo
 7
 8  echo "La data di oggi è `date`, questa è la settimana `date +%V`."
 9  echo
10
11  echo "Attualmente sono connessi questi utenti:"
12  w | cut -d " " -f 1 - | grep -v USER | sort -u
13  echo
14
15  echo "Questo è `uname -s` che gira su un processore `uname -m`."
16  echo
17
18  echo "Questo è il tempo di attività:"
19  uptime
20  echo
21
22  echo "È tutto gente!"
```

Uno script inizia sempre con gli stessi due caratteri,, `#!/`. Successivamente viene definita la shell che eseguirà i comandi successivi alla prima riga. Questo script inizia con la cancellazione dello

schermo sulla riga 2. La riga 3 stampa un messaggio, informando l'utente su cosa sta per accadere. La riga 5 saluta l'utente. Le righe 6, 9, 13, 16 e 20 sono presenti solo per una visualizzazione ordinata dell'output. La riga 8 stampa la data corrente e il numero della settimana. La riga 11 è di nuovo un messaggio informativo, come le righe 3, 18 e 22. La riga 12 formatta l'output di `w`; la riga 15 mostra le informazioni sul sistema operativo e sulla CPU. La riga 19 fornisce le informazioni sul tempo di attività e sul carico della cpu.

Sia **echo** che **printf** sono comandi interni di Bash. Il primo esce sempre con uno stato 0 e stampa semplicemente gli argomenti seguiti da un carattere di fine riga sullo standard output, mentre il secondo consente la definizione di una stringa di formattazione e fornisce un codice di stato di uscita diverso da zero in caso di errore.

Questo è lo stesso script che usa il **printf** nativo:

```
tom:~> cat mysystem.sh
#!/bin/bash
clear
printf "Questa è un'informazione fornita da mysystem.sh. Il programma comincia adesso.\n"

printf "Ciao, $USER.\n\n"

printf "La data di oggi è `date`, questa è la settimana `date +%V`.\n\n"

printf "Attualmente sono connessi questi utenti:\n"
w | cut -d " " -f 1 - | grep -v USER | sort -u
printf "\n"

printf "Questo è `uname -s` che gira su un processore `uname -m`.\n\n"

printf "Questo è il tempo di attività:\n"
uptime
printf "\n"

printf "È tutto gente!\n"
```

La creazione di script user friendly tramite l'inserimento di messaggi è trattata nel [Capitolo 8](#).



Posizione standard della Bourne Again shell

Questo implica che il programma **bash** sia installato in `/bin`.



Se lo stdout non è disponibile

Se si esegue uno script da cron, si forniscono i nomi completi dei path e si reindirizza l'output e gli errori. Poiché la shell viene eseguita in modalità non interattiva, qualsiasi errore causerà l'uscita prematura dello script se non vi è stato posto rimedio.

I capitoli seguenti tratteranno i dettagli degli script di cui sopra.

1.5.6. Esempio di script init

Uno script di inizializzazione avvia i servizi di sistema su macchine UNIX e Linux. Esempi comuni sono il demone di sistema del log, il demone della gestione dell'energia, i demoni del nome e della mail. Questi script, noti anche come script di avvio [startup], sono memorizzati in una locazione specifica sul sistema, come `/etc/rc.d/init.d` o `/etc/init.d`. Init, il processo iniziale, legge i

suoi file di configurazione e decide quali servizi lanciare o fermare in ciascun livello di esecuzione. Un livello di esecuzione è una configurazione di processi; ogni sistema ha un singolo livello utente di esecuzione, ad esempio, per l'esecuzione di attività amministrative, per le quali il sistema deve essere il più possibile inutilizzato, come il ripristino di un file system critico da un backup. In genere anche i livelli di esecuzione di reboot e shutdown vengono configurati.

Le attività [task] da eseguire all'avvio o all'arresto di un servizio sono elencate negli script di startup. È uno dei compiti dell'amministratore di sistema configurare **init**, in modo che i servizi vengano avviati e arrestati al momento giusto. Di fronte a questa attività, è necessaria una buona comprensione delle procedure di avvio e arresto del sistema. Si consiglia di leggere le pagine man di **init** e **inittab** prima di modificare gli script di inizializzazione.

Ecco un esempio semplicissimo, che emetterà un suono all'avvio e all'arresto della macchina:

```
#!/bin/bash

# This script is for /etc/rc.d/init.d
# Link in rc3.d/S99audio-greeting and rc0.d/K01audio-greeting

case "$1" in
'start')
    cat /usr/share/audio/at_your_service.au > /dev/audio
    ;;
'stop')
    cat /usr/share/audio/oh_no_not_again.au > /dev/audio
    ;;
esac
exit 0
```

L'istruzione `case`, usata spesso in questo tipo di script, è descritta nella [Sezione 7.2.5](#).

1.6. Sommario

Bash è la shell di GNU, compatibile con la Bourne shell e comprende molte funzioni utili di altre shell. Quando la shell viene avviata, legge i suoi file di configurazione. I più importanti sono:

- `/etc/profile`
- `~/.bash_profile`
- `~/.bashrc`

Bash si comporta in modo diverso quando è in modalità interattiva e ha anche una modalità conforme a POSIX e una con restrizioni.

I comandi della shell possono essere suddivisi in tre gruppi: le funzioni, i comandi interni (o nativi) e i comandi esistenti in una directory del sistema. Bash supporta ulteriori comandi interni non presenti nella semplice Bourne Shell.

Gli script sono costituiti da tali comandi disposti secondo i dettami della sintassi della shell. Gli script vengono letti ed eseguiti riga per riga e dovrebbero avere una struttura logica.

1.7. Esercizi

Questi sono alcuni esercizi di 'riscaldamento' per il prossimo capitolo:

1. Dove si trova il programma **bash** sul sistema?
2. Usare l'opzione `--version` per scoprire quale versione si sta utilizzando.
3. Quali file di configurazione della shell vengono letti quando si accede al sistema utilizzando l'interfaccia utente grafica e quindi aprendo un terminale?
4. Le seguenti shell sono interattive? Sono shell di login?
 - o Una shell aperta cliccando sullo sfondo del desktop grafico, selezionando "Terminal" o simili da un menu.
 - o Una shell che si ottiene dopo aver eseguito il comando **ssh localhost**.
 - o Una shell che si ottiene quando si accede alla console in modalità testo.
 - o Una shell ottenuta dal comando **xterm &**.
 - o Una shell aperta dallo script **mysystem.sh**.
 - o Una shell che si ottiene su un host remoto, per la quale non si dovevano fornire login e/o password perché si usa SSH e forse le chiavi SSH.
5. Spiegare perché **bash** non esce quando si digita **Ctrl+C** sulla riga di comando?
6. Visualizzare il contenuto della lista [stack] delle directory.
7. Se non è ancora così, impostare il prompt in modo che mostri la propria posizione nella gerarchia del file system, ad esempio aggiungendo questa riga a `~/.bashrc`:

```
export PS1="\u@\h \w> "
```

8. Visualizzare i comandi con hash per la propria sessione corrente di shell.
9. Quanti processi sono attualmente in esecuzione sul sistema? Usare **ps** e **wc**, la prima riga di output di **ps** non è un processo!
10. Come visualizzare il nome host del sistema? Solo il nome, niente di più!

Capitolo 2. Scrittura e debug di script

Dopo aver letto questo capitolo, si sarà in grado di:

- Scrivere un semplice script
- Definire il tipo di shell che dovrebbe eseguire lo script
- Mettere i commenti in uno script
- Modificare i permessi su uno script
- Eseguire e debuggare uno script

2.1. Creazione ed esecuzione di uno script

2.1.1. Scrivere e dare un nome

Uno script shell è una sequenza di comandi di cui si fa un uso ripetuto. Questa sequenza viene in genere eseguita immettendo il nome dello script sulla riga di comando. In alternativa, gli script si possono utilizzare per automatizzare le attività con la funzione cron. Un altro uso degli script è nella procedura di avvio [boot] e arresto [shutdown] di UNIX, in cui il funzionamento dei demoni e dei servizi è definito negli script di inizializzazione.

Per creare uno script di shell, si apre un nuovo file vuoto nell'editor. Qualsiasi editor di testo andrà bene: **vim**, **emacs**, **gedit**, **dtpad** eccetera, sono tutti validi. Potrebbe essere il caso di scegliere un editor più avanzato come **vim** o **emacs**, dato che questi possono essere configurati per riconoscere la sintassi della shell e di Bash e possono essere di grande aiuto per prevenire gli errori che i principianti commettono frequentemente, come dimenticare parentesi e punto e virgola.



Evidenziazione della sintassi in vim

Per attivare l'evidenziazione della sintassi in **vim**, usare il comando

:syntax enable

o

:sy enable

o

:syn enable

Si può aggiungere questa impostazione al file `.vimrc` per renderlo permanente.

Inserire i comandi UNIX nel nuovo file vuoto, come se si stessero immettendo sulla riga di comando. Come discusso nel capitolo precedente (vedere [Sezione 1.3](#)), i comandi possono essere funzioni shell, nativi di shell, comandi UNIX e altri script.

Dare allo script un nome ragionevole che suggerisca cosa faccia. Assicurarsi che il nome dello script non sia in conflitto con i comandi esistenti. Per evitare che si crei confusione, i nomi degli script spesso finiscono con `.sh`; anche così, potrebbero esserci altri script sul sistema con lo stesso nome di quello scelto. Controllare usando **which**, **whereis** e altri comandi per trovare informazioni su programmi e file:

which -a script_name

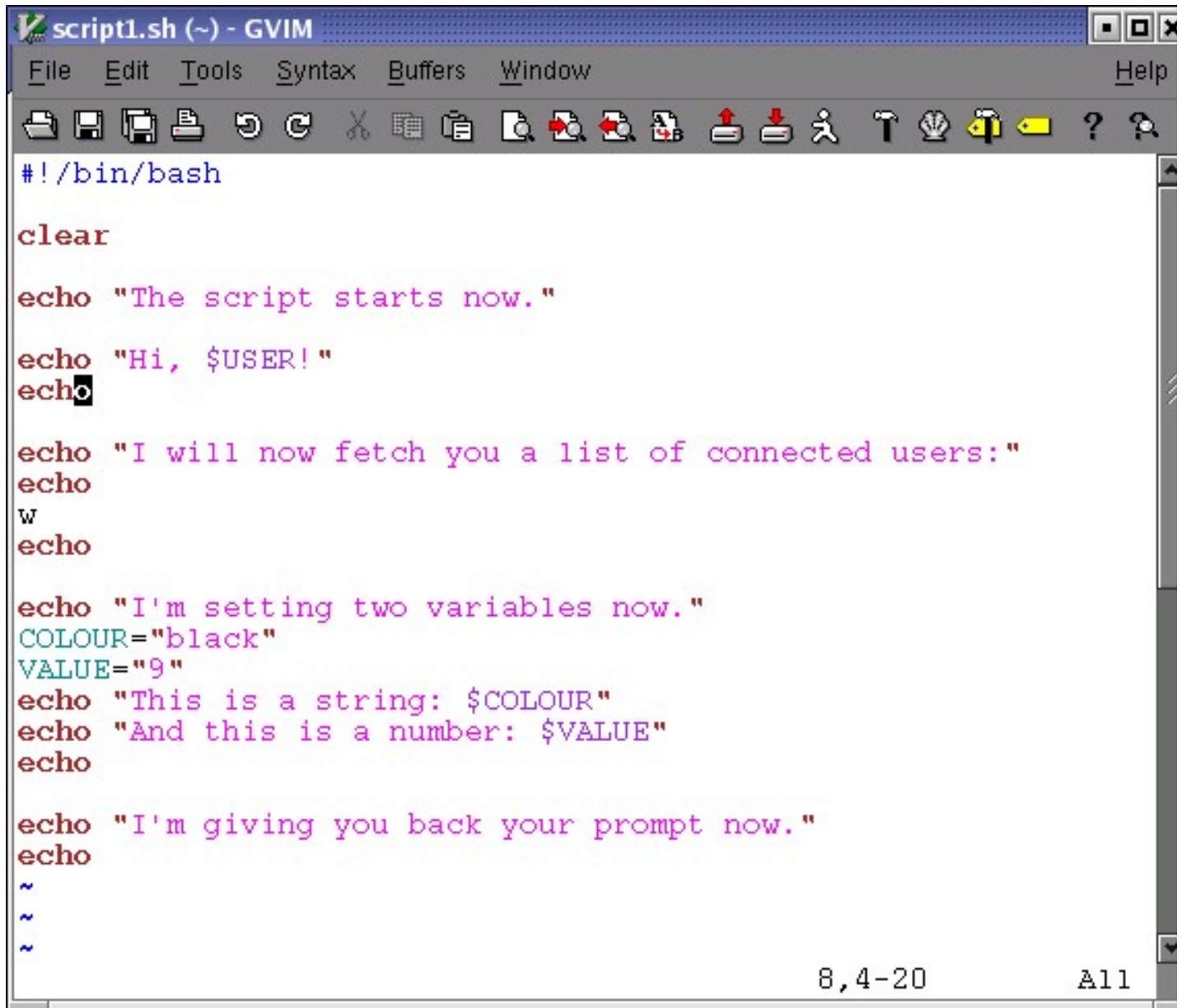
whereis script_name

locate script_name

2.1.2. script1.sh

In questo esempio si usa il comando interno di Bash **echo** per informare l'utente su cosa sta per accadere, prima che venga eseguita l'attività che creerà l'output. Si consiglia vivamente di informare gli utenti su cosa sta facendo uno script, per evitare che si preoccupino *perché lo script non sta facendo nulla*. Torneremo sull'argomento della notifica agli utenti nel [Capitolo 8](#).

Figura 2-1. script1.sh



```
#!/bin/bash

clear

echo "The script starts now."

echo "Hi, $USER!"
echo

echo "I will now fetch you a list of connected users:"
echo
W
echo

echo "I'm setting two variables now."
COLOUR="black"
VALUE="9"
echo "This is a string: $COLOUR"
echo "And this is a number: $VALUE"
echo

echo "I'm giving you back your prompt now."
echo
~
~
~
```

Scrivere questo script anche per te stessi. Potrebbe essere una buona idea creare una directory `~/scripts` per contenere gli script. Aggiungere la directory al contenuto della variabile `PATH`:

```
export PATH="$PATH:~/scripts"
```

Se si è all'inizio con Bash, usare un editor di testo che utilizzi colori diversi per i diversi costrutti della shell. L'evidenziazione della sintassi è supportata da **vim**, **gvim**, **(x)emacs**, **kwrite** e molti altri editor; controllare la documentazione dell'editor scelto.

Prompt diversi

I prompt durante questo corso variano a seconda dell'umore dell'autore. Questo assomiglia molto più a situazioni di vita reale rispetto al prompt standard `$`. L'unica convenzione a cui ci atteniamo è che il prompt *root* termini con un cancelletto (`#`).

2.1.3. Esecuzione dello script

Lo script dovrebbe avere le autorizzazioni di esecuzione per gli owner (proprietari) corretti per essere eseguibile. Quando si impostano i permessi, controllare di aver veramente ottenuto i permessi desiderati. Fatto ciò, lo script può essere eseguito come qualsiasi altro comando:

```
willy:~/scripts> chmod u+x script1.sh

willy:~/scripts> ls -l script1.sh
-rwxrw-r-- 1 willy willy 456 Dec 24 17:11 script1.sh

willy:~> script1.sh
The script starts now.
Hi, willy!

I will now fetch you a list of connected users:

  3:38pm up 18 days,  5:37,  4 users,  load average: 0.12, 0.22, 0.15
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
root      tty2      -               Sat 2pm  4:25m  0.24s  0.05s  -bash
willy     :0        -               Sat 2pm  ?      0.00s  ?      -
willy     pts/3     -               Sat 2pm  3:33m  36.39s 36.39s  BitchX willy
ir
willy     pts/2     -               Sat 2pm  3:33m  0.13s  0.06s
/usr/bin/screen

I'm setting two variables now.
This is a string: black
And this is a number: 9

I'm giving you back your prompt now.

willy:~/scripts> echo $COLOUR

willy:~/scripts> echo $VALUE

willy:~/scripts>
```

Questo è il modo più comune per eseguire uno script. È preferibile eseguire lo script in questo modo in una subshell. Le variabili, le funzioni e gli alias creati in questa subshell sono noti solo alla particolare sessione bash di quella subshell. Quando quella shell esce e il genitore riprende il controllo, tutto viene ripulito e tutte le modifiche allo stato della shell apportate dallo script vengono dimenticate.

Se non è stata messa la directory `scripts` nel `PATH` e nemmeno `.` (la directory corrente) è nel `PATH`, di può attivare lo script in questo modo:

`./script_name.sh`

Uno script può anche essere eseguito esplicitamente da una data shell, ma generalmente lo si fa solo se si vuole ottenere un comportamento speciale, come controllare se lo script funziona con un'altra shell o avere delle stampe per il debug:

`rbash script_name.sh`

`sh script_name.sh`

`bash -x script_name.sh`

La shell specificata si avvierà come sottoshell della shell corrente ed eseguirà lo script. Questo viene fatto quando si desidera che lo script si avvii con opzioni specifiche o in condizioni specifiche che non sono definite nello script.

Se si vuol eseguire lo script nella shell corrente, la si può rendere *source*:

source script_name.sh

```
source = .
```

Il comando interno Bash **source** è un sinonimo per il comando Bourne shell **.** (punto).

In questo caso lo script non necessita dell'autorizzazione per l'esecuzione. I comandi vengono eseguiti nel contesto della shell corrente, quindi qualsiasi modifica apportata all'ambiente sarà visibile al termine dell'esecuzione dello script:

```
willy:~/scripts> source script1.sh
--output omitted--

willy:~/scripts> echo $VALUE
9

willy:~/scripts>
```

2.2. Nozioni di base sugli script

2.2.1. Quale shell eseguirà lo script?

Nell'eseguire uno script in una sotto-shell, è necessario definire quale shell deve eseguirlo. Il tipo previsto in fase di scrittura dello script potrebbe non essere quello predefinito sul sistema, quindi i comandi inseriti potrebbero causare errori se eseguiti dalla shell sbagliata.

La prima riga dello script determina la shell avviata. I primi due caratteri della prima riga devono essere **#!/**, seguono poi il path della shell che dovrà interpretare i comandi che seguono. Anche le righe vuote sono considerate righe, quindi non iniziare lo script con una riga vuota.

Ai fini di questo corso, tutti gli script inizieranno con la riga

#!/bin/bash

Come notato prima, ciò implica che l'eseguibile Bash si trova in `/bin`

2.2.2. Aggiungere i commenti

Si dovrebbe essere consapevoli del fatto di non essere l'unica persona a leggere il codice. Molti utenti e amministratori di sistema eseguono script scritti da altri. Se vogliono vedere come è stato fatto, i commenti sono utili per guidare il lettore.

Inoltre i commenti semplificano la vita. Diciamo che si è dovuto leggere molte pagine man per ottenere un risultato particolare con alcuni comandi che usati nello script. Non ci si ricorderà come

ha funzionato se ci sarà bisogno di cambiare lo script dopo settimane o mesi, a meno che non sia stato annotato nel codice cosa sia stato fatto, come è stato fatto e/o perché.

Prendere l'esempio `script1.sh` e copiarlo in `commented-script1.sh`, che modifichiamo in modo che i commenti riflettano ciò che fa lo script. Tutto ciò che la shell incontra dopo un cancelletto su una riga viene ignorato e visibile solo all'apertura del file dello script:

```
#!/bin/bash
# Questo script ripulisce il terminale, visualizza un saluto e fornisce
informazioni
# sugli utenti attualmente connessi. Le due variabili di esempio vengono
impostate e visualizzate.

clear                                # pulisce la finestra del terminale

echo "The script starts now."

echo "Hi, $USER!"                   # il simbolo del dollaro '$' viene usato per
ottenere il contenuto della variabile
echo

echo "I will now fetch you a list of connected users:"
echo
w                                    # mostra chi è connesso e
echo                                # cosa stanno facendo

echo "I'm setting two variables now."
COLOUR="black"                      # imposta una variabile locale
VALUE="9"                           # imposta una variabile locale
echo "This is a string: $COLOUR"     # mostra il contenuto della
variabile
echo "And this is a number: $VALUE"  # mostra il contenuto della
variabile
echo

echo "I'm giving you back your prompt now."
echo
```

In uno script decente, le prime righe di solito descrivono cosa aspettarsi. Quindi ciascun grande blocco di comandi verrà commentato secondo necessità per motivi di chiarezza. Gli script di `init` di Linux, ad esempio, nella directory `init.d` del sistema, sono generalmente ben commentati poiché devono essere leggibili e modificabili da chiunque utilizzi Linux.

2.3. Debugging di script Bash

2.3.1. Il debugging di tutto lo script

Quando le cose non vanno secondo i piani, è necessario determinare esattamente cosa causa il fallimento dello script. Bash fornisce molte funzionalità di debug, la più comune è avviare la sub-shell con l'opzione `-x`, che eseguirà l'intero script in modalità debug. Il trace di ogni comando più i suoi argomenti vengono stampati sullo standard output dopo che i comandi sono stati espansi ma prima che vengano eseguiti.

Questo è lo script `commented-script1.sh` eseguito in modalità debug. Notare ancora che i commenti aggiunti non sono visibili nell'output dello script.

```
willy:~/scripts> bash -x script1.sh
```

```
+ clear

+ echo 'The script starts now.'
The script starts now.
+ echo 'Hi, willy!'
Hi, willy!
+ echo

+ echo 'I will now fetch you a list of connected users:'
I will now fetch you a list of connected users:
+ echo

+ w
 4:50pm up 18 days,  6:49,  4 users,  load average: 0.58, 0.62, 0.40
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
root      tty2      -               Sat 2pm  5:36m  0.24s  0.05s  -bash
willy     :0        -               Sat 2pm  ?       0.00s  ?       -
willy     pts/3     -               Sat 2pm  43:13  36.82s 36.82s  BitchX willy
ir
willy     pts/2     -               Sat 2pm  43:13  0.13s  0.06s
/usr/bin/screen
+ echo

+ echo 'I\'\'m setting two variables now.'
I'm setting two variables now.
+ COLOUR=black
+ VALUE=9
+ echo 'This is a string: '
This is a string:
+ echo 'And this is a number: '
And this is a number:
+ echo

+ echo 'I\'\'m giving you back your prompt now.'
I'm giving you back your prompt now.
+ echo
```

Ora c'è un debugger completo per Bash, disponibile su [SourceForge](https://sourceforge.net/projects/bashdb/). Queste funzionalità di debug sono disponibili nella maggior parte delle versioni moderne di Bash, a partire dalla 3.x.

2.3.2. Debug di parte(i) dello script

Usando il comando interno **set** di Bash si possono eseguire in modalità normale quelle parti dello script che si ritengono prive di errori e visualizzare le informazioni di debug solo per le zone problematiche. Diciamo che non siamo sicuri di cosa farà il comando **w** nell'esempio `commented-script1.sh`, quindi potremmo racchiuderlo nello script in questo modo:

```
set -x                # attiva il debugging da qui
w
set +x                # ferma il debugging da qui
```

L'output è simile a questo:

```
willy: ~/scripts> script1.sh
The script starts now.
Hi, willy!

I will now fetch you a list of connected users:

+ w
```

```

5:00pm up 18 days, 7:00, 4 users, load average: 0.79, 0.39, 0.33
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
root      tty2      -               Sat 2pm  5:47m  0.24s  0.05s  -bash
willy     :0        -               Sat 2pm  ?      0.00s  ?      -
willy     pts/3     -               Sat 2pm  54:02  36.88s 36.88s BitchX willyke
willy     pts/2     -               Sat 2pm  54:02  0.13s  0.06s
/usr/bin/screen
+ set +x

I'm setting two variables now.
This is a string:
And this is a number:

I'm giving you back your prompt now.

willy: ~/scripts>

```

Su può attivare e disattivare la modalità di debug tutte le volte che si vuole all'interno dello stesso script.

La tabella seguente offre una panoramica di altre utili opzioni di Bash:

Tabella 2-1. Panoramica del set di opzioni di debug

Notazione breve	Notazione lunga	Risultato
set -f	set -o noglob	Disabilita la generazione del nome del file utilizzando i metacaratteri (globbing).
set -v	set -o verbose	Stampa le righe di input della shell mentre vengono lette.
set -x	set -o xtrace	Stampa i trace del comando prima di eseguirlo.

Il meno '-' viene utilizzato per attivare un'opzione shell e un più '+' per disattivarla. Non farsi confondere!

Nell'esempio seguente, vengono mostrate queste opzioni sulla riga di comando:

```

willy:~/scripts> set -v

willy:~/scripts> ls
ls
commented-scripts.sh  script1.sh

willy:~/scripts> set +v
set +v

willy:~/scripts> ls *
commented-scripts.sh  script1.sh

willy:~/scripts> set -f

willy:~/scripts> ls *
ls: *: No such file or directory

willy:~/scripts> touch *

willy:~/scripts> ls
*  commented-scripts.sh  script1.sh

willy:~/scripts> rm *

```

```
willy:~/scripts> ls
commented-scripts.sh  script1.sh
```

In alternativa, queste modalità possono essere indicate nello script stesso, aggiungendo le opzioni desiderate alla dichiarazione della shell della prima riga. Le opzioni possono essere combinate, come di solito accade con i comandi UNIX:

#!/bin/bash -xv

Una volta trovata la parte difettosa dello script, si possono aggiungere le istruzioni **echo** prima di ogni comando di cui non si è certi, in modo da vedere esattamente dove e perché le cose non funzionano. Nello script di esempio `commented-script1.sh` si potrebbe fare così, assumendo sempre che la visualizzazione degli utenti dia problemi:

```
echo "messaggio di debug: ora si tenta di avviare il comando w"; w
```

Negli script più avanzati, è possibile inserire l'**eco** per visualizzare il contenuto delle variabili nelle diverse fasi dello script, in modo da rilevare eventuali difetti:

```
echo "Variable VARNAME is now set to $VARNAME."
```

2.4. Sommario

Uno script di shell è una serie riutilizzabile di comandi inseriti in un file di testo eseguibile. Qualsiasi editor di testo può essere utilizzato per scrivere gli script.

Gli script iniziano con **#!/** seguito dal path della shell che esegue i comandi dallo script. I commenti vengono aggiunti a uno script per riferimento futuro e anche per renderlo comprensibile ad altri utenti. È meglio avere troppe spiegazioni che non averne abbastanza.

Il debug di uno script può essere eseguito utilizzando le opzioni della shell. Le opzioni della shell possono essere utilizzate per il debug parziale o per l'analisi dell'intero script. Anche l'inserimento di comandi **echo** in posizioni strategiche è una comune tecnica di risoluzione dei problemi.

2.5. Esercizi

Questo esercizio aiuterà a creare il primo script.

1. Scrivere uno script usando l'editor preferito. Lo script dovrebbe visualizzare il path della propria home-directory e il tipo di terminale in uso. Inoltre, deve mostrare tutti i servizi avviati nel runlevel 3 sul sistema. (suggerimento: usare `HOME`, `TERM` e `ls /etc/rc3.d/S*`)
2. Aggiungere i commenti nello script.
3. Aggiungere informazioni per gli utenti dello script.
4. Modificare i permessi sullo script in modo da poterlo eseguire.
5. Eseguire lo script in modalità normale e in modalità debug. Dovrebbe funzionare senza errori.
6. Inserire degli errori nello script: guardare cosa succede se si scrivono in modo errato i comandi, se si tralascia la prima riga o ci si mette qualcosa di incomprensibile, o se si

sbaglia l'ortografia dei nomi delle variabili o si scrivono in caratteri minuscoli dopo che sono stati dichiarati in maiuscolo. Controllare cosa dicono i commenti di debug su questo.

Capitolo 3. L'ambiente [environment] Bash

In questo capitolo discuteremo i vari modi in cui l'ambiente Bash può essere modificato:

- Modifica dei file di inizializzazione della shell
- Utilizzo delle variabili
- Utilizzo di dirsi stili per il testo tra virgolette
- Eseguire calcoli aritmetici
- Assegnazione di alias
- Utilizzo dell'espansione e della sostituzione

3.1. File di inizializzazione della shell

3.1.1. File di configurazione a livello di sistema

3.1.1.1. /etc/profile

Quando viene invocato in modo interattivo con l'opzione `--login` o quando viene invocato come **sh**, Bash legge le istruzioni `/etc/profile`. Queste di solito impostano le variabili della shell `PATH`, `USER`, `MAIL`, `HOSTNAME` e `HISTSIZE`.

Su alcuni sistemi, il valore **umask** è configurato in `/etc/profile`; su altri sistemi questo file contiene puntatori ad altri file di configurazione come:

- `/etc/inputrc`, il file di inizializzazione Readline a livello di sistema in cui è possibile configurare lo stile sonoro `[bell-style]` della riga di comando.
- la directory `/etc/profile.d`, che contiene i file che configurano il comportamento a livello di sistema di specifici programmi.

Tutte le impostazioni che si vogliono applicare a tutti gli ambienti degli utenti dovrebbero stare in questo file. Potrebbe assomigliare a questo:

```
# /etc/profile

# System wide environment and startup programs, for login setup

PATH=$PATH:/usr/X11R6/bin

# No core files by default
ulimit -S -c 0 > /dev/null 2>&1

USER=`id -un`
LOGNAME=$USER
MAIL="/var/spool/mail/$USER"

HOSTNAME=`/bin/hostname`
HISTSIZE=1000
```

```
# Keyboard, bell, display style: the readline config file:
if [ -z "$INPUTRC" -a ! -f "$HOME/.inputrc" ]; then
    INPUTRC=/etc/inputrc
fi

PS1="\u@\h \W"

export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE INPUTRC PS1

# Source initialization files for specific programs (ls, vim, less, ...)
for i in /etc/profile.d/*.sh ; do
    if [ -r "$i" ]; then
        . $i
    fi
done

# Settings for program initialization
source /etc/java.conf
export NPX_PLUGIN_PATH="$JRE_HOME/plugin/ns4plugin:/usr/lib/netscape/plugins"

PAGER="/usr/bin/less"

unset i
```

Questo file di configurazione imposta alcune variabili di ambiente della shell di base e alcune variabili richieste dagli utenti che eseguono Java e/o applicazioni Java nel proprio browser web. Vedere la [Sezione 3.2](#).

Consultare il [Capitolo 7](#) per ulteriori informazioni sul condizionale **if** utilizzato in questo file; il [Capitolo 9](#) tratta dei cicli come il costrutto **for**.

Il sorgente Bash contiene il file di esempio `profile` per un uso generale o individuale. Questi e quello nell'esempio sopra necessitano di modifiche affinché funzionino nel proprio ambiente!

3.1.1.2. /etc/bashrc

Sui sistemi che offrono più tipi shell, potrebbe essere opportuno inserire configurazioni specifiche di Bash in tale file, poiché `/etc/profile` viene letto anche da altre shell, come la Bourne. Gli errori generati da shell che non comprendono la sintassi Bash vengono prevenuti suddividendo i file di configurazione per i diversi tipi di shell. In tali casi, il `~/ .bashrc` dell'utente potrebbe puntare a `/etc/bashrc` per includerlo nel processo di inizializzazione della shell al momento del login.

Si potrebbe anche scoprire che `/etc/profile`, sul sistema, contiene solo l'ambiente della shell e le impostazioni di avvio del programma, mentre `/etc/bashrc` contiene le definizioni a livello di sistema per le funzioni della shell e gli degli alias. È possibile fare riferimento al file `/etc/bashrc` in `/etc/profile` o nei singoli file utente di inizializzazione della shell.

Il sorgente contiene il file di esempio `bashrc`, oppure se ne potrebbe reperire una copia in `/usr/share/doc/bash-2.05b/startup-files`. Questo fa parte del `bashrc` fornito con la documentazione di Bash:

```
alias ll='ls -l'
alias dir='ls -ba'
alias c='clear'
alias ls='ls --color'
```

```
alias mroe='more'
alias pdw='pwd'
alias sl='ls --color'

pskill()
{
    local pid

    pid=$(ps -ax | grep $1 | grep -v grep | gawk '{ print $1 }')
    echo -n "killing $1 (process $pid)..."
    kill -9 $pid
    echo "slaughtered."
}
```

Oltre agli alias generali, contiene utili alias che fanno funzionare i comandi anche se si scrivono male. Discuteremo gli alias nella [Sezione 3.5.2](#). Questo file contiene una funzione, **pskill**; le funzioni saranno studiate in dettaglio nel [Capitolo 11](#).

3.1.2. File di configurazione utente individuali



Non ci sono questi file?!

Questi file potrebbero non stare nella home directory per impostazione predefinita; se necessario crearli.

3.1.2.1. ~/.bash_profile

Questo è il file di configurazione da preferire per la configurazione individuale degli ambienti utente. In questo file, gli utenti possono aggiungere ulteriori opzioni di configurazione o modificare le impostazioni predefinite:

```
franky~> cat .bash_profile
#####
#
# .bash_profile file
#
# Executed from the bash shell when you log in.
#
#####

source ~/.bashrc
source ~/.bash_login
case "$OS" in
    IRIX)
        stty sane dec
        stty erase
        ;;
    # SunOS)
    #     stty erase
    #     ;;
    *)
        stty sane
        ;;
esac
```

Questo utente configura il carattere backspace '\ per l'accesso su diversi sistemi operativi. Oltre a ciò, vengono letti il `.bashrc` e il `.bash_login` dell'utente.

3.1.2.2. ~/.bash_login

Questo file contiene impostazioni specifiche che normalmente vengono eseguite solo quando si accede al sistema. Nell'esempio, lo usiamo per configurare il valore **umask** e per mostrare un elenco di utenti connessi al momento del login. Questo utente ottiene anche il calendario per il mese corrente:

```
#####  
#  
#   Bash_login file  
#  
#   commands to perform from the bash shell at login time  
#   (sourced from .bash_profile)  
#  
#####  
#   file protection  
umask 002      # all to me, read to group and others  
#   miscellaneous  
w  
cal `date +"%m"` `date +"%Y"`
```

In assenza di ~/.bash_profile, verrà letto questo file.

3.1.2.3. ~/.profile

In assenza di ~/.bash_profile e ~/.bash_login, viene letto ~/.profile. Può contenere le stesse configurazioni, che sono poi accessibili anche da altre shell. Si tenga presente che le altre shell potrebbero non comprendere la sintassi di Bash.

3.1.2.4. ~/.bashrc

Oggi è più comune utilizzare una shell senza login (non-login), ad esempio quando si accede graficamente utilizzando le finestre del terminale X. All'apertura di tale finestra, l'utente non deve fornire un nome utente o una password; non viene eseguita alcuna autenticazione. Bash cerca ~/.bashrc quando ciò accade, poi vi si fa riferimento anche nei file letti all'accesso, il che significa che non è necessario inserire le stesse impostazioni in più file.

In questo .bashrc dell'utente sono definiti un paio di alias e vengono impostate delle variabili per specifici programmi dopo la lettura di /etc/bashrc a livello di sistema:

```
franky ~> cat .bashrc  
# /home/franky/.bashrc  
  
# Source global definitions  
if [ -f /etc/bashrc ]; then  
    . /etc/bashrc  
  
fi  
  
# shell options
```

```

set -o noclobber

# my shell variables

export PS1="\[\033[1;44m\]\u \w\[\033[0m\] "
export PATH="$PATH:~/bin:~/scripts"

# my aliases

alias cdrecord='cdrecord -dev 0,0,0 -speed=8'
alias ss='ssh octarine'
alias ll='ls -la'

# mozilla fix

MOZILLA_FIVE_HOME=/usr/lib/mozilla
LD_LIBRARY_PATH=/usr/lib/mozilla:/usr/lib/mozilla/plugins
MOZ_DIST_BIN=/usr/lib/mozilla
MOZ_PROGRAM=/usr/lib/mozilla/mozilla-bin
export MOZILLA_FIVE_HOME LD_LIBRARY_PATH MOZ_DIST_BIN MOZ_PROGRAM

# font fix
alias xt='xterm -bg black -fg white &'

# BitchX settings
export IRCNAME="frnk"

# THE END
franky ~>

```

Altri esempi si trovano nel pacchetto Bash. Si rammenti che i file di esempio potrebbero richiedere delle modifiche per funzionare nel tuo ambiente.

Gli alias sono discussi nella [Sezione 3.5](#).

3.1.2.5. ~/.bash_logout

Questo file contiene istruzioni specifiche per la procedura di logout. Nell'esempio, la finestra del terminale viene cancellata al logout. Questo è utile per le connessioni remote, che lasceranno una finestra pulita dopo averle chiuse.

```

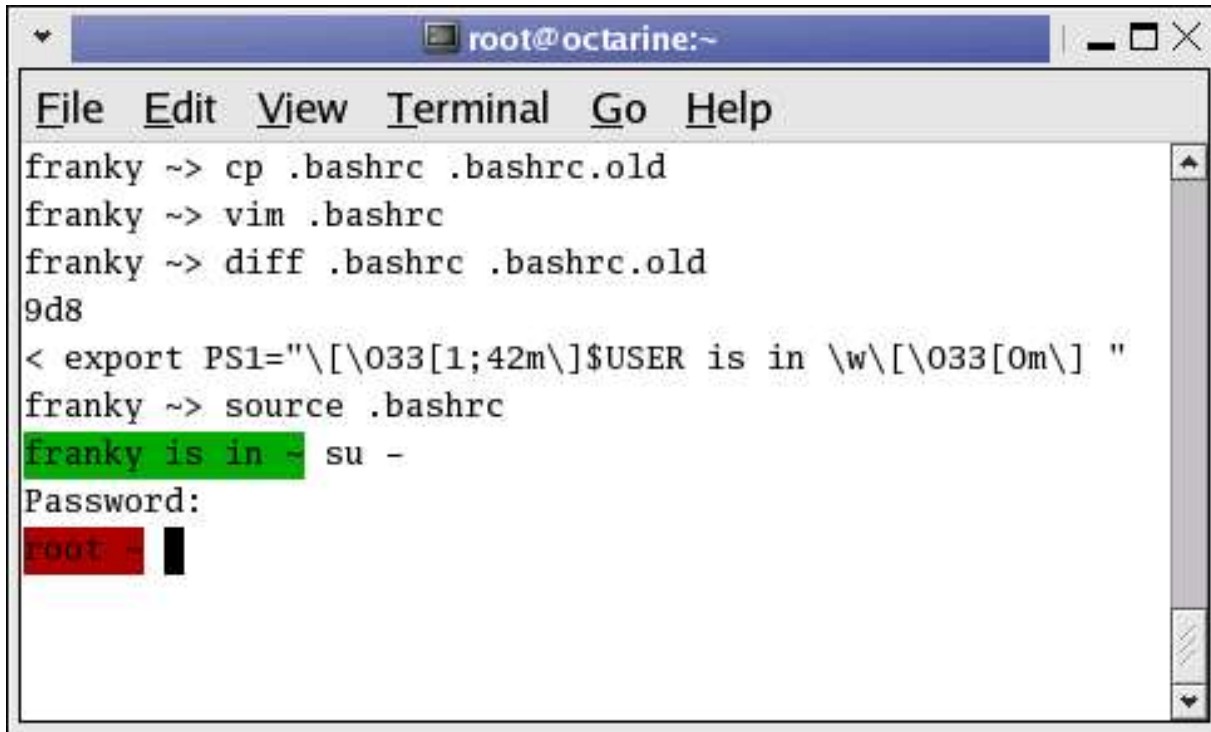
franky ~> cat .bash_logout
#####
#                                                                    #
#   Bash_logout file                                              #
#                                                                    #
#   commands to perform from the bash shell at logout time      #
#                                                                    #
#####
clear
franky ~>

```

3.1.3. Modifica dei file di configurazione della shell

Quando si apportano modifiche a uno dei file di cui sopra, gli utenti devono riconnettersi al sistema o eseguire il **source** del file modificato affinché le modifiche abbiano effetto. Interpretando lo script in questo modo, le modifiche vengono applicate alla sessione della shell corrente:

Figura 3-1. Prompt diversi per utenti diversi



La maggior parte degli script di shell vengono eseguiti in un ambiente privato: le variabili non vengono ereditate dai processi figlio a meno che non vengano esportate dalla shell padre. Il 'source' di un file contenente i comandi della shell è un modo per applicare le modifiche al proprio ambiente e impostare le variabili nella shell corrente.

Questo esempio mostra anche l'uso di impostazioni di prompt diverse da parte di utenti diversi. In questo caso, il rosso significa pericolo. Quando si ha un prompt verde, non c'è da preoccuparsi troppo.

Notare che **source resourcefile** è uguale a **. resourcefile**.

Se ci si dovesse perdere in tutti questi file di configurazione e trovarsi di fronte a impostazioni la cui origine non sia chiara, usare le istruzioni **echo**, proprio come per il debug degli script; vedere la [Sezione 2.3.2](#). Si potrebbero aggiungere righe come questa:

```
echo "Now executing .bash_profile.."
```

o come questa:

```
echo "Now setting PS1 in .bashrc:"
export PS1="[some value]"
echo "PS1 is now set to $PS1"
```

3.2. Variabili

3.2.1. Tipi di variabili

Come visto negli esempi sopra, le variabili di shell sono in caratteri maiuscoli per convenzione. Bash mantiene un elenco di due tipi di variabili:

3.2.1.1. Variabili globali

Le variabili globali, o variabili di ambiente, sono disponibili in tutte le shell. I comandi **env** o **printenv** possono essere utilizzati per visualizzare le variabili di ambiente. Questi programmi vengono forniti col pacchetto *sh-utils*.

Di seguito è riportato un tipico output:

```
franky ~> printenv
CC=gcc
CDPATH=.:~/usr/local:/usr:/
CFLAGS=-O2 -fomit-frame-pointer
COLORTERM=gnome-terminal
CXXFLAGS=-O2 -fomit-frame-pointer
DISPLAY=:0
DOMAIN=hq.garrels.be
e=
TOR=vi
FCEDIT=vi
FIGIGNORE=.o:~
G_BROKEN_FILENAMES=1
GDK_USE_XFT=1
GDMSESSION=Default
GNOME_DESKTOP_SESSION_ID=Default
GTK_RC_FILES=/etc/gtk/gtkrc:/nethome/franky/.gtkrc-1.2-gnome2
GWMCOLOR=darkgreen
GWMTERM=xterm
HISTFILESIZE=5000
history_control=ignoredups
HISTSIZE=2000
HOME=/nethome/franky
HOSTNAME=octarine.hq.garrels.be
INPUTRC=/etc/inputrc
IRCNAME=franky
JAVA_HOME=/usr/java/j2sdk1.4.0
LANG=en_US
LD_FLAGS=-s
LD_LIBRARY_PATH=/usr/lib/mozilla:/usr/lib/mozilla/plugins
LESSCHARSET=latin1
LESS=-edfMQ
LESSOPEN=|/usr/bin/lesspipe.sh %s
LEX=flex
LOCAL_MACHINE=octarine
LOGNAME=franky
LS_COLORS=no=00:fi=00:di=01;34:ln=01;36:pi=40;33:so=01;35:bd=40;33;01:cd=40;33;01:or=01;05;37;41:mi=01;05;37;41:ex=01;32:*.cmd=01;32:*.exe=01;32:*.com=01;32:*.btm=01;32:*.bat=01;32:*.sh=01;32:*.csh=01;32:*.tar=01;31:*.tgz=01;31:*.arj=01;31:*.taz=01;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.bz2=01;31:*.bz=01;31:*.tz=01;31:*.rpm=01;31:*.cpio=01;31:*.jpg=01;35:*.gif=01;35:*.bmp=01;35:*.xbm=01;35:*.xpm=01;35:*.png=01;35:*.tif=01;35:
MACHINES=octarine
MAILCHECK=60
MAIL=/var/mail/franky
MANPATH=/usr/man:/usr/share/man:/usr/local/man:/usr/X11R6/man
MEAN_MACHINES=octarine
MOZ_DIST_BIN=/usr/lib/mozilla
MOZILLA_FIVE_HOME=/usr/lib/mozilla
MOZ_PROGRAM=/usr/lib/mozilla/mozilla-bin
MTOOLS_FAT_COMPATIBILITY=1
MYMALLOC=0
NNTPPORT=119
NNTPSERVER=news
```

```

NPX_PLUGIN_PATH=/plugin/ns4plugin/:/usr/lib/netscape/plugins
OLDPWD=/nethome/franky
OS=Linux
PAGER=less
PATH=/nethome/franky/bin.Linux:/nethome/franky/bin:/usr/local/bin:/usr/local/sbin:/usr/X11R6/bin:/usr/bin:/usr/sbin:/bin:/sbin:.
PS1=\[\033[1;44m\]franky is in \w\[\033[0m\]
PS2=More input>
PWD=/nethome/franky
SESSION_MANAGER=local/octarine.hq.garrels.be:/tmp/.ICE-unix/22106
SHELL=/bin/bash
SHELL_LOGIN=--login
SHLVL=2
SSH_AGENT_PID=22161
SSH_ASKPASS=/usr/libexec/openssh/gnome-ssh-askpass
SSH_AUTH_SOCK=/tmp/ssh-XXmhQ4fC/agent.22106
START_WM=twm
TERM=xterm
TYPE=type
USERNAME=franky
USER=franky
_=/usr/bin/printenv
VISUAL=vi
WINDOWID=20971661
XAPPLRESDIR=/nethome/franky/app-defaults
XAUTHORITY=/nethome/franky/.Xauthority
XENVIRONMENT=/nethome/franky/.Xdefaults
XFILESEARCHPATH=/usr/X11R6/lib/X11/%L/%T/%N%C%S:/usr/X11R6/lib/X11/%l/%T/%N%C%S:/usr/X11R6/lib/X11/%T/%N%C%S:/usr/X11R6/lib/X11/%L/%T/%N%S:/usr/X11R6/lib/X11/%l/%T/%N%S:/usr/X11R6/lib/X11/%T/%N%S
XKEYSYMDB=/usr/X11R6/lib/X11/XKeysymDB
XMODIFIERS=@im=none
XTERMID=
XWINHOME=/usr/X11R6
X=X11R6
YACC=bison -y

```

3.2.1.2. Variabili locali

Le variabili locali sono disponibili solo nella shell corrente. Utilizzando il comando interno **set**, senza alcuna opzione, verrà visualizzato un elenco di tutte le variabili (incluse quelle di ambiente) e le funzioni. L'output verrà ordinato in base alle impostazioni internazionali correnti e visualizzato in un formato riutilizzabile.

Di seguito è riportato un file diff realizzato confrontando l'output di **printenv** e **set**, dopo aver ommesso le funzioni visualizzate anche dal comando **set**:

```

franky ~> diff set.sorted printenv.sorted | grep "<" | awk '{ print $2 }'
BASE=/nethome/franky/.Shell/hq.garrels.be/octarine.alias
BASH=/bin/bash
BASH_VERSINFO=( [0]="2"
BASH_VERSION='2.05b.0 (1)-release'
COLUMNS=80
DIRSTACK=()
DO_FORTUNE=
EUID=504
GROUPS=()
HERE=/home/franky
HISTFILE=/nethome/franky/.bash_history
HOSTTYPE=i686
IFS=$'
LINES=24

```



```

MACHTYPE=i686-pc-linux-gnu
OPTERR=1
OPTIND=1
OSTYPE=linux-gnu
PIPESTATUS=( [0]="0" )
PPID=10099
PS4='+
PWD_REAL='pwd
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-
comments:monitor
THERE=/home/franky
UID=504

```



Awk

il linguaggio di programmazione GNU Awk è spiegato nel [Capitolo 6](#)

3.2.1.3. Variabili per contenuto

Oltre a dividere le variabili in locali e globali, possiamo anche dividerle in categorie in base al tipo di contenuto che questa contiene. A questo proposito, le variabili sono di 4 tipi:

- Variabili stringa
- Variabili intere
- Variabili Costanti
- Variabili array

Discuteremo questi tipi nel [Capitolo 10](#). Per ora, lavoreremo con valori interi e stringa per le variabili.

3.2.2. Creazione di variabili

Le variabili fanno distinzione tra maiuscole e minuscole per default la prima lettera è in maiuscolo. Dare alle variabili locali un nome minuscolo è una convenzione che a volte viene applicata. Tuttavia, si è liberi di utilizzare i nomi che si desiderano o di mischiare maiuscole e minuscole. Le variabili possono anche contenere cifre, ma non è consentito un nome che inizia con una cifra:

```

prompt> export lnumber=1
bash: export: `lnumber=1': not a valid identifier

```

Per impostare una variabile nella shell, si usa

VARNAME="value"

Mettere spazi attorno al segno di uguale causerà degli errori. È buona abitudine mettere le stringhe tra virgolette quando si assegnano valori alle variabili: questo ridurrà la possibilità di commettere errori.

Alcuni esempi di utilizzo di maiuscole e minuscole, numeri e spazi:

```

franky ~> MYVAR1="2"

franky ~> echo $MYVAR1
2

```

```
franky ~> first_name="Franky"

franky ~> echo $first_name
Franky

franky ~> full_name="Franky M. Singh"

franky ~> echo $full_name
Franky M. Singh

franky ~> MYVAR-2="2"
bash: MYVAR-2=2: command not found

franky ~> MYVAR1 ="2"
bash: MYVAR1: command not found

franky ~> MYVAR1= "2"
bash: 2: command not found

franky ~> unset MYVAR1 first_name full_name

franky ~> echo $MYVAR1 $first_name $full_name
<--no output-->

franky ~>
```

3.2.3. Esportare le variabili

Una variabile creata come quelle nell'esempio sopra è disponibile solo per la shell corrente. È una variabile locale: i processi figli della shell corrente non saranno a conoscenza di questa variabile. Per passare le variabili a una sub-shell, dobbiamo *esportarle* usando il comando interno **export**. Le variabili esportate vengono dette variabili di ambiente. L'impostazione e l'esportazione vengono generalmente eseguite in un unico passaggio:

```
export VARNAME="value"
```

Una subshell può modificare le variabili che ha ereditato dal genitore, ma le modifiche apportate dal figlio non influiscono sul genitore. Ciò è mostrato nell'esempio:

```
franky ~> full_name="Franky M. Singh"

franky ~> bash

franky ~> echo $full_name

franky ~> exit

franky ~> export full_name

franky ~> bash

franky ~> echo $full_name
Franky M. Singh

franky ~> export full_name="Charles the Great"

franky ~> echo $full_name
Charles the Great
```

```
franky ~> exit

franky ~> echo $full_name
Franky M. Singh

franky ~>
```

Quando si tenta di leggere per la prima volta il valore di `full_name` in una subshell, questa non è presente (**echo** mostra una stringa nulla). La subshell finisce, e `full_name` viene esportato nel genitore - una variabile può essere esportata dopo che le è stato assegnato un valore. Poi viene avviata una nuova subshell, in cui è visibile la variabile esportata dal genitore. La variabile viene modificata per contenere un altro nome, ma il valore per questa variabile nel genitore rimane lo stesso.

3.2.4. Variabili riservate

3.2.4.1. Variabili riservate della shell Bourne

Bash utilizza determinate variabili di shell allo stesso modo della shell Bourne. In alcuni casi, Bash assegna un valore predefinito alla variabile. La tabella seguente fornisce una panoramica di queste variabili shell semplici:

Tabella 3-1. Variabili shell Bourne riservate

Nome della variabile	Definizione
CDPATH	Un elenco di directory separate da due punti utilizzato come percorso di ricerca per il comando interno cd .
HOME	La directory home dell'utente corrente; l'impostazione predefinita per il cd interno. Il valore di questa variabile viene utilizzato anche dall'espansione della tilde.
IFS	Un elenco di caratteri che separano i campi; usato quando la shell divide le parole come parte dell'espansione.
MAIL	Se questo parametro è impostato su un nome file e la variabile <code>MAILPATH</code> non è impostata, Bash informa l'utente dell'arrivo della posta nel file specificato.
MAILPATH	Un elenco separato da due punti di nomi di file che la shell controlla periodicamente per le nuove mail.
OPTARG	Il valore dell'ultimo argomento dell'opzione elaborato dal getopts integrato.
OPTIND	L'indice dell'ultimo argomento dell'opzione elaborato dal getopts integrato.
PATH	Un elenco di directory separate da due punti in cui la shell cerca i comandi.
PS1	La stringa principale del prompt. Il valore predefinito è <code>"s-v\\$ "</code> .
PS2	La stringa secondaria del prompt. Il valore di default è <code>"> "</code> .

3.2.4.2. Variabili Bash riservate

Queste variabili sono impostate o utilizzate da Bash, ma altre shell normalmente non le trattano in modo speciale.

Tabella 3-2. Variabili Bash riservate

Nome della variabile	Definizione
auto_resume	Questa variabile controlla come la shell interagisce con l'utente e il controllo del job.
BASH	Il percorso completo utilizzato per eseguire l'istanza corrente di Bash.
BASH_ENV	Se questa variabile è impostata quando viene invocato Bash per eseguire uno script di shell, il suo valore viene espanso e utilizzato come nome di un file di avvio da leggere prima di eseguire lo script.
BASH_VERSION	Il numero di versione dell'istanza corrente di Bash.
BASH_VERSINFO	Una variabile array di sola lettura i cui membri contengono informazioni sulla versione per l'istanza corrente di Bash.
COLUMNS	Utilizzata dal comando select per determinare la larghezza del terminale durante la stampa delle liste di selezione. Impostato automaticamente alla ricezione di un segnale <i>SIGWINCH</i> .
COMP_CWORD	Un indice in <code>\${COMP_WORDS}</code> della parola che contiene la posizione corrente del cursore.
COMP_LINE	La riga di comando corrente.
COMP_POINT	L'indice della posizione corrente del cursore rispetto all'inizio del comando corrente.
COMP_WORDS	Una matrice variabile costituita dalle singole parole nella riga di comando corrente.
COMPREPLY	Un array da cui Bash legge i possibili completamenti generati da una funzione shell invocata dalla funzione di completamento programmabile.
DIRSTACK	Un array contenente il contenuto corrente dello stack della directory.
EUID	L'effettivo ID utente numerico dell'utente corrente.
FCEDIT	L'editor utilizzato come predefinito dall'opzione <code>-e</code> del comando interno fc .
FIGIGNORE	Un elenco di suffissi separati da due punti da ignorare quando si esegue il completamento del nome file.
FUNCNAME	Il nome di qualsiasi funzione di shell attualmente in esecuzione.
GLOBIGNORE	Un elenco di pattern separati da due punti che definiscono l'insieme di nomi di file da ignorare nell'espansione del nome di file.
GROUPS	Un array contenente l'elenco dei gruppi di cui l'utente corrente è membro.
histchars	Fino a tre caratteri che controllano l'espansione della cronologia, la sostituzione rapida e la <i>tokenization</i> .
HISTCMD	Il numero della cronologia, o indice nell'elenco della cronologia, del comando corrente.
HISTCONTROL	Definisce se un comando viene aggiunto al file di cronologia.
HISTFILE	Il nome del file in cui viene salvata la cronologia dei comandi. Il valore di default è <code>~/.bash_history</code> .
HISTFILESIZE	Il numero massimo di righe contenute nel file di cronologia, il valore predefinito è 500.
HISTIGNORE	Un elenco di pattern separati da due punti utilizzati per decidere quali righe di comando devono essere salvate nell'elenco della cronologia.

Nome della variabile	Definizione
HISTSIZE	Il numero massimo di comandi da ricordare nell'elenco della cronologia, il valore predefinito è 500.
HOSTFILE	Contiene il nome di un file nello stesso formato di <code>/etc/hosts</code> che dovrebbe essere letto quando la shell deve completare un nome host.
HOSTNAME	Il nome dell'host corrente.
HOSTTYPE	Una stringa che descrive la macchina su cui è in esecuzione Bash.
IGNOREEOF	Controlla l'azione della shell alla ricezione di un carattere <i>EOF</i> come unico input.
INPUTRC	Il nome del file di inizializzazione Readline, che sovrascrive il valore di default <code>/etc/inputrc</code> .
LANG	Utilizzato per determinare la categoria della lingua locale per qualsiasi categoria non specificatamente selezionata con una variabile che inizia con <code>LC_</code> .
LC_ALL	Questa variabile sovrascrive il valore di <code>LANG</code> e qualsiasi altra variabile <code>LC_</code> che specifica una categoria di lingua locale.
LC_COLLATE	Questa variabile determina la 'collation', nell'ordinamento dei risultati dell'espansione del nome file e determina il comportamento delle espressioni di 'range', delle classi di equivalenza e delle 'collating sequence' all'interno dell'espansione del nome file e del 'pattern matching'.
LC_CTYPE	Questa variabile determina l'interpretazione dei caratteri e il comportamento delle classi di caratteri all'interno dell'espansione dei nomi dei file e del 'pattern matching'.
LC_MESSAGES	Questa variabile determina la lingua locale utilizzata per tradurre le stringhe tra virgolette precedute da un segno "\$".
LC_NUMERIC	Questa variabile determina la categoria delle impostazioni internazionali utilizzata per la formattazione dei numeri.
LINENO	Il numero di riga nello script o nella funzione di shell attualmente in esecuzione.
LINES	Utilizzato dal comando interno select per determinare la lunghezza della colonna per la stampa degli elenchi di selezione.
MACHTYPE	Una stringa che descrive completamente il tipo di sistema su cui è in esecuzione Bash, nel formato standard GNU CPU-COMPANY-SYSTEM.
MAILCHECK	La frequenza (in secondi) con cui la shell deve controllare la posta nei file specificati nelle variabili <code>MAILPATH</code> o <code>MAIL</code> .
OLDPWD	La directory di lavoro precedente impostata dal comando interno cd .
OPTERR	Se impostato a 1, Bash visualizza i messaggi di errore generati dal comando interno getopts .
OSTYPE	Una stringa che descrive il sistema operativo su cui è in esecuzione Bash.
PIPESTATUS	Un array contenente un elenco di valori dello stato di uscita dai processi nella pipeline in primo piano eseguita più di recente (che può contenere solo un singolo comando).
POSIXLY_CORRECT	Se questa variabile è nell'ambiente all'avvio di bash , la shell entra in modalità POSIX.

Nome della variabile	Definizione
PPID	L'ID del processo padre della shell.
PROMPT_COMMAND	Se impostato, il valore viene interpretato come un comando da eseguire prima della stampa di ogni prompt primario (PS1).
PS3	Il valore di questa variabile viene utilizzato come prompt per il comando select . Il default è "#? "
PS4	Il valore è il prompt stampato prima dell'eco della riga di comando quando è impostata l'opzione -x; il valore di default è "+ "
PWD	La directory di lavoro corrente come impostata dal comando cd .
RANDOM	Ogni volta che si fa riferimento a questo parametro, viene generato un numero intero casuale compreso tra 0 e 32767. L'assegnazione di un valore a questa variabile 'semina' il generatore di numeri casuali.
REPLY	La variabile predefinita per il comando read
SECONDS	Questa variabile si espande al numero di secondi dall'avvio della shell.
SHELLOPTS	Un elenco separato da due punti delle opzioni abilitate della shell.
SHLVL	Incrementato di uno ogni volta che viene avviata una nuova istanza di Bash.
TIMEFORMAT	Il valore di questo parametro viene utilizzato come stringa di formato specificando come devono essere visualizzate le informazioni sui tempi per le pipeline precedute dalla parola riservata time .
TMOUT	Se impostato su un valore maggiore di zero, TMOUT viene considerato come il timeout predefinito per il comando read . In una shell interattiva, il valore viene interpretato come il numero di secondi di attesa dell'input dopo l'emissione del prompt primario. Bash termina dopo quel numero di secondi se l'input non arriva.
UID	L'ID utente numerico reale dell'utente corrente.

Controllare le pagine man, info o doc di Bash per ulteriori informazioni. Alcune variabili sono di sola lettura, altre vengono impostate automaticamente e altre perdono il loro significato se impostate su un valore diverso da quello predefinito.

3.2.5. Parametri Speciali

La shell gestisce diversi parametri in modo speciale. Questi parametri possono essere solo referenziati; l'assegnazione non è consentita.

Tabella 3-3. Variabili bash speciali

Carattere	Definizione
\$*	Si espande nei parametri posizionali, a partire da uno. Quando l'espansione avviene tra virgolette, si espande in una singola parola col valore di ciascun parametro separato dal primo carattere della variabile speciale IFS .
\$@	Si espande nei parametri posizionali, a partire da uno. Quando l'espansione avviene tra virgolette, ogni parametro si espande in una parola separata.
\$#	Si espande col numero di parametri posizionali in decimale.
\$?	Si espande con lo stato di uscita della pipeline in primo piano eseguita più recente.

Carattere	Definizione
\$-	Un trattino si espande nei flag dell'opzione corrente come specificato al momento dell'invocazione, dal comando integrato set o da quelli impostati dalla shell stessa (come -i).
\$\$	Si espande nell'ID del processo della shell.
\$!	Si espande nell'ID processo del comando in background (asincrono) eseguito più di recente.
\$0	Si espande col nome della shell o dello script della shell.
\$_	La variabile 'underscore' viene impostata all'avvio della shell e contiene il nome assoluto del file della shell o dello script in esecuzione come passato nell'elenco degli argomenti. Successivamente, si espande all'ultimo argomento del comando precedente, dopo l'espansione. È anche impostato col percorso completo di ogni comando eseguito e posizionato nell'ambiente esportato in quel comando. Quando si controlla la posta, questo parametro contiene il nome del file di posta.

\$* vs. @\$

L'implementazione di "\$*" è sempre stata un problema e realisticamente avrebbe dovuto essere sostituita con il comportamento di "\$@". In quasi tutti i casi in cui i programmatori usano "\$*", intendono "\$@". "\$*" Può causare bug e persino falle di sicurezza nel software.

I parametri posizionali sono le parole che seguono il nome di uno script di shell. Vengono inseriti nelle variabili \$1, \$2, \$3 e così via. Se necessario, le variabili vengono aggiunte a un array interno. \$# contiene il numero totale di parametri, come mostrato da questo semplice script:

```
#!/bin/bash

# positional.sh
# Questo script legge 3 parametri posizionali e li stampa.

POSPAR1="$1"
POSPAR2="$2"
POSPAR3="$3"

echo "$1 is the first positional parameter, \"$1.\"
echo "$2 is the second positional parameter, \"$2.\"
echo "$3 is the third positional parameter, \"$3.\"
echo
echo "The total number of positional parameters is $#."
```

All'esecuzione si potrebbe dare un numero qualsiasi di argomenti:

```
franky ~> positional.sh one two three four five
one is the first positional parameter, $1.
two is the second positional parameter, $2.
three is the third positional parameter, $3.

The total number of positional parameters is 5.

franky ~> positional.sh one two
one is the first positional parameter, $1.
two is the second positional parameter, $2.
  is the third positional parameter, $3.

The total number of positional parameters is 2.
```

Maggiori informazioni sulla valutazione di questi parametri si trovano nel [Capitolo 7](#) e nella [Sezione 9.7](#).

Alcuni esempi sugli altri parametri speciali:

```
franky ~> grep dictionary /usr/share/dict/words
dictionary

franky ~> echo $_
/usr/share/dict/words

franky ~> echo $$
10662

franky ~> mozilla &
[1] 11064

franky ~> echo $!
11064

franky ~> echo $0
bash

franky ~> echo $?
0

franky ~> ls doesnotexist
ls: doesnotexist: No such file or directory

franky ~> echo $?
1

franky ~>
```

L'utente *franky* inizia a inserire il comando **grep**, che provoca l'assegnazione della variabile `_`. L'ID di processo della sua shell è 10662. Dopo aver messo un job in background, `!` contiene l'ID di processo del lavoro in background. La shell in esecuzione è **bash**. Quando viene commesso un errore, `?` contiene un codice di uscita diverso da 0 (zero).

3.2.6. Riciclo di script con variabili

Oltre a rendere lo script più leggibile, le variabili permetteranno anche di impiegare più velocemente uno script in un altro ambiente o per un altro scopo. Si consideri il seguente esempio, uno script molto semplice che esegue un backup della directory home di *franky* su un server remoto:

```
#!/bin/bash

# Questo script fa un backup della mia home directory.

cd /home

# Questo crea l'archivio
tar cf /var/tmp/home_franky.tar franky > /dev/null 2>&1

# Prima di tutto rimuove il vecchio file bzip2. Reindirizza gli errori perché
# generano problemi se l'archivio
# non esiste. Poi crea un nuovo file compresso.
rm /var/tmp/home_franky.tar.bz2 2> /dev/null
bzip2 /var/tmp/home_franky.tar
```



```
# Copia il file su un altro host: ci sono le chiavi ssh per farlo funzionare
senza intervento.
scp /var/tmp/home_franky.tar.bz2 bordeaux:/opt/backup/franky > /dev/null 2>&1

# Crea un timestamp in un logfile.
date >> /home/franky/log/home_backup.log
echo backup succeeded >> /home/franky/log/home_backup.log
```

Prima di tutto, è più probabile che si commettano errori se si inseriscono manualmente i nomi dei file e delle directory ogni volta che ce n'è bisogno. In secondo luogo, supponiamo che *franky* voglia dare questo script a *carol*, allora carol dovrà fare alcune modifiche prima di poter usare lo script per eseguire il backup della sua home directory. Lo stesso vale se *franky* vuole usare questo script per fare il backup di altre directory. Per un facile riutilizzo, rendere variabili tutti i file, le directory, i nomi utente, i nomi dei server ecc. Pertanto, sarà necessario modificare un valore solo una volta, senza dover scorrere l'intero script per verificare dove c'è un parametro. Questo è un esempio:

```
#!/bin/bash

# Questo script fa un backup della mia home directory.

# Cambiare il valore delle variabili secondo le proprie esigenze:
BACKUPDIR=/home
BACKUPFILES=franky
TARFILE=/var/tmp/home_franky.tar
BZIPFILE=/var/tmp/home_franky.tar.bz2
SERVER=bordeaux
REMOTEDIR=/opt/backup/franky
LOGFILE=/home/franky/log/home_backup.log

cd $BACKUPDIR

# Questo crea l'archivio
tar cf $TARFILE $BACKUPFILES > /dev/null 2>&1

# Prima di tutto rimuove il vecchio file bzip2. Reindirizza gli errori perché
generano problemi se l'archivio
# non esiste. Poi crea un nuovo file compresso.
rm $BZIPFILE 2> /dev/null
bzip2 $TARFILE

# Copia il file su un altro host: ci sono le chiavi ssh per farlo funzionare
senza intervento.
scp $BZIPFILE $SERVER:$REMOTEDIR > /dev/null 2>&1

# Crea un timestamp in un logfile.
date >> $LOGFILE
echo backup succeeded >> $LOGFILE
```



Grandi directory e scarsa larghezza di banda

Quanto sopra è puramente un esempio comprensibile da tutti, utilizzando una piccola directory e un host sulla stessa sottorete. A seconda della larghezza di banda, della dimensione della directory e della posizione del server remoto, può essere necessario molto tempo per eseguire i backup utilizzando questo meccanismo. Per directory più grandi e larghezze di banda inferiori, utilizzare **rsync** per mantenere sincronizzate le directory da entrambi i lati.

3.3. Caratteri tra virgolette

3.3.1. Perché?

Molti caratteri hanno significati speciali in un contesto o in un altro. Le virgolette vengono utilizzate per rimuovere il significato speciale di caratteri o parole: le virgolette possono disabilitare il trattamento speciale per i caratteri speciali, possono impedire che le parole riservate vengano riconosciute come tali e possono disabilitare l'espansione dei parametri.

3.3.2. Caratteri di 'escape'

I caratteri di escape vengono utilizzati per rimuovere il significato speciale da un singolo carattere. In Bash, viene utilizzata una barra rovesciata non quotata, `\`. Essa conserva il valore letterale del carattere successivo che segue, ad eccezione di *newline*. Se compare un carattere di nuova riga subito dopo la barra rovesciata, segna la continuazione di una riga quando è più lunga della larghezza del terminale; il backslash viene rimosso dal flusso di input ed effettivamente ignorato.

```
franky ~> date=20021226

franky ~> echo $date
20021226

franky ~> echo \$date
$date
```

In questo esempio, la variabile `date` viene creata e impostata per contenere un valore. Il primo **eco** mostra il valore della variabile, ma nel secondo il simbolo del dollaro è 'escaped'.

3.3.3. Apici singoli

Le virgolette singole (`"`) vengono utilizzate per preservare il valore letterale di ciascun carattere racchiuso. Una singola virgoletta potrebbe non apparire tra virgolette singole, anche se preceduta da un backslash.

Continuiamo con l'esempio precedente:

```
franky ~> echo '$date'
$date
```

3.3.4. Virgolette doppie

Usando le doppie virgolette viene mantenuto il valore letterale di tutti i caratteri racchiusi, ad eccezione del simbolo del dollaro, dei backtick (virgolette singole all'indietro, ```) e della barra rovesciata.

Il simbolo del dollaro e i backtick mantengono il loro significato speciale all'interno delle doppie virgolette.

La backslash mantiene il suo significato solo se seguita da dollaro, backtick, virgolette doppie, backslash o newline (a capo). All'interno delle virgolette, le barre rovesciate vengono rimosse dal flusso di input quando sono seguite da uno di questi caratteri. Le backslash che precedono i caratteri che non hanno un significato speciale non vengono modificate per essere elaborate dall'interprete della shell.

Una virgoletta doppia può essere virgolettata tra virgolette doppie facendola precedere da un backslash.

```
franky ~> echo "$date"
20021226

franky ~> echo "`date`"
Sun Apr 20 11:22:06 CEST 2003

franky ~> echo "I'd say: \"Go for it!\""
I'd say: "Go for it!"

franky ~> echo "\""
More input>"

franky ~> echo "\"\""
\
```

3.3.5. ANSI-C tra virgolette

Le parole nella forma "\$'STRING'" vengono trattate in modo speciale. La parola viene espansa in una stringa, i caratteri di 'escaped' col backslash vengono sostituiti come specificato dallo standard ANSI-C. Le sequenze di escape col backslash si possono trovare nella documentazione Bash.

3.3.6. Lingua 'locale'

Una stringa tra virgolette doppie preceduta dal simbolo del dollaro farà sì che la stringa venga tradotta in base al linguaggio locale corrente. Se il linguaggio locale corrente è "C" o "POSIX", il simbolo del dollaro viene ignorato. Se la stringa viene tradotta e sostituita, la sostituzione è tra virgolette.

3.4. Espansione della shell

3.4.1. In generale

Dopo che il comando è stato suddiviso in *token* (vedi [Sezione 1.4.1.1](#)), questi token o parole vengono espansi o risolti. Vengono eseguiti otto tipi di espansione, di cui parleremo nelle sezioni successive, nell'ordine in cui vengono espansi.

Dopo tutte le espansioni, viene eseguita la rimozione delle virgolette.

3.4.2. Espansione delle parentesi graffe "Brace expansion"

L'espansione delle parentesi graffe è un meccanismo mediante il quale possono essere generate stringhe arbitrarie. I pattern da espandere con parentesi graffe assumono la forma di un *PREAMBLE* opzionale, seguito da una serie di stringhe separate da virgole tra una coppia di parentesi graffe, seguite da un *POSTSCRIPT* opzionale. Il preambolo è preceduto da ogni stringa contenuta all'interno delle parentesi graffe, e il postscript viene poi aggiunto a ciascuna stringa risultante, espandendosi da sinistra a destra.

Le espansioni delle parentesi graffe possono essere nidificate. I risultati di ogni stringa espansa non vengono ordinati; viene mantenuto l'ordine da sinistra a destra:

```
franky ~> echo sp{el,il,al}l
spell spill spall
```

L'espansione della parentesi graffa viene eseguita prima di qualsiasi altra espansione e tutti i caratteri speciali di altre espansioni vengono conservati nel risultato. Essa è rigorosamente testuale. Bash non applica alcuna interpretazione sintattica al contesto dell'espansione o al testo tra le parentesi graffe. Per evitare conflitti con l'espansione dei parametri, la stringa "\${" non è considerata idonea per l'espansione delle parentesi graffe.

Un'espansione di parentesi graffa correttamente formata deve contenere parentesi graffe di apertura e chiusura senza virgolette e almeno una virgola senza virgolette. Qualsiasi espansione delle graffe formata in modo errato viene lasciata invariata.

3.4.3. Espansione della tilde

Se una parola inizia con una tilde senza virgolette ("~"), tutti i caratteri fino alla prima barra senza virgolette (o tutti i caratteri, se non ci sono barre senza virgolette) vengono considerati *tilde-prefisso*. Se nessuno dei caratteri nel prefisso tilde è virgolettato, i caratteri nel prefisso tilde che seguono la tilde vengono trattati come un possibile nome di login. Se questo nome di login è la stringa nulla, la tilde viene sostituita con il valore della variabile shell `HOME`. Se `HOME` non è impostato, viene invece sostituita la directory home dell'utente che esegue la shell. In caso contrario, il prefisso tilde viene sostituito con la directory home associata alla login specificata.

Se il prefisso tilde è "~+", il valore della variabile shell `PWD` sostituisce il prefisso della tilde. Se il prefisso della tilde è "~-", il valore della variabile shell `OLDPWD`, se impostato, viene sostituito.

Se i caratteri che seguono la tilde sono costituiti da un numero N, eventualmente preceduto da un "+" o da un "-", il prefisso della tilde viene sostituito con l'elemento corrispondente dallo stack delle directory, come verrebbe visualizzato dal comando **dirs** invocato con i caratteri che seguono la tilde come argomento. Se il prefisso della tilde, senza la tilde, è costituito da un numero senza "+" o "-" iniziale, si presume un "+".

Se la login non è valida o l'espansione della tilde fallisce, la parola rimane invariata.

Ogni assegnazione di variabile viene verificata per i prefissi della tilde senza virgolette immediatamente dopo un ":" o "=". In questi casi viene eseguita anche l'espansione della tilde. Ne risulta che si possono usare nomi di file con tilde nelle assegnazioni a `PATH`, `MAILPATH` e `CDPATH`, e la shell assegna il valore espanso.

Esempio:

```
franky ~> export PATH="$PATH:~/testdir"
```

~/testdir verrà espanso in \$HOME/testdir, quindi se \$HOME è /var /home/franky, la directory /var/home/franky/testdir verrà aggiunta al contenuto della variabile PATH.

3.4.4. Parametro della shell ed espansione variabile

Il carattere "\$" introduce l'espansione dei parametri, la sostituzione dei comandi o l'espansione aritmetica. Il nome del parametro o il simbolo da espandere può essere racchiuso tra parentesi graffe, che sono facoltative ma servono a proteggere la variabile da espandere da caratteri immediatamente successivi che potrebbero essere interpretati come parte del nome.

Quando vengono utilizzate le parentesi graffe, la graffa finale corrispondente è la prima "}" non 'escaped' da un backslash o all'interno di una stringa tra virgolette e non all'interno di un'espansione aritmetica implicita, sostituzione di comandi o espansione di parametri.

La forma base dell'espansione dei parametri è "\${PARAMETER}". Viene sostituito il valore di "PARAMETER". Le parentesi graffe sono obbligatorie quando "PARAMETER" è un parametro posizionale con più di una cifra, oppure quando "PARAMETER" è seguito da un carattere che non deve essere interpretato come parte del suo nome.

Se il primo carattere di "PARAMETER" è un punto esclamativo, Bash usa il valore della variabile formato dal resto di "PARAMETER" come nome della variabile; questa variabile viene quindi espansa e quel valore viene utilizzato nel resto della sostituzione, anziché il valore di "PARAMETER" stesso. Questo è noto come *espansione indiretta*.

Sicuramente si avrà familiarità con l'espansione diretta dei parametri, poiché accade sempre, anche nei casi più semplici, come quello sopra o il seguente:

```
franky ~> echo $SHELL
/bin/bash
```

La seguente è un'espansione indiretta:

```
franky ~> echo ${!N*}
NNTPPORT NNTPSERVER NPX_PLUGIN_PATH
```

Si noti che non è lo stesso di **echo \$N***.

Il costrutto seguente consente la creazione della variabile se non esiste ancora:

\${VAR:=value}

Esempio:

```
franky ~> echo $FRANKY

franky ~> echo ${FRANKY:=Franky}
Franky
```

Tuttavia, i parametri speciali, tra cui i parametri posizionali, non possono essere assegnati in questo modo.

Discuteremo ulteriormente l'uso delle parentesi graffe per il trattamento delle variabili nel [Capitolo 10](#). Ulteriori informazioni si possono trovare anche nelle pagine informative di Bash.

3.4.5. Sostituzione del comando

La sostituzione del comando consente all'output di un comando di sostituire il comando stesso. La sostituzione del comando si ha quando un comando è racchiuso in questo modo:

`$(command)`

o in questo modo usando i backtick:

``command``

Bash esegue l'espansione eseguendo COMMAND e rimpiazzando la sostituzione del comando con l'output standard del comando, eliminando eventuali newline finali. Le newlines interne non vengono eliminate, ma possono essere rimosse durante la suddivisione in parole.

```
franky ~> echo `date`  
Thu Feb 6 10:06:20 CET 2003
```

Quando viene utilizzata sostituzione vecchio tipo tra virgolette rovesciate, la backslash mantiene il suo significato letterale tranne quando è seguita da "\$", "\"" o "\". I primi backtick non preceduti da un backslash terminano la sostituzione del comando. Quando si utilizza il modulo "\$ (COMMAND)", tutti i caratteri tra parentesi costituiscono il comando; nessuno è trattato in modo speciale.

Le sostituzioni del comando possono essere nidificate. Per nidificare quando si utilizza il modulo tra virgolette, evitare i backtick interni con backslash.

Se la sostituzione appare tra virgolette, la suddivisione in parole e l'espansione del nome del file non vengono eseguite sui risultati.

3.4.6. Espansione aritmetica

L'espansione aritmetica consente la valutazione di un'espressione aritmetica e la sostituzione del risultato. Il formato per l'espansione aritmetica è:

`$((EXPRESSION))`

L'espressione viene trattata come se fosse tra virgolette, ma le virgolette tra parentesi non vengono trattate in modo speciale. Tutti i token nell'espressione subiscono l'espansione dei parametri, la sostituzione dei comandi e la rimozione delle virgolette. Le sostituzioni aritmetiche possono essere nidificate.

La valutazione delle espressioni aritmetiche viene eseguita con interi a larghezza fissa senza alcun controllo per l'overflow, sebbene la divisione per zero venga intercettata e riconosciuta come errore. Gli operatori sono più o meno gli stessi del linguaggio di programmazione C. In ordine di precedenza decrescente, l'elenco si presenta così:

Tabella 3-4. Operatori aritmetici

Operatore	Significato
VAR++ and VAR--	post-incremento e post-decremento di variabile
++VAR and --VAR	pre-incremento e pre-decremento di variabile
- e +	meno e più unario
! e ~	negazione logica di bit
**	elevamento a potenza
*, / e %	moltiplicazione, divisione, resto
+ e -	addizione, sottrazione
<< e >>	slittamento di bit a sinistra e a destra
<=, >=, < e >	operatori di confronto
== e !=	uguaglianza e disuguaglianza
&	AND di bit
^	OR esclusivo di bit
	OR di bit
&&	AND logico
	OR logico
expr ? expr : expr	valutazione condizionale
=, *=, /=, %=, +=, -=, <<=, >>=, &=, ^= and =	assegnazione
,	separatore tra espressioni

Le variabili di shell sono utilizzabili come operandi; l'espansione del parametro viene eseguita prima che l'espressione venga valutata. All'interno di un'espressione, le variabili possono anche essere referenziate per nome senza utilizzare la sintassi di espansione dei parametri. Il valore di una variabile viene valutato come un'espressione aritmetica quando vi si fa riferimento. Non è necessario che una variabile abbia il suo attributo intero attivato per essere utilizzata in un'espressione.

Le costanti con uno 0 (zero) iniziale vengono interpretate come numeri ottali. Uno "0x" o uno "0X" iniziale denota l'esadecimale. Altrimenti, i numeri assumono la forma "[BASE'#']N", dove "BASE" è un numero decimale compreso tra 2 e 64 che rappresenta la base aritmetica e N è un numero in quella base. Se "BASE'#'" viene omissso, viene utilizzata la base 10. Le cifre maggiori di 9 sono rappresentate dalle lettere minuscole, dalle lettere maiuscole, "@" e "_", in questo ordine. Se "BASE" è minore o uguale a 36, le lettere minuscole e maiuscole possono essere utilizzate in modo intercambiabile per rappresentare i numeri tra 10 e 35.

Gli operatori vengono valutati secondo l'ordine di precedenza. Le sottoespressioni tra parentesi vengono valutate per prime e possono sovrascrivere le regole di precedenza descritte.

Ove possibile, gli utenti di Bash dovrebbero provare a utilizzare la sintassi con parentesi quadre:

\$[EXPRESSION]

Tuttavia, questo calcolerà solo il risultato di *EXPRESSION* e non eseguirà test:

```
franky ~> echo ${365*24}
8760
```

Vedere la [Sezione 7.1.2.2](#), ed altre, per esempi pratici negli script.

3.4.7. Sostituzione del processo

La sostituzione dei processi è supportata sui sistemi che supportano le 'named pipe' (FIFO) o il metodo `/dev/fd` per denominare i file aperti. Prende la forma di

`<(LIST)`

o

`>(LIST)`

Il processo `LIST` viene eseguito con il suo input o output connesso a una FIFO o a qualche file in `/dev/fd`. Il nome di questo file viene passato come argomento al comando corrente come risultato dell'espansione. Se viene utilizzato la forma `>(LIST)`, la scrittura nel file fornirà l'input per `LIST`. Se viene utilizzato la forma `<(LIST)`, il file passato come argomento deve essere letto per ottenere l'output di `LIST`. Notare che nessuno spazio può apparire tra i segni `<` o `>` e la parentesi sinistra, altrimenti il costrutto verrebbe interpretato come un reindirizzamento.

Quando disponibile, la sostituzione del processo viene eseguita contemporaneamente all'espansione dei parametri e delle variabili, alla sostituzione dei comandi e all'espansione aritmetica.

Maggiori informazioni nella [Sezione 8.2.3](#).

3.4.8. Divisione della parola

La shell esegue la scansione dei risultati dell'espansione dei parametri, della sostituzione dei comandi e dell'espansione aritmetica che non si trovano tra virgolette per la suddivisione in parole.

La shell tratta ogni carattere di `$IFS` come delimitatore e suddivide i risultati delle altre espansioni in parole su questi caratteri. Se `IFS` non è impostato o il suo valore è esattamente `"<space><tab><newline>"`, il default, allora qualsiasi sequenza di `IFS` caratteri serve per delimitare le parole. Se `IFS` ha un valore diverso da quello predefinito, le sequenze dei caratteri 'whitespace' "spazio" e "Tab" vengono ignorate all'inizio e fine della parola, purché il carattere di whitespace sia nel valore di `IFS` (un `IFS` whitespace). Qualsiasi carattere in `IFS` che non è `IFS` whitespace, insieme a qualsiasi carattere di whitespace `IF` adiacente, delimita un campo. Anche una sequenza di whitespace `IFS` viene considerata come delimitatore. Se il valore di `IFS` è null, non si ha la suddivisione in parole.

Gli argomenti nulli espliciti (`""` o `""`) vengono mantenuti. Gli argomenti nulli impliciti non virgolettati, risultanti dall'espansione di parametri che non hanno valori, vengono rimossi. Se un parametro senza alcun valore viene espanso tra virgolette, ne risulta un argomento nullo e viene mantenuto.



Espansione e suddivisione in parole

Se non si verifica alcuna espansione, non viene eseguita alcuna suddivisione.

3.4.9. Espansione del nome del file

Dopo la suddivisione in parole, a meno che non sia stata impostata l'opzione `-f` (vedere la [Sezione 2.3.2](#)), Bash scansiona ogni parola alla ricerca dei caratteri `"*`, `"?"` e `"["`. Se appare uno di questi caratteri, la parola viene considerata come un *PATTERN* e sostituita con un elenco in ordine alfabetico di nomi di file che corrispondono al modello. Se non vengono trovati nomi di file corrispondenti e l'opzione shell `nullglob` è disabilitata, la parola rimane invariata. Se l'opzione `nullglob` è impostata e non vengono trovate corrispondenze, la parola viene rimossa. Se l'opzione shell `nocaseglob` è abilitata, la corrispondenza viene eseguita indipendentemente dalla maiuscola dei caratteri alfabetici.

Quando viene utilizzato un pattern per la generazione del nome file, il carattere `"."` all'inizio di un nome file o immediatamente dopo una barra deve essere esplicitamente abbinato, a meno che l'opzione shell `dotglob` non sia settata. Quando si fa corrispondere un nome di file, il carattere barra deve essere sempre associato in modo esplicito. In altri casi, il carattere `"."` non viene trattato in modo speciale.

La variabile di shell `GLOBIGNORE` può essere utilizzata per restringere l'insieme dei nomi di file che corrispondono a un pattern. Se `GLOBIGNORE` è settato, ogni nome di file corrispondente che coincide anche a uno dei pattern in `GLOBIGNORE` viene rimosso dall'elenco delle corrispondenze. I nomi dei file `.` e `..` vengono sempre ignorati, anche quando è impostato `GLOBIGNORE`. Tuttavia, l'impostazione di `GLOBIGNORE` ha l'effetto di abilitare l'opzione della `dotglob`, quindi tutti gli altri nomi di file che iniziano con `"."` corrisponderanno [match]. Per ottenere il vecchio comportamento di ignorare i nomi di file che iniziano con `"."`, rendere `"*"` uno dei pattern in `GLOBIGNORE`. L'opzione `dotglob` è disabilitata quando `GLOBIGNORE` non è settata.

3.5. Gli Alias

3.5.1. Cosa sono gli alias?

Un alias consente di sostituire una parola con una stringa quando viene utilizzata come prima parola di un comando semplice. La shell mantiene un elenco degli alias che possono essere impostati e non con i comandi interni **alias** e **unalias**. Inserire **alias** senza opzioni per visualizzare un elenco di alias noti alla shell corrente.

```
franky: ~> alias
alias ..='cd ..'
alias ...='cd ../../'
alias ....='cd ../../..'
alias PAGER='less -r'
alias Txxterm='export TERM=xterm'
alias XARGS='xargs -r'
alias cdrecord='cdrecord -dev 0,0,0 -speed=8'
alias e='vi'
alias egrep='grep -E'
alias ewformat='fdformat -n /dev/fd0u1743; ewfsck'
alias fgrep='grep -F'
alias ftp='ncftp -d15'
alias h='history 10'
alias fformat='fdformat /dev/fd0H1440'
alias j='jobs -l'
alias ksane='setterm -reset'
alias ls='ls -F --color=auto'
alias m='less'
```

```
alias md='mkdir'
alias od='od -Ax -ta -txC'
alias p='pstree -p'
alias ping='ping -vcl'
alias sb='ssh blubber'
alias sl='ls'
alias ss='ssh octarine'
alias tar='gtar'
alias tmp='cd /tmp'
alias unaliasall='unalias -a'
alias vi='eval `resize`;vi'
alias vt100='export TERM=vt100'
alias which='type'
alias xt='xterm -bg black -fg white &'

franky ~>
```

Gli alias sono utili per specificare la versione predefinita di un comando che esiste in diverse versioni sul sistema o per specificare le opzioni predefinite di un comando. Un altro uso degli alias è per correggere l'ortografia errata.

La prima parola di ogni comando semplice, se non virgolettata, viene controllata per vedere se ha un alias. In caso affermativo, quella parola viene sostituita dal testo dell'alias. Il nome dell'alias e il testo sostitutivo possono contenere qualsiasi input valido della shell, inclusi i metacaratteri, con l'eccezione che il nome dell'alias non può contenere "=". La prima parola del testo sostitutivo viene testata per gli alias, ma una parola identica a un alias in fase di espansione non viene espansa una seconda volta. Ciò significa che si può avere un alias da **ls** a **ls -F**, per esempio, e Bash non tenterà di espanderlo ricorsivamente. Se l'ultimo carattere del valore dell'alias è uno spazio o un carattere di tabulazione, anche la parola del comando successiva all'alias verrà controllata per l'espansione dell'alias.

Gli alias non vengono espansi quando la shell non è interattiva, a meno che sia impostata l'opzione `expand_aliases` utilizzando il comando **shopt**.

3.5.2. Creazione e rimozione degli alias

Gli alias vengono creati utilizzando il comando **alias** nella shell. Per un uso permanente, inserire l'**alias** in uno dei file di inizializzazione della shell; se si inserisce semplicemente l'alias sulla riga di comando, viene riconosciuto solo all'interno della shell corrente

```
franky ~> alias dh='df -h'

franky ~> dh
Filesystem      Size  Used Avail Use% Mounted on
/dev/hda7       1.3G  272M 1018M  22% /
/dev/hda1       121M   9.4M  105M   9% /boot
/dev/hda2       13G   8.7G   3.7G  70% /home
/dev/hda3       13G   5.3G   7.1G  43% /opt
none            243M    0   243M   0% /dev/shm
/dev/hda6       3.9G   3.2G   572M  85% /usr
/dev/hda5       5.2G   4.3G   725M  86% /var

franky ~> unalias dh

franky ~> dh
bash: dh: command not found

franky ~>
```

Bash legge sempre almeno una riga completa di input prima di eseguire qualsiasi comando su tale riga. Gli alias vengono espansi quando viene letto un comando, non quando viene eseguito. Pertanto, una definizione di alias che appare sulla stessa riga di un altro comando non ha effetto finché non viene letta la successiva riga di input. I comandi che seguono la definizione dell'alias su quella riga non sono interessati dal nuovo alias. Questo comportamento è anche un problema quando vengono eseguite le funzioni. Gli alias vengono espansi quando viene letta una definizione di funzione, non quando viene eseguita la funzione, perché una definizione di funzione è essa stessa un comando composto. Di conseguenza, gli alias definiti in una funzione non sono disponibili fino a quando tale funzione non viene eseguita. Per sicurezza, mettere sempre le definizioni degli alias su una riga separata e non usare **alias** nei comandi composti.

Gli alias non vengono ereditati dai processi figlio. La Bourne Shell (**sh**) non riconosce gli alias.

Maggiori informazioni sulle funzioni sono nel [Capitolo 11](#).

Le funzioni sono più veloci

Gli alias vengono esaminati dopo le funzioni e quindi la risoluzione è più lenta. Sebbene gli alias siano più facili da comprendere, le funzioni della shell sono da preferire agli alias per quasi tutti gli scopi.

3.6. Altre opzioni di Bash

3.6.1. Opzioni di visualizzazione

Abbiamo già discusso un paio di opzioni Bash utili per il debug degli script. In questa sezione, daremo uno sguardo più approfondito alle opzioni di Bash.

Usare `-o` per impostare con **set** per visualizzare tutte le opzioni della shell:

```
willy:~> set -o
allexport          off
braceexpand       on
emacs             on
errexit           off
hashall           on
histexpand        on
history           on
ignoreeof         off
interactive-comments on
keyword           off
monitor           on
noclobber         off
noexec            off
noglob            off
nolog             off
notify            off
nounset           off
onecmd            off
physical          off
posix             off
privileged        off
verbose           off
vi               off
xtrace            off
```

Vedere le pagine di info di Bash, sezione Shell Built-in Commands->The Set Built-i per una descrizione di ciascuna opzione. Molte opzioni si possono abbreviare con un solo carattere: l'opzione `xtrace`, ad esempio, equivale a specificare `set -x`.

3.6.2. Modifica delle opzioni

Le opzioni possono essere impostate in modo diverso dal default quando si chiama la shell, oppure possono essere impostate mentre la shell è in vita. Si possono anche includere nei file di configurazione delle risorse della shell.

Il seguente comando esegue uno script in modalità compatibile con POSIX:

```
willy:~/scripts> bash --posix script.sh
```

Per modificare temporaneamente l'ambiente corrente o per l'utilizzo in uno script, si preferisce usare `set`. Usare `-` (trattino) per abilitare un'opzione, `+` per disabilitarla:

```
willy:~/test> set -o noclobber
willy:~/test> touch test
willy:~/test> date > test
bash: test: cannot overwrite existing file
willy:~/test> set +o noclobber
willy:~/test> date > test
```

L'esempio sopra mostra l'opzione `noclobber`, che impedisce la sovrascrittura dei file esistenti da parte dalle operazioni di reindirizzamento. Lo stesso vale per le opzioni mono-carattere, ad esempio `-u`, che considererà le variabili non impostate come un errore quando impostate uscendo dalla shell, se non è interattiva, incontrando tali errori:

```
willy:~> echo $VAR
willy:~> set -u
willy:~> echo $VAR
bash: VAR: unbound variable
```

Questa opzione è utile anche per rilevare un'assegnazione errata al contenuto alle variabili: lo stesso errore si verificherà anche, ad esempio, quando si assegna una stringa di caratteri a una variabile che è stata dichiarata esplicitamente come contenente solo valori interi.

L'ultimo esempio, mostra l'opzione `noglob`, che impedisce l'espansione dei caratteri speciali:

```
willy:~/testdir> set -o noglob
willy:~/testdir> touch *
willy:~/testdir> ls -l *
-rw-rw-r-- 1 willy willy 0 Feb 27 13:37 *
```

3.7. Sommario

L'ambiente Bash può essere configurato globalmente e per singolo utente. Vengono utilizzati vari file di configurazione per mettere a punto il comportamento della shell.

Questi file contengono opzioni di shell, settaggi per variabili, definizioni di funzioni e vari altri elementi per creare un ambiente personalizzato.

Fatta eccezione per la quelli riservati della Bourne shell, Bash e dei parametri speciali, i nomi delle variabili possono essere scelti più o meno liberamente.

Poiché molti caratteri hanno significati doppi o addirittura tripli, a seconda dell'ambiente, Bash utilizza un sistema di virgolette per togliere un significato speciale da uno o più caratteri.

Bash utilizza vari metodi per espandere gli elementi sulla riga di comando per determinare quali comandi eseguire.

3.8. Esercizi

Per questo esercizio, si dovranno leggere le pagine man di **useradd**, perché useremo la directory `/etc/skel` per contenere i file di default per la configurazione della shell, che vengono copiati sulla home directory di ogni utente appena creato.

Per prima cosa faremo alcuni esercizi generali sull'impostazione e la visualizzazione delle variabili.

1. Creare 3 variabili, `VAR1`, `VAR2` e `VAR3`; inizializzarle per contenere i valori "tredici", "13" e "Buon Compleanno" rispettivamente.
2. Visualizzare i valori di tutte e tre le variabili.
3. Sono variabili locali o globali?
4. Rimuovere `VAR3`.
5. Si riescono a vedere le due variabili rimanenti in una nuova finestra di terminale?
6. Modificare `/etc/profile` in modo che tutti gli utenti vengano salutati all'accesso (provare questo).
7. Per l'account `root`, impostarne il prompt su qualcosa come "Pericolo!! root sta facendo qualcosa in \w", preferibilmente in un colore brillante come rosso o rosa o con colori invertiti.
8. Assicurarsi che gli utenti appena creati ricevano anche un bel prompt personalizzato che li informi su quale sistema e in quale directory stanno lavorando. Verificare le modifiche aggiungendo un nuovo utente e accedendo con le nuove credenziali.
9. Scrivere uno script in cui assegnano due valori interi a due variabili. Lo script dovrebbe calcolare la superficie di un rettangolo che ha queste proporzioni. Dovrebbe generare commenti e un output elegante.

Non dimenticare di eseguire **chmod** sugli script!

Capitolo 4. Espressioni regolari

In questo capitolo si parla di:

- Uso delle espressioni regolari
- Metacaratteri delle espressioni regolari
- Trovare pattern nei file o nell'output
- Serie di caratteri e classi in Bash

4.1. Espressioni regolari

4.1.1. Cosa sono le espressioni regolari?

Un'espressione regolare è un pattern [modello] che descrive un insieme di stringhe. Le espressioni regolari sono costruite in modo analogo alle espressioni aritmetiche utilizzando vari operatori per combinare espressioni più piccole.

Gli elementi costitutivi fondamentali sono le espressioni regolari che corrispondono a un singolo carattere. La maggior parte dei caratteri, comprese tutte le lettere e le cifre, sono espressioni regolari che corrispondono a se stessi. Qualsiasi metacarattere con un significato speciale può essere citato [quoted] precedendolo con una barra rovesciata.

4.1.2. Metacaratteri delle espressioni regolari

Un'espressione regolare può essere seguita da uno dei tanti operatori di ripetizione (metacaratteri):

Tabella 4-1. Operatori delle espressioni regolari

Operatore	Effetto
.	Corrisponde a qualsiasi singolo carattere.
?	L'elemento precedente è facoltativo ed è presente, al massimo, una volta.
*	L'elemento precedente è presente zero o più volte.
+	L'elemento precedente è presente una o più volte.
{N}	L'elemento precedente è presente esattamente N volte.
{N,}	L'elemento precedente è presente N o più volte.
{N,M}	L'elemento precedente è presente almeno N volte, ma non più di M volte.
-	rappresenta l'intervallo se non è il primo o l'ultimo in un elenco o il punto finale di un intervallo in un elenco.
^	Corrisponde alla stringa vuota all'inizio di una riga; rappresenta anche i caratteri non compresi nell'intervallo di un elenco.
\$	Corrisponde alla stringa vuota alla fine di una riga.
\b	Corrisponde alla stringa vuota al lato di una parola.
\B	Corrisponde alla stringa vuota purché non si trovi ai lati di una parola.
\<	Trova la stringa vuota all'inizio della parola.
\>	Trova la stringa vuota alla fine della parola.

È possibile concatenare due espressioni regolari; l'espressione risultante corrisponde a qualsiasi stringa formata concatenando due sotto-stringhe che corrispondono rispettivamente alle sottoespressioni concatenate.

Due espressioni regolari possono essere unite dall'operatore "infisso" "|"; l'espressione regolare risultante corrisponde a qualsiasi stringa che corrisponda a una delle due sottoespressioni.

La ripetizione ha la precedenza sulla concatenazione, che a sua volta ha la precedenza sulle alternative. Un'intera sottoespressione può essere racchiusa tra parentesi per sovrascrivere queste regole di precedenza.

4.1.3. Espressioni regolari di base ed estese

Nelle espressioni regolari di base i metacaratteri "?", "+", "{", "|", "(" e ")" perdono il loro significato speciale; usare in sostituzione le versioni con barra rovesciata "\?", "\+", "\{", "\|", "\(", and "\)".

Controllare nella documentazione del sistema se i comandi che utilizzano le espressioni regolari supportano quelle estese.

4.2. Esempi usando grep

4.2.1. Cos'è grep?

grep cerca nei file di input le righe che contengono una corrispondenza con un determinato elenco di pattern. Quando trova una corrispondenza in una riga, copia la riga nell'output standard (per impostazione di default) o in qualsiasi altro tipo di output richiesto tramite le opzioni.

Sebbene **grep** si aspetti di eseguire la corrispondenza sul testo, non ha limiti sulla lunghezza della riga di input oltre alla memoria disponibile e può corrispondere a caratteri arbitrari all'interno di una riga. Se il byte finale di un file di input non è un *newline*, **grep** ne fornisce tacitamente uno. Poiché *newline* è anche un separatore per l'elenco dei pattern, non c'è modo di far corrispondere i caratteri *newline* in un testo.

Qualche esempio:

```
cathy ~> grep root /etc/passwd
root:x:0:0:root:/root:/bin/bash
operator:x:11:0:operator:/root:/sbin/nologin

cathy ~> grep -n root /etc/passwd
1:root:x:0:0:root:/root:/bin/bash
12:operator:x:11:0:operator:/root:/sbin/nologin

cathy ~> grep -v bash /etc/passwd | grep -v nologin
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
news:x:9:13:news:/var/spool/news:
mailnull:x:47:47::/var/spool/mqueue:/dev/null
xfs:x:43:43:X Font Server:/etc/X11/fs:/bin/false
rpc:x:32:32:Portmapper RPC user:/bin/false
nscd:x:28:28:NSCD Daemon:/bin/false
named:x:25:25:Named:/var/named:/bin/false
squid:x:23:23::/var/spool/squid:/dev/null
ldap:x:55:55:LDAP User:/var/lib/ldap:/bin/false
apache:x:48:48:Apache:/var/www:/bin/false
```

```
cathy ~> grep -c false /etc/passwd
7

cathy ~> grep -i ps ~/.bash* | grep -v history
/home/cathy/.bashrc:PS1="\[\033[1;44m\]$USER is in \w\[\033[0m\] "
```

Col primo comando, l'utente *cathy* visualizza le righe di `/etc/passwd` contenenti la stringa *root*.

Quindi visualizza i numeri di riga che contengono tale stringa di ricerca.

Col terzo comando controlla quali utenti non stanno utilizzando **bash**, ma gli account con la shell **nologin** non vengono visualizzati.

Poi conta il numero di account che hanno `/bin/false` come shell.

L'ultimo comando mostra le righe di tutti i file nella home directory che iniziano con `~/.bash`, escludendo le corrispondenze contenenti la stringa *history*, in modo da escludere le corrispondenze da `~/.bash_history` che potrebbe contenere la stessa stringa, in maiuscolo o minuscolo. Si noti che la ricerca è per la *stringa* "ps" e non per il *comando* ps.

Ora vediamo cos'altro possiamo fare con **grep**, usando le espressioni regolari.

4.2.2. Grep e le espressioni regolari



Se non si è su Linux

Usiamo **grep** di GNU in questi esempi, che supporta le espressioni regolari estese. GNU **grep** è il default sui sistemi Linux. Se si sta lavorando su sistemi proprietari, controllare con l'opzione `-v` quale versione si sta utilizzando. GNU **grep** può essere scaricato da <http://gnu.org/directory/>.

4.2.2.1. Ancoraggi [anchor] a linee e a parole

Dall'esempio precedente, ora vogliamo visualizzare esclusivamente le righe che iniziano con la stringa "root":

```
cathy ~> grep ^root /etc/passwd
root:x:0:0:root:/root:/bin/bash
```

Se vogliamo vedere quali account non hanno alcuna shell assegnata, cerchiamo le righe che terminano con ":"

```
cathy ~> grep :$ /etc/passwd
news:x:9:13:news:/var/spool/news:
```

Per verificare che `PATH` sia esportato in `~/.bashrc`, prima si selezionano le righe "export" e poi si cercano le righe che iniziano con la stringa "PATH", in modo da non visualizzare `MANPATH` e altri possibili percorsi:

```
cathy ~> grep export ~/.bashrc | grep '^<PATH'
export
PATH="/bin:/usr/lib/mh:/lib:/usr/bin:/usr/local/bin:/usr/ucb:/usr/sbin:$PATH"
```


Allo stesso modo, `\>` corrisponde alla fine di una parola.

Volendo trovare una stringa che sia una parola separata (racchiusa tra spazi), è meglio usare `-w`, come in questo esempio in cui vengono visualizzate le informazioni per la partizione di root:

```
cathy ~> grep -w / /etc/fstab
LABEL=/                                /                                ext3      defaults    1 1
```

Se questa opzione non viene utilizzata, verranno visualizzate tutte le righe della tabella del file system.

4.2.2.2. Classi di caratteri

Un'espressione tra parentesi quadre è un elenco di caratteri racchiusi tra "[" e "]". Corrisponde a qualsiasi singolo carattere in quell'elenco; se il primo carattere dell'elenco è il caret, "^", allora cerca qualsiasi carattere NON presente nell'elenco. Ad esempio, l'espressione regolare "[0123456789]" corrisponde a qualsiasi singola cifra.

All'interno di un'espressione tra parentesi, una *range expression* [espressione di intervallo] è composta da due caratteri separati da un trattino. Corrisponde a qualsiasi singolo carattere ordinato tra i due caratteri, inclusi, utilizzando la sequenza di 'collating' e il set di caratteri della lingua locale. Ad esempio, nelle impostazioni C di default della lingua, "[a-d]" è equivalente a "[abcd]". Molte lingue ordinano i caratteri alfabeticamente e con tali impostazioni "[a-d]" non è sempre equivalente a "[abcd]"; potrebbe essere equivalente a "[aBbCcDd]", per esempio. Per ottenere l'interpretazione tradizionale delle espressioni tra parentesi, è possibile utilizzare la lingua del C impostando la variabile d'ambiente `LC_ALL` al valore "C".

Infine, alcune nomi di classi di caratteri sono predefinite all'interno delle espressioni tra parentesi. Vedere le pagine man o info di **grep** per maggiori informazioni su tali espressioni predefinite.

```
cathy ~> grep [yf] /etc/group
sys:x:3:root,bin,adm
tty:x:5:
mail:x:12:mail,postfix
ftp:x:50:
nobody:x:99:
floppy:x:19:
xfs:x:43:
nfsnobody:x:65534:
postfix:x:89:
```

Nell'esempio vengono visualizzate tutte le righe contenenti o il carattere "y" o "f".

4.2.2.3. Caratteri jolly [Wildcard]

Si usa "." per una corrispondenza con un singolo carattere. Volendo ottenere un elenco di tutte le parole del dizionario inglese di cinque caratteri che iniziano con "c" e terminano con "h" (utile per risolvere i cruciverba):

```
cathy ~> grep '\<c...h\>' /usr/share/dict/words
catch
clash
```

```
cloth
coach
couch
cough
crash
crush
```

Per visualizzare le righe contenenti il carattere punto '.', usare l'opzione `-F` di **grep**.

Per indicare più caratteri, usare l'asterisco. Questo esempio seleziona tutte le parole che iniziano con "c" e terminano con "h" dal dizionario del sistema:

```
cathy ~> grep '\<c.*h\>' /usr/share/dict/words
caliph
cash
catch
cheesecloth
cheetah
--output omitted--
```

Per trovare il carattere asterisco in un file o in un output, usare le virgolette singole. Cathy nell'esempio seguente prova prima a trovare il carattere asterisco in `/etc/profile` senza usare le virgolette, che non restituisce alcuna riga. Usando le virgolette, viene generato l'output:

```
cathy ~> grep * /etc/profile

cathy ~> grep '*' /etc/profile
for i in /etc/profile.d/*.sh ; do
```

4.3. Pattern matching con le funzionalità di Bash

4.3.1. Sequenze di caratteri

A parte **grep** e le espressioni regolari, c'è una buona dose di pattern matching da fare direttamente nella shell, senza dover usare un programma esterno.

Come già si saprà, l'asterisco (*) e il punto interrogativo (?) corrispondono rispettivamente a qualsiasi stringa o a qualsiasi carattere singolo. Porre tra apici questi caratteri speciali per abbinarli letteralmente:

```
cathy ~> touch "*"

cathy ~> ls "*"
*
```

Ma si possono anche usare le parentesi quadre per far corrispondere qualsiasi carattere o range di caratteri, se le coppie di caratteri sono separate da un trattino. Un esempio:

```
cathy ~> ls -ld [a-cx-z]*
drwxr-xr-x  2 cathy  cathy      4096 Jul 20  2002 app-defaults/
drwxrwxr-x  4 cathy  cathy      4096 May 25  2002 arabic/
drwxrwxr-x  2 cathy  cathy      4096 Mar  4  18:30 bin/
drwxr-xr-x  7 cathy  cathy      4096 Sep  2  2001 crossover/
drwxrwxr-x  3 cathy  cathy      4096 Mar 22  2002 xml/
```

Elenca tutti i file nella directory home di *cathy*, iniziando con "a", "b", "c", "x", "y" o "z".

Se il primo carattere tra parentesi graffe è "!" o "^", verrà trovata una corrispondenza con qualsiasi carattere non presente nella lista. Per trovare il trattino ("-"), includerlo come primo o ultimo carattere nel set. L'ordinamento dipende dalla lingua corrente e dal valore della variabile `LC_COLLATE`, se impostata. Tieni presente che altre versioni della lingua potrebbero interpretare "[a-cx-z]" come "[aBbCcXxYyZz]" se l'ordinamento viene eseguito secondo l'ordine alfabetico. Per essere sicuri di avere l'interpretazione tradizionale dei range, forzare questo comportamento impostando `LC_COLLATE` o `LC_ALL` con "C".

4.3.2. Classi di caratteri

Le classi di caratteri si possono specificare all'interno delle parentesi quadre, utilizzando la sintassi `[:CLASS:]`, dove `CLASS` è definita nello standard POSIX e ha uno dei valori

"alnum", "alpha", "ascii", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper", "word" oppure "xdigit".

Qualche esempio:

```
cathy ~> ls -ld [[:digit:]]*
drwxrwxr-x    2 cathy  cathy          4096 Apr 20 13:45 2/

cathy ~> ls -ld [[:upper:]]*
drwxrwxr--    3 cathy  cathy          4096 Sep 30  2001 Nautilus/
drwxrwxr-x    4 cathy  cathy          4096 Jul 11  2002 OpenOffice.org1.0/
-rw-rw-r--    1 cathy  cathy          997376 Apr 18 15:39 Schedule.sdc
```

Quando l'opzione shell `extglob` è abilitata (usando il comando **shopt**), vengono riconosciuti diversi operatori di pattern matching estesi. Maggiori informazioni nelle pagine informative di Bash, sezione Basic shell features->Shell Expansions->Filename Expansion->Pattern Matching.

4.4. Sommario

Le espressioni regolari sono strumenti potenti per selezionare righe particolari da file o output. Molti comandi UNIX usano espressioni regolari: **vim**, **perl**, il database PostgreSQL e così via. Sono disponibili in qualsiasi lingua o applicazione utilizzando librerie esterne e hanno persino trovato la loro strada in sistemi non UNIX. Ad esempio, le espressioni regolari vengono utilizzate nel foglio di calcolo Excel fornito con la suite Microsoft Windows Office. In questo capitolo abbiamo un primo approccio col comando **grep**, indispensabile in qualsiasi ambiente UNIX.



Il comando **grep** può fare molto di più dei pochi compiti che discussi qui; l'abbiamo usato solo come esempio per le espressioni regolari. La versione GNU di **grep** viene fornita con molta documentazione, che consigliamo vivamente di leggerla!

Bash ha funzionalità integrate per i matching pattern e può riconoscere classi e intervalli di caratteri.

4.5. Esercizi

Questi esercizi aiuteranno a padroneggiare le espressioni regolari.

1. Visualizzare un elenco di tutti gli utenti sul sistema che accedono con la shell Bash per default.
2. Dalla directory `/etc/group`, mostrare tutte le righe che iniziano con la stringa "daemon".
3. Stampare tutte le righe dello stesso file che non contengono la stringa.
4. Visualizzare le informazioni su localhost dal file `/etc/hosts`, mostrare il numero o i numeri di riga corrispondenti alla stringa di ricerca e contare il numero di occorrenze della stringa.
5. Visualizza un elenco di sottodirectory `/usr/share/doc` contenenti informazioni sulle shell.
6. Quanti file `README` contengono queste sottodirectory? Non contare nulla sotto forma di "README.a_string".
7. Creare un elenco dei file nella directory home che sono stati modificati meno di 10 ore fa, utilizzando **grep**, ma tralasciare le directory.
8. Mettere questi comandi in uno script che genererà un output comprensibile.
9. Si riesce a trovare un'alternativa a **wc -l**, usando **grep**?
10. Usando la tabella del file system (`/etc/fstab` per esempio), elencare i dispositivi del disco locale.
11. Creare uno script che controlli se un utente esiste in `/etc/passwd`. Per ora, si può specificare il nome utente nello script, non si deve lavorare con argomenti e condizionali in questa fase.
12. Visualizzare i file di configurazione in `/etc` che contengono numeri nei loro nomi.

Capitolo 5. Sed, l'editor dei i flussi di GNU [stream editor]

Alla fine di questo capitolo si conosceranno i seguenti argomenti:

- Che cos'è **sed**?
- Uso interattivo di **sed**
- Espressioni regolari e modifica del lo stream [flusso]
- Utilizzo dei comandi **sed** negli script



Questa è un'introduzione

Queste spiegazioni sono tutt'altro che complete e certamente non intendono essere utilizzate come manuale utente definitivo per **sed**. Questo capitolo è stato incluso solo per mostrare alcuni degli argomenti più interessanti nei prossimi capitoli e perché ogni utente esperto dovrebbe avere una conoscenza di base delle cose che possono essere fatte con questo editor.

Per informazioni dettagliate, fare riferimento alle info su **sed** e alle pagine man.

5.1. Introduzione

5.1.1. Che cos'è sed?

Uno Stream EDitor viene utilizzato per eseguire trasformazioni elementari sul testo letto da un file o da una pipe. Il risultato viene inviato allo standard output. La sintassi per il comando **sed** non ha una specifica del file di output, ma i risultati possono essere salvati in un file utilizzando il reindirizzamento dell'output. L'editor non modifica l'input originale.

Ciò che distingue **sed** da altri editor, come **vi** e **ed**, è la sua capacità di filtrare il testo alimentato da una pipeline. Non è necessario interagire con l'editor mentre è in esecuzione; questo è il motivo per cui **sed** è talvolta chiamato *editor batch*. Questa funzione consente l'uso di comandi di modifica negli script, facilitando notevolmente le attività di modifiche ripetitive. Quando si deve sostituire il testo in un gran numero di file, **sed** è di grande aiuto.

5.1.2. Comandi di sed

il programma **sed** può eseguire sostituzioni e cancellazioni di pattern di testo utilizzando espressioni regolari, come quelle usate con il comando **grep**; vedere la [Sezione 4.2](#).

I comandi di modifica sono simili a quelli usati nell'editor **vi**:

Tabella 5-1. Comandi di modifica di sed

Comando	Risultato
a\	Aggiungi il testo sotto la riga corrente.
c\	Cambia il testo nella riga corrente con il nuovo testo.
d	Elimina il testo.
i\	Inserisci il testo sopra la riga corrente.
p	Stampa il testo.
r	Leggi un file.
s	Cerca e sostituisci il testo.
w	Scrivi in un file.

Oltre ai comandi di modifica, puoi dare a **sed** delle opzioni. Un estratto è nella tabella seguente:

Tabella 5-2. Opzioni di sed

Opzione	Effetto
-e SCRIPT	Aggiungere i comandi in SCRIPT al set di comandi da eseguire durante l'elaborazione dell'input.
-f	Aggiungere i comandi contenuti nel file SCRIPT-FILE al set di comandi da eseguire durante l'elaborazione dell'input.
-n	Modalità silenziosa.
-V	Stampa la versione ed esce.

Le pagine info di **sed** contengono maggiori informazioni; elenchiamo qui solo i comandi e le opzioni utilizzati più di frequente.

5.2. Editing interattivo

5.2.1. Linee di stampa contenenti un pattern

Questo si può fare con **grep**, ovviamente, ma non si può fare un "trova e sostituisci" usando quel comando. Questo è solo per iniziare.

Questo è il file di testo di esempio:

```
sandy ~> cat -n example
 1 This is the first line of an example text.
 2 It is a text with errors.
 3 Lots of errors.
 4 So much errors, all these errors are making me sick.
 5 This is a line not containing any errors.
 6 This is the last line.

sandy ~>
```

Vogliamo che **sed** trovi tutte le righe che contengono il nostro pattern di ricerca, in questo caso "errors". Usiamo il **p** per ottenere il risultato:

```
sandy ~> sed '/errors/p' example
This is the first line of an example text.
It is a text with errors.
It is a text with errors.
Lots of errors.
Lots of errors.
So much errors, all these errors are making me sick.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

Come si noterà, **sed** stampa l'intero file, ma le righe contenenti la stringa ricercata vengono stampate due volte. Questo non è quello che vogliamo. Per stampare solo quelle righe che corrispondono al nostro pattern, usare l'opzione **-n**:

```
sandy ~> sed -n '/errors/p' example
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.

sandy ~>
```

5.2.2. Eliminazione di righe di input contenenti un pattern

Usiamo lo stesso file di testo di esempio. Ora vogliamo solo vedere le righe *non* contenenti la stringa di ricerca:

```
sandy ~> sed '/errors/d' example
```

```
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

Il comando **d** determina l'esclusione delle righe dalla visualizzazione.

Le linee trovate che iniziano con un determinato pattern e terminano con un secondo pattern vengono mostrate in questo modo:

```
sandy ~> sed -n '/^This.*errors.$/p' example
This is a line not containing any errors.

sandy ~>
```

Si noti che l'ultimo punto '.' deve essere 'escaped' per effettivamente corrispondere. Nel nostro esempio l'espressione corrisponde a qualsiasi carattere, incluso l'ultimo punto.

5.2.3. Range di linee

Questa volta vogliamo eliminare le righe contenenti gli errori. Nell'esempio queste sono le righe dalla 2 alla 4. Specificare tale intervallo, oltre al comando **d**:

```
sandy ~> sed '2,4d' example
This is the first line of an example text.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

Per stampare il file partendo da una determinata riga fino alla fine del file, utilizzare un comando simile a questo:

```
sandy ~> sed '3,$d' example
This is the first line of an example text.
It is a text with errors.

sandy ~>
```

Questo stampa solo le prime due righe del file di esempio.

Il comando seguente stampa la prima riga che contiene il pattern "a text", fino alla riga successiva che contiene il pattern "a line":

```
sandy ~> sed -n '/a text/,/This/p' example
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.

sandy ~>
```

5.2.4. Ricerca e sostituzione con sed

Nel file di esempio, cercheremo e sostituiremo gli errori invece di (de)selezionare solo le righe che contengono la stringa di ricerca.

```
sandy ~> sed 's/erors/errors/' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

Come si può vedere, questo non è esattamente l'effetto desiderato: nella riga 4, solo la prima occorrenza della stringa di ricerca è stata sostituita e resta ancora un "errore". Usare il comando **g** per indicare a **sed** che dovrebbe esaminare l'intera riga invece di fermarsi alla prima occorrenza della stringa:

```
sandy ~> sed 's/erors/errors/g' example
This is the first line of an example text.
It is a text with errors.
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the last line.

sandy ~>
```

Per inserire una stringa all'inizio di ogni riga di un file, ad esempio per mettere tra virgolette:

```
sandy ~> sed 's/^/> /' example
> This is the first line of an example text.
> It is a text with errors.
> Lots of errors.
> So much errors, all these errors are making me sick.
> This is a line not containing any errors.
> This is the last line.

sandy ~>
```

Inserire una stringa alla fine di ogni riga:

```
sandy ~> sed 's/$/EOL/' example
This is the first line of an example text.EOL
It is a text with errors.EOL
Lots of errors.EOL
So much errors, all these errors are making me sick.EOL
This is a line not containing any errors.EOL
This is the last line.EOL

sandy ~>
```

Più comandi di ricerca e sostituzione vengono separati da singole opzioni **-e**:

```
sandy ~> sed -e 's/erors/errors/g' -e 's/last/final/g' example
This is the first line of an example text.
It is a text with errors.
```



```
Lots of errors.
So much errors, all these errors are making me sick.
This is a line not containing any errors.
This is the final line.

sandy ~>
```

Si tenga presente che per default **sed** stampa i risultati nell'output standard, molto probabilmente nella finestra del terminale. Volendo salvare l'output in un file, reindirizzarlo:

```
sed option 'some/expression' file_to_process > sed_output_in_a_file
```



Altri esempi

Un sacco di esempi di **sed** si possono trovare negli script di startup della macchina, che di solito sono in `/etc/init.d` o `/etc/rc.d/` `init.d`. Passare alla directory contenente gli initscript del sistema ed inserire il seguente comando:

```
grep sed *
```

5.3. Editing non-interattivo

5.3.1. Lettura di comandi sed da un file

Più comandi **sed** possono essere inseriti in un file ed eseguiti utilizzando l'opzione `-f`. Quando si crea un file di questo tipo, assicurarsi che:

- Non esistono spazi bianchi finali alla fine delle righe.
- Non vengono utilizzate virgolette.
- Quando si inserisce il testo da aggiungere o sostituire, tutto tranne l'ultima riga termina con una barra rovesciata.

5.3.2. Scrivere i file di output

La scrittura dell'output viene eseguita utilizzando l'operatore di reindirizzamento dell'output `>`. Questo è uno script di esempio utilizzato per creare file HTML molto semplici da file di testo.

```
sandy ~> cat script.sed
1i\
<html>\
<head><title>sed generated html</title></head>\
<body bgcolor="#ffffff">\
<pre>
$a\
</pre>\
</body>\
</html>

sandy ~> cat txt2html.sh
#!/bin/bash

# This is a simple script that you can use for converting text into HTML.
# First we take out all newline characters, so that the appending only happens
# once, then we replace the newlines.
```

```

echo "converting $1..."

SCRIPT="/home/sandy/scripts/script.sed"
NAME="$1"
TEMPFILE="/var/tmp/sed.$PID.tmp"
sed "s/\n/^M/" $1 | sed -f $SCRIPT | sed "s/^M/\n/" > $TEMPFILE
mv $TEMPFILE $NAME

echo "done."

sandy ~>

```

\$1 contiene il primo argomento di un dato comando, in questo caso il nome del file da convertire:

```

sandy ~> cat test
line1
line2
line3

```

Maggiori informazioni sui parametri posizionali nel [Capitolo 7](#).

```

sandy ~> txt2html.sh test
converting test...
done.

sandy ~> cat test
<html>
<head><title>sed generated html</title></head>
<body bgcolor="#ffffff">
<pre>
line1
line2
line3
</pre>
</body>
</html>

sandy ~>

```

Non è proprio così che si fa; questo esempio dimostra solo le capacità di **sed**. Vedere la [Sezione 6.3](#) per una soluzione più decente a questo problema, utilizzando i costrutti **awk** *BEGIN* e *END*.



Easy sed

Gli editor avanzati, che supportano l'evidenziazione della sintassi, possono riconoscere la sintassi **sed**. Questo può essere di grande aiuto se si tende a dimenticare le barre rovesciate e simili.

5.4. Sommario

Lo stream editor **sed** è un potente strumento a riga di comando, in grado di gestire flussi di dati: può prendere linee di input da una pipe. Questo lo rende adatto per un uso non interattivo. L'editor **sed** utilizza comandi in stile **vi** e accetta espressioni regolari.

Lo strumento **sed** può leggere i comandi dalla riga di comando o da uno script. Viene spesso utilizzato per eseguire azioni di ricerca e sostituzione su righe contenenti un pattern.

5.5. Esercizi

Questi esercizi hanno lo scopo di dimostrare ulteriormente cosa può fare **sed**.

1. Stampare un elenco di file nella directory `script`, che terminano con `".sh"`. Si tenga presente che si potrebbe dover eseguire l'unalias `ls`. Mettere il risultato in un file temporaneo.
2. Creare un elenco di file in `/usr/bin` che hanno la lettera "a" come secondo carattere. Mettere il risultato in un file temporaneo.
3. Elimina le prime 3 righe di ogni file temporaneo.
4. Stampa sullo standard output solo le righe che contengono il pattern "an".
5. Creare un file contenente i comandi **sed** per eseguire le due attività precedenti. Aggiungere un comando extra a questo file che aggiunge una stringa come `*** Questo potrebbe avere qualcosa a che fare con le pagine man e man ***` nella riga che precede ogni occorrenza della stringa "uomo". Controllare i risultati.
6. Per l'input viene utilizzato un lungo elenco della directory radice, `/`. Creare un file contenente i comandi **sed** che controllano i link simbolici e i file semplici. Se un file è un link simbolico, precederlo con una riga come `--Questo è un collegamento simbolico--`. Se il file è un semplice file, aggiungere una stringa sulla stessa riga, aggiungendo un commento come `<--- questo è un file semplice`.
7. Creare uno script che mostri le righe contenenti spazi bianchi finali da un file. Questo script dovrebbe usare uno script **sed** e mostrare informazioni sensibili all'utente.

Capitolo 6. Il linguaggio di programmazione GNU awk

In questo capitolo parleremo di:

- Che cos'è gawk?
- Utilizzo dei comandi gawk sulla riga di comando
- Come formattare il testo con gawk
- Come gawk usa le espressioni regolari
- Gawk negli script
- Gawk e le variabili



Per rendere la cosa più divertente

Come con **sed**, sono stati scritti interi libri sulle varie versioni di **awk**. Questa introduzione è lunga dall'essere completa ed è intesa solo per comprendere gli esempi nei capitoli seguenti. Per ulteriori informazioni, è meglio iniziare con la documentazione fornita con GNU awk: "GAWK: Effective AWK Programming: A User's Guide for GNU Awk".

6.1. Iniziare con gawk

6.1.1. Cos'è il gawk?

Gawk è la versione GNU del noto programma UNIX **awk**, un altro popolare editor di stream. Poiché il programma **awk** è spesso solo un link a **gawk**, lo chiameremo **awk**.

Il compito base di **awk** è quello di cercare nei file righe o altre unità di testo contenenti uno o più pattern. Quando una riga corrisponde a uno dei pattern, vengono eseguite azioni speciali su tale riga.

I programmi in **awk** sono diversi dai programmi nella maggior parte degli altri linguaggi, perché i programmi **awk** sono "data-driven" [guidati dai dati]: si descrivono i dati da elaborare e cosa farne quando vengono trovati. La maggior parte degli altri linguaggi sono "procedurali". Si descrive, in grande dettaglio, ogni passo che il programma deve compiere. Quando si lavora con linguaggi procedurali, di solito è molto più difficile descrivere chiaramente i dati che il programma elaborerà. Per questo motivo, i programmi **awk** sono spesso piacevolmente facili da leggere e scrivere.



Cosa significa veramente?

Negli anni '70, si sono riuniti tre programmatori per creare questo linguaggio. I loro nomi erano Aho, Kernighan e Weinberger. Hanno preso il primo carattere di ciascuno dei loro nomi e li hanno messi insieme. Quindi il nome del linguaggio avrebbe potuto essere "wak".

6.1.2. Comandi Gawk

Quando si esegue **awk**, si indica a**awk** un *programma* che dice a **awk** cosa fare. Il programma consiste in una serie di *regole*. (Può anche contenere definizioni di funzioni, cicli, condizioni e altri costrutti della programmazione, funzionalità avanzate che per ora ignoreremo). Ogni regola specifica un pattern da cercare e un'azione da eseguire dopo aver trovato il pattern.

Ci sono diversi modi per eseguire **awk**. Se il programma è breve, è più semplice eseguirlo dalla riga di comando:

```
awk PROGRAM inputfile(s)
```

Se devono essere apportate più modifiche, forse regolarmente e su più file, è più semplice inserire i comandi **awk** in uno script. Questo si legge così:

```
awk -f PROGRAM-FILE inputfile(s)
```

6.2. Il programma print

6.2.1. Stampa dei campi selezionati

Il comando **print** in **awk** emette i dati selezionati dal file di input.

Quando **awk** legge una riga di un file, divide la riga in campi in base al *separatore di campi di input* specificato, `FS`, che è una variabile **awk** (vedere la [Sezione 6.3.2](#)). Questa variabile è predefinita per essere uno o più spazi o tabulazioni.

Le variabili \$1, \$2, \$3, ..., \$N contengono i valori del primo, secondo, terzo fino all'ultimo campo di una riga di input. La variabile \$0 (zero) contiene il valore dell'intera riga. Questo è illustrato nell'immagine sottostante, dove vediamo sei colonne nell'output del comando **df**:

Figura 6-1. I campi in awk

```

kelly@octarine:~
File Edit View Terminal Go Help
[kelly@octarine kelly]$ df -h
Filesystem      Size  Used Avail Use% Mounted on
/dev/hda7       1.3G  274M 1016M  22% /
/dev/hda1       121M   9.4M  105M   9% /boot
/dev/hda2       13G   8.7G   3.7G  70% /home
/dev/hda3       13G   5.6G   6.8G  45% /opt
none            243M     0   243M   0% /dev/shm
/dev/hda6       3.9G   3.3G   480M  88% /usr
/dev/hda5       5.2G   4.6G   431M  92% /var
  
```

Below the table, red brackets and labels indicate the field numbers:

- \$1: Filesystem
- \$2: Size
- \$3: Used
- \$4: Avail
- \$5: Use%
- \$6: Mounted on

Nell'output di **ls -l** ci sono 9 colonne. L'istruzione **print** utilizza questi campi come segue:

```

kelly@octarine ~/test> ls -l | awk '{ print $5 $9 }'
160orig
121script.sed
120temp_file
126test
120twolines
441txt2html.sh

kelly@octarine ~/test>
  
```

Questo comando ha stampato la quinta colonna di un lungo elenco di file, che contiene la dimensione del file, e l'ultima colonna, il nome del file. Questo output non è molto leggibile a meno che non si utilizzi il modo ufficiale di fare riferimento alle colonne, ovvero separare quelle che si desiderano stampare con una virgola. In tal caso, il carattere separatore di output di default, di solito uno spazio, verrà inserito tra ciascun campo di output.



Configurazione locale

Si noti che la configurazione dell'output del comando **ls -l** potrebbe essere diversa su altri sistemi. La visualizzazione dell'ora e della data dipende dalle impostazioni internazionali.

6.2.2. Campi di formattazione

Senza formattazione, utilizzando solo il separatore, l'output appare piuttosto scadente. Inserendo un paio di tab e una stringa per indicare di quale output si tratta lo farà sembrare molto migliore:

```
kelly@octarine ~/test> ls -ldh * | grep -v total | \
awk '{ print "Size is " $5 " bytes for " $9 }'
Size is 160 bytes for orig
Size is 121 bytes for script.sed
Size is 120 bytes for temp_file
Size is 126 bytes for test
Size is 120 bytes for twolines
Size is 441 bytes for txt2html.sh

kelly@octarine ~/test>
```

Notare l'uso della backslash, che fa continuare l'input lungo sulla riga successiva senza che la shell lo interpreti come un comando separato. Mentre l'input dalla riga di comando può essere di lunghezza virtualmente illimitata, il monitor non lo è e la carta stampata certamente non lo è. L'uso della backslash consente anche di copiare e incollare le righe sopra in una finestra di terminale.

L'opzione `-h` per `ls` viene utilizzata per fornire formati di dimensioni leggibili dall'uomo per file più grandi. L'output di un lungo elenco che mostra la quantità totale di blocchi nella directory viene dato quando l'argomento è una directory. Questa riga è inutile per noi, quindi aggiungiamo un asterisco. Aggiungiamo anche l'opzione `-d` per lo stesso motivo, nel caso in cui l'asterisco si espanda in una directory.

La backslash in questo esempio indica la continuazione di una riga. Vedere la [Sezione 3.3.2](#).

Si possono eliminare un numero qualsiasi di colonne e persino invertirne l'ordine. Nell'esempio seguente questo è utilizzato per mostrare le partizioni più critiche:

```
kelly@octarine ~> df -h | sort -rnk 5 | head -3 | \
awk '{ print "Partition " $6 "\t: " $5 " full!" }'
Partition /var : 86% full!
Partition /usr : 85% full!
Partition /home : 70% full!

kelly@octarine ~>
```

La tabella seguente fornisce una panoramica dei caratteri di formattazione speciali:

Tabella 6-1. Caratteri di formattazione per gawk

Sequenza	Significato
<code>\a</code>	Carattere Bell (suono)
<code>\n</code>	Carattere newline (andata a capo)
<code>\t</code>	Tabulatore

Virgolette, simboli del dollaro e altri metacaratteri dovrebbero essere evitati con un backslash.

6.2.3. Il comando print e le espressioni regolari

Un'espressione regolare può essere utilizzata come pattern racchiudendola tra slash ('/.../').

L'espressione regolare viene poi controllata rispetto all'intero testo di ciascun record. La sintassi è la seguente:

awk 'EXPRESSION { PROGRAM }' file(s)

L'esempio seguente mostra solo le informazioni sul disco locale, i file system di rete non vengono mostrati:

```
kelly is in ~> df -h | awk '/dev\/hd/ { print $6 "\t: " $5 }'
/           : 46%
/boot       : 10%
/opt        : 84%
/usr        : 97%
/var        : 73%
/.vll       : 8%

kelly is in ~>
```

Le barre [slash] devono essere 'escaped', perché hanno un significato speciale per **awk**.

Di seguito un altro esempio in cui cerchiamo nella directory `/etc` i file che terminano con ".conf" e iniziano con "a" o "x", utilizzando espressioni regolari estese:

```
kelly is in /etc> ls -l | awk '/<(a|x).*\.conf$/ { print $9 }'
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf

kelly is in /etc>
```

Questo esempio illustra il significato speciale del punto nelle espressioni regolari: il primo indica che vogliamo cercare qualsiasi carattere dopo la prima stringa di ricerca, il secondo è un carattere 'escaped' perché fa parte di una stringa da trovare (la fine del file nome).

6.2.4. Pattern speciali

Per far precedere l'output da commenti, utilizzare l'istruzione **BEGIN**:

```
kelly is in /etc> ls -l | \
awk 'BEGIN { print "Files found:\n" } /<[a|x).*\.conf$/ { print $9 }'
Files found:
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf

kelly is in /etc>
```

L'istruzione **END** può essere aggiunta per inserire il testo dopo che l'intero input è stato elaborato:

```
kelly is in /etc> ls -l | \
```

```
awk '/\<[a|x].*\..conf$/ { print $9 } END { print \
"Can I do anything else for you, mistress?" }'
amd.conf
antivir.conf
xcdroast.conf
xinetd.conf
Can I do anything else for you, mistress?

kelly is in /etc>
```

6.2.5. Script Gawk

Poiché i comandi tendono ad allungarsi un po', si potrebbe volerli inserire in uno script, in modo che siano riutilizzabili. Uno script **awk** contiene istruzioni **awk** che definiscono pattern e azioni.

A titolo illustrativo, creeremo un report che mostra le nostre partizioni più piene. Vedere la [Sezione 6.2.2.](#)

```
kelly is in ~> cat diskrep.awk
BEGIN { print "*** WARNING WARNING WARNING ***" }
/\<[8|9][0-9]%/ { print "Partition " $6 "\t: " $5 " full!" }
END { print "*** Give money for new disks URGENTLY! ***" }

kelly is in ~> df -h | awk -f diskrep.awk
*** WARNING WARNING WARNING ***
Partition /usr : 97% full!
*** Give money for new disks URGENTLY! ***

kelly is in ~>
```

awk prima stampa un messaggio iniziale, poi formatta tutte le righe che contengono un otto o un nove all'inizio di una parola, seguito da un altro numero e un segno di percentuale. Viene poi aggiunto un messaggio finale.



Evidenziazione della sintassi

Awk è un linguaggio di programmazione. La sua sintassi è riconosciuta dalla maggior parte degli editor che possono eseguire l'evidenziazione della sintassi per altri linguaggi, come C, Bash, HTML, ecc.

6.3. Variabili di Gawk

Poiché **awk** elabora il file di input, utilizza diverse variabili. Alcune sono modificabili, altre sono di sola lettura.

6.3.1. Il separatore del campo di input

Il *separatore di campo*, che può essere un singolo carattere o un'espressione regolare, controlla il modo in cui **awk** suddivide un record di input in campi. Il record di input viene scansionato per le sequenze di caratteri che corrispondono alla definizione del separatore; i campi stessi sono il testo tra le corrispondenze.

Il separatore di campo è rappresentato dalla variabile interna `FS`. Notare che è qualcosa di diverso dalla variabile `IFS` usata dalle shell compatibili con POSIX.

Il valore della variabile separatore di campo può essere modificato in **awk** con l'operatore di assegnazione `=`. Spesso il momento giusto per farlo è all'inizio dell'esecuzione prima che qualsiasi input sia stato elaborato, in modo che il primissimo record venga letto con il separatore appropriato. Per farlo, usare il pattern speciale **BEGIN**.

Nell'esempio seguente, creiamo un comando che visualizza tutti gli utenti sul sistema con una descrizione:

```
kelly is in ~> awk 'BEGIN { FS=":" } { print $1 "\t" $5 }' /etc/passwd
--output omitted--
kelly    Kelly Smith
franky   Franky B.
eddy     Eddy White
willy    William Black
cathy    Catherine the Great
sandy    Sandy Li Wong

kelly is in ~>
```

In uno script **awk**, sarebbe simile a questo:

```
kelly is in ~> cat printnames.awk
BEGIN { FS=":" }
{ print $1 "\t" $5 }

kelly is in ~> awk -f printnames.awk /etc/passwd
--output omitted--
```

Scegliere con attenzione i separatori dei campi di input per evitare problemi. Un esempio per illustrare questo: supponiamo che si riceva un input sotto forma di righe che assomigliano a questa:

"Sandy L. Wong, 64 Zoo St., Antwerp, 2000X"

Si scrive una riga di comando o uno script, che stampa il nome della persona in quel record:

```
awk 'BEGIN { FS="," } { print $1, $2, $3 }' inputfile
```

Ma una persona potrebbe avere un dottorato di ricerca [PhD], e potrebbe essere scritto così:

"Sandy L. Wong, PhD, 64 Zoo St., Antwerp, 2000X"

L'**awk** darà l'output sbagliato per questa riga. Se necessario, usare un **awk** o **sed** extra per uniformare l'input dei dati.

Il separatore di campi di input di default è uno o più spazi bianchi o tabulazioni.

6.3.2. I separatori di output

6.3.2.1. Il separatore di campo di output

I campi sono normalmente separati da spazi nell'output. Questo diventa evidente quando si usa la sintassi corretta per il comando **print**, dove gli argomenti sono separati da virgole:

```
kelly@octarine ~/test> cat test
record1      data1
record2      data2

kelly@octarine ~/test> awk '{ print $1 $2}' test
record1data1
record2data2

kelly@octarine ~/test> awk '{ print $1, $2}' test
record1 data1
record2 data2

kelly@octarine ~/test>
```

Se non si inseriscono le virgole, **print** tratterà gli elementi da visualizzare come un unico argomento, omettendo così l'uso del *separatore di output* predefinito, `OFS`.

Qualsiasi stringa di caratteri può essere utilizzata come separatore del campo di output impostando questa variabile interna.

6.3.2.2. Il separatore dei record di output

L'output di un'intera istruzione **print** è detto *record di output*. Ogni comando **print** genera un record di output, quindi emette una stringa chiamata *separatore di record di output* [output record separator], `ORS`. Il valore di default per questa variabile è `"\n"`, un carattere 'andata a capo'. Pertanto, ogni istruzione **print** genera una riga diversa.

Per modificare il modo in cui i campi di output e i record sono separati, assegnare nuovi valori a `OFS` e a `ORS`:

```
kelly@octarine ~/test> awk 'BEGIN { OFS=";" ; ORS="\n-->\n" } \
{ print $1,$2}' test
record1;data1
-->
record2;data2
-->

kelly@octarine ~/test>
```

Se il valore di `ORS` non contiene newline, l'output del programma viene eseguito insieme su un'unica riga.

6.3.3. Il numero dei record

La variabile interna `NR` contiene il numero di record che vengono elaborati. Viene incrementato dopo aver letto una nuova riga di input. La si può usare alla fine per contare il numero totale di record, o in ciascun record in uscita:

```
kelly@octarine ~/test> cat processed.awk
BEGIN { OFS="-" ; ORS="\n--> done\n" }
```

```
{ print "Record number " NR ": \t" $1,$2 }
END { print "Number of records processed: " NR }
```

```
kelly@octarine ~/test> awk -f processed.awk test
Record number 1:      record1-data1
--> done
Record number 2:      record2-data2
--> done
Number of records processed: 2
--> done

kelly@octarine ~/test>
```

6.3.4. Variabili definite dall'utente

Oltre alle variabili interne, se ne possono definire altre. Quando **awk** incontra un riferimento a una variabile che non esiste (che non è predefinita), la variabile viene creata e inizializzata con una stringa nulla. Per tutti i riferimenti successivi, il valore della variabile è l'ultimo valore assegnato. Le variabili possono essere una stringa o un valore numerico. Il contenuto dei campi di input può anche essere assegnato alle variabili.

I valori possono essere assegnati direttamente con l'operatore = oppure è possibile utilizzare il valore corrente della variabile in combinazione con altri operatori:

```
kelly@octarine ~> cat revenues
20021009      20021013      consultancy      BigComp      2500
20021015      20021020      training      EduComp      2000
20021112      20021123      appdev      SmartComp      10000
20021204      20021215      training      EduComp      5000

kelly@octarine ~> cat total.awk
{ total=total + $5 }
{ print "Send bill for " $5 " dollar to " $4 }
END { print "-----\nTotal revenue: " total }
```

```
kelly@octarine ~> awk -f total.awk test
Send bill for 2500 dollar to BigComp
Send bill for 2000 dollar to EduComp
Send bill for 10000 dollar to SmartComp
Send bill for 5000 dollar to EduComp
-----
Total revenue: 19500

kelly@octarine ~>
```

Sono accettate anche abbreviazioni di tipo C come **VAR+= valore**.

6.3.5. Altri esempi

L'esempio della [Sezione 5.3.2](#) diventa molto più semplice quando usiamo uno script **awk**:

```
kelly@octarine ~/html> cat make-html-from-text.awk
BEGIN { print "<html>\n<head><title>Awk-generated HTML</title></head>\n<body\n\nbgcolor=\"#ffffff\">\n\n<pre>" }
{ print $0 }
END { print "</pre>\n</body>\n</html>" }
```

E il comando da eseguire è anche molto più semplice quando si usa **awk** invece di **sed**:

```
kelly@octarine ~/html> awk -f make-html-from-text.awk testfile > file.html
```

Esempi awk sul sistema

Facciamo nuovamente riferimento alla directory contenente gli initscript sul sistema. Immettere un comando simile al seguente per vedere esempi più pratici dell'uso diffusissimo del comando **awk**:

```
grep awk /etc/init.d/*
```

6.3.6. Il programma printf

Per un controllo più preciso sul formato di output rispetto a quello normalmente fornito da **print**, utilizzare **printf**. Il comando **printf** può essere usato per specificare la larghezza del campo da usare per ogni elemento, così come varie scelte di formattazione per i numeri (come la base di output usare, se stampare un esponente, se stampare un segno e quante cifre stampare dopo la virgola). Questo viene fatto fornendo una stringa, chiamata *stringa di formato*, che controlla come e dove stampare gli altri argomenti.

La sintassi è la stessa dell'istruzione **printf** del linguaggio C; vedere la guida introduttiva al C. Le pagine di info **gawk** contengono spiegazioni complete.

6.4. Sommario

L'utility **gawk** interpreta un linguaggio di programmazione speciale, gestendo semplici compiti di riformattazione dei dati con poche righe di codice. È la versione gratuita del comando UNIX **awk**.

Questo strumento legge le righe dei dati di input e può facilmente mettere l'output in colonne. Il programma **print** è il più comune per filtrare e formattare i campi definiti.

La dichiarazione al volo delle variabili è semplice e consente un facile calcolo di somme, statistiche e altre operazioni sul flusso di input elaborato. Variabili e comandi possono essere inseriti in script **awk** per l'elaborazione in background.

Altre cose che da sapere su **awk**:

- Il linguaggio resta famoso su UNIX e simili, ma per eseguire compiti simili, Perl è ora più usato. Tuttavia, **awk** ha una curva di apprendimento molto più ripida (il che significa che si impara molto in un tempo molto breve). In altre parole, Perl è più difficile da imparare.
- Sia Perl che **awk** hanno in comune la reputazione di essere incomprensibili, anche agli stessi autori dei programmi che utilizzano questi linguaggi. Quindi documentare il codice!

6.5. Esercizi

Questi sono alcuni esempi pratici in cui **awk** può risultare utile.

1. Per il primo esercizio, l'input è costituito da linee nella seguente forma:

```
Username:Firstname:Lastname:Telephone number
```

2. Crea uno script **awk** che converta tale riga in un record LDAP in questo formato:

```
dn: uid=Username, dc=example, dc=com
cn: Firstname Lastname
sn: Lastname
telephoneNumber: Telephone number
```

3. Creare un file contenente un paio di record di test e controllare.
4. Crea uno script Bash usando **awk** e i comandi UNIX standard che mostreranno i primi tre utenti dello spazio su disco nel file system `/home` (se non c'è la directory contenente le 'home' su una partizione separata, creare lo script per la partizione `/`; questo è presente su ogni sistema UNIX). Innanzitutto, eseguire i comandi dalla riga di comando. Poi inserirli in uno script. Lo script dovrebbe creare un output sensato (come se lo si dovesse far leggere al capo). Se tutto funziona, chiedere allo script di inviare i risultati via email (usando per esempio **mail** `-s Utilizzo dello spazio su disco` `<you@your_comp>` `<risultato>`).

Se il demone 'quota' è in esecuzione, usare tale informazione; in caso contrario, usare **find**.

5. Creare un output in stile XML da un elenco separato da **Tab** nel seguente formato:

```
Meaning very long line with a lot of description  
meaning another long line  
  
othermeaning    more longline  
  
testmeaning  
loooooooooooooooooooooooooooooooooooooooooooooooooooooooooooooong line,  
but i mean really loooooooooooooooooooooooooooooooooooooooooooooong.
```

L'output si dovrebbe leggere:

[illegible]

Inoltre, se si conosce l'XML, scrivere uno script BEGIN e END per completare la tabella. Oppure farlo in HTML.

Capitolo 7. Istruzioni condizionali

In questo capitolo discuteremo l'uso delle istruzioni condizionali negli script Bash. Ciò include i seguenti argomenti:

- L'istruzione **if**
 - Utilizzo dello stato di uscita di un comando
 - Confronto e test di input e file
 - I costrutti **if/then/else**
 - I costrutti **if/then/elif/else**
 - Utilizzo e test dei parametri posizionali
 - Istruzioni **if** nidificate
 - Espressioni booleane
 - Utilizzo delle istruzioni **case**
-

7.1. Introduzione di **if**

7.1.1. In generale

A volte è necessario specificare diverse azioni da intraprendere in uno script di shell, a seconda del successo o del fallimento di un comando. Il costrutto **if** consente di specificare tali condizioni.

La sintassi più compatta del comando **if** è:

if TEST-COMMANDS; then CONSEQUENT-COMMANDS; fi

Viene eseguito l'elenco **COMANDI-DI-TEST**, e se lo stato di ritorno è zero, viene eseguita la lista di **COMANDI-CONSEQUENTI**. Lo stato di ritorno è lo stato di uscita dell'ultimo comando eseguito, o zero se nessuna condizione è stata verificata come vera.

Il **COMANDO-DI-TEST** spesso implica test di confronto numerici o di stringhe, ma può anche essere qualsiasi comando che restituisce uno stato pari a zero quando ha esito positivo e un altro stato quando fallisce. Le espressioni unarie vengono spesso usate per esaminare lo stato di un file. Se l'argomento **FILE** di uno dei primari è nella forma `/dev/fd/N`, allora viene controllato il descrittore del file "N". Per i test possono essere utilizzati anche `stdin`, `stdout` e `stderr` e i rispettivi descrittori di file.

7.1.1.1. Espressioni usate con **if**

La tabella seguente contiene una panoramica dei cosiddetti "primari" che compongono il comando o elenco di comandi **COMANDO-DI-TEST**. Questi primari sono messi tra parentesi quadre per indicare il test di un'espressione condizionale.

Tabella 7-1. Espressioni primarie

Primario	Significato
[-a FILE]	Vero se il FILE esiste.
[-b FILE]	Vero se il FILE esiste ed è un file speciale a blocchi [block-special].
[-c FILE]	Vero se FILE esiste ed è un file di caratteri speciali.
[-d FILE]	Vero se FILE esiste ed è una directory.
[-e FILE]	Vero se il FILE esiste.
[-f FILE]	Vero se FILE esiste ed è un file normale.
[-g FILE]	Vero se FILE esiste e il suo bit SGID è settato.
[-h FILE]	Vero se FILE esiste ed è un link simbolico.
[-k FILE]	Vero se FILE esiste e il suo bit permanente [sticky] è settato.
[-p FILE]	Vero se FILE esiste ed è una named pipe (FIFO).
[-r FILE]	Vero se FILE esiste ed è leggibile.
[-s FILE]	Vero se FILE esiste e ha una dimensione maggiore di zero.
[-t FD]	Vero se il descrittore di file FD è aperto e fa riferimento a un terminale.
[-u FILE]	Vero se FILE esiste e il suo bit SUID (imposta ID utente) è settato.
[-w FILE]	Vero se FILE esiste ed è scrivibile.
[-x FILE]	Vero se FILE esiste ed è eseguibile.
[-O FILE]	Vero se FILE esiste ed è di proprietà dell'effettivo ID utente .
[-G FILE]	Vero se FILE esiste ed è di proprietà dell'effettivo ID del gruppo.
[-L FILE]	Vero se FILE esiste ed è un link simbolico.
[-N FILE]	Vero se FILE esiste ed è stato modificato dall'ultima lettura.
[-S FILE]	Vero se FILE esiste ed è un socket.
[FILE1 -nt FILE2]	Vero se FILE1 è stato modificato più recentemente di FILE2 o se FILE1 esiste e FILE2 no.
[FILE1 -ot FILE2]	Vero se FILE1 è più vecchio di FILE2 o se FILE2 esiste e FILE1 non esiste.
[FILE1 -ef FILE2]	Vero se FILE1 e FILE2 fanno riferimento allo stesso dispositivo e allo stesso numero di inode.
[-o OPTIONNAME]	Vero se l'opzione shell "OPTIONNAME" è abilitata.
[-z STRING]	Vero se la lunghezza di "STRING" è zero.
[-n STRING] or [STRING]	Vero se la lunghezza di "STRING" è diversa da zero.
[STRING1 == STRING2]	Vero se le stringhe sono uguali. "=" può essere utilizzato al posto di "==" per la stretta conformità a POSIX.
[STRING1 != STRING2]	Vero se le stringhe non sono uguali.
[STRING1 < STRING2]	Vero se "STRING1" viene prima di "STRING2" alfabeticamente nella lingua corrente.
[STRING1 > STRING2]	Vero se "STRING1" viene dopo "STRING2" alfabeticamente nella lingua corrente.
[ARG1 OP ARG2]	"OP" è uno tra -eq, -ne, -lt, -le, -gt o -ge. Questi operatori binari aritmetici restituiscono 'true' se "ARG1" è uguale a, non uguale a, minore,

Primario	Significato
	minore o uguale, maggiore, maggiore o uguale a "ARG2", rispettivamente. "ARG1" e "ARG2" sono interi.

Le espressioni possono essere combinate utilizzando i seguenti operatori, elencati in ordine decrescente di precedenza:

Tabella 7-2. Combinazione di espressioni

Operazione	Effetto
[! EXPR]	Vero se EXPR è falso.
[(EXPR)]	Restituisce il valore di EXPR . Può essere usato per sovrascrivere la normale precedenza degli operatori.
[EXPR1 -a EXPR2]	Vero se sia EXPR1 che EXPR2 sono vere.
[EXPR1 -o EXPR2]	Vero se o EXPR1 o EXPR2 è vero.

Il costrutto interno [(o **test**) valuta le espressioni condizionali utilizzando un insieme di regole basate sul numero di argomenti. Ulteriori informazioni su questo argomento possono essere trovate nella documentazione di Bash. Così come l'**if** è chiuso con **fi**, la parentesi quadra di apertura dovrebbe essere chiusa dopo l'elenco delle condizioni.

7.1.1.2. Comandi che seguono l'istruzione then

L'elenco **COMANDI-CONSEQUENTI** che segue l'istruzione **then** può essere qualsiasi comando UNIX valido, qualsiasi programma eseguibile, qualsiasi script di shell eseguibile o qualsiasi istruzione di shell, ad eccezione dell'istruzione di chiusura **fi**. È importante ricordare che **then** e **fi** sono considerati istruzioni separate nella shell. Pertanto, quando appaiono sulla riga di comando, sono separati da un punto e virgola.

In uno script, le diverse parti dell'istruzione **if** sono generalmente ben separate. Di seguito, un paio di semplici esempi.

7.1.1.3. Controllo dei file

Il primo esempio verifica l'esistenza di un file:

```
anny ~> cat msgcheck.sh
#!/bin/bash

echo "This scripts checks the existence of the messages file."
echo "Checking..."
if [ -f /var/log/messages ]
then
    echo "/var/log/messages exists."
fi
echo
echo "...done."
```



```

anny ~> ./msgcheck.sh
This scripts checks the existence of the messages file.
Checking...
/var/log/messages exists.

...done.

```

7.1.1.4. Controllo delle opzioni della shell

Per aggiungere nei file di configurazione di Bash:

```

# These lines will print a message if the noclobber option is set:

if [ -o noclobber ]
then
    echo "Your files are protected against accidental overwriting using
redirection."
fi

```

L'ambiente [environment]

L'esempio sopra funziona se inserito nella riga di comando:

```

anny ~> if [ -o noclobber ] ; then echo ; echo "your files are protected
against overwriting." ; echo ; fi

your files are protected against overwriting.

anny ~>

```

Tuttavia, se si testano le condizioni che dipendono dall'environment, si potrebbero ottenere risultati diversi inserendo lo stesso comando in uno script, perché lo script aprirà una nuova shell, in cui le variabili e le opzioni previste potrebbero non essere impostate automaticamente.

7.1.2. Una semplice applicazione di if

7.1.2.1. Testare lo stato d'uscita

La variabile `?` contiene lo stato di uscita del comando eseguito in precedenza (l'ultimo processo in primo piano completato).

L'esempio seguente mostra un semplice test:

```

anny ~> if [ $? -eq 0 ]
More input> then echo 'That was a good job!'
More input> fi
That was a good job!

anny ~>

```

L'esempio seguente dimostra che **COMANDI-DI-TEST** potrebbe essere qualsiasi comando UNIX che restituisce uno stato di uscita e che **if** restituisce nuovamente uno stato di uscita pari a zero:

```

anny ~> if ! grep $USER /etc/passwd

```

```
More input> then echo "your user account is not managed locally"; fi
your user account is not managed locally

anny > echo $?
0

anny >
```

Lo stesso risultato si può ottenere nel seguente modo:

```
anny > grep $USER /etc/passwd

anny > if [ $? -ne 0 ] ; then echo "not a local account" ; fi
not a local account

anny >
```

7.1.2.2. Confronti numerici

Gli esempi seguenti utilizzano confronti numerici:

```
anny > num=`wc -l work.txt`

anny > echo $num
201

anny > if [ "$num" -gt "150" ]
More input> then echo ; echo "you've worked hard enough for today."
More input> echo ; fi

you've worked hard enough for today.

anny >
```

Questo script viene eseguito da cron ogni domenica. Se il numero della settimana è pari, ti ricorda di buttare la spazzatura:

```
#!/bin/bash

# Calculate the week number using the date command:
WEEKOFFSET=$((date +%V) % 2)

# Test if we have a remainder. If not, this is an even week so send a message.
# Else, do nothing.

if [ $WEEKOFFSET -eq "0" ]; then
    echo "Domenica sera, metti fuori i bidoni della spazzatura." | mail -s
    "Buttare la spazzatura" your@your_domain.org
fi
```

7.1.2.3. Confronti tra stringhe

Un esempio di confronto tra stringhe per testare l'ID utente:

```
if [ "$(whoami)" != 'root' ]; then
    echo "You have no permission to run $0 as non-root user."
```

```
    exit 1;
fi
```

Con Bash, si può abbreviare il costrutto. L'equivalente compatto del test sopra è il seguente:

```
[ "$(whoami)" != 'root' ] && ( echo you are using a non-privileged account;
exit 1 )
```

Simile all'espressione "&&" che indica cosa fare se il test risulta vero, "||" specifica cosa fare se il test è falso.

Le espressioni regolari possono essere utilizzate anche nelle comparazioni:

```
anny > gender="female"

anny > if [[ "$gender" == f* ]]
More input> then echo "Pleasure to meet you, Madame."; fi
Pleasure to meet you, Madame.

anny >
```

Programmatori Veri

La maggior parte dei programmatori preferirà utilizzare il comando interno **test**, che equivale a utilizzare le parentesi quadre per il confronto, in questo modo:

```
test "$(whoami)" != 'root' && (echo you are using a non-privileged
account; exit 1)
```

No exit?

Se invochi l'**exit** in una sotto-shell, non verranno passate le variabili al genitore. Usa { e } invece di (e) se non si vuole che Bash esegua il fork di una sotto-shell

Vedere le pagine di info per Bash per ulteriori informazioni sul pattern matching con i costrutti "((EXPRESSION))" e "[[EXPRESSION]]".

7.2. Utilizzo più avanzato di if

7.2.1. costrutti if/then/else

7.2.1.1. Esempio fittizio

Questo è il costrutto da utilizzare per fare un'azione se i comandi **if** risultano veri, e un'altra se risulta falso. Un esempio:

```
freddy scripts> gender="male"

freddy scripts> if [[ "$gender" == "f*" ]]
More input> then echo "Pleasure to meet you, Madame."
More input> else echo "How come the lady hasn't got a drink yet?"
More input> fi
How come the lady hasn't got a drink yet?

freddy scripts>
```

[] e [[]]

Contrariamente a [, [[] impedisce la suddivisione in parole dei valori delle variabili. Quindi, se VAR="var con spazi", non c'è bisogno di raddoppiare le virgolette \$VAR in un test - anche se usare le virgolette rimane una buona abitudine. Inoltre, [[] impedisce l'espansione del path, quindi le stringhe letterali con caratteri jolly non tentano di espandersi in nomi di file. Utilizzando [, == e != vengono interpretate le stringhe a destra come [shell glob pattern] da confrontare col valore a sinistra, ad esempio: [[] "value" == val*]].

Come la lista **COMANDI-CONSEQUENTI** che segue l'istruzione **then**, l'elenco **COMANDI-CONSEQUENTI-ALTERNATIVI** che segue l'istruzione **else** può contenere qualsiasi comando in stile UNIX che restituisca uno stato di uscita.

Un altro esempio, ampliando quello della [Sezione 7.1.2.1](#):

```
anny ~> su -
Password:
[root@elegance root]# if ! grep ^$USER /etc/passwd 1> /dev/null
> then echo "your user account is not managed locally"
> else echo "your account is managed from the local /etc/passwd file"
> fi
your account is managed from the local /etc/passwd file
[root@elegance root]#
```

Passiamo all'account *root* per mostrare l'effetto dell'istruzione **else**: *root* è solitamente un account locale mentre l'account utente potrebbe essere gestito da un sistema centrale, come un server LDAP.

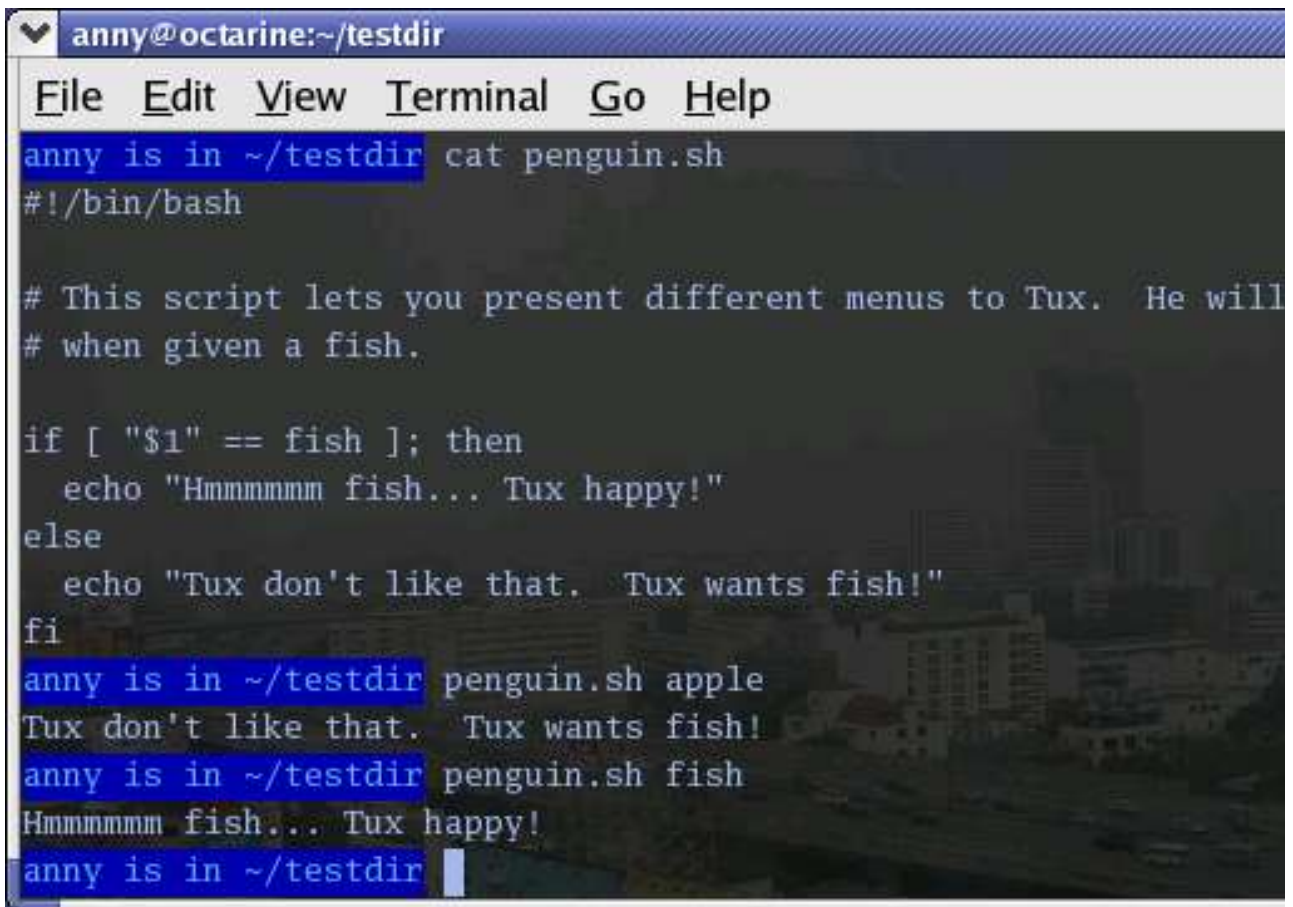
7.2.1.2. Controllo degli argomenti sulla riga di comando

Invece di impostare una variabile e poi eseguire uno script, spesso è più elegante inserire i valori per le variabili nella riga di comando.

Si usano i parametri posizionali \$1, \$2, ..., \$N per questo scopo. \$# si riferisce al numero di argomenti sulla riga di comando. \$0 si riferisce al nome dello script.

Quello che segue è un semplice esempio:

Figura 7-1. Test di un argomento sulla riga di comando con if



```

anny@octarine:~/testdir
File Edit View Terminal Go Help
anny is in ~/testdir cat penguin.sh
#!/bin/bash

# This script lets you present different menus to Tux.  He will
# when given a fish.

if [ "$1" == fish ]; then
    echo "HMMMMMMM fish... Tux happy!"
else
    echo "Tux don't like that.  Tux wants fish!"
fi

anny is in ~/testdir penguin.sh apple
Tux don't like that.  Tux wants fish!
anny is in ~/testdir penguin.sh fish
HMMMMMMM fish... Tux happy!
anny is in ~/testdir

```

Ecco un altro esempio, utilizzando due argomenti:

```

anny ~> cat weight.sh
#!/bin/bash

# This script prints a message about your weight if you give it your
# weight in kilos and height in centimeters.

weight="$1"
height="$2"
idealweight=$((height - 110))

if [ $weight -le $idealweight ] ; then
    echo "You should eat a bit more fat."
else
    echo "You should eat a bit more fruit."
fi

anny ~> bash -x weight.sh 55 169
+ weight=55
+ height=169
+ idealweight=59
+ '[' 55 -le 59 ']'
+ echo 'You should eat a bit more fat.'
You should eat a bit more fat.

```

7.2.1.3. Testare il numero di argomenti

L'esempio seguente mostra come modificare lo script precedente in modo che stampi un messaggio se vengono forniti più o meno di 2 argomenti:

```

anny ~> cat weight.sh
#!/bin/bash

# This script prints a message about your weight if you give it your
# weight in kilos and height in centimeters.

if [ ! $# == 2 ]; then
    echo "Usage: $0 weight_in_kilos length_in_centimeters"
    exit
fi

weight="$1"
height="$2"
idealweight=$((height - 110))

if [ $weight -le $idealweight ] ; then
    echo "You should eat a bit more fat."
else
    echo "You should eat a bit more fruit."
fi

anny ~> weight.sh 70 150
You should eat a bit more fruit.

anny ~> weight.sh 70 150 33
Usage: ./weight.sh weight_in_kilos length_in_centimeters

```

Al primo si fa riferimento con \$1, al secondo con \$2 e così via. Il numero totale di argomenti è memorizzato in \$#.

Consultare la [Sezione 7.2.5](#) per un modo più elegante di stampare i messaggi sull'utilizzo.

7.2.1.4. Verificare l'esistenza di un file

Questo test viene eseguito in molti script, perché è inutile avviare molti programmi se si sa che non funzioneranno:

```

#!/bin/bash

# This script gives information about a file.

FILENAME="$1"

echo "Properties for $FILENAME:"

if [ -f $FILENAME ]; then
    echo "Size is $(ls -lh $FILENAME | awk '{ print $5 }')"
    echo "Type is $(file $FILENAME | cut -d":" -f2 -)"
    echo "Inode number is $(ls -li $FILENAME | cut -d" " -f1 -)"
    echo "$((df -h $FILENAME | grep -v Mounted | awk '{ print "On",$1", \\"
which is mounted as the",$6,"partition."}'))"
else
    echo "File does not exist."
fi

```

Notare che il file è riferito tramite una variabile; in questo caso è il primo argomento dello script. In alternativa, quando non vengono forniti argomenti, le posizioni dei file vengono solitamente memorizzate in variabili all'inizio di uno script e il loro contenuto viene indicato utilizzando tali

variabili. Pertanto, quando si desidera modificare il nome di un file in uno script, è necessario farlo una sola volta.

Nomi di file con spazi

L'esempio precedente avrà esito negativo se il valore di `$1` può essere parserizzato come più parole. In tal caso, il comando **if** può essere corretto utilizzando le virgolette doppie attorno al nome del file o utilizzando `[[` invece di `[`.

7.2.2. Costrutti if/then/elif/else

7.2.2.1. In generale

Questa è la forma completa dell'istruzione **if**:

if TEST-COMMANDS; **then**

CONSEQUENT-COMMANDS;

elif MORE-TEST-COMMANDS; **then**

MORE-CONSEQUENT-COMMANDS;

else ALTERNATE-CONSEQUENT-COMMANDS;

fi

Viene eseguita la lista **TEST-COMMANDS** e se lo stato di ritorno è zero, viene eseguita la lista **CONSEQUENT-COMMANDS**. Se **TEST-COMMANDS** restituisce uno stato diverso da zero, ogni lista **elif** viene eseguita a turno, e se lo stato di uscita è zero, vengono eseguiti i corrispondenti **MORE-CONSEQUENT-COMMANDS** e il comando viene completato. Se **else** è seguito da un elenco di **ALTERNATE-CONSEQUENT-COMMANDS** e il comando finale nell'ultima clausola **if** o **elif** ha uno stato di uscita diverso da zero, allora viene eseguito **ALTERNATE-CONSEQUENT-COMMANDS**. Lo stato di ritorno è lo stato di uscita dell'ultimo comando eseguito, o zero se nessuna condizione è stata verificata come vera.

7.2.2.2. Esempio

Questo è un esempio inseribile nel crontab per l'esecuzione quotidiana:

```
anny /etc/cron.daily> cat disktest.sh
#!/bin/bash

# This script does a very simple test for checking disk space.

space=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 |
cut -d "%" -f1 -`
alertvalue="80"

if [ "$space" -ge "$alertvalue" ]; then
    echo "At least one of my disks is nearly full!" | mail -s "daily diskcheck"
    root
else
```

```
echo "Disk space normal" | mail -s "daily diskcheck" root
fi
```

7.2.3. Istruzioni if nidificate

All'interno dell'istruzione **if**, si può utilizzare un'altra istruzione **if**. Si possono usare tanti livelli di **if** annidati quanti se ne possono gestire logicamente.

Questo è un esempio per testare gli anni bisestili:

```
anny ~/testdir> cat testleap.sh
#!/bin/bash
# This script will test if we're in a leap year or not.

year=`date +%Y`

if [ ${year} % 400 -eq 0 ]; then
    echo "This is a leap year. February has 29 days."
elif [ ${year} % 4 -eq 0 ]; then
    if [ ${year} % 100 -ne 0 ]; then
        echo "This is a leap year, February has 29 days."
    else
        echo "This is not a leap year. February has 28 days."
    fi
else
    echo "This is not a leap year. February has 28 days."
fi

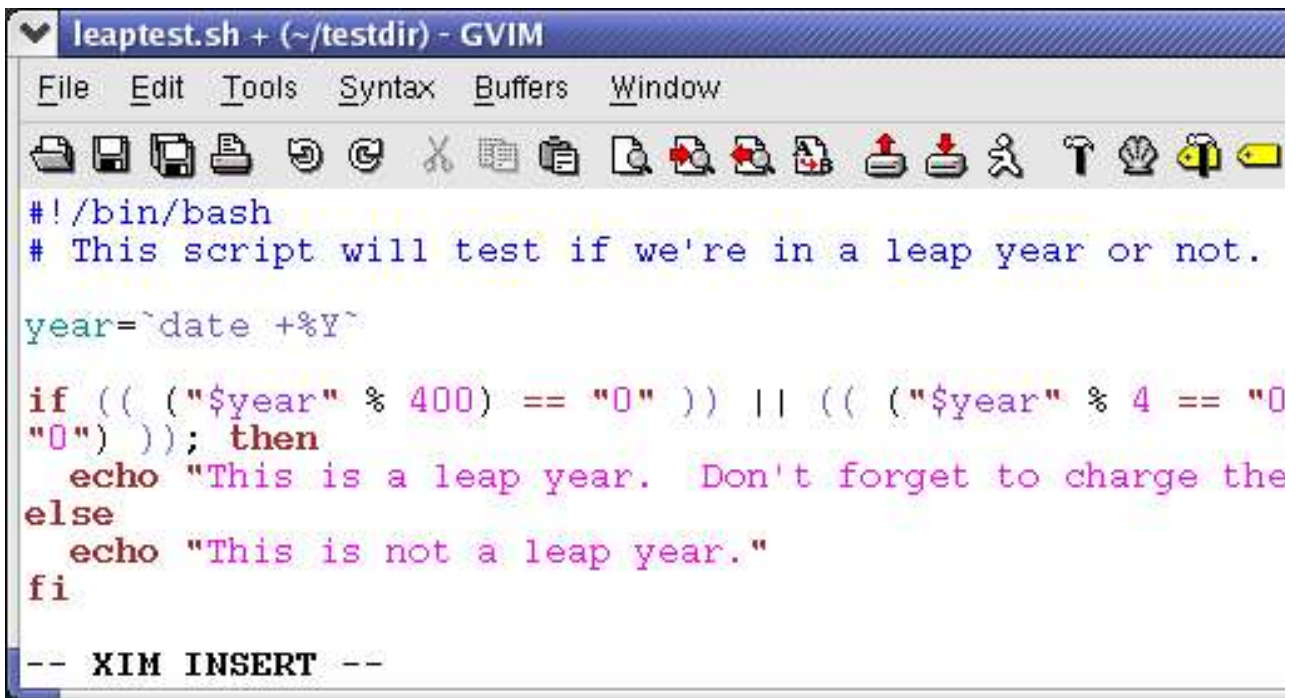
anny ~/testdir> date
Tue Jan 14 20:37:55 CET 2003

anny ~/testdir> testleap.sh
This is not a leap year.
```

7.2.4. Operazioni booleane

Lo script precedente può essere abbreviato utilizzando gli operatori booleani "AND" (&&) e "OR" (||).

Figura 7-2. Esempio con gli operatori booleani



```

leaptest.sh + (~/testdir) - GVIM
File Edit Tools Syntax Buffers Window
#!/bin/bash
# This script will test if we're in a leap year or not.

year=`date +%Y`

if (( (" $year" % 400) == "0" )) || (( (" $year" % 4 == "0" )); then
    echo "This is a leap year. Don't forget to charge the
else
    echo "This is not a leap year."
fi

-- XIM INSERT --

```

Usiamo le doppie parentesi per testare un'espressione aritmetica, vedere [Sezione 3.4.6](#). Questo è equivalente all'istruzione **let**. Si rimarrà perplessi usando le parentesi quadre, se si prova qualcosa come `[$year % 400]`, perché qui le parentesi quadre non rappresentano da sole un comando effettivo.

Tra gli altri editor, **gvim** è uno di quelli che supportano gli schemi di colori in base al formato del file; tali editor sono utili per rilevare errori nel codice.

7.2.5. Uso delle istruzioni **exit** e **if**

Abbiamo già incontrato brevemente l'istruzione **exit** nella [Sezione 7.2.1.3](#). Essa termina l'esecuzione dell'intero script. Viene spesso utilizzata se l'input richiesto dall'utente non è corretto, se un'istruzione non è stata eseguita correttamente o se si è verificato qualche altro errore.

L'istruzione **exit** accetta un argomento opzionale. Tale argomento è il codice intero dello stato di uscita, che viene passato al genitore e memorizzato nella variabile `$?`.

Uno zero significa che lo script è stato eseguito correttamente. Qualsiasi altro valore può essere utilizzato dai programmatori per restituire messaggi diversi al genitore, in modo che possano essere intraprese azioni diverse in base al fallimento o al successo del processo figlio. Se non viene fornito alcun argomento al comando **exit**, la shell genitore utilizza il valore corrente della variabile `$?`.

Di seguito è riportato un esempio con uno script `penguin.sh` leggermente adattato, che invia il suo stato di uscita al genitore, `feed.sh`:

```

anny ~/testdir> cat penguin.sh
#!/bin/bash

# This script lets you present different menus to Tux. Sarà felice solo
# quando gli viene dato un pesce. We've also added a dolphin and (presumably)
a camel.

```

```

if [ "$menu" == "fish" ]; then
    if [ "$animal" == "penguin" ]; then
        echo "HMMMMMM fish... Tux happy!"
    elif [ "$animal" == "dolphin" ]; then
        echo "Pweetpeettreetppeterdepweet!"
    else
        echo "*prrrrrrrrt*"
    fi
else
    if [ "$animal" == "penguin" ]; then
        echo "Tux don't like that.  Tux wants fish!"
        exit 1
    elif [ "$animal" == "dolphin" ]; then
        echo "Pweepwishpeeterdepweet!"
        exit 2
    else
        echo "Will you read this sign?!"
        exit 3
    fi
fi

```

Questo script viene richiamato nel successivo, che quindi esporta le sue variabili `menu` e `animal`:

```

anny ~/testdir> cat feed.sh
#!/bin/bash
# This script acts upon the exit status given by penguin.sh

export menu="$1"
export animal="$2"

feed="/nethome/anny/testdir/penguin.sh"

$feed $menu $animal

case $? in
1)
    echo "Guard: You'd better give'm a fish, less they get violent..."
    ;;
2)
    echo "Guard: It's because of people like you that they are leaving earth all
the time..."
    ;;
3)
    echo "Guard: Buy the food that the Zoo provides for the animals, you ***, how
do you think we survive?"
    ;;
*)
    echo "Guard: Don't forget the guide!"
    ;;
esac

anny ~/testdir> ./feed.sh apple penguin
Tux don't like that.  Tux wants fish!
Guard: You'd better give'm a fish, less they get violent...

```

Come si può vedere, i codici di stato di uscita possono essere scelti liberamente. I comandi esistenti di solito hanno una serie di codici definiti; vedere il manuale del programmatore per ogni comando per maggiori informazioni.

7.3. L'istruzione case

7.3.1. Condizioni semplificate

Le istruzioni **if** nidificate potrebbero essere utili, ma non appena ci si trova di fronte a un paio di possibili azioni da intraprendere, tendono a confondere. Per i condizionali più complessi, si usa la sintassi **case**:

case *EXPRESSION in CASE1) COMMAND-LIST;; CASE2) COMMAND-LIST;; ... CASEN)*
COMMAND-LIST;; esac

Ogni **case** è un'espressione che corrisponde a un pattern. Vengono eseguiti i comandi nella **COMMAND-LIST** per la prima corrispondenza. Il simbolo "|" viene utilizzato per separare più pattern e l'operatore ")" termina un elenco di pattern. Ciascun **case** più i relativi comandi viene detti *clausola*. Ogni clausola deve essere terminata con ";;". Ogni istruzione **case** termina con l'istruzione **esac**.

Nell'esempio, dimostriamo l'uso dei **case** per inviare un avviso più selettivo con lo script `disktest.sh`:

```
anny ~/testdir> cat disktest.sh
#!/bin/bash

# This script does a very simple test for checking disk space.

space=`df -h | awk '{print $5}' | grep % | grep -v Use | sort -n | tail -1 |
cut -d "%" -f1 -`

case $space in
[1-6]*)
    Message="All is quiet."
    ;;
[7-8]*)
    Message="Start thinking about cleaning out some stuff. There's a partition
that is $space % full."
    ;;
9[1-8])
    Message="Better hurry with that new disk... One partition is $space % full."
    ;;
99)
    Message="I'm drowning here! There's a partition at $space %!"
    ;;
*)
    Message="I seem to be running with an nonexistent amount of disk space..."
    ;;
esac

echo $Message | mail -s "disk report `date`" anny

anny ~/testdir>
You have new mail.

anny ~/testdir> tail -16 /var/spool/mail/anny
From anny@octarine Tue Jan 14 22:10:47 2003
Return-Path: <anny@octarine>
Received: from octarine (localhost [127.0.0.1])
    by octarine (8.12.5/8.12.5) with ESMTP id h0ELAlBG020414
    for <anny@octarine>; Tue, 14 Jan 2003 22:10:47 +0100
Received: (from anny@localhost)
    by octarine (8.12.5/8.12.5/Submit) id h0ELAltn020413
```

```

    for anny; Tue, 14 Jan 2003 22:10:47 +0100
Date: Tue, 14 Jan 2003 22:10:47 +0100
From: Anny <anny@octarine>
Message-Id: <200301142110.h0ELAltn020413@octarine>
To: anny@octarine
Subject: disk report Tue Jan 14 22:10:47 CET 2003

Start thinking about cleaning out some stuff.  There's a partition that is 87 %
full.

anny ~/testdir>

```

Ovviamente basterebbe aprire il programma di posta per controllare i risultati; questo è solo per dimostrare che lo script invia una mail decente con le righe di intestazione "To:", "Subject:" e "From:" .

Molti altri esempi che utilizzano le istruzioni **case** si possono trovare nella directory degli script di inizializzazione del sistema. Gli script di avvio utilizzano i casi **start** e **stop** per eseguire o arrestare i processi di sistema. Un esempio teorico si trova nella sezione successiva.

7.3.2. Esempio di initscript

Gli initscript spesso fanno uso di istruzioni **case** per avviare, arrestare e interrogare i servizi di sistema. Questo è un estratto dello script che avvia Anacron, un demone che esegue comandi periodicamente con una frequenza specificata in giorni.

```

case "$1" in
    start)
        start
        ;;

    stop)
        stop
        ;;

    status)
        status anacron
        ;;

    restart)
        stop
        start
        ;;

    condrestart)
        if test "x`pidof anacron`" != x; then
            stop
            start
        fi
        ;;

    *)
        echo $"Usage: $0 {start|stop|restart|condrestart|status}"
        exit 1
esac

```

Le attività da eseguire in ogni caso, come l'arresto e l'avvio del demone, sono definite nelle funzioni, parzialmente prese dal file `/etc/rc.d/init.d/functions`. Vedere il [Capitolo 11](#) per ulteriori spiegazioni.

7.4. Sommario

In questo capitolo abbiamo imparato come costruire le condizioni nei nostri script in modo che diverse azioni possano essere intraprese in caso di successo o fallimento di un comando. Le azioni possono essere determinate utilizzando l'istruzione **if**. Ciò consente di eseguire comparazioni aritmetiche e tra stringhe e testare il codice di uscita, l'input e i file necessari allo script.

Un semplice test **if/then/fi** spesso precede i comandi in uno script di shell per impedire la generazione dell'output, in modo che lo script possa essere facilmente eseguito in background o tramite il cron. Definizioni delle condizioni più complesse vengono solitamente inserite in un'istruzione **case**.

In caso di successo del test delle condizioni, lo script può informare esplicitamente il genitore utilizzando lo stato **exit 0**. In caso di errore, può essere restituito qualsiasi altro numero. In base al codice di ritorno, il programma padre può intraprendere l'azione appropriata.

7.5. Esercizi

Ecco alcune idee per iniziare a utilizzare **if** negli script:

1. Usare un costrutto **if/then/elif/else** che stampa le informazioni sul mese corrente. Lo script dovrebbe stampare il numero di giorni nel mese corrente e fornire informazioni sugli anni bisestili se il mese è febbraio.
2. Fare lo stesso, con un'istruzione **case** e un uso alternativo del comando **date**.
3. Modificare `/etc/profile` in modo da ricevere un saluto speciale quando ci si connetti al sistema come *root*.
4. Modificare lo script `leaptest.sh` della [Sezione 7.2.4](#) in modo che richieda un argomento, l'anno. Verificare che venga fornito esattamente e solo un argomento.
5. Scrivere uno script chiamato `whichdaemon.sh` che controlli se i demoni **httpd** e **init** sono in esecuzione sul sistema. Se è in esecuzione un **httpd**, lo script dovrebbe stampare un messaggio del tipo "Questa macchina sta eseguendo un server web." Usare **ps** per controllare processi.
6. Scrivere uno script che esegua un backup della directory home su una macchina remota usando **scp**. Lo script dovrebbe riportare in un file di log, ad esempio `~/log/homebackup.log`. Se non si ha una seconda macchina su cui copiare il backup, usare **scp** per provare a copiarlo su localhost. Ciò richiede chiavi SSH tra i due host, altrimenti si deve fornire una password. La creazione delle chiavi SSH è spiegata in **man ssh-keygen**.
7. Adattare lo script del primo esempio nella [Sezione 7.3.1](#) per includere il caso di utilizzo di spazio su disco esattamente del 90% e un utilizzo di spazio su disco inferiore al 10%.

Lo script dovrebbe usare **tar cf** per la creazione del backup e **gzip** o **bzip2** per comprimere il file `.tar`. Mettere tutti i nomi di file nelle variabili. Inserire il nome del server remoto e della directory remota in una variabile. In questo modo sarà più facile riutilizzare lo script o apportare modifiche in futuro.

Lo script dovrebbe verificare l'esistenza di un archivio compresso. Se esiste, rimuoverlo prima per impedire la generazione di output.

Lo script dovrebbe anche controllare lo spazio disponibile su disco. Si tenga presente che in qualsiasi momento si potrebbero avere i dati nella home directory, i dati nel file `.tar` e i dati nell'archivio compresso tutti insieme sul disco. Se non c'è abbastanza spazio su disco, uscire con un messaggio di errore nel file di registro

Lo script dovrebbe ripulire l'archivio compresso prima di uscire.

Capitolo 8. Scrivere script interattivi

In questo capitolo discuteremo come interagire con gli utenti dei nostri script:

- Stampa di messaggi e spiegazioni facili da usare
- Catturare l'input dell'utente
- Richiesta di input da parte dell'utente
- Utilizzo dei descrittori di file per leggere e scrivere su più file

8.1. Visualizzazione dei messaggi dell'utente

8.1.1. Interattivo o no?

Alcuni script vengono eseguiti senza alcuna interazione da parte dell'utente. I vantaggi degli script non interattivi includono:

- Lo script viene eseguito ogni volta in modo prevedibile.
- Lo script può essere eseguito in background.

Molti script, tuttavia, richiedono input da parte dell'utente o forniscono output all'utente mentre è in esecuzione. I vantaggi degli script interattivi sono, tra gli altri:

- È possibile creare script più flessibili.
- Gli utenti possono personalizzare lo script durante l'esecuzione o farlo comportare in modi diversi.
- Lo script può segnalare i suoi progressi durante l'esecuzione.

Quando si scrivono script interattivi, non lesinare mai sui commenti. Uno script che stampa i messaggi appropriati è molto più intuitivo e può essere più facilmente sottoposto a debug. Uno script potrebbe fare un lavoro perfetto, ma si riceveranno molte chiamate di supporto se non si informa l'utente su ciò che sta accadendo. Quindi si devono includere messaggi che dicono all'utente di attendere l'output perché è in corso un calcolo. Se possibile, provare a dare un'indicazione di quanto tempo l'utente dovrà attendere. Se l'attesa dovesse richiedere regolarmente molto tempo durante l'esecuzione di una determinata attività, si potrebbe prendere in considerazione l'integrazione di alcune indicazioni di elaborazione nell'output dello script.

Quando si richiede l'input all'utente, è meglio fornire troppe informazioni sul tipo di dati da inserire che troppo poche. Questo vale anche per il controllo degli argomenti e per il messaggio sull'uso allegato.

Bash ha i comandi **echo** e **printf** per fornire commenti agli utenti, e anche se si ha ormai una certa familiarità con l'uso di **echo**, discuteremo altri esempi nelle sezioni successive.

8.1.2. Utilizzo del comando echo

Il comando interno **echo** scrive i suoi argomenti, separati da spazi e terminati con un carattere di nuova riga. Lo stato di ritorno è sempre zero. **echo** richiede un paio di opzioni:

- **-e**: interpreta i caratteri escape con la backslash.
- **-n**: sopprime l'andata a capo finale.

Come esempio di commenti aggiunti, miglioreremo un po' `feed.sh` e `penguin.sh` dalla [Sezione 7.2.1.2](#):

```
michel ~/test> cat penguin.sh
#!/bin/bash

# Questo script consente di presentare diversi menu a Tux.. Sarà felice solo
# quando gli viene dato un pesce. Per renderlo più divertente, abbiamo
# aggiunto un altro paio di animali.

if [ "$menu" == "fish" ]; then
    if [ "$animal" == "penguin" ]; then
        echo -e "HMMMMMM fish... Tux happy!\n"
    elif [ "$animal" == "dolphin" ]; then
        echo -e "\a\a\aPweetpeettreettpeterdepweet!\a\a\a\n"
    else
        echo -e "*prrrrrrrrt*\n"
    fi
else
    if [ "$animal" == "penguin" ]; then
        echo -e "Tux don't like that. Tux wants fish!\n"
        exit 1
    elif [ "$animal" == "dolphin" ]; then
        echo -e "\a\a\a\a\aPweepwishpeeterdepweet!\a\a\a"
        exit 2
    else
        echo -e "Will you read this sign?! Don't feed the \"$animal\"s!\n"
        exit 3
    fi
fi

michel ~/test> cat feed.sh
#!/bin/bash
# This script acts upon the exit status given by penguin.sh

if [ "$#" != "2" ]; then
    echo -e "Usage of the feed script:\t$0 food-on-menu animal-name\n"
    exit 1
else

    export menu="$1"
    export animal="$2"

    echo -e "Feeding $menu to $animal...\n"

    feed="/nethome/anny/testdir/penguin.sh"

    $feed $menu $animal

result="$?"

    echo -e "Done feeding.\n"
```

```

case "$result" in
  1)
    echo -e "Guard: \"You'd better give'm a fish, less they get violent...\"\n"
    ;;
  2)
    echo -e "Guard: \"No wonder they flee our planet...\"\n"
    ;;
  3)
    echo -e "Guard: \"Buy the food that the Zoo provides at the entry, you
***\"\n"
    echo -e "Guard: \"You want to poison them, do you?\"\n"
    ;;
  *)
    echo -e "Guard: \"Don't forget the guide!\"\n"
    ;;
esac

fi

echo "Leaving..."
echo -e "\a\aThanks for visiting the Zoo, hope to see you again soon!\n"

michel ~/test> feed.sh apple camel
Feeding apple to camel...

Will you read this sign?!  Don't feed the camels!

Done feeding.

Guard: "Buy the food that the Zoo provides at the entry, you ***"

Guard: "You want to poison them, do you?"

Leaving...
Thanks for visiting the Zoo, hope to see you again soon!

michel ~/test> feed.sh apple
Usage of the feed script:      ./feed.sh food-on-menu animal-name

```

Maggiori informazioni sui caratteri di escape si possono trovare nella [Sezione 3.3.2](#). La tabella seguente fornisce una panoramica delle sequenze riconosciute dal comando **echo**:

Tabella 8-1. Sequenze di escape usate dal comando echo

Sequenza	Significato
\a	Allarme (campanello).
\b	Backspace.
\c	Sopprime l'andata a capo finale.
\e	Escape.
\f	Form feed.
\n	Newline.
\r	Carriage return.
\t	Tab orizzontale.
\v	Tab verticale.
\\	Backslash.
\0NNN	Il carattere a otto bit il cui valore in ottale è NNN (da zero a tre cifre ottali).

Sequenza	Significato
\NNN	Il carattere a otto bit il cui valore in ottale è NNN (da una a tre cifre ottali).
\xHH	Il carattere a otto bit il cui valore è in esadecimale (una o due cifre esadecimali).

Per ulteriori informazioni sul comando **printf** e sul modo in cui consente di formattare l'output, vedere le pagine info di Bash. Si tenga presente che potrebbero esserci differenze tra le diverse versioni di Bash.

8.2. Catturare l'input dell'utente

8.2.1. Uso del comando nativo read

Il comando nativo **read** è la controparte dei comandi **echo** e **printf**. La sintassi del comando **read** è la seguente:

```
read [options] NAME1 NAME2 ... NAMEN
```

Viene letta una riga dall'input standard, o dal descrittore di file fornito come argomento con l'opzione **-u**. La prima parola della riga è assegnata al primo nome, **NAME1**, la seconda parola al secondo nome, e così via, con le restanti parole e i loro separatori interposti assegnati all'ultimo nome, **NAMEN**. Se ci sono meno parole lette dal flusso di input rispetto ai nomi, a quelli rimanenti vengono assegnati valori vuoti.

I caratteri nel valore della variabile **IFS** vengono utilizzati per suddividere la riga di input in parole o token; vedere la [Sezione 3.4.8](#). Il carattere backslash può essere utilizzato per rimuovere qualsiasi significato speciale al successivo carattere letto e per la continuazione di riga.

Se non viene fornito alcun nome, la riga letta viene assegnata alla variabile **REPLY**.

Il codice di ritorno del comando **read** è zero, a meno che non venga incontrato un carattere di fine file, se **read** va in time out o se viene fornito un descrittore di file non valido come argomento dell'opzione **-u**.

Le seguenti opzioni sono supportate dal **read** nativo di Bash:

Tabella 8-2. Parametri per il read nativo

Opzione	Significato
-a ANAME	Le parole vengono assegnate a indici sequenziali dell'array ANAME , a partire da 0. Tutti gli elementi vengono rimossi da ANAME prima dell'assegnazione. Gli altri argomenti NAME vengono ignorati.
-d DELIM	Il primo carattere di DELIM viene usato per terminare la riga di input, anziché il newline.
-e	readline viene usato per ottenere la riga.
-n NCHARS	read torna dopo aver letto NCHARS caratteri anziché aspettare il completamento della riga di input.
-p PROMPT	Mostra PROMPT , senza un newline finale, prima di tentare di leggere qualsiasi input. Il prompt viene visualizzato solo se l'input proviene da un terminale.

Opzione	Significato
-r	Se viene data questa opzione, il backslash non funge da carattere di escape. Il backslash viene considerato parte della riga. In particolare, una coppia backslash-newline non può essere utilizzata come continuazione di riga.
-s	Modalità silenziosa. Se l'input proviene da un terminale, i caratteri non vengono ripetuti.
-t TIMEOUT	Causa il timeout di read e restituisce un errore se non viene letta una riga completa di input entro TIMEOUT secondi. Questa opzione non ha effetto se read non sta leggendo l'input dal terminale o da una pipe.
-u FD	Leggi l'input dal descrittore di file FD .

Questo è un semplice esempio, che migliora lo script `leaptest.sh` del capitolo precedente:

```
michel ~/test> cat leaptest.sh
#!/bin/bash
# This script will test if you have given a leap year or not.

echo "Type the year that you want to check (4 digits), followed by [ENTER]:"

read year

if (( ("year" % 400) == "0" )) || (( ("year" % 4 == "0") && ("year" % 100 != "0") )); then
    echo "year is a leap year."
else
    echo "This is not a leap year."
fi

michel ~/test> leaptest.sh
Type the year that you want to check (4 digits), followed by [ENTER]:
2000
2000 is a leap year.
```

8.2.2. Richiesta di input da parte dell'utente

L'esempio seguente mostra come utilizzare i prompt per spiegare cosa deve immettere l'utente.

```
michel ~/test> cat friends.sh
#!/bin/bash

# This is a program that keeps your address book up to date.

friends="/var/tmp/michel/friends"

echo "Hello, $USER". This script will register you in Michel's friends
database."

echo -n "Enter your name and press [ENTER]: "
read name
echo -n "Enter your gender and press [ENTER]: "
read -n 1 gender
echo

grep -i "$name" "$friends"

if [ $? == 0 ]; then
    echo "You are already registered, quitting."
    exit 1
```

```

elif [ "$gender" == "m" ]; then
    echo "You are added to Michel's friends list."
    exit 1
else
    echo -n "How old are you? "
    read age
    if [ $age -lt 25 ]; then
        echo -n "Which colour of hair do you have? "
        read colour
        echo "$name $age $colour" >> "$friends"
        echo "You are added to Michel's friends list.  Thank you so much!"
    else
        echo "You are added to Michel's friends list."
        exit 1
    fi
fi
fi

michel ~/test> cp friends.sh /var/tmp; cd /var/tmp

michel ~/test> touch friends; chmod a+w friends

michel ~/test> friends.sh
Hello, michel.  Questo script vi registrerà nel database degli amici di Michel.
Enter your name and press [ENTER]: michel
Enter your gender and press [ENTER] :m
You are added to Michel's friends list.

michel ~/test> cat friends

```

Si noti che qui non viene omesso alcun output. Lo script memorizza solo le informazioni sulle persone a cui Michel è interessato, ma dirà sempre che si è stati aggiunti all'elenco, a meno che non ci sia già dentro.

Altre persone possono ora iniziare ad eseguire lo script:

```

[anny@octarine tmp]$ friends.sh
Hello, anny.  Questo script vi registrerà nel database degli amici di Michel.
Enter your name and press [ENTER]: anny
Enter your gender and press [ENTER] :f
How old are you? 22
Which colour of hair do you have? black
You are added to Michel's friends list.

```

After a while, the friends list begins to look like this:

```

tille 24 black
anny 22 black
katya 22 blonde
maria 21 black
--output omitted--

```

Naturalmente, questa situazione non è l'ideale, dal momento che tutti possono modificare (ma non eliminare) i file di Michel. È possibile risolvere questo problema utilizzando modalità di accesso speciali sul file di script, vedere [SUID e SGID](#) nella guida Introduzione a Linux.

8.2.3. Reindirizzamento e descrittori di file

8.2.3.1. In generale

Come sapete dall'utilizzo di base della shell, l'input e l'output di un comando possono essere reindirizzati prima che venga eseguito, utilizzando una notazione speciale - gli operatori di reindirizzamento - interpretati dalla shell. Il reindirizzamento può essere utilizzato anche per aprire e chiudere i file per l'ambiente di esecuzione della shell corrente.

Il reindirizzamento può verificarsi anche in uno script, in modo che possa ricevere l'input da un file, ad esempio, o inviare l'output a un file. Successivamente, l'utente può rivedere il file di output o può riutilizzarlo in un altro script come input.

L'input e l'output del file sono realizzati da handle interi che tengono traccia di tutti i file aperti per un determinato processo. Questi valori numerici sono noti come descrittori di file. I descrittori di file più noti sono *stdin*, *stdout* e *stderr*, con i numeri di descrittore di file 0, 1 e 2, rispettivamente. Questi numeri e i rispettivi dispositivi sono riservati. Bash può prendere anche porte TCP o UDP su host di rete come descrittori di file.

L'output seguente mostra come i descrittori di file riservati puntano ai dispositivi effettivi:

```
michel ~> ls -l /dev/std*
lrwxrwxrwx 1 root root 17 Oct 2 07:46 /dev/stderr -> ../proc/self/fd/2
lrwxrwxrwx 1 root root 17 Oct 2 07:46 /dev/stdin -> ../proc/self/fd/0
lrwxrwxrwx 1 root root 17 Oct 2 07:46 /dev/stdout -> ../proc/self/fd/1

michel ~> ls -l /proc/self/fd/[0-2]
lrwx----- 1 michel michel 64 Jan 23 12:11 /proc/self/fd/0 -> /dev/pts/6
lrwx----- 1 michel michel 64 Jan 23 12:11 /proc/self/fd/1 -> /dev/pts/6
lrwx----- 1 michel michel 64 Jan 23 12:11 /proc/self/fd/2 -> /dev/pts/6
```

Notare che ogni processo ha la propria vista dei file sotto */proc/self*, poiché in realtà è un link simbolico a */proc/<process_ID>*.

Si potrebbe voler controllare **info MAKEDEV** e **info proc** per ulteriori informazioni sulle subdirectory */proc* e sul modo in cui il sistema gestisce i descrittori di file standard per ogni esecuzione processi.

Quando si esegue un determinato comando, vengono eseguiti i seguenti passaggi, nell'ordine:

- Se lo standard output di un comando precedente viene reindirizzato [piped] allo standard input del comando corrente, allora */proc/<current_process_ID>/fd/0* viene aggiornato per indirizzare la stessa pipe anonima di */proc/<previous_process_ID>/fd/1*.
- Se lo standard output del comando corrente viene reindirizzato [piped] allo standard input del comando successivo, allora */proc/<current_process_ID>/fd/1* viene aggiornato per indirizzare un'altra pipe anonima.
- Il reindirizzamento per il comando corrente viene elaborato da sinistra a destra.
- Il reindirizzamento "N>&M" o "N<&M" dopo un comando ha l'effetto di creare o aggiornare il link simbolico */proc/self/fd/N* con lo stesso target del link simbolico */proc/self/fd/M*.
- I reindirizzamenti "N> file" e "N< file" hanno l'effetto di creare o aggiornare il link simbolico */proc/self/fd/N* con il file di destinazione.
- La chiusura del descrittore del file "N>&-" ha l'effetto di eliminare il link simbolico */proc/self/fd/N*.
- Solo ora viene eseguito il comando corrente.

Eseguendo uno script dalla riga di comando, non cambia molto perché il processo della shell figlia utilizzerà gli stessi descrittori di file del genitore. Quando tale genitore non è disponibile, ad esempio quando si esegue uno script utilizzando la funzione *cron*, i descrittori di file standard sono

pipe o altri file (temporanei), a meno che non venga utilizzata una qualche forma di reindirizzamento. Ciò si vede nell'esempio seguente, che mostra l'output di un semplice script **at**:

```
michel ~> date
Fri Jan 24 11:05:50 CET 2003

michel ~> at 1107
warning: commands will be executed using (in order)
a) $SHELL b) login shell c) /bin/sh
at> ls -l /proc/self/fd/ > /var/tmp/fdtest.at
at> <EOT>
job 10 at 2003-01-24 11:07

michel ~> cat /var/tmp/fdtest.at
total 0
lr-x----- 1 michel michel 64 Jan 24 11:07 0 ->
/var/spool/at/!0000c010959eb (deleted)
l-wx----- 1 michel michel 64 Jan 24 11:07 1 -> /var/tmp/fdtest.at
l-wx----- 1 michel michel 64 Jan 24 11:07 2 ->
/var/spool/at/spool/a0000c010959eb
lr-x----- 1 michel michel 64 Jan 24 11:07 3 -> /proc/21949/fd
```

E uno con **cron**:

```
michel ~> crontab -l
# DO NOT EDIT THIS FILE - edit the master and reinstall.
# (/tmp/crontab.21968 installed on Fri Jan 24 11:30:41 2003)
# (Cron version -- $Id: chap8.xml,v 1.9 2006/09/28 09:42:45 tille Exp $)
32 11 * * * ls -l /proc/self/fd/ > /var/tmp/fdtest.cron

michel ~> cat /var/tmp/fdtest.cron
total 0
lr-x----- 1 michel michel 64 Jan 24 11:32 0 -> pipe:[124440]
l-wx----- 1 michel michel 64 Jan 24 11:32 1 -> /var/tmp/fdtest.cron
l-wx----- 1 michel michel 64 Jan 24 11:32 2 -> pipe:[124441]
lr-x----- 1 michel michel 64 Jan 24 11:32 3 -> /proc/21974/fd
```

8.2.3.2. Reindirizzamento degli errori

Dagli esempi precedenti, è chiaro che si possono fornire file di input e output per uno script (vedere la [Sezione 8.2.4](#) per ulteriori informazioni), ma alcuni tendono a dimenticare gli errori di reindirizzamento - output da cui si potrebbe dipendere in seguito. Inoltre, se si è fortunati, verranno inviati degli errori e mostrate le eventuali cause di errore. Se non si è così fortunati, gli errori causeranno il fallimento dello script e non verranno catturati né inviati da nessuna parte, quindi non si potrà eseguire alcun debug.

Quando si reindirizzano gli errori, si tenga presente che l'ordine di precedenza è significativo. Per esempio, questo comando impartito in `/var/spool`

```
ls -l * 2> /var/tmp/unaccessible-in-spool
```

reindirizzerà l'output standard del comando **ls** nel file `unaccessible-in-spool` in `/var/tmp`. Il comando

```
ls -l * > /var/tmp/spoolist 2>&1
```

indirizzerà sia l'input standard che lo standard error nel file `spoolist`. Il comando

```
ls -l * 2 >& 1 > /var/tmp/spoollist
```

indirizza solo l'output standard nel file di destinazione, poiché lo standard error viene copiato nell'output standard prima che quest'ultimo venga reindirizzato.

Per comodità, gli errori vengono spesso reindirizzati verso `/dev/null`, se è sicuro che non saranno necessari. Si possono trovare centinaia di esempi negli script di startup del sistema.

Bash consente di reindirizzare sia lo standard output che lo standard error nel file il cui nome è il risultato dell'espansione di `FILE` con questo costrutto:

&> FILE

Questo è l'equivalente di `> FILE 2>&1`, utilizzato negli esempi precedenti. È anche spesso combinato con il reindirizzamento verso `/dev/null`, ad esempio quando si vuole solo che un comando venga eseguito, indipendentemente dall'output o dagli errori prodotti.

8.2.4. Input e output del File

8.2.4.1. Utilizzo di `/dev/fd`

La directory `/dev/fd` contiene voci denominate `0`, `1`, `2` e così via. Aprire il file `/dev/fd/N` equivale a duplicare il descrittore di file `N`. Se il sistema fornisce `/dev/stdin`, `/dev/stdout` e `/dev/stderr`, si noterà che sono equivalenti a `/dev/fd/0`, `/dev/fd/1` e `/dev/fd/2`, rispettivamente.

L'uso principale dei file `/dev/fd` è dalla shell. Questo meccanismo consente ai programmi che utilizzano path come argomenti di gestire lo standard input e lo standard output allo stesso modo degli altri path. Se `/dev/fd` non è disponibile nel sistema, si dovrà trovare un modo per aggirare il problema. Questo può essere fatto per esempio usando un trattino (`-`) per indicare che un programma dovrebbe leggere da una pipe. Un esempio:

```
michel ~> filter body.txt.gz | cat header.txt - footer.txt
Questo testo viene stampato all'inizio di ogni stampa e ringrazia il sysadmin
per aver creato una così grande infrastruttura di stampa.

Testo da filtrare.

Questo testo viene inserito alla fine di ogni stampa.
```

Il comando `cat` prima legge il file `header.txt`, poi il suo standard input che è l'output del comando `filter` e infine il file `footer.txt`. Il significato speciale del trattino come argomento della riga di comando per fare riferimento allo standard input o allo standard output è un equivoco che si è insinuato in molti programmi. Potrebbero anche esserci problemi quando si specifica il trattino come primo argomento, poiché potrebbe essere interpretato come un'opzione per il comando precedente. L'uso di `/dev/fd` consente l'uniformità e previene la confusione:

```
michel ~> filter body.txt | cat header.txt /dev/fd/0 footer.txt | lp
```

In questo esempio pulito, tutto l'output viene inoltre reindirizzato a `lp` per inviarlo alla stampante predefinita.

8.2.4.2. Read ed exec

8.2.4.2.1. Assegnare descrittori di file ai file

Un altro modo di vedere i descrittori di file è considerarli come un modo per assegnare un valore numerico a un file. Invece di usare il nome del file, si può usare il numero del descrittore. Il comando nativo **exec** può essere usato per sostituire la shell del processo corrente o per alterare i descrittori di file della shell corrente. Ad esempio, può essere utilizzato per assegnare un descrittore a un file. Usare

exec fdN> file

per assegnare il descrittore di file N a `file` per l'output, e

exec fdN< file

per assegnare il descrittore N a `file` per l'input. Dopo che un descrittore è stato assegnato a un file, può essere utilizzato con gli operatori di reindirizzamento della shell, come illustrato nell'esempio seguente:

```
michel ~> exec 4> result.txt

michel ~> filter body.txt | cat header.txt /dev/fd/0 footer.txt >& 4

michel ~> cat result.txt
Questo testo viene stampato all'inizio di ogni stampa e ringrazia il sysadmin
per aver creato una così grande infrastruttura di stampa.

Testo da filtrare.

Questo testo viene inserito alla fine di ogni stampa.
```

Descrittore di file 5

L'uso di questo descrittore di file potrebbe causare problemi, vedere [Guida avanzata allo scripting Bash](#), capitolo 16. Si consiglia vivamente di non utilizzarlo.

8.2.4.2.2. Read negli script

Quello che segue è un esempio che mostra come sia possibile alternare tra l'input del file e l'input della riga di comando:

```
michel ~/testdir> cat sysnotes.sh
#!/bin/bash

# Questo script crea un indice di importanti file di configurazione, li mette
insieme in
# un file di backup e consente di aggiungere dei commenti per ogni file.

CONFIG=/var/tmp/sysconfig.out
rm "$CONFIG" 2>/dev/null

echo "Output will be saved in $CONFIG."

# crea fd 7 cn lo stesso target di fd 0 (salva il "valore" di stdin)
```

```

exec 7<&0

# aggiorna fd 0 al file target /etc/passwd
exec < /etc/passwd

# Legge la prima riga di /etc/passwd
read rootpasswd

echo "Saving root account info..."
echo "Your root account info:" >> "$CONFIG"
echo $rootpasswd >> "$CONFIG"

# aggiorna fd 0 al target fd 7 (il vecchio target di fd 0); cancella fd 7
exec 0<&7 7<&-

echo -n "Enter comment or [ENTER] for no comment: "
read comment; echo $comment >> "$CONFIG"

echo "Saving hosts information..."

# prima prepara un file hosts che non contenga commenti
TEMP="/var/tmp/hosts.tmp"
cat /etc/hosts | grep -v "^#" > "$TEMP"

exec 7<&0
exec < "$TEMP"

read ip1 name1 alias1
read ip2 name2 alias2

echo "Your local host configuration:" >> "$CONFIG"

echo "$ip1 $name1 $alias1" >> "$CONFIG"
echo "$ip2 $name2 $alias2" >> "$CONFIG"

exec 0<&7 7<&-

echo -n "Enter comment or [ENTER] for no comment: "
read comment; echo $comment >> "$CONFIG"
rm "$TEMP"

michel ~/testdir> sysnotes.sh
Output will be saved in /var/tmp/sysconfig.out.
Saving root account info...
Enter comment or [ENTER] for no comment: hint for password: blue lagoon
Saving hosts information...
Enter comment or [ENTER] for no comment: in central DNS

michel ~/testdir> cat /var/tmp/sysconfig.out
Your root account info:
root:x:0:0:root:/root:/bin/bash
hint for password: blue lagoon
Your local host configuration:
127.0.0.1 localhost.localdomain localhost
192.168.42.1 tintagel.kingarthur.com tintagel
in central DNS

```

8.2.4.3. Chiusura dei descrittori di file

Poiché i processi figlio ereditano i descrittori di file aperti, è buona norma chiudere un descrittore quando non è più necessario. Questo viene fatto usando

exec fd<&-

come sintassi. Nell'esempio sopra, il descrittore 7, che è stato assegnato allo standard input, viene chiuso ogni volta che l'utente deve avere accesso al dispositivo di input standard effettivo, solitamente la tastiera.

Quello che segue è un semplice esempio che reindirizza solo lo standard error a una pipe:

```
michel ~> cat listdirs.sh
#!/bin/bash

# This script prints standard output unchanged, while standard error is
# redirected for processing by awk.

INPUTDIR="$1"

# fd 6 targets fd 1 target (console out) in current shell
exec 6>&1

# fd 1 targets pipe, fd 2 targets fd 1 target (pipe),
# fd 1 targets fd 6 target (console out), fd 6 closed, execute ls
ls "$INPUTDIR"/* 2>&1 >&6 6>&- \
    # Closes fd 6 for awk, but not for ls.

| awk 'BEGIN { FS=":" } { print "YOU HAVE NO ACCESS TO" $2 }' 6>&-

# fd 6 closed for current shell
exec 6>&-
```

8.2.4.4. Documenti *here*

Spesso, lo script potrebbe richiamare un altro programma o script che richiede un input. Il documento *here* fornisce un modo per istruire la shell a leggere l'input dalla sorgente corrente fino a quando non viene trovata una riga contenente solo la stringa cercata (senza spazi finali). Tutte le righe lette fino a quel punto vengono quindi utilizzate come input standard per un comando.

Il risultato è che non è necessario chiamare file separati; si possono usare caratteri speciali della shell e che è meglio di un mucchio di **echo**:

```
michel ~> cat startsurf.sh
#!/bin/bash

# This script provides an easy way for users to choose between browsers.

echo "These are the web browsers on this system:"

# Start here document
cat << BROWSERS
mozilla
links
lynx
konqueror
opera
netscape
BROWSERS
# End here document

echo -n "Which is your favorite? "
read browser

echo "Starting $browser, please wait..."
$browser &
```

```

michel ~> startsurf.sh
These are the web browsers on this system:
mozilla
links
lynx
konqueror
opera
netscape
Which is your favorite? opera
Starting opera, please wait...

```

Sebbene si parli di un *documento here*, dovrebbe essere un costrutto all'interno dello stesso script. Questo è un esempio che installa un pacchetto automaticamente, anche se normalmente lo si dovrebbe confermare:

```

#!/bin/bash

# This script installs packages automatically, using yum.

if [ $# -lt 1 ]; then
    echo "Usage: $0 package."
    exit 1
fi

yum install $1 << CONFIRM
Y
CONFIRM

```

E questo è come gira lo script. Quando viene richiesto il prompt con la stringa "Is this ok [y/N]", o script risponde automaticamente "y":

```

[root@picon bin]# ./install.sh tuxracer
Gathering header information file(s) from server(s)
Server: Fedora Linux 2 - i386 - core
Server: Fedora Linux 2 - i386 - freshrpms
Server: JPackage 1.5 for Fedora Core 2
Server: JPackage 1.5, generic
Server: Fedora Linux 2 - i386 - updates
Finding updated packages
Downloading needed headers
Resolving dependencies
Dependencies resolved
I will do the following:
[install: tuxracer 0.61-26.i386]
Is this ok [y/N]: EnterDownloading Packages
Running test transaction:
Test transaction complete, Success!
tuxracer 100 % done 1/1
Installed:  tuxracer 0.61-26.i386
Transaction(s) Complete

```

8.3. Sommario

In questo capitolo, abbiamo appreso come dare informazioni e come richiedere l'input all'utente. Questo di solito viene fatto usando l'accoppiata **echo/read**. Si è anche discusso di come i file possono essere utilizzati come input e output utilizzando i descrittori di file e il reindirizzamento e come ciò possa essere combinato per ottenere l'input dall'utente.

Abbiamo sottolineato l'importanza di fornire negli script una messaggistica completa agli utenti. Come sempre, quando altre persone usano gli script, è meglio dare troppe informazioni anziché troppo poche. I documenti *here* sono dei costrutti shell che consentono la creazione di liste, mantenendo le scelte degli utenti. Questo costrutto può essere utilizzato anche per eseguire attività in background, che altrimenti sarebbero interattive, senza alcun intervento.

8.4. Esercizi

Questi esercizi sono applicazioni pratiche dei costrutti discussi in questo capitolo. Quando si scrivono gli script, è possibile eseguire il test utilizzando una directory di prova che non contenga troppi dati. Si scrive ogni passaggio, poi si testa quella parte di codice, invece di scrivere tutto in una volta.

1. Scrivere uno script che richieda l'età dell'utente. Se è uguale o superiore a 16, stampa un messaggio dicendo che questo utente è autorizzato a bere alcolici. Se l'età dell'utente è inferiore a 16 anni, stampa un messaggio che dice all'utente quanti anni deve attendere prima di poter bere legalmente.

Come extra, calcolare la quantità di birra che un' persona di età superiore ai 18 anni ha bevuto statisticamente (100 litri/anno) e stampare queste informazioni.

2. Scrivere uno script che accetta un file come argomento. Usare un documento *here* che propone all'utente un paio di scelte per comprimere il file. Le scelte possibili potrebbero essere **gzip**, **bzip2**, **compress** e **zip**.
3. Scrivere uno script chiamato `homebackup` che automatizza il **tar** in modo che la persona che esegue lo script utilizzi sempre le opzioni desiderate (`cvp`) e la directory di destinazione del backup (`/var/backups`) per eseguire un backup della propria home directory. Implementare le seguenti funzionalità:
 - Verificare il numero di argomenti. Lo script dovrebbe potersi eseguire senza argomenti. Se sono presenti degli argomenti, uscire dopo aver stampato un messaggio sull'utilizzo.
 - Determinare se la directory `backup` ha spazio libero sufficiente per contenere il backup.
 - Chiedere all'utente se desidera un backup completo o incrementale. Se l'utente non dispone ancora di un file di backup completo, stampare un messaggio che informa che verrà eseguito un backup completo. In caso di backup incrementale, eseguire questa operazione solo se il backup completo non è più vecchio di una settimana.
 - Comprimere il backup utilizzando uno strumento qualsiasi di compressione. Informare l'utente che lo script sta eseguendo questa operazione, poiché potrebbe volerci del tempo, durante il quale l'utente potrebbe iniziare a preoccuparsi se sullo schermo non viene visualizzato alcun output.
 - Stampare un messaggio che informa l'utente sulla dimensione del backup compresso.

Vedere **info tar** o [Introduction to Linux](#), capitolo 9: "Preparing your data" per le informazioni basilari.

4. Scrivere uno script chiamato `simple-useradd.sh` che aggiunga un utente locale al sistema. Questo script dovrebbe:
 - Prendere solo un argomento, oppure uscire dopo aver stampato un messaggio sull'utilizzo.
 - Controllare `/etc/passwd` e decidere su quale user ID libero. Stampare un messaggio contenente tale ID.

- Creare un gruppo privato per l'utente, controllando il file `/etc/group`. Stampare un messaggio con l'ID del gruppo.
 - Raccogliere informazioni dall'utente operatore: un commento che descrive tale utente, la scelta da una lista di shell (test di accettabilità, altrimenti si esce stampando un messaggio), la data di scadenza per l'account, gruppi extra di cui il nuovo utente dovrebbe essere membro.
 - Con le informazioni ottenute, aggiungere una riga a `/etc/passwd`, `/etc/group` e `/etc/shadow`; creare la home directory dell'utente (con i permessi corretti!); aggiungere l'utente ai gruppi secondari desiderati.
 - Impostare la password per questo utente con una stringa nota predefinita.
5. Riscrivere lo script della [Sezione 7.2.1.4](#) in modo che legga l'input dall'utente invece di prenderlo dal primo argomento.
-

Capitolo 9. Lavori ripetitivi

Al termine di questo capitolo, si sarà in grado di

- Usare i cicli **for**, **while** e **until** e decidere ciclo scegliere all'occasione.
 - Utilizzare le funzioni native Bash **break** e **continue**.
 - Scrivere scripts utilizzando il comando **select**.
 - Scrivere script che accettano un numero variabile di argomenti.
-

9.1. Il ciclo for

9.1.1. Come funziona?

Il ciclo **for** è il primo dei tre costrutti di ciclo della shell. Questo ciclo consente di specificare un elenco di valori. Viene eseguito un elenco di comandi per ogni valore nella lista.

La sintassi per questo ciclo è:

for *NAME* [**in** *LIST*]; **do** *COMMANDS*; **done**

Se non è presente [**in** *LIST*], viene rimpiazzata con **in \$@** e **for** esegue i **COMMANDS** una volta per ciascun parametro posizionale impostato (cfr. la [Sezione 3.2.5](#) e [Sezione 7.2.1.2](#)).

Lo stato restituito è quello d'uscita dell'ultimo comando eseguito. Se non vengono eseguiti comandi perché *LIST* non contiene alcun elemento, lo stato di ritorno è zero.

NAME può essere qualsiasi nome di variabile, sebbene viene spesso usato *i*. *LIST* può essere qualsiasi elenco di parole, stringhe o numeri, che può essere un letterale o generato da qualsiasi comando. I **COMMANDS** da eseguire possono anche essere qualsiasi del sistema operativo, script, programmi o istruzioni della shell. La prima volta nel ciclo, *NAME* viene impostato col primo elemento della *LIST*. La seconda volta, il suo valore viene impostato sul secondo elemento nell'elenco e così via. Il ciclo termina quando *NAME* ha esaurito tutti i valori della *LIST* e in tale *LIST* non ne sono rimasti altri.

9.1.2. Esempi

9.1.2.1. Uso della sostituzione del comando per specificare gli elementi della LIST

Il primo è un esempio di riga di comando, che illustra l'uso di un ciclo **for** per eseguire una copia di backup di ogni file `.xml`. Dopo l'immissione del comando, è possibile iniziare a lavorare sui sorgenti:

```
[carol@octarine ~/articles] ls *.xml
file1.xml  file2.xml  file3.xml

[carol@octarine ~/articles] ls *.xml > list

[carol@octarine ~/articles] for i in `cat list`; do cp "$i" "$i".bak ; done

[carol@octarine ~/articles] ls *.xml*
file1.xml  file1.xml.bak  file2.xml  file2.xml.bak  file3.xml  file3.xml.bak
```

Qui si elencano i file in `/sbin` che siano solo dei semplici file di testo e possibilmente script:

```
for i in `ls /sbin`; do file /sbin/$i | grep ASCII; done
```

9.1.2.2. Uso del contenuto di una variabile per specificare gli elementi della LIST

Il seguente è uno script applicativo specifico per la conversione di file HTML, conformi a un determinato schema, in un file PHP. La conversione avviene eliminando le prime 25 righe e le ultime 21, sostituendole con due tag PHP inserendo le righe di intestazione e quelle a fine pagina:

```
[carol@octarine ~/html] cat html2php.sh
#!/bin/bash
# specific conversion script for my html files to php
LIST=$(ls *.html)
for i in "$LIST"; do
    NEWNAME=$(ls "$i" | sed -e 's/html/php/')
    cat beginfile > "$NEWNAME"
    cat "$i" | sed -e '1,25d' | tac | sed -e '1,21d' | tac >> "$NEWNAME"
    cat endfile >> "$NEWNAME"
done
```

Dato che non si esegue un conteggio delle righe, non c'è modo di conoscere il numero di riga da cui iniziare a eliminare le righe fino al raggiungimento della fine. Il problema viene risolto con **tac**, che inverte le righe di un file.

Il comando **basename**

Anziché usare **sed** per sostituire il suffisso `html` con `php`, sarebbe più pulito usare il comando **basename**. Leggerne la pagina man per ulteriori approfondimenti.

Caratteri strani

Ci si imbatte in problemi se l'elenco si espande con nomi di file contenenti spazi o altri caratteri irregolari. Un costrutto più ideale per ottenere l'elenco sarebbe quello di utilizzare la funzione di globbing della shell, in questo modo:

```
for i in $PATHNAME/*; do
```

```
commands
done
```

9.2. Il ciclo while

9.2.1. Che cos'è?

Il costrutto **while** consente l'esecuzione ripetitiva di un elenco di comandi, purché il comando che controlla il ciclo **while** venga eseguito correttamente (stato di uscita pari a zero). La sintassi è:

while CONTROL-COMMAND; do CONSEQUENT-COMMANDS; done

CONTROL-COMMAND può essere qualsiasi comando che possa uscire con uno stato di successo o errore. I **CONSEQUENT-COMMANDS** possono essere programmi, script o costrutti shell.

Non appena il **CONTROL-COMMAND** fallisce, si esce dal ciclo. In uno script, viene eseguito il comando successivo all'istruzione **done**.

Lo stato di ritorno è quello di uscita dell'ultimo comando **CONSEQUENT-COMMANDS**, o zero se non ne è stato eseguito nessuno.

9.2.2. Esempi

9.2.2.1. Semplice esempio di while

Ecco un esempio per gli impazienti:

```
#!/bin/bash

# This script opens 4 terminal windows.

i="0"

while [ $i -lt 4 ]
do
xterm &
i=$((i+1))
done
```

9.2.2.2. Cicli nidificati

L'esempio seguente è stato scritto per copiare le immagini realizzate con una webcam in una directory web. Ogni cinque minuti viene scattata una foto. Ogni ora viene creata una nuova directory, contenente le immagini per quell'ora. Ogni giorno viene creata una nuova directory contenente 24 sottodirectory. Lo script gira in background.

```
#!/bin/bash

# This script copies files from my homedirectory into the webserver directory.
# (use scp and SSH keys for a remote directory)
# A new directory is created every hour.
```

```
PICSDIR=/home/carol/pics
WEBDIR=/var/www/carol/webcam

while true; do
    DATE=`date +%Y%m%d`
    HOUR=`date +%H`
    mkdir $WEBDIR/"$DATE"

    while [ $HOUR -ne "00" ]; do
        DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
        mkdir "$DESTDIR"
        mv $PICDIR/*.jpg "$DESTDIR"/
        sleep 3600
        HOUR=`date +%H`
    done
done
```

Da notare l'uso dell'istruzione **true**. Ciò significa: continua l'esecuzione finché non si è interrotti forzatamente (con **kill** o con **Ctrl+C**).

Questo piccolo script può essere usato per i test di simulazione; genera file:

```
#!/bin/bash

# This generates a file every 5 minutes

while true; do
    touch pic-`date +%s`.jpg
    sleep 300
done
```

Da notare l'uso del comando **date** per generare tutti i tipi di nomi di file e di directory. Vedere la pagina man per approfondimenti.



Utilizzo del sistema

L'esempio precedente è a puro scopo dimostrativo. È possibile eseguire facilmente controlli regolari utilizzando la funzione di sistema *cron*. Non dimenticare di reindirizzare l'output e gli errori quando si eseguono gli script dal crontab!

9.2.2.3. Uso dell'input da tastiera per controllare il ciclo while

Questo script può essere interrotto dall'utente immettendo una sequenza **Ctrl+C**:

```
#!/bin/bash

# This script provides wisdom

FORTUNE=/usr/games/fortune

while true; do
    echo "On which topic do you want advice?"
    cat << topics
    politics
    startrek
    kernelnewbies
    sports
    bofh-excuses
    magic
```

```

love
literature
drugs
education
topics

echo
echo -n "Make your choice: "
read topic
echo
echo "Free advice on the topic of $topic: "
echo
$FORTUNE $topic
echo

done

```

Viene usato un documento *here* per presentare all'utente le possibili scelte. Inoltre, il test con **true** ripete i comandi della lista **CONSEQUENT-COMMANDS** più e più volte.

9.2.2.4. Calcolo di una media

Questo script calcola la media dell'input dell'utente, questo viene testato prima di essere elaborato: se l'input non rientra nell'intervallo, viene stampato un messaggio. Se viene digitato **q** si esce dal ciclo:

```

#!/bin/bash

# Calculate the average of a series of numbers.

SCORE="0"
AVERAGE="0"
SUM="0"
NUM="0"

while true; do

    echo -n "Enter your score [0-100%] ('q' for quit): "; read SCORE;

    if (("SCORE" < "0") || ("SCORE" > "100")); then
        echo "Be serious. Common, try again: "
    elif [ "$SCORE" == "q" ]; then
        echo "Average rating: $AVERAGE%."
        break
    else
        SUM=$((SUM + SCORE))
        NUM=$((NUM + 1))
        AVERAGE=$((SUM / NUM))
    fi

done

echo "Exiting."

```

Da notare come le variabili nelle ultime righe vengono lasciate senza virgolette per eseguire i calcoli.

9.3. Il ciclo until

9.3.1. Che cos'è?

Il ciclo **until** è molto simile al **while**, tranne per il fatto che il ciclo viene eseguito finché il **TEST-COMMAND** viene eseguito con successo. Finché questo comando fallisce, il ciclo continua. La sintassi è la stessa del ciclo **while**:

until TEST-COMMAND; do CONSEQUENT-COMMANDS; done

Quello di ritorno è lo stato di uscita dell'ultimo comando eseguito nell'elenco **COMANDI-CONSEQUENT**, oppure zero se non ne è stato eseguito nessuno. Anche qui **TEST-COMMAND** può essere qualsiasi comando che possa uscire con uno stato di successo o di errore, e **CONSEQUENT-COMMANDS** può essere qualsiasi comando UNIX, script o costruito shell.

Come spiegato in precedenza, il ";" può essere sostituito con uno o più andate a capo ovunque appaia.

9.3.2. Esempio

Uno script migliorato `picturesort.sh` (vedere la [Sezione 9.2.2.2](#)), che verifica lo spazio disponibile su disco. Se non è sufficiente, rimuove le immagini dei mesi precedenti:

```
#!/bin/bash

# This script copies files from my homedirectory into the webserver directory.
# A new directory is created every hour.
# If the pics are taking up too much space, the oldest are removed.

while true; do
    DISKFUL=$(df -h $WEBDIR | grep -v File | awk '{print $5}' | cut -d "%"
-f1 -)

    until [ $DISKFUL -ge "90" ]; do

        DATE=`date +%Y%m%d`
        HOUR=`date +%H`
        mkdir $WEBDIR/"$DATE"

        while [ $HOUR -ne "00" ]; do
            DESTDIR=$WEBDIR/"$DATE"/"$HOUR"
            mkdir "$DESTDIR"
            mv $PICDIR/*.jpg "$DESTDIR"/
            sleep 3600
            HOUR=`date +%H`
        done

        DISKFULL=$(df -h $WEBDIR | grep -v File | awk '{ print $5 }' | cut -d
%" -f1 -)
        done

        TOREMOVE=$(find $WEBDIR -type d -a -mtime +30)
        for i in $TOREMOVE; do
            rm -rf "$i";
        done

    done
```

Da notare l'inizializzazione delle variabili `hour` e `diskfull` e l'uso delle opzioni con `ls` e `date` per ottenere un elenco corretto per `in_toremove`.

9.4. Cicli e reindirizzamento dell'I/O

9.4.1. Reindirizzamento dell'input

Invece di controllare un ciclo verificando il risultato di un comando o tramite l'input dell'utente, è possibile specificare un file da cui leggere l'input per controllare il ciclo. In questi casi, spesso, **read** è il comando del controllo. L'esecuzione del ciclo continua finché ci sono righe di input che lo alimentano. Non appena sono state lette tutte le righe di input, si esce dal loop.

Poiché il costrutto del ciclo è considerato una struttura di comando (come un **while TEST-COMMAND; do CONSEQUENT-COMMANDS; done**), il reindirizzamento dovrebbe avvenire dopo il comando **done**, in modo che sia conforme al formato

```
command < file
```

Questo tipo di reindirizzamento funziona anche con altri tipi di loop.

9.4.2. Reindirizzamento dell'output

Nell'esempio seguente, l'output del comando **find** viene usato come input per il comando **read** per controllare un ciclo **while**:

```
[carol@octarine ~/testdir] cat archiveoldstuff.sh
#!/bin/bash

# This script creates a subdirectory in the current directory, to which old
# files are moved.
# Might be something for cron (if slightly adapted) to execute weekly or
# monthly.

ARCHIVENR=`date +%Y%m%d`
DESTDIR="$PWD/archive-$ARCHIVENR"

mkdir "$DESTDIR"

# using quotes to catch file names containing spaces, using read -d for more
# fool-proof usage:
find "$PWD" -type f -a -mtime +5 | while read -d $'\000' file
do
gzip "$file"; mv "$file".gz "$DESTDIR"
echo "$file archived"
done
```

I file vengono compressi prima di essere spostati nella directory di archiviazione.

9.5. Break e continue

9.5.1. Il break nativo

L'istruzione **break** viene usata per interrompere il ciclo corrente prima della sua normale conclusione. Questo viene fatto quando non si sa in anticipo quante volte il ciclo dovrà essere eseguito, ad esempio perché dipende dall'input dell'utente.

L'esempio seguente mostra un ciclo **while** che può essere interrotto. Questa è una versione leggermente migliorata dello script `wisdom.sh` della [Sezione 9.2.2.3](#).

```
#!/bin/bash

# This script provides wisdom
# You can now exit in a decent way.

FORTUNE=/usr/games/fortune

while true; do
echo "On which topic do you want advice?"
echo "1.  politics"
echo "2.  startrek"
echo "3.  kernelnewbies"
echo "4.  sports"
echo "5.  bofh-excuses"
echo "6.  magic"
echo "7.  love"
echo "8.  literature"
echo "9.  drugs"
echo "10. education"
echo

echo -n "Enter your choice, or 0 for exit: "
read choice
echo

case $choice in
1)
    $FORTUNE politics
    ;;
2)
    $FORTUNE startrek
    ;;
3)
    $FORTUNE kernelnewbies
    ;;
4)
    echo "Sports are a waste of time, energy and money."
    echo "Go back to your keyboard."
    echo -e "\t\t\t\t -- \"Unhealthy is my middle name\" Soggie."
    ;;
5)
    $FORTUNE bofh-excuses
    ;;
6)
    $FORTUNE magic
    ;;
7)
    $FORTUNE love
    ;;
8)
    $FORTUNE literature
```

```

;;
9)
$FORTUNE drugs
;;
10)
$FORTUNE education
;;
0)
echo "OK, see you!"
break
;;
*)
echo "That is not a valid choice, try a number from 0 to 10."
;;
esac
done

```

si tenga presente che **break** esce dal ciclo, non dallo script. Questo può essere dimostrato aggiungendo un comando **echo** alla fine dello script. Tale **echo** verrà eseguito anche con un input che causa l'esecuzione di un **break** (quando l'utente digita "0").

Nei cicli nidificati, **break** consente di indicare il ciclo da cui uscire. Per approfondimenti, consultare le pagine **info**.

9.5.2. Il continue nativo

L'istruzione **continue** riprende l'iterazione di un ciclo **for**, **while**, **until** o **select**.

In un ciclo **for**, la variabile di controllo assume il valore dell'elemento successivo nell'elenco. In un ciclo **while** o **until**, però, l'esecuzione riprende con **TEST-COMMAND** all'inizio del ciclo.

9.5.3. Esempi

Nell'esempio seguente, i nomi dei file vengono convertiti in minuscolo. Se non è necessaria alcuna conversione, un'istruzione **continue** riavvia l'esecuzione del ciclo. Questi comandi non consumano molte risorse di sistema e, molto probabilmente, problemi simili possono essere risolti usando **sed** e **awk**. Tuttavia, è utile conoscere questo tipo di costruzione quando si eseguono lavori pesanti, che potrebbero non essere nemmeno necessari inserendo i test nelle posizioni corrette in uno script, risparmiando risorse di sistema.

```

[carol@octarine ~/test] cat tolower.sh
#!/bin/bash

# This script converts all file names containing upper case characters into
file# names containing only lower cases.

LIST="$(ls)"

for name in "$LIST"; do

if [[ "$name" != *[:upper:]* ]]; then
continue
fi

ORIG="$name"

```

```
NEW=`echo $name | tr 'A-Z' 'a-z'`

mv "$ORIG" "$NEW"
echo "new name for $ORIG is $NEW"
done
```

Questo script presenta almeno uno svantaggio: sovrascrive i file esistenti. L'opzione `noclobber` per Bash è utile solo quando si verifica il reindirizzamento. L'opzione `-b` del comando `mv` fornisce maggiore sicurezza, ma è solo in caso di una sovrascrittura accidentale, come mostrato in questo test:

```
[carol@octarine ~/test] rm *

[carol@octarine ~/test] touch test Test TEST

[carol@octarine ~/test] bash -x tolower.sh
++ ls
+ LIST=test
Test
TEST
+ [[ test != *[:upper:]* ]]
+ continue
+ [[ Test != *[:upper:]* ]]
+ ORIG=Test
++ echo Test
++ tr A-Z a-z
+ NEW=test
+ mv -b Test test
+ echo 'new name for Test is test'
new name for Test is test
+ [[ TEST != *[:upper:]* ]]
+ ORIG=TEST
++ echo TEST
++ tr A-Z a-z
+ NEW=test
+ mv -b TEST test
+ echo 'new name for TEST is test'
new name for TEST is test

[carol@octarine ~/test] ls -a
./ ../ test test~
```

Il `tr` fa parte del pacchetto *textutils*; può eseguire tutti i tipi di trasformazione dei caratteri.

9.6. Creazione di menù col select nativo

9.6.1. In generale

9.6.1.1. Uso di select

Il costrutto **select** facilita la generazione di menù. La sintassi è abbastanza simile al ciclo **for**:

```
select WORD [in LIST]; do RESPECTIVE-COMMANDS; done
```

`LIST` espandendosi, genera un elenco di elementi. L'espansione viene stampata nello "standard error"; ogni voce è preceduta da un numero. Se non è presente **in LIST**, vengono stampati i parametri posizionali, come se fosse stato specificato **in \$@**. `LIST` viene stampato una sola volta.

Dopo aver scritto tutti gli elementi, viene stampato il prompt `PS3` e viene letta una riga dallo standard input. Se questa riga è composta da un numero corrispondente a uno degli elementi, il valore di `WORD` viene impostato col nome di quell'elemento. Se la riga è vuota, vengono riscritti gli elementi e il prompt `PS3`. Se viene letto un carattere *EOF* (End Of File), il ciclo termina. Poiché la maggior parte degli utenti non ha la più pallida idea di quale combinazione di tasti venga utilizzata per la sequenza *EOF*, è più agevole avere un comando per il **break** tra gli elementi. Qualsiasi altro valore della riga letta imposterà `WORD` come stringa nulla.

La riga letta viene salvata nella variabile `REPLY`.

Dopo ogni selezione vengono eseguiti i **RESPECTIVE-COMMANDS** fin quando non si legge il numero che rappresenta il **break**. Con questo si esce dal ciclo.

9.6.1.2. Esempi

Questo è un semplicissimo esempio, ma, come si può vedere, non è molto intuitivo:

```
[carol@octarine testdir] cat private.sh
#!/bin/bash

echo "This script can make any of the files in this directory private."
echo "Enter the number of the file you want to protect:"

select FILENAME in *;
do
    echo "You picked $FILENAME ($REPLY), it is now only accessible to you."
    chmod go-rwx "$FILENAME"
done

[carol@octarine testdir] ./private.sh
This script can make any of the files in this directory private.
Enter the number of the file you want to protect:
1) archive-20030129
2) bash
3) private.sh
#? 1
You picked archive-20030129 (1)
#?
```

Impostando il prompt `PS3` ed aggiungendo la possibilità di uscire ne migliora l'utilizzo:

```
#!/bin/bash

echo "This script can make any of the files in this directory private."
echo "Enter the number of the file you want to protect:"

PS3="Your choice: "
QUIT="QUIT THIS PROGRAM - I feel safe now."
touch "$QUIT"

select FILENAME in *;
do
    case $FILENAME in
        "$QUIT")
            echo "Exiting."
            break
        ;;
        *)
    esac
done
```

```

        echo "You picked $FILENAME ($REPLY) "
        chmod go-rwx "$FILENAME"
        ;;
    esac
done
rm "$QUIT"

```

9.6.2. I submenu

Qualsiasi istruzione all'interno di un costrutto **select** può essere un altro ciclo **select**, consentendo la creazione di uno o più sotto-menù all'interno di un menù.

Per default, la variabile `PS3` non viene modificata entrando in un ciclo **select** nidificato. Volendo un prompt diverso nel sotto-menù, lo si imposti nei tempi giusti.

9.7. Lo shift nativo

9.7.1. Che cosa fa?

Il comando **shift** è uno di quelli nativi della Bourne shell forniti con Bash. Questo comando accetta un argomento, un numero. I parametri posizionali vengono spostati a sinistra di N volte. I parametri posizionali dal $N+1$ a $\$#$ vengono rinominati coi nomi delle variabili da $\$1$ a $\$# - N + 1$.

Si supponga che un comando si aspetti 10 argomenti, e N è 4, allora $\$4$ diventa $\$1$, $\$5$ diventa $\$2$ e così via. $\$10$ diventa $\$7$ e gli originali $\$1$, $\$2$ e $\$3$ vengono buttati via.

Se N è zero o maggiore di $\$#$, i parametri posizionali non vengono modificati (il numero totale di argomenti, vedere la [Sezione 7.2.1.2](#)) e il comando non ha effetto. Se N non è presente, si assume che sia 1. Lo stato restituito è zero a meno che N non sia maggiore di $\$#$ o inferiore a zero; altrimenti è diverso da zero.

9.7.2. Esempi

Un'istruzione **shift** viene in generalmente utilizzata quando non è noto a priori il numero di argomenti di un comando, ad esempio quando gli utenti possono fornire tutti gli argomenti che desiderano. In questi casi, gli argomenti vengono solitamente elaborati in un ciclo **while** loop con **(($\$#$))** come condizione di test. Tale condizione è vera fin quando il numero di argomenti è maggiore di zero. La variabile $\$1$ e il comando **shift** elaborano qualsiasi argomento. Il numero di argomenti si riduce ad ogni esecuzione di **shift** fino a diventare zero, dopodiché il ciclo **while** termina.

L'esempio seguente, `cleanup.sh`, usa il comando **shift** per elaborare ogni file nell'elenco generato da **find**:

```

#!/bin/bash

# This script can clean up files that were last accessed over 365 days ago.

USAGE="Usage: $0 dir1 dir2 dir3 ... dirN"

if [ "$#" == "0" ]; then
    echo "$USAGE"

```

```

        exit 1
fi

while (( "$#" )); do

if [[ $(ls "$1") == "" ]]; then
    echo "Empty directory, nothing to be done."
else
    find "$1" -type f -a -atime +365 -exec rm -i {} \;
fi

shift

done

```

-exec vs. xargs

Il comando **find**, riportato sopra, è sostituibile con quanto segue:

find options | xargs [commands_to_execute_on_found_files]

Il comando **xargs** compila ed esegue righe di comando dallo standard input. Ha il vantaggio che la riga di comando viene riempita fino al raggiungimento del limite di sistema. Solo allora verrà chiamato il comando da eseguire, nell'esempio sopra questo sarebbe **rm**. Se ci sono più argomenti, verrà utilizzata una nuova riga di comando, finché non sarà più piena o finché non ci saranno più argomenti. La stessa cosa utilizzando le chiamate **find -exec** sulla riga di comando da eseguire su ciascun file trovato. Pertanto, l'utilizzo di **xargs** velocizza notevolmente gli script e le prestazioni della macchina.

Nel prossimo esempio, è stato modificato lo script dalla [Sezione 8.2.4.4](#) in modo che accetti più pacchetti da installare contemporaneamente:

```

#!/bin/bash
if [ $# -lt 1 ]; then
    echo "Usage: $0 package(s)"
    exit 1
fi
while (($#)); do
    yum install "$1" << CONFIRM
Y
CONFIRM
shift
done

```

9.8. Sommario

In questo capitolo abbiamo discusso di come i comandi ripetitivi possono essere inclusi in un ciclo. I cicli più comuni vengono creati utilizzando le istruzioni **for**, **while** o **until**, o una combinazione di essi. Il ciclo **for** esegue un task un determinato numero di volte. Se non si sanno quante volte si deve eseguire un comando, allora o si usa il ciclo **until** o **while** per specificare quando deve finire il ciclo.

I cicli possono essere interrotti o reiterati coi comandi **break** e **continue**.

Si può usare un file come input per un ciclo utilizzando l'operatore di reindirizzamento dell'input, i cicli possono anche leggere l'output dei comandi inseriti nel loop utilizzando una pipe.

Il costrutto **select** viene usato per stampare menù negli script interattivi. È possibile ciclare sugli argomenti della riga di comando di uno script col comando **shift**.

9.9. Esercizi

Da ricordare: quando si creano script, si deve lavorare per passi verificandone ciascuno prima di inserirlo nello script.

1. Creare uno script che prenderà una (ricorsiva) di file in `/etc` in modo che un amministratore di sistema principiante li possa modificare senza timore.
2. Scrivere uno script che prenda un solo argomento, il nome di una directory. Se il numero di argomenti è maggiore o inferiore a uno, stampare un messaggio sull'uso. Se l'argomento non è una directory, stampare un altro messaggio. Per la directory specificata, stampare i cinque file più grandi e i cinque più recentemente modificati.
3. Spiegare perché è così importante mettere le variabili tra virgolette doppie nell'esempio della [Sezione 9.4.2](#)?
4. Scrivere uno script simile a quello della [Sezione 9.5.1](#), ma pensare ad un modo per uscire dopo che l'utente ha eseguito 3 cicli.
5. Ideare una soluzione migliore del `move -b` per lo script della [Sezione 9.5.3](#) per evitare la sovrascrittura dei file esistenti. Per esempio, controllando se il file esiste già. Non fare lavoro inutile!
6. Riscrivere lo script `whichdaemon.sh` della [Sezione 7.2.4](#), in modo che:
 - Stampi un elenco di server da controllare, come Apache, il server SSH, il demone NTP, un demone dei nomi, un demone di gestione dell'alimentazione e così via.
 - Per ciascuna scelta che l'utente può fare, stampare delle informazioni sensibili, come il nome del server web, NTP le informazioni del trace, e così via.
 - Facoltativamente, dare possibilità agli utenti di controllare server diversi da quelli elencati. In questi casi, verificare che almeno il processo specificato sia in esecuzione.
 - Esaminare lo script della [Sezione 9.2.2.4](#). Notare come viene elaborato il carattere di input diverso da `q`. Riscrivere lo script in modo che stampi un messaggio se i caratteri vengono forniti come input.

Capitolo 10. Approfondimenti sulle variabili

In questo capitolo, discuteremo l'uso avanzato di variabili e argomenti. Al termine, si sarà in grado di

- Dichiarare e utilizzare un array di variabili
 - Specificare il tipo di variabile che si vuol usare
 - Rendi le variabili di sola lettura
 - Usare **set** per assegnare un valore a una variabile
-

10.1. Tipi di variabili

10.1.1. Assegnazione generica dei valori

Come si è visto, Bash comprende molti tipi di variabili e parametri. Finora, non ci è preoccupati molto del tipo di variabili assegnate, ma le variabili potrebbero contenere qualsiasi valore. Un semplice esempio di riga di comando lo dimostra:

```
[bob in ~] VARIABLE=12

[bob in ~] echo $VARIABLE
12

[bob in ~] VARIABLE=string

[bob in ~] echo $VARIABLE
string
```

Ci sono casi in cui si vuol evitare questo tipo di comportamento, ad esempio quando si gestiscono numeri di telefono e altri numeri. Oltre a numeri interi e le variabili, si potrebbe voler specificare una variabile che sia una costante. Questo viene solitamente fatto all'inizio di uno script, quando si dichiara il valore della costante. Dopodiché, ci sono solo riferimenti al nome della variabile costante, in modo che se la costante dovrà essere modificata, lo si dovrà fare in un solo punto. Una variabile può anche essere una serie di variabili di qualsiasi tipo, un cosiddetto *array* di variabili (VAR0VAR1, VAR2, ... VARN).

10.1.2. Uso del declare nativo

Con un'istruzione **declare** si può limitare l'assegnazione dei valori a delle variabili.

La sintassi per **declare** è la seguente:

declare **OPTION**(s) **VARIABLE=value**

Vengono utilizzate le seguenti opzioni per determinare il tipo di dato che la variabile può contenere e per assegnarle gli attributi:

Tabella 10-1. Opzioni per la declare nativa

Opzione	Significato
-a	La variabile è un array.
-f	Ammessi solo nomi di funzioni.
-i	La variabile va trattata come un intero; la valutazione aritmetica viene eseguita quando alla variabile viene assegnato un valore (vedere la Sezione 3.4.6).
-p	Visualizza gli attributi e i valori di ciascuna variabile. Quando viene utilizzato -p, le altre opzioni vengono ignorate.
-r	Rende la variabile a sola lettura. A queste variabili quindi non possono essere assegnati valori da successive istruzioni di assegnazione, né possono essere annullate.
-t	Assegna a ciascuna variabile l'attributo <i>trace</i> .

Opzione	Significato
-x	Contrassegna ciascuna variabile per l'export in comandi successivi tramite l'environment.

Utilizzando + anziché -, disattiva l'attributo. Quando viene utilizzato in una funzione, **declare** crea variabili locali.

L'esempio seguente mostra come l'assegnazione di un tipo ad una variabile influenza il valore.

```
[bob in ~] declare -i VARIABLE=12

[bob in ~] VARIABLE=string

[bob in ~] echo $VARIABLE
0

[bob in ~] declare -p VARIABLE
declare -i VARIABLE="0"
```

Da notare che Bash ha un'opzione per dichiarare un valore numerico, ma nessuna per dichiarare valori di tipo stringa. Questo perché, per default, se non vengono fornite specifiche, una variabile può contenere qualsiasi tipo di dato:

```
[bob in ~] OTHERVAR=blah

[bob in ~] declare -p OTHERVAR
declare -- OTHERVAR="blah"
```

Non appena si limita l'assegnazione di valori a una variabile, questa potrà contenere solo quel tipo di dati. Le possibili restrizioni sono interi, costanti o array.

Consultare le pagine informative di Bash per gli stati restituiti.

10.1.3. Costanti

In Bash, le costanti vengono create rendendo una variabile a sola lettura. Il **readonly** nativo contrassegna ogni variabile specificata come imm modificabile. La sintassi è:

readonly **OPTION** **VARIABLE** (**s**)

I valori di queste variabili non potranno più essere modificati con una successiva assegnazione. Se c'è l'opzione -f, ogni variabile farà riferimento a una funzione shell; vedere il [Capitolo 11](#). Se c'è -a, ogni variabile farà riferimento a un array di variabili. Se non viene dato alcun argomento, o se c'è -p, verrà visualizzato un elenco con tutte le variabili a sola lettura. Con l'opzione -p, l'output può essere riciclato come input.

Lo stato restituito è zero, a meno che non sia stata specificata un'opzione non valida, una delle variabili o funzioni non esiste o sia se sia stato fornito -f per un nome di variabile invece che per un nome di funzione.

```
[bob in ~] readonly TUX=penguinpower

[bob in ~] TUX=Mickeysoft
```

```
bash: TUX: readonly variable
```

10.2. Variabili array

10.2.1. Creazione di array

Un array è una variabile che contiene più valori. Qualsiasi variabile può essere utilizzata come array. Non esiste un limite massimo alla dimensione di un array, né alcun requisito che le variabili membro debbano essere indicizzate o assegnate in modo contiguo. Gli array sono a base zero: il primo elemento è indicizzato con il numero 0.

La dichiarazione indiretta viene eseguita utilizzando la seguente sintassi per dichiarare una variabile:

ARRAY [INDEXNR]=value

INDEXNR viene considerato come un'espressione aritmetica che deve restituire un numero positivo.

La dichiarazione esplicita di un array viene eseguita utilizzando **declare**:

declare -a ARRAYNAME

È accettata anche una dichiarazione con un numero di indice, ma tale numero verrà ignorato. Gli attributi dell'array possono essere specificati utilizzando **declare** o **readonly**. Gli attributi si applicano a tutte le variabili nell'array; non si possono avere array misti.

Gli array possono anche essere creati utilizzando assegnazioni composte con questo formato:

ARRAY=(value1 value2 ... valueN)

Ogni valore, quindi ha il formato *[indexnumber]=string*. Il numero di indice è facoltativo. Se viene fornito, gli viene assegnato quell'indice; in caso contrario l'indice dell'elemento assegnato è il numero dell'ultimo indice assegnato, più uno. Questo formato è accettato anche da **declare**. Se non vengono forniti numeri di indice, l'indicizzazione inizia da zero.

L'aggiunta di membri mancanti o aggiuntivi in un array si esegue utilizzando la sintassi:

ARRAYNAME [indexnumber]=value

Da ricordare che **read** ha l'opzione **-a**, che consente di leggere e assegnare valori per le variabili membro di un array.

10.2.2. Dereferenziazione delle variabili in un array

Per fare riferimento al contenuto di un elemento in una matrice, si utilizzano le parentesi graffe. Ciò è necessario, come si vede dall'esempio seguente, per evitare che la shell interpreti gli operatori di espansione. Se il numero dell'indice è **@** o *****, si fa riferimento a tutti i membri dell'array.

```
[bob in ~] ARRAY=(one two three)
```

```
[bob in ~] echo ${ARRAY[*]}
one two three

[bob in ~] echo $ARRAY[*]
one[*]

[bob in ~] echo ${ARRAY[2]}
three

[bob in ~] ARRAY[3]=four

[bob in ~] echo ${ARRAY[*]}
one two three four
```

Fare riferimento al contenuto di una variabile membro di un array senza fornire un numero di indice equivale a fare riferimento al contenuto del primo elemento, quello a cui si fa riferimento con il numero di indice zero.

10.2.3. Cancellazione di variabili array

Per distruggere gli array o le variabili membro di un array, si usa **unset**:

```
[bob in ~] unset ARRAY[1]

[bob in ~] echo ${ARRAY[*]}
one three four

[bob in ~] unset ARRAY

[bob in ~] echo ${ARRAY[*]}
<--no output-->
```

10.2.4. Esempi di array

È difficile trovare esempi pratici dell'uso degli array. Per esempio, si trovano molti script che in realtà non fanno nulla sul sistema ma che usano gli array per calcolare le serie matematiche. E questo sarebbe uno degli esempi più interessanti... la maggior parte degli script mostra semplicemente cosa si può fare con un array in modo semplice e teorico.

La ragione di questa ottusità è che gli array sono strutture piuttosto complesse. Si scoprirà che la maggior parte degli esempi pratici per i quali è possibile utilizzare gli array sono già implementati sul sistema utilizzando gli array, tuttavia a un livello più basso, nel linguaggio di programmazione C in cui sono scritti la maggior parte dei comandi UNIX. Un buon esempio è il comando Bash **history**. Chi è interessato, può controllare la directory `built-ins` nell'albero dei sorgenti di Bash e dare un'occhiata a `fc.def`, che viene elaborato durante la compilazione dei comandi nativi [built-in].

Un altro motivo per cui è difficile trovare buoni esempi è che non tutte le shell supportano gli array, interrompendo, quindi, la compatibilità.

Dopo molti giorni di ricerca, ho finalmente trovato questo esempio funzionante presso un provider Internet. Distribuisce i file di configurazione del server web Apache sugli host in una web farm:

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

```

if [ $(whoami) != 'root' ]; then
    echo "Must be root to run $0"
    exit 1;
fi
if [ -z $1 ]; then
    echo "Usage: $0 </path/to/httpd.conf>"
    exit 1
fi

httpd_conf_new=$1
httpd_conf_path="/usr/local/apache/conf"
login=htuser

farm_hosts=(web03 web04 web05 web06 web07)

for i in ${farm_hosts[@]}; do
    su $login -c "scp $httpd_conf_new ${i}:${httpd_conf_path}"
    su $login -c "ssh $i sudo /usr/local/apache/bin/apachectl graceful"
done
exit 0

```

I primi due test vengono eseguiti per verificare se è l'utente corretto che sta eseguendo lo script e con gli argomenti corretti. I nomi degli host da configurare sono elencati nell'array `farm_hosts`. A tutti questi host viene poi fornito il file di configurazione di Apache, dopodiché il demone viene riavviato. Si noti l'uso dei comandi della suite Secure Shell, che crittografano le connessioni agli host remoti.

Grazie, Eugene e collega, per questo contributo.

Dan Richter ha contribuito con il seguente esempio. Questo è il problema che ha dovuto affrontare:

"...Nella mia azienda, abbiamo delle demo sul nostro sito web e ogni settimana qualcuno deve testarle tutte. Per questo c'è un job di cron che riempie un array con i possibili candidati, usa **date +%W** per trovare la settimana dell'anno, ed esegue un'operazione modulo per trovare l'indice corretto. La persona fortunata verrà avvisata via e-mail".

E questo è stato il suo modo per risolverlo:

```

#!/bin/bash
# This is get-tester-address.sh
#
# First, we test whether bash supports arrays.
# (Support for arrays was only added recently.)
#
whotest[0]='test' || (echo 'Failure: arrays not supported in this version of
bash.' && exit 2)

#
# Our list of candidates. (Feel free to add or
# remove candidates.)
#
wholist=(
    'Bob Smith <bob@example.com>'
    'Jane L. Williams <jane@example.com>'
    'Eric S. Raymond <esr@example.com>'
    'Larry Wall <wall@example.com>'
    'Linus Torvalds <linus@example.com>'
)
#
# Count the number of possible testers.

```

```
# (Loop until we find an empty string.)
#
count=0
while [ "x${wholist[count]}" != "x" ]
do
    count=$(( $count + 1 ))
done

#
# Now we calculate whose turn it is.
#
week=`date '+%W'`      # The week of the year (0..53).
week=${week#0}        # Remove possible leading zero.

let "index = $week % $count" # week modulo count = the lucky person

email=${wholist[index]}    # Get the lucky person's e-mail address.

echo $email                # Output the person's e-mail address.
```

Questo script viene poi utilizzato in altri script, come il seguente, che utilizza un documento *here*:

```
email=`get-tester-address.sh` # Find who to e-mail.
hostname=`hostname`          # This machine's name.

#
# Send e-mail to the right person.
#
mail $email -s '[Demo Testing]' <<EOF
The lucky tester this week is: $email

Reminder: the list of demos is here:
    http://web.example.com:8080/DemoSites

(This e-mail was generated by $0 on ${hostname}.)
EOF
```

10.3. Operazioni sulle variabili

10.3.1. Aritmetica delle variabili

Se ne è già discusso nella [Sezione 3.4.6](#).

10.3.2. Lunghezza di una variabile

Con la sintassi `${#VAR}` si calca il numero di caratteri in una variabile. Se `VAR` è `"*"` o `"@"`, tale valore verrà sostituito col numero di parametri posizionali o il numero degli elementi in un array generale. Questo è mostrato nell'esempio seguente:

```
[bob in ~] echo $SHELL
/bin/bash

[bob in ~] echo ${#SHELL}
9

[bob in ~] ARRAY=(one two three)
```

```
[bob in ~] echo ${#ARRAY}
3
```

10.3.3. Trasformazione di variabili

10.3.3.1. Sostituzione

`${VAR:-WORD}`

Se `VAR` non è definito o è nullo, viene sostituita l'espansione di `WORD`; altrimenti si sostituisce il valore di `VAR`:

```
[bob in ~] echo ${TEST:-test}
test

[bob in ~] echo $TEST

[bob in ~] export TEST=a_string

[bob in ~] echo ${TEST:-test}
a_string

[bob in ~] echo ${TEST2:-$TEST}
a_string
```

Questo modulo viene spesso utilizzato nei test condizionali, ad esempio:

```
[ -z "${COLUMNS:-}" ] && COLUMNS=80
```

è una notazione abbreviata di

```
if [ -z "${COLUMNS:-}" ]; then
    COLUMNS=80
fi
```

Vedere la [Sezione 7.1.2.3](#) per ulteriori informazioni su questo tipo di condizioni di test.

Se il trattino (-) viene sostituito con il segno di uguale (=), il valore viene assegnato al parametro se non esiste:

```
[bob in ~] echo $TEST2

[bob in ~] echo ${TEST2:=$TEST}
a_string

[bob in ~] echo $TEST2
a_string
```

La seguente sintassi verifica l'esistenza di una variabile. Se non è impostato, l'espansione di `WORD` viene stampata su standard out e le shell non interattive vengono chiuse. Una dimostrazione:

```
[bob in ~] cat vartest.sh
#!/bin/bash

# This script tests whether a variable is set.  If not,
```



```
# it exits printing a message.

echo ${TESTVAR:? "There's so much I still wanted to do..."}
echo "TESTVAR is set, we can proceed."

[ bob in testdir ] ./vartest.sh
./vartest.sh: line 6: TESTVAR: There's so much I still wanted to do...

[ bob in testdir ] export TESTVAR=present

[ bob in testdir ] ./vartest.sh
present
TESTVAR is set, we can proceed.
```

L'utilizzo di "+" al posto del punto esclamativo imposta la variabile con l'espansione di *WORD*; se non esiste, non succede nulla.

10.3.3.2. Rimozione di sotto-stringhe

Per rimuovere un numero di caratteri pari a *OFFSET* da una variabile, si usa la sintassi:

`${VAR:OFFSET:LENGTH}`

Il parametro *LENGTH* definisce quanti caratteri mantenere, a partire dal primo carattere dopo il punto di offset. Se si omette *LENGTH*, viene preso il resto del contenuto della variabile:

```
[ bob in ~ ] export STRING="thisisaverylongname"

[ bob in ~ ] echo ${STRING:4}
isaverylongname

[ bob in ~ ] echo ${STRING:6:5}
avery
```

`${VAR#WORD}`

and

`${VAR##WORD}`

Questi costrutti vengono utilizzati per eliminare il "pattern matching" dell'espansione di *WORD* in *VAR*. *WORD* viene espansa per produrre un pattern proprio come nell'espansione del nome file. Se il pattern corrisponde all'inizio del valore espanso di *VAR*, il risultato dell'espansione è il valore espanso di *VAR* con il pattern corrispondente più breve ("#") quello più lungo (indicato con "##").

Se *VAR* è * o @, l'operazione di rimozione del pattern viene applicata a turno a ciascun parametro posizionale e l'espansione è l'elenco risultante.

Se *VAR* è un array iscritto con "*" o "@", l'operazione di rimozione del pattern viene applicata a turno a ciascun membro dell'array e l'espansione è l'elenco risultante. Questo viene mostrato negli esempi seguenti:

```
[ bob in ~ ] echo ${ARRAY[*]}
one two one three one four
```

```
[bob in ~] echo ${ARRAY[*]#one}
two three four

[bob in ~] echo ${ARRAY[*]#t}
one wo one hree one four

[bob in ~] echo ${ARRAY[*]#t*}
one wo one hree one four

[bob in ~] echo ${ARRAY[*]##t*}
one one one four
```

L'effetto opposto si ottiene utilizzando "%" e "%%", come nell'esempio seguente. *WORD* deve corrispondere a una parte finale della stringa:

```
[bob in ~] echo $STRING
thisisaverylongname

[bob in ~] echo ${STRING%name}
thisisaverylong
```

10.3.3.3. Sostituzione di parti di nomi di variabili

Questo viene fatto usando

`${VAR/PATTERN/STRING}`

o

`${VAR//PATTERN/STRING}`

come sintassi. La prima forma sostituisce solo la prima corrispondenza, la seconda sostituisce tutte le corrispondenze di *PATTERN* con *STRING*:

```
[bob in ~] echo ${STRING/name/string}
thisisaverylongstring
```

Maggiori informazioni possono essere trovate nelle pagine informative di Bash.

10.4. Sommario

Normalmente, una variabile può contenere qualsiasi tipo di dato, a meno che non sia stata dichiarata in modo esplicito. Le variabili costanti vengono impostate utilizzando il comando **readonly**.

Un array contiene un insieme di variabili. Se viene dichiarato un tipo di dato, tutti gli elementi nell'array verranno impostati per contenere solo questo tipo di dato.

Le funzionalità di Bash consentono la sostituzione e la trasformazione delle variabili "al volo". Le operazioni standard includono il calcolo della lunghezza di una variabile, l'aritmetica sulle variabili, la sostituzione del contenuto della variabile e la sostituzione di parte del contenuto.

10.5. Esercizi

Ecco alcuni stimoli per il cervello:

1. Scrivere uno script che faccia quanto segue:
 - Visualizza il nome dello script in esecuzione.
 - Visualizza il primo, il terzo e il decimo argomento assegnati allo script.
 - Visualizza il numero totale di argomenti passati allo script.
 - Se ci sono più di tre parametri posizionali, usa **shift** per spostare tutti i valori di 3 posizioni a sinistra.
 - Stampa tutti i valori degli argomenti rimanenti.
 - Stampa il numero di argomenti.

Provare con zero, uno, tre e più di dieci argomenti.

2. Scrivere uno script che implementi un semplice browser web (in modalità testo), utilizzando **wget** e **links -dump** per mostrare le pagine HTML all'utente. L'utente avrà 3 scelte: inserire un URL, inserire **b** per tornare indietro e **q** per uscire. Gli ultimi 10 URL immessi dall'utente vengono archiviati in un array, dal quale l'utente può ripristinare l'URL utilizzando la funzionalità di *back*.

Capitolo 11. Funzioni

In questo capitolo parleremo

- Cosa sono le funzioni
- Creazione e visualizzazione di funzioni dalla riga di comando
- Funzioni negli script
- Passare argomenti alle funzioni
- Quando usare le funzioni

11.1. Introduzione

11.1.1. Cosa sono le funzioni?

Le funzioni della shell, o *routine*, sono un modo per raggruppare i comandi per l'esecuzione successiva, usando un unico nome per questo gruppo. Il nome della routine deve essere univoco all'interno della shell o dello script. Tutti i comandi che compongono una funzione vengono eseguiti come normali comandi. Quando si richiama una funzione come nome di comando semplice, viene eseguito l'elenco dei comandi associati a quel nome di funzione. Una funzione viene eseguita all'interno della shell in cui è stata dichiarata: nessun nuovo processo viene creato per interpretare i comandi.

Speciali comandi nativi si trovano prima delle funzioni della shell durante la ricerca dei comandi. I comandi speciali nativi sono: **break**, **:**, **..**, **continue**, **eval**, **exec**, **exit**, **export**, **readonly**, **return**, **set**, **shift**, **trap** e **unset**.

11.1.2. Sintassi delle funzioni

Le funzioni utilizzano la sintassi

```
function FUNCTION { COMMANDS; }
```

o

```
FUNCTION () { COMMANDS; }
```

Entrambe definiscono una funzione di shell **FUNCTION**. L'uso del comando **function** è facoltativo; tuttavia, se non viene utilizzato, sono necessarie le parentesi.

I comandi elencati tra parentesi graffe costituiscono il corpo della funzione. Questi comandi vengono eseguiti ogni volta che viene specificato **FUNCTION** come nome di un comando.. Lo stato di uscita è quello di uscita dell'ultimo comando eseguito nel corpo.



Errori comuni

Le parentesi graffe devono essere separate dal corpo da spazi, altrimenti vengono interpretate in modo errato.

Il corpo di una funzione deve terminare con un punto e virgola o una andata a capo.

11.1.3. Parametri posizionali nelle funzioni

Le funzioni sono come mini-script: possono accettare parametri, possono usare variabili note solo all'interno della funzione (usando il comando shell **local**) e possono restituire valori alla shell chiamante.

Una funzione ha anche un sistema per interpretare i parametri posizionali. Tuttavia, i parametri posizionali passati a una funzione non sono gli stessi passati a un comando o a uno script.

Quando una funzione viene eseguita, gli argomenti della funzione diventano i parametri posizionali durante la sua esecuzione. Il parametro speciale **#** che si espande al numero di parametri posizionali viene aggiornato per riflettere la modifica. Il parametro posizionale **0** resta invariato. La variabile Bash **FUNCNAME** viene impostata col nome della funzione, durante l'esecuzione.

Se viene eseguito il comando **return** in una funzione, questa termina e l'esecuzione riprende col comando successivo alla chiamata della funzione. Al termine di una funzione, i valori dei parametri posizionali e del parametro speciale **#** vengono ripristinati con i valori che avevano prima dell'esecuzione della funzione. Se viene fornito un argomento numerico per **return**, tale sarà lo stato restituito. Un semplice esempio:

```
[lydia@cointreau ~/test] cat showparams.sh
#!/bin/bash

echo "This script demonstrates function arguments."
echo

echo "Positional parameter 1 for the script is $1."
echo

test ()
```

```
{
echo "Positional parameter 1 in the function is $1."
RETURN_VALUE=$?
echo "The exit code of this function is $RETURN_VALUE."
}

test other_param

[lydia@cointreau ~/test] ./showparams.sh parameter1
This script demonstrates function arguments.

Positional parameter 1 for the script is parameter1.

Positional parameter 1 in the function is other_param.
The exit code of this function is 0.

[lydia@cointreau ~/test]
```

Si noti che il valore di ritorno o codice di uscita della funzione è spesso memorizzato in una variabile, in modo che possa essere analizzato in un secondo momento. Gli script init sul sistema usano spesso la tecnica di sondare la variabile `RETVAL` in un test condizionale, come questo:

```
if [ $RETVAL -eq 0 ]; then
    <start the daemon>
```

O come questo esempio dallo script `/etc/init.d/amd`, in cui vengono utilizzate le funzionalità di ottimizzazione di Bash:

```
[ $RETVAL = 0 ] && touch /var/lock/subsys/amd
```

I comandi dopo **&&** vengono eseguiti solo quando il test risulta essere vero; questo è il modo più breve per rappresentare una struttura **if/then/fi**.

Il codice di ritorno della funzione viene spesso utilizzato come codice di uscita dell'intero script. Si vedranno molti initscripts che terminano con qualcosa come **exit \$RETVAL**.

11.1.4. Visualizzazione delle funzioni

Tutte le funzioni conosciute dalla shell corrente possono essere visualizzate usando il comando **set** senza opzioni. Le funzioni vengono mantenute dopo essere state utilizzate, a meno che si esegua **unset** dopo l'uso. Il comando **which** mostra anche le funzioni:

```
[lydia@cointreau ~] which zless
zless is a function
zless ()
{
    zcat "$@" | "$PAGER"
}

[lydia@cointreau ~] echo $PAGER
less
```

Questo è il tipo di funzione che è solitamente configurata nei file di configurazione delle risorse della shell dell'utente. Le funzioni sono più flessibili degli alias e forniscono un modo semplice e facile per adeguare l'ambiente dell'utente.

Eccone uno per gli utenti DOS:

```
dir ()
{
    ls -F --color=auto -lF --color=always "$@" | less -r
}
```

11.2. Esempi di funzioni negli script

11.2.1. Riciclo

Ci sono molti script sul sistema che usano le funzioni come un modo strutturato per gestire una serie di comandi. Su alcuni sistemi Linux, ad esempio, c'è il file di definizione `/etc/rc.d/init.d/functions`, a cui fanno riferimento tutti gli script init. Usando questo metodo, attività comuni come il controllo dell'esecuzione di un processo, l'avvio o l'arresto di un demone e così via, devono essere scritte solo una volta, in modo generico. Se è necessario eseguire la stessa attività di nuovo, il codice viene riutilizzato.

Si potrebbe creare il proprio file `/etc/functions` che contiene tutte le funzioni usate regolarmente sul sistema, in diversi script. Basta mettere la riga

```
./etc/functions
```

da qualche parte all'inizio dello script e si possono riciclare le funzioni.

11.2.2. Impostazione del path

Questa sezione potrebbe trovarsi nel file `/etc/profile`. La funzione **pathmunge** viene definita e quindi utilizzata per impostare il percorso `perroot` e per altri utenti:

```
pathmunge () {
    if ! echo $PATH | /bin/egrep -q "^(|:)$1($|:)" ; then
        if [ "$2" = "after" ] ; then
            PATH=$PATH:$1
        else
            PATH=$1:$PATH
        fi
    fi
}

# Path manipulation
if [ `id -u` = 0 ] ; then
    pathmunge /sbin
    pathmunge /usr/sbin
    pathmunge /usr/local/sbin
fi

pathmunge /usr/X11R6/bin after

unset pathmunge
```

La funzione prende il suo primo argomento come nome di un percorso. Se tale percorso non è ancora nel path corrente, viene aggiunto. Il secondo argomento della funzione definisce se il percorso verrà aggiunto prima o dopo la definizione corrente di `PATH`.

Gli utenti normali ottengono solo che si aggiunge `/usr/X11R6/bin` ai `path`, mentre *root* ha un paio di directory extra contenenti comandi di sistema. Dopo essere stata utilizzata, la funzione viene de-impostata [`unset`] in modo da non conservarla.

11.2.3. Backup remoti

L'esempio seguente viene utilizzato per il backup dei file per dei libri. Utilizza chiavi SSH per abilitare la connessione remota. Sono definite due funzioni, **buplinux** e **bupbash**, ciascuna delle quali crea un file `.tar`, che viene poi compresso e spedito al server remoto. Successivamente, la copia locale viene eliminata.

La domenica viene eseguito solo **bupbash**.

```
#!/bin/bash

LOGFILE="/nethome/tille/log/backupsript.log"
echo "Starting backups for `date`" >> "$LOGFILE"

buplinux()
{
  DIR="/nethome/tille/xml/db/linux-basics/"
  TAR="Linux.tar"
  BZIP="$TAR.bz2"
  SERVER="rincewind"
  RDIR="/var/www/intra/tille/html/training/"

  cd "$DIR"
  tar cf "$TAR" src/*.xml src/images/*.png src/images/*.eps
  echo "Compressing $TAR..." >> "$LOGFILE"
  bzip2 "$TAR"
  echo "...done." >> "$LOGFILE"
  echo "Copying to $SERVER..." >> "$LOGFILE"
  scp "$BZIP" "$SERVER:$RDIR" > /dev/null 2>&1
  echo "...done." >> "$LOGFILE"
  echo -e "Done backing up Linux course:\nSource files, PNG and EPS
  images.\nRubbish removed." >> "$LOGFILE"
  rm "$BZIP"
}

bupbash()
{
  DIR="/nethome/tille/xml/db/"
  TAR="Bash.tar"
  BZIP="$TAR.bz2"
  FILES="bash-programming/"
  SERVER="rincewind"
  RDIR="/var/www/intra/tille/html/training/"

  cd "$DIR"
  tar cf "$TAR" "$FILES"
  echo "Compressing $TAR..." >> "$LOGFILE"
  bzip2 "$TAR"
  echo "...done." >> "$LOGFILE"
  echo "Copying to $SERVER..." >> "$LOGFILE"
  scp "$BZIP" "$SERVER:$RDIR" > /dev/null 2>&1
  echo "...done." >> "$LOGFILE"

  echo -e "Done backing up Bash course:\n$FILES\nRubbish removed." >> "$LOGFILE"
  rm "$BZIP"
}
```

```

DAY=`date +%w`

if [ "$DAY" -lt "2" ]; then
    echo "It is `date +%A`, only backing up Bash course." >> "$LOGFILE"
    bupbash
else
    buplinux
    bupbash
fi

echo -e "Remote backup `date` SUCCESS\n-----" >> "$LOGFILE"

```

L'esecuzione di questo script avviene da cron, ovvero senza interazione con l'utente, e reindirizzando lo standard error dal comando **scp** a `/dev/null`.

Si potrebbe obiettare che tutti i diversi passi possono essere raggruppati in un unico comando come

```
tar c dir_to_backup/ | bzip2 | ssh server "cat > backup.tar.bz2"
```

Questa però non è una buona soluzione se si fosse interessati ai risultati intermedi, che potrebbero essere riletti in caso di errore dello script.

L'espressione

```
command &> file
```

è equivalente a

```
command > file 2>&1
```

11.3. Sommario

Le funzioni forniscono un modo semplice per raggruppare comandi che si devono eseguire ripetutamente. Quando una funzione è in esecuzione, i parametri posizionali vengono modificati in quelli della funzione. Quando si ferma, vengono ripristinati a quelli del programma chiamante. Le funzioni sono come mini-script e, proprio come uno script, generano codici di uscita o di ritorno.

Sebbene questo fosse un breve capitolo, contiene importanti nozioni necessarie per raggiungere lo stato di pigrizia finale che è l'obiettivo tipico di qualsiasi amministratore di sistema.

11.4. Esercizi

Ecco alcune cose utili che si possono fare con le funzioni:

1. Aggiungere una funzione al file di configurazione `~/.bashrc` che automatizza la stampa delle pagine man. Il risultato dovrebbe essere che digitando qualcosa come **printman** `<command>`, venga stampata la prima pagina man appropriata. Verificarlo utilizzando una stampante virtuale a scopo di test.

Come extra, dare la possibilità all'utente di fornire il numero di sezione della pagina man da stampare.

2. Creare una sottodirectory nella home in cui possano memorizzare le definizioni delle funzioni. Mettere un paio di funzioni in tale directory. Funzioni utili potrebbero essere, tra le altre, avere gli stessi comandi di del DOS o di un UNIX commerciale lavorando con Linux, o viceversa. Queste funzioni dovrebbero poi essere importate nell'environment della shell quando viene letto `~/ .bashrc`.

Capitolo 12. Catturare i segnali

In questo capitolo tratteremo i seguenti argomenti:

- Segnali disponibili
- Uso dei segnali
- Uso dell'istruzione **trap**
- Come impedire agli utenti di interrompere i programmi

12.1. Segnali

12.1.1. Introduzione

12.1.1.1. Trovare la pagina man di signal

Il sistema contiene una pagina man che elenca tutti i segnali disponibili, ma a seconda del proprio sistema operativo, potrebbe essere aperta in un modo diverso. Sulla maggior parte dei sistemi Linux, questo sarà **man 7 signal**. Nel dubbio, individuare la pagina man e la sezione esatta usando comandi come

```
man -k signal | grep list
```

o

```
apropos signal | grep list
```

I nomi dei segnali possono essere trovati usando **kill -l**.

12.1.1.2. I segnali alla shell Bash

In assenza di un trap, una shell Bash interattiva ignora *SIGTERM* e *SIGQUIT*. *SIGINT* viene catturato e gestito e, se il controllo del job è attivo, vengono ignorati anche *SIGTTIN*, *SIGTTOU* e *SIGTSTP*. Anche i comandi eseguiti come risultato di una sostituzione di comando ignorano questi segnali, quando vengono generati da tastiera.

Per default *SIGHUP* esce da una shell. Una shell interattiva invierà un *SIGHUP* a tutti i job, in esecuzione o fermati; consultare la documentazione su **disown** se si vuol disabilitare questo

comportamento di default per un particolare processo. Usare l'opzione `huponexit` per terminare (kill) tutti i job dopo la ricezione di un *SIGHUP*, utilizzando il comando **shopt**.

12.1.1.3. Invio di segnali con la shell

I seguenti segnali possono essere inviati tramite la shell Bash:

Tabella 12-1. Segnali di controllo in Bash

Combinazione standard di tasti	Significato
Ctrl+C	Il segnale di interruzione invia <i>SIGINT</i> al job principale in esecuzione in primo piano.
Ctrl+Y	Il carattere di <i>sospensione ritardata</i> . Provoca l'arresto di un processo in esecuzione quando tenta di leggere l'input dal terminale. Il controllo viene restituito alla shell, l'utente può mettere il processo in primo piano, in background o terminarlo. La sospensione ritardata è disponibile solo sui sistemi operativi la supportano.
Ctrl+Z	Il segnale di <i>sospensione</i> , invia un <i>SIGTSTP</i> ad un programma in esecuzione, fermandolo e restituendo il controllo alla shell.

Impostazioni del terminale

Controllare le impostazioni della propria **stty**. La sospensione e il ripristino dell'output sono generalmente disabilitati se si utilizzano emulazioni di terminale "moderne". Lo standard **xterm** supporta **Ctrl+S** e **Ctrl+Q** per default.

12.1.2. Utilizzo di segnali con kill

La maggior parte delle shell moderne, compreso Bash, ha una funzione **kill**. In Bash, sono accettati come opzioni sia i nomi che i numeri dei segnali e gli argomenti possono essere ID di job o di processo. Uno stato di uscita può essere segnalato utilizzando l'opzione `-1`: zero quando almeno un segnale è stato inviato con successo, diverso da zero se si è verificato un errore.

Utilizzando il comando **kill** da `/usr/bin`, il sistema potrebbe abilitare opzioni extra, come la possibilità di killare i processi da un ID utente diverso dal proprio e specificare i processi per nome, come con **pgrep** e **pkill**.

Entrambi i comandi **kill** inviano il segnale *TERM* se non ci sono opzioni.

Questo è un elenco dei segnali più comuni:

Tabella 12-2. Segnali kill comuni

Nome del segnale	Valore del segnale	Effetto
SIGHUP	1	Interruzione
SIGINT	2	Interruzione dalla tastiera
SIGKILL	9	segnale kill

Nome del segnale	Valore del segnale	Effetto
SIGTERM	15	Segnale di terminazione
SIGSTOP	17,19,23	Stop del processo



SIGKILL e SIGSTOP

SIGKILL e *SIGSTOP* non possono essere catturati, bloccati o ignorati.

Quando si interrompe un processo o una serie di processi, è buona norma iniziare a provare con il segnale meno pericoloso, *SIGTERM*. In questo modo, i programmi che provvedono ad un arresto ordinato hanno la possibilità di seguire le procedure per cui sono stati progettati da eseguire quando ricevono il segnale *SIGTERM*, come pulire e chiudere i file aperti. Se si invia un *SIGKILL* ad un processo, si toglie ogni possibilità che il processo esegua una pulizia e uno spegnimento ordinati, il che potrebbe avere conseguenze spiacevoli.

Ma se una terminazione pulita non funziona, i segnali *INT* o *KILL* potrebbero essere l'unico modo. Ad esempio, quando un processo non 'muore' usando **Ctrl+C**, è meglio usare il **kill -9** su quell'ID di processo:

```
maud: ~> ps -ef | grep stuck_process
maud   5607   2214  0 20:05 pts/5    00:00:02 stuck_process

maud: ~> kill -9 5607

maud: ~> ps -ef | grep stuck_process
maud   5614   2214  0 20:15 pts/5    00:00:00 grep stuck_process
[1]+  Killed                  stuck_process
```

Quando un processo avvia più istanze, **killall** potrebbe semplificare le cose. Richiede la stessa opzione del comando **kill**, ma si applica a tutte le istanze di un determinato processo. Fare pratica con questo comando prima di usarlo in un ambiente di produzione, poiché potrebbe non funzionare come previsto su alcuni Unix commerciali.

12.2. Trap

12.2.1. In generale

Potrebbero verificarsi situazioni in cui non si desidera che gli utenti dei propri script escano prematuramente utilizzando sequenze di interruzione da tastiera, ad esempio perché è necessario fornire un input o eseguire la pulizia. L'istruzione **trap** cattura queste sequenze e può essere programmata per eseguire una serie di comandi dopo averle catturate.

La sintassi per l'istruzione **trap** è semplice:

trap [COMMANDS] [SIGNALS]

Questo indica al comando **trap** di catturare i *SIGNALS* elencati, che possono essere nomi di segnali con o senza il prefisso *SIG*, o i numeri. Se un segnale è *0* o *EXIT*, vengono eseguiti i **COMMANDS** all'uscita della shell. Se uno dei segnali è *DEBUG*, i **COMMANDS** vengono eseguiti dopo ciascun comando semplice. Un segnale può anche essere indicato come *ERR*; in questo caso i **COMMANDS** vengono eseguiti ogni volta che un comando semplice esce con uno stato diverso da zero. Notare che questi comandi non verranno eseguiti quando lo stato di uscita diverso da zero

proviene da una parte di un'istruzione **if**, o da cicli **while** o **until**. Né verranno eseguiti se il risultato di un *AND* (&&) o un *OR* (||) logico è diverso da zero, nemmeno quando lo stato di ritorno di un comando viene invertito con l'operatore *!*.

Lo stato di ritorno del comando **trap** stesso è zero a meno che non venga rilevata un segnale non valido. Il comando **trap** accetta un paio di opzioni, che sono documentate nelle pagine informative di Bash.

Ecco un esempio molto semplice, che cattura il **Ctrl+C** dell'utente, stampando un messaggio. Quando si tenta di terminare questo programma senza indicare il segnale *KILL*, non succederà niente:

```
#!/bin/bash
# traptest.sh

trap "echo Booh!" SIGINT SIGTERM
echo "pid is $$"

while :                # This is the same as "while true".
do
    sleep 60           # This script is not really doing anything.
done
```

12.2.2. Come Bash interpreta le trap

Quando Bash riceve un segnale per il quale è stata impostata una trap si attende il completamento del comando corrente in esecuzione. Quando Bash è in attesa di un comando asincrono tramite **wait**, la ricezione di un segnale per il quale è stata impostata una trap farà tornare immediatamente **wait** con uno stato di uscita maggiore di 128, subito dopo il quale viene eseguita la trap.

12.2.3. Altri esempi

12.2.3.1. Rilevare quando viene utilizzata una variabile

Durante il debug di lunghi script, si potrebbe voler assegnare a una variabile l'attributo *trace* e catturare i messaggi *DEBUG* per quella variabile. Normalmente la variabile si dichiarerebbe con **VARIABLE=VALUE**. La sostituzione della dichiarazione della variabile con le seguenti righe potrebbe fornire informazioni preziose su ciò che sta facendo lo script:

```
declare -t VARIABLE=value

trap "echo VARIABLE is being used here." DEBUG

# rest of the script
```

12.2.3.2. Rimozione della spazzatura all'uscita

Il comando **whatis** si basa su un database che viene regolarmente creato utilizzando lo script `makewhatis.cron` con `cron`:

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

```

LOCKFILE=/var/lock/makewhatis.lock

# Previous makewhatis should execute successfully:

[ -f $LOCKFILE ] && exit 0

# Upon exit, remove lockfile.

trap "{ rm -f $LOCKFILE ; exit 255; }" EXIT

touch $LOCKFILE
makewhatis -u -w
exit 0

```

12.3. Sommario

I segnali possono essere inviati ai programmi usando il comando **kill** o con scorciatoie da tastiera. Questi segnali possono essere catturati, eseguendo delle azioni, utilizzando l'istruzione **trap**.

Alcuni programmi ignorano i segnali. L'unico segnale che nessun programma può ignorare è *KILL*.

12.4. Esercizi

Qualche esempio pratico:

1. Creare uno script che scriva un'immagine del boot su un dischetto utilizzando **dd**. Se l'utente tenta di interrompere lo script con **Ctrl+C**, visualizzare un messaggio che indichi che tale azione renderà il dischetto inutilizzabile.
2. Scrivere uno script che automatizza l'installazione di un pacchetto di terze parti. Il pacchetto deve essere scaricato da Internet. Deve essere decompresso, spaccettato e compilato se è il caso. Solo l'installazione effettiva del pacchetto dovrebbe non essere interrompibile.

Appendice A. Funzionalità della shell

Questo documento fornisce una panoramica delle caratteristiche comuni della shell (le stesse in ogni tipo di shell) e delle diverse funzionalità della shell (funzionalità specifiche della shell).

A.1. Funzionalità comuni

Le seguenti funzionalità sono standard in ogni shell. Si noti che i comandi **stop**, **suspend**, **jobs**, **bg** e **fg** sono disponibili solo su sistemi che supportano il controllo dei job.

Tabella A-1. Funzionalità Comuni alla Shell

Comando	Significato
>	Reindirizzamento dell'output

Comando	Significato
>>	Accodamento ad un file
<	Reindirizzamento dell'input
<<	Documento "Here" (reindirizza l'input)
	Pipe di output
&	Esegue il processo in background.
;	Separatore di comandi sulla stessa riga
*	Corrisponde a qualsiasi carattere/i nel nome del file
?	Corrisponde a un solo carattere nel nome del file
[]	Corrisponde a qualsiasi carattere racchiuso
()	Esegue in una subshell
` `	Sostituisce con l'output del comando racchiuso
" "	Virgoletta parziale (consente l'espansione di variabili e comandi)
' '	Virgolette piene (nessuna espansione)
\	Quota il carattere successivo
\$var	Usa il valore della variabile
\$\$	Id del processo
\$0	Il nome del comando
\$n	n-esimo argomento (n da 0 a 9)
#	Inizio commento
bg	Esecuzione in background
break	Interrompe del ciclo
cd	Cambia directory
continue	Riprende un ciclo
echo	Mostra l'output
eval	Valuta gli argomenti
exec	Esegue una nuova shell
fg	Esecuzione in primo piano (foreground)
jobs	Mostra i job attivi
kill	Termina i job in esecuzione
newgrp	cambia il gruppo
shift	Slitta i parametri posizionali
stop	Sospende un job in background
suspend	Sospende un job in foreground
time	Il comando per l'orario
umask	Imposta una serie di permessi
unset	Cancella le definizioni di variabili o funzioni
wait	Attendi il completamento di un job in background

A.2. Funzionalità diverse

La tabella seguente mostra le principali differenze tra le shell standard (**sh**), Bourne Again SHell (**bash**), Korn shell (**ksh**) e la C shell (**csh**).

Compatibilità tra shell

Poiché la Bourne Again SHell è un superset della **sh**, tutti i comandi **sh** funzioneranno anche in **bash** - ma non viceversa. **bash** ha molte più funzioni proprie e, come mostra la tabella seguente, sono comprese molte funzioni da altre shell.

Dato che la Turbo C è un superset della **csh**, tutti i comandi della **csh** funzioneranno anche in **tcsh**, ma non il contrario.

Tabella A-2. Funzionalità Shell differenti

sh	bash	ksh	csh	Significato/Azione
\$	\$	\$	%	Prompt utente di default
	>	>	>!	Reindirizzamento forzato
> file 2>&1	&> file 0 > file 2>&1	> file 2>&1	>& file	Reindirizza stdout e stderr verso file
	{ }		{ }	Espande gli elementi nella lista
`command`	`command` o \$(command)	\$(command)	`command`	Sostituisce con l'output del command racchiuso
\$HOME	\$HOME	\$HOME	\$home	Directory home
	~	~	~	Simbolo della directory home
	~+, ~-, dirs	~+, ~-	=-, =N	Accesso allo stack della directory
var=value	VAR=value	var=value	set var=value	Assegnazione di variabile
export var	export VAR=value	export var=val	setenv var val	Impostazione di una variabile d'ambiente
	\${nnnn}	\${nn}		È possibile fare riferimento a più di 9 argomenti
"\$@"	"\$@"	"\$@"		Tutti gli argomenti come parole separate

sh	bash	ksh	csh	Significato/Azione
\$#	\$#	\$#	\$#argv	Il numero degli argomenti
\$?	\$?	\$?	\$status	Stato di uscita dell'ultimo comando eseguito
\$!	\$!	\$!		PID del processo in background più recente
\$-	\$-	\$-		Opzioni attuali
. file	source file or . file	. file	source file	Legge i comandi nel file
	alias x='y'	alias x=y	alias x y	Il nome x sta per il comando y
case	case	case	switch o case	Sceglie un'alternativa
done	done	done	end	Fine di un'istruzione di un loop
esac	esac	esac	endsw	Fine di un case o di uno switch
exit n	exit n	exit n	exit (expr)	Esce con uno stato
for/do	for/do	for/do	foreach	Cicla attraverso delle variabili
	set -f, set -o nullglob dotglob nocaseglob noglob		noglob	Ignora i caratteri di sostituzione per la generazione del nome file
hash	hash	alias -t	hashstat	Visualizza comandi hash (alias tracciati)
hash cmds	hash cmds	alias -t cmds	rehash	Ricorda le posizioni dei comandi
hash -r	hash -r		unhash	Dimentica le posizioni dei comandi
	history	history	history	Elenca i comandi precedenti
	ArrowUp+Enter or !!	r	!!	Ripete il comando precedente
	!str	r str	!str	Ripete l'ultimo comando che inizia con "str"
	!cmd:s/x/y/	r x=y cmd	!cmd:s/x/y/	Sostituisce "x" con "y" nel

sh	bash	ksh	csh	Significato/Azione
				comando più recente che inizia con "cmd", poi lo esegue.
if [\$i -eq 5]	if [\$i -eq 5]	if ((i==5))	if (\$i==5)	Esempio di condizione di test
fi	fi	fi	endif	Chiude l'istruzione if
ulimit	ulimit	ulimit	limit	Imposta i limiti delle risorse
pwd	pwd	pwd	dirs	Stampa la directory di lavoro
read	read	read	\$<	Legge dal terminale
trap 2	trap 2	trap 2	onintr	Ignora le interruzioni
	unalias	unalias	unalias	Rimuove gli alias
until	until	until		Comincia un ciclo until
while/do	while/do	while/do	while	Comincia un ciclo while

La Bourne Again SHell ha molte altre funzionalità non elencate qui. Questa tabella serve solo per dare un'idea di come questa shell includa tutte le idee utili di altre shell: non ci sono spazi vuoti nella colonna per **bash**. Maggiori informazioni sulle funzionalità che si trovano solo in Bash possono essere recuperate dalle pagine informative di Bash, nella sezione "Bash Features".

Maggiori informazioni:

Si dovrebbe almeno leggere un manuale, dato il manuale della shell. Sarebbe preferibile **info bash**, **bash** dato che la shell GNU è più semplice per chi inizia. Lo si stampi e lo si studi quando si ha del tempo.

Glossario

Questa sezione contiene una panoramica in ordine alfabetico dei comandi comuni di UNIX. Maggiori informazioni sull'utilizzo sono reperibili nelle pagine man o info.

A

a2ps

File di formato per la stampa su una stampante PostScript.

acroread

Visualizzatore PDF.

adduser

Crea un nuovo utente o aggiorna i default per i nuovi utenti.

alias

Crea un alias di shell per un comando.

anacron

Esegue i comandi periodicamente, non presuppone che la macchina continui a funzionare.

apropos

Cerca le stringhe nel database whatis.

apt-get

Utilità di gestione dei pacchetti APT.

aspell

Correttore ortografico.

at, atq, atrm

Accoda, esamina o elimina i job per l'esecuzione successiva.

aumix

Regola il mixer audio.

(g)awk

Scansione di pattern ed elaborazione del linguaggio.

B

bash

Bourne Again SHell.

batch

Accoda, esamina o elimina i job per l'esecuzione successiva.

bg

Esegue un job in background.

bitmap

Editor di bitmap e convertitore per X window System.

bzip2

U compressore di file block-sorting.

C**cat**

Concatena i file e li stampa sullo standard output.

cd

Cambia directory.

cdp/cdplay

Un programma interattivo testuale per il controllo e la riproduzione di CD Rom audio sotto Linux.

cdparanoia

Un'utilità di lettura di CD audio che include funzionalità aggiuntive per la verifica dei dati.

cdrecord

Registra un CD-R.

chattr

Cambia gli attributi di un file

chgrp

Cambia la proprietà di un gruppo

chkconfig

Aggiorna o interroga le informazioni sul livello di esecuzione dei servizi di sistema.

chmod

Cambia i permessi di accesso di un file

chown

Cambia il proprietario e il gruppo di un file

compress

Comprime i file.

cp

Copia file e directory.

crontab

Gestisce i file crontab.

csch

Apri una C shell.

cut

Rimuovere le sezioni da ciascuna riga di uno o più file.

D

date

Stampa o imposta l'ora e la data del sistema.

dd

Converte e copia un file (disk dump).

df

Riporta l'utilizzo del disco di sistema.

dhcpcd

Demone client del DHCP.

diff

Cerca le differenze tra due file.

dig

Interroga server DNS ed esegue delle ricerche.

dmesg

Visualizza o gestisce i messaggi contenuti nel buffer del kernel.

du

Stima l'utilizzo dello spazio per i file.

E

echo

Mostra una riga di testo.

ediff

Traduttore da diff a inglese.

egrep

Un grep esteso.

eject

Smonta ed espelle il supporto rimovibile.

emacs

Avvia l'editor Emacs.

exec

Richiama sotto-processi.

exit

Esce dalla shell corrente.

export

Aggiunge delle funzioni all'environment della shell.

F**fax2ps**

Converte un facsimile TIFF in PostScript.

fdformat

Formatta un floppy disk.

fdisk

Gestore per la tabella delle partizioni per Linux.

fetchmail

Recupera la posta da un server compatibile con POP, IMAP, ETRN o ODMR.

fg

Porta un job in primo piano (foreground).

file

Determina il tipo di file.

find

Cerca i file.

formail

(Ri)formattatore di mail.

fortune

Stampa un detto casuale, si spera risulti interessante.

ftp

Servizio per trasferire file (noion sicuro a meno che non venga utilizzato un account anonimo!).

G

galeon

Browser web grafico.

gdm

Gnome Display Manager.

(min/a)getty

Dispositivi di console di controllo.

gimp

Programma per la gestione di immagini.

grep

Scrive le righe che corrispondono ad un pattern.

grub

La shell grub (boot loader del progetto GNU).

gv

Un visualizzatore per PostScript PDF.

gzip

Comprime ed espande file.

H

halt

Ferma il sistema.

head

Visualizza la prima parte dei file.

help

Mostra l'help su un comando nativo della shell.

host

Utility di ricerca DNS.

httpd

Server Apache per l'hypertext transfer protocol.

I**id**

Stampa UID e GID reali ed effettive.

ifconfig

Configura l'interfaccia di rete e ne mostra la configurazione.

info

Legge i documenti Info.

init

Controlla il processo di inizializzazione.

iostat

Mostra le statistiche sull'I/O.

ip

Mostra/modifica lo stato dell'interfaccia di rete.

ipchains

Amministrazione del firewall IP.

iptables

Amministrazione dei filtri dei pacchetti IP.

J

jar

Tool per gli archivi Java.

jobs

Elenca i job in background.

K

kdm

Desktop manager per KDE.

kill(all)

Termina uno o più processi.

ksh

Apri una Korn shell.

L

ldapmodify

Modifica una voce di un LDAP.

ldapsearch

Tool per la ricerca in un LDAP.

less

more con delle funzionalità.

lilo

Boot loader di Linux .

links

Browser WWW testuale.

ln

Crea collegamenti (link) tra file.

loadkeys

Carica le tabelle di traduzione della tastiera.

locate

Cerca i file.

logout

Chiude la shell corrente.

lp

Invia le richieste al servizio di stampa LP.

lpc

Programma di controllo della stampante di linea.

lpq

Programma per esaminare la coda di stampa.

lpr

Stampa offline.

lprm

Rimuove le richieste di stampe.

ls

Lista il contenuto di una directory.

lynx

Browser WWW testuale.

M

mail

Spedisce e riceve mail.

man

Legge le pagine man.

mcopy

Copia i file MSDOS da/a Unix.

mdir

Mostra una directory MSDOS.

memusage

Mostra l'utilizzo della memoria.

memusagestat

Mostra le statistiche sull'uso della memoria.

mesg

Regola l'accesso in scrittura al proprio terminale.

mformat

Aggiungere un file system MSDOS a un floppy disk formattato di basso livello.

mkbootdisk

Crea un floppy di avvio autonomo per il sistema corrente.

mkdir

Crea una directory.

mkisofs

Crea un filesystem ibrido ISO9660.

more

Filtro per visualizzare un testo una schermata alla volta.

mount

Monta un file system o visualizza informazioni sui file system montati.

mozilla

Browser Web.

mt

Controlla il funzionamento dell'unità a nastro magnetico.

mtr

Strumento diagnostico della rete.

mv

Rinomina i file.

N

named

Server dei nomi di dominio Internet.

ncftp

Programma browser per servizi ftp (non sicuro!).

netstat

Stampa connessioni di rete, tabelle di routing, statistiche dell'interfaccia, connessioni mascherate (masquerade) e appartenenti a multicast.

nfsstat

Stampa statistiche sui file system in rete.

nice

Esegue un programma con priorità di scheduling modificata.

nmap

Strumento di esplorazione della rete e scanner per la sicurezza.

ntsysv

Semplice interfaccia per la configurazione dei livelli di esecuzione.

P

passwd

Cambia la password.

pdf2ps

Traduttore da Ghostscript PDF a PostScript.

perl

Practical Extraction and Report Language.

pg

Paginazione del testo in output.

ping

Invia una richiesta di eco ad un host.

pr

Converte il testo per la stampa.

printenv

Stampa tutte le parti di un environment.

procmail

Elaboratore di mail autonomo.

ps

Riposta lo stato di un processo.

pstree

Mostra l'alberatura dei processi.

pwd

Stampa la directory di lavoro corrente.

Q

quota

Mostra l'utilizzo e i limiti del disco.

R

rcp

Copia remota (non sicura!)

rdesktop

Client per il Remote Desktop Protocol.

reboot

Ferma e riavvia il sistema.

renice

Altera la priorità di un processo in esecuzione.

rlogin

Remote login (telnet, non sicuro!).

rm

Rimuove un file.

rmdir

Rimuove una directory.

rpm

RPM Package Manager.

rsh

Remote shell (non sicura!).

S**scp**

Secure remote copy.

screen

Screen manager con emulazione VT100.

set

Mostra, imposta o cambia una variabile.

setterm

Imposta gli attributi di un terminale.

sftp

Ftp sicuro (crittografato).

sh

Apri una standard shell.

shutdown

Spegne un sistema.

sleep

Aspetta per un determinato periodo.

slocate

Versione Security Enhanced del GNU Locate.

slrnn

Client testuale per Usenet.

snort

Strumento di rilevamento delle intrusioni di rete.

sort

Ordina le righe di un file di testo.

ssh

Secure shell.

ssh-keygen

Generazione della chiave di autenticazione.

stty

Modifica e stampa le impostazioni della linea terminale

su

Cambia utente.

T

tac

Concatena e stampa i file al contrario.

tail

Visualizza l'ultima parte dei file.

talk

Chat con un altro utente.

tar

Utility di archiviazione.

tcsh

Apri una shell Turbo C.

telnet

Interfaccia utente per il protocollo TELNET (non sicura!).

tex

Formattazione e impaginazione del testo.

time

Temporizza un comando semplice o fornisce l'uso di risorse.

tin

Programma per leggere notizie.

top

Visualizza i principali processi della CPU.

touch

Cambia i timestamp dei file.

traceroute

Stampa il percorso dei pacchetti verso l'host di rete.

tripwire

Un controllo di integrità dei file per sistemi UNIX.

twm

Tab Window Manager per X Window System.

U**ulimit**

Controllo delle risorse.

umask

Imposta la maschera di creazione del file utente.

umount

Smonta un file system.

uncompress

Decomprime file compressi.

uniq

Rimuovere le righe duplicate da un file ordinato.

update

Demone del kernel per svuotare i buffer sporchi sul disco.

uptime

Visualizza il tempo di attività del sistema e il carico medio.

userdel

Elimina un account utente e relativi file.

V

vi(m)

Avvia l'editor vi (migliorato).

vimtutor

Il tutor Vim.

vmstat

Riporta le statistiche sulla memoria virtuale.

W

w

Mostra chi è connesso e cosa sta facendo.

wall

Invia un messaggio a tutti i terminali.

wc

Stampa il numero di byte, parole o righe nei file.

which

Mostra il percorso completo dei comandi (shell).

who

Mostra chi ha effettuato l'accesso.

who am i

Stampa l'ID utente effettivo.

whois

Interroga un database whois o nickname.

write

Invia un messaggio a un altro utente.

X

xauth

Utilità dell'authority file.

xcdroast

Front-end grafico per cdrecord.

xclock

Orologio analogico/digitale per X.

xconsole

Monitora i messaggi della console di sistema con X.

xdm

X Display Manager con supporto per XDMCP, selettore dell'host.

xdvi

Visualizzatore DVI.

xf86

server per font X.

xhost

Programma di controllo degli accessi al server per X

xinetd

Il demone dei servizi Internet estesi.

xload

Visualizzazione del carico medio del sistema per X.

xlsfonts

Visualizzatore dell'elenco dei server dei font per X.

xmms

Lettore audio per X.

xpdf

Visualizzatore PDF.

xterm

Emulatore di terminale per X.

Z

zcat

Comprime ed espande file.

zgrep

Cerca nei file, eventualmente compressi, un'espressione regolare.

zmore

Filtro per la visualizzazione del testo compresso.

Indice

A

alias

[Section 3.5.1](#)

ANSI-C tra virgolette

[Sezione 3.3.5](#)

argomenti

[Sezione 7.2.1.2](#)

espansione aritmetica

[Sezione 3.4.7](#)

operatori aritmetici

[Sezione 3.4.7](#)

array

[Sezione 10.2.1](#)

awk

[Sezione 6.1](#)

awkprogram

[Sezione 6.1.2](#)

B

bash

[Sezione 1.2](#)

.bash_login

[Sezione 3.1.2.2](#)

.bash_logout

[Sezione 3.1.2.5](#)

.bash_profile

[Sezione 3.1.2.1](#)

.bashrc

[Sezione 3.1.2.4](#)

batch editor

[Sezione 5.1.1](#)

break

[Sezione 9.5.1](#)

boolean operators

[Sezione 7.2.4](#)

Bourne shell

[Sezione 1.1.2](#)

brace expansion

[Sezione 3.4.3](#)

built-in commands

[Sezione 1.3.2](#)

C

Istruzioni case

[Sezione 7.2.5](#)

Classi di caratteri

[Sezione 4.2.2.2](#), [Sezione 4.3.2](#)

Processo figlio (child)

[Sezione 1.3.1](#)

espressioni combinate

[Sezione 7.1.1.1](#)

sostituzione del comando

[Sezione 3.4.6](#)

commenti

[Sezione 2.2.2](#)

istruzioni condizionali

[Sezione 7.1](#)

file di configurazione

[Sezione 3.1](#)

costanti

[Sezione 10.1.3](#)

continue

[Sezione 9.5.2](#)

segnali di controllo

[Sezione 12.1.1.3](#)

creazione delle variabili

[Sezione 3.2.2](#)

csh

La C shell, [Sezione 1.1.2](#)

D

debugging degli script

[Sezione 2.3](#)

declare

[Sezione 10.1.2](#), [Sezione 10.2.1](#)

Virgolette doppie

[Sezione 3.3.4](#)

E

echo

[Sezione 1.5.5](#), [Sezione 2.1.2](#), [Sezione 2.3.2](#), [Sezione 8.1.2](#)

editor

[Sezione 2.1.1](#)

else

[Sezione 7.2.1](#)

emacs

[Sezione 2.1.1](#)

env

[Sezione 3.2.1.1](#)

esac

[Sezione 7.2.5](#)

caratteri di escape

[Sezione 3.3.2](#)

sequenze di escape

[Sezione 8.1.2](#)

/etc/bashrc

[Sezione 3.1.1.2](#)

/etc/passwd

[Sezione 1.1.2](#)

/etc/profile

[Sezione 3.1.1](#)

/etc/shells

[Sezione 1.1.2](#)

exec

[Sezione 1.3.1](#), [Sezione 8.2.4.2](#)

permessi di esecuzione

[Sezione 2.1.3](#)

esecuzione

[Sezione 2.1.3](#)

exit

[Sezione 7.2.5](#)

stato di uscita

[Sezione 7.1.2.1](#)

espansione

[Sezione 1.4.1.5](#), [Sezione 3.4](#)

export

[Sezione 3.2.3](#)

espressioni regolari estese

[Sezione 4.1.3](#)

F

descrittori di file

[Sezione 8.2.3](#), [Sezione 8.2.4.1](#)

espansione dei nomi dei file

[Sezione 3.4.9](#)

ricerca e sostituzione

[Sezione 5.2.4](#)

for

[Sezione 9.1](#)

fork

[Sezione 1.3.1](#)

funzioni

[Sezione 11.1.1](#)

G

gawk

[Sezione 6.1.1](#)

comandi gawk

[Sezione 6.1.2](#)

campi gawk

[Sezione 6.2.1](#)

formattazione gawk

[Sezione 6.2.2](#)

script gawk

[Sezione 6.2.5](#)

variabili gawk

[Sezione 6.3](#)

gedit

[Sezione 2.1.1](#)

variabili globali

[Sezione 3.2.1.1](#)

globbing

[Sezione 2.3.2](#)

grep

[Sezione 4.2.1](#)

H

documento here

[Sezione 8.2.4.4](#)

I

if

[Sezione 7.1.1](#)

init

[Sezione 1.3.1](#), [Sezione 1.5.6](#)

file di inizializzazione

[Sezione 3.1](#)

separatore dei campi di input

[Sezione 3.2.4.1](#), [Sezione 3.2.5](#), [Sezione 6.3](#)

editing interattivo

[Sezione 5.2](#)

script interattivi

[Sezione 8.1](#)

shell interattiva

[Sezione 1.2.2.2.1](#), [Sezione 1.2.2.2.2](#), [Sezione 1.2.2.3.3](#)

invocazione

[Sezione 1.2.2.1](#)

J

K

kill

[Sezione 12.1.2](#)

killall

[Sezione 12.1.2](#)

ksh

Korn shell, [Sezione 1.1.2](#)

L

lunghezza di una variabile

[Sezione 10.3.2](#)

ancoraggi alle righe

[Sezione 4.2.2.1](#)

Lingua 'locale'

[Sezione 3.3.6](#)

locate

[Sezione 2.1.1](#)

flusso logico

[Sezione 1.5.4](#)

shell logica

[Sezione 1.2.2.2.1](#)

M

menù

[Sezione 9.6](#)

metacaratteri

[Sezione 4.1.2](#)

N

istruzioni if nidificate

[Sezione 7.2.3](#)

noglob

[Sezione 2.3.2](#)

editing non-interattivo

[Sezione 5.3](#)

shell non-interattiva

[Sezione 1.2.2.2.3](#)

shell senza login (non-login)

[Sezione 1.2.2.2.2](#)

confronti numerici

[Sezione 7.1.2.2](#)

O

opzioni

[Sezione 3.6.1](#)

separatore di campo di output

[Sezione 6.3.2.1](#)

separatore di record di output

[Sezione 6.3.2.2](#)

P

espansione del parametro

[Sezione 3.4.5](#)

PATH

[Sezione 2.1.2](#)

pattern matching

[Sezione 4.3](#)

parametri posizionali

[Sezione 3.2.5](#), [Sezione 11.1.3](#)

POSIX

[Sezione 1.2.1](#)

Modalità POSIX

[Sezione 1.2.2.2.5](#)

espressioni primarie

[Sezione 7.1.1.1](#)

printenv

[Sezione 3.2.1.1](#)

printf

[Sezione 1.5.5](#), [Sezione 6.3.6](#)

sostituzione di processo

[Sezione 3.4.8](#)

.profile

[Sezione 3.1.2.3](#)

prompt

[Sezione 3.1.3](#)

Q

caratteri tra virgolette

[Sezione 3.3](#)

R

reindirizzamento

[Sezione 1.4.1.7](#), [Sezione 3.6.2](#), [Sezione 8.2.3](#), [Sezione 9.4](#)

rbash

[Sezione 1.2.2.10](#)

read

[Sezione 8.2.1](#)

readonly

[Sezione 10.1.3](#)

operatori delle espressioni regolari

[Sezione 4.1.2](#), [Sezione 5.2](#), [Sezione 6.2.4](#)

espressioni regolari

[Sezione 4.1](#)

invocazione remota

[Sezione 1.2.2.2.6](#)

rimozione degli alias

[Sezione 3.5.2](#)

variabili riservate

[Sezione 3.2.4](#)

return

[Sezione 11.1.3](#)

S

sed

[Sezione 5.1](#)

comandi sed di editing

[Sezione 5.1.2](#)

opzioni di sed

[Sezione 5.1.2](#)

script sed

[Sezione 5.3.2](#)

select

[Sezione 9.6](#)

set

[Sezione 3.2.1.2](#), [Sezione 3.6.1](#), [Sezione 11.1.4](#)

shift

[Sezione 9.7](#)

segnali

[Sezione 12.1.1](#)

apici singoli

[Sezione 3.3.3](#)

sorgente

[Sezione 2.1.3](#)

parametri speciali

[Sezione 3.2.5](#)

variabili speciali

[Sezione 3.2.5](#)

errore standard

[Sezione 8.2.3.1](#)

input standard

[Sezione 8.2.3.1](#)

output standard

[Sezione 8.2.3.1](#)

confronto di stringhe

[Sezione 7.1.2.3](#)

stty

[Sezione 12.1.1](#)

totto-menù

[Sezione 9.6.2](#)

subshell

[Sezione 2.2.1](#)

sostituzione

[Sezione 10.3.3.1](#), [Sezione 10.3.3.3](#)

sotto-stringa

[Sezione 10.3.3.2](#)

sintassi

[Sezione 1.4.1.1](#)

T

tcsh

[Sezione 1.1.2](#)

terminologia

[Sezione 1.5.3](#)

then

[Sezione 7.1.1.2](#)

espansione della tilde

[Sezione 3.4.4](#)

trasformazione di variabili

[Sezione 10.3.3](#)

trap

[Sezione 12.2.1](#)

true

[Sezione 9.2.2.2](#)

U

unalias

[Sezione 3.5.1](#), [Sezione 3.5.2](#)

unset

[Sezione 3.2.2](#), [Sezione 10.2.3](#), [Sezione 11.1.4](#)

until

[Sezione 9.3](#)

input utente

[Sezione 8.2.1](#), [Sezione 8.2.2](#)

messaggi utente

[Sezione 8.1.1](#)

V

variabili

[Sezione 3.2](#), [Sezione 10.1](#)

espansione di variabile

[Sezione 3.4.5](#)

verbose

[Sezione 2.3.2](#)

vi(m)

[Sezione 2.1.1](#)

W

wait

[Sezione 12.2.2](#)

whereis

[Sezione 2.1.1](#)

which

[Sezione 2.1.1](#)

while

[Sezione 9.2](#)

caratteri jolly [wildcard]

[Sezione 4.2.2.3](#)

ancoraggi alle parole

[Sezione 4.2.2.1](#)

divisione di parole

[Sezione 3.4.9](#)

X

xtrace

[Sezione 2.3.1](#), [Sezione 2.3.2](#)

Y

Z