Capitolo 1

La shell bash

Bash è un interprete di comandi: fornisce un ambiente in cui gli utenti possono eseguire comandi e lanciare applicazioni. Come penetration tester e professionisti della sicurezza, ci capita spesso di scrivere script bash per automatizzare un'ampia gamma di attività, cosa che rende la shell bash uno strumento essenziale per un hacker. In questo capitolo, imposterete il vostro ambiente di sviluppo bash, esplorerete alcuni utili comandi Linux da includere nei futuri script e apprenderete i fondamenti della sintassi del linguaggio: le variabili, gli array, gli stream, gli argomenti e gli operatori.

In questo capitolo

Configurazione dell'ambiente

Prima di iniziare a studiare la shell bash, avrete bisogno sia di una shell bash in esecuzione in un terminale sia di un editor di testo. Potete accedervi da qualsiasi sistema operativo, seguendo le istruzioni presentate in questo paragrafo.

NOTA

A partire dal Capitolo 4, utilizzerete Kali Linux per eseguire i comandi bash e completare i laboratori di hacking. Se desiderate configurare Kali ora, consultate i passaggi presentati nel Capitolo 3.

Accesso alla shell bash

Se state usando Linux o macOS, la shell bash dovrebbe già essere disponibile. Su Linux, aprite l'applicazione *Terminale* con Ctrl-Alt-T. Su macOS, potete trovare il terminale dall'icona Launchpad sul dock di sistema.
Kali e macOS usano di default la Z Shell, quindi, quando aprite una nuova finestra del terminale, dovrete digitare `exec bash` per passare alla shell bash prima di eseguire i comandi. Se volete cambiare la shell di default in bash, in modo da non dover sempre cambiare manualmente shell, potete impiegare il comando `chsh -s /bin/bash`.
Se utilizzate Windows, potete impiegare WSL (*Windows Subsystem for Linux*), che consente di eseguire distribuzioni Linux e di accedere a un ambiente bash. La pagina ufficiale della documentazione di Microsoft WSL (`https://learn.microsoft.com/it-it/windows/wsl/install`) spiega come installarlo.
Un'alternativa a WSL è *Cygwin*, che emula un ambiente Linux fornendo una raccolta di utility Linux e funzionalità di chiamata al sistema. Per installare Cygwin, visitate `https://www.cygwin.com/install.html` per scaricare il file di setup, poi seguite la procedura guidata di installazione.
Cygwin si installa, di default, nella cartella `C:\cygwin64\` di Windows. Per eseguire i vostri script bash, salvate gli script nella directory contenente il vostro nome utente in `C:\cygwin64\home`. Per esempio, se il vostro nome utente è `david`, dovete salvare i vostri script in `C:\cygwin64\home\david`. Quindi, dal terminale di Cygwin, sarete in grado di cambiare directory e accedere alla directory home per eseguire i vostri script.

Installazione di un editor di testo

Per iniziare a realizzare script bash, avrete bisogno di un editor di testo, preferibilmente dotato di funzionalità utili come l'evidenziazione della sintassi. Potete scegliere fra editor di testo basati su terminale e a interfaccia utente grafica. Gli editor di testo basati su terminale (come vi e GNU nano) sono utili, perché durante un penetration test potrebbero essere le uniche opzioni disponibili quando avete bisogno di sviluppare uno script sul momento.
Se preferite gli editor di testo grafici, Sublime Text (`https://www.sublimetext.com`) è un'opzione interessante. In Sublime Text, potete attivare la funzionalità di evidenziazione della sintassi per gli script bash facendo clic su *Plain Text* nell'angolo inferiore destro e scegliendo *Bash* dall'elenco a discesa dei linguaggi. Se state usando un altro editor di testo, fate riferimento alla sua documentazione ufficiale per scoprire come attivare l'evidenziazione della sintassi bash.

Esplorazione della shell

Ora che avete un ambiente bash operativo, è il momento di imparare alcune basi. Certamente svilupperete gli script nel vostro editor di testo, ma probabilmente vi ritroverete anche a eseguire frequentemente singoli comandi nel terminale. Questo perché, spesso, avrete bisogno di vedere come viene eseguito un comando e il tipo di output che produce prima di includerlo in uno script. Cominciamo eseguendo alcuni comandi della shell bash. Innanzitutto, immettete il seguente comando, per verificare che la shell bash sia disponibile sul vostro sistema:

```
$ bash --version
```

La versione in output dipende dal sistema operativo in uso.

Controllo delle variabili d'ambiente

Quando viene eseguita in un terminale, la shell bash carica un set di *variabili d'ambiente* a ogni nuova sessione. I programmi possono usare queste variabili d'ambiente per vari scopi, come scoprire l'identità dell'utente che esegue lo script, la posizione della sua home directory e la sua shell di default.
Per visualizzare l'elenco delle variabili d'ambiente impostate da bash, eseguire il comando `env` direttamente dalla shell (Listato 1.1).

Listato 1.1 Elenco delle variabili d'ambiente della shell bash.

```
$ env

SHELL=/bin/bash
LANGUAGE=en_CA:en
DESKTOP_SESSION=ubuntu
PWD=/home/user
--snip--
```

Potete leggere il contenuto delle singole variabili d'ambiente usando il comando `echo`, che invia l'output sul terminale. Per esempio, per mostrare la shell di default dell'utente, usate la variabile d'ambiente `SHELL` preceduta da un simbolo di dollaro (`$`) e circondata da parentesi graffe (`{}`). Questo farà sì che la shell bash espanda la variabile mostrando il valore che le è stato assegnato, come mostrato nel Listato 1.2.

Listato 1.2 Visualizzazione del valore di una variabile d'ambiente sul terminale.

```
$ echo ${SHELL}

/bin/bash
```

Ecco alcune delle variabili d'ambiente di default disponibili.

- `BASH_VERSION` La versione della shell bash in esecuzione.
- `BASHPID` L'identificatore di processo (PID) del processo bash corrente.
- `GROUPS` Un elenco dei gruppi cui appartiene l'utente corrente.
- `HOSTNAME` Il nome dell'host.
- `OSTYPE` Il tipo di sistema operativo.
- `PWD` La directory di lavoro corrente.
- `RANDOM` Un numero casuale da 0 a 32.767.
- `UID` L'ID utente (UID) dell'utente corrente.
- `SHELL` Il percorso completo della shell.

Gli esempi seguenti mostrano come controllare i valori di alcune di queste variabili d'ambiente:

```
$ echo ${RANDOM}
8744

$ echo ${UID}
1000

$ echo ${OSTYPE}
linux-gnu
```

Questi comandi generano, rispettivamente, un numero casuale, l'ID dell'utente corrente e il tipo di sistema operativo. Potete trovare l'elenco completo delle variabili d'ambiente su `https://www.gnu.org/software/bash/manual/html_node/Bash-Variables.html`.

Esecuzione di comandi Linux

Gli script bash che scriverete in questo libro impiegheranno comuni strumenti Linux, quindi se non avete ancora dimestichezza con i comandi di navigazione dalla riga di comando e le utility di editing dei file, come `cd`, `ls`, `chmod`, `mkdir` e `touch`, provate a esplorarli usando il comando `man` (*manual*). Potete specificarlo prima di qualsiasi comando Linux per aprire una guida a terminale che spiega l'uso e le opzioni di quel comando, come mostrato nel Listato 1.3.

Listato 1.3 Accesso alla pagina man di un comando.

```
$ man ls

NAME
      ls - list directory contents

SYNOPSIS
      ls [OPTION]... [FILE]...

DESCRIPTION
      List information about the FILEs (the current directory by default).
      Sort entries alphabetically if none of -cftuvSUX nor
      --sort is specified.
      Mandatory arguments to long options are mandatory for short options too.
      -a, --all
      do not ignore entries starting with.
--snip--
```

I comandi Linux possono accettare molti tipi di input sulla riga di comando. Per esempio, potete digitare `ls` senza argomenti per vedere i file e le directory, o passargli argomenti per esempio, per visualizzare l'elenco dei file tutti su una riga.
Gli argomenti vengono passati sulla riga di comando utilizzando una sintassi breve o lunga, a seconda del comando in questione. La sintassi *breve* utilizza un singolo trattino

(-) seguito da uno o più caratteri. Il seguente esempio utilizza `ls` per elencare i file e le directory con la sintassi breve degli argomenti:

```
$ ls -l
```

Alcuni comandi consentono di fornire più argomenti unendoli insieme oppure elencandoli separatamente:

```
$ ls -la
$ ls -l -a
```

Tenete presente che alcuni comandi potrebbero generare errori, se provate a unire due argomenti dopo un singolo trattino, quindi usate il comando `man` per scoprire la sintassi consentita.
Alcune opzioni dei comandi potrebbero permettere di utilizzare una sintassi in *formato lungo*, come il comando `--help` per elencare le opzioni disponibili. La sintassi degli argomenti in formato lungo è preceduta dal simbolo del trattino doppio (--):

```
$ ls --help
```

A volte, per comodità, lo stesso argomento di un comando supporta sia la sintassi breve sia quella lunga. Per esempio, `ls` supporta l'argomento `-a` (*all*) per visualizzare tutti i file, inclusi quelli nascosti (i file il cui nome inizia con un punto sono considerati nascosti in Linux). Tuttavia, potreste anche passare l'argomento `--all` e il risultato sarebbe identico:

```
$ ls -a
$ ls --all
```

Eseguiamo alcuni semplici comandi Linux per vedere le varianti delle opzioni. Innanzitutto, create una directory con `mkdir`:

```
$ mkdir directory1
```

Ora create due directory con `mkdir`:

```
$ mkdir directory2 directory3
```

Quindi, elencate i processi utilizzando `ps` con la sintassi breve degli argomenti, fornendo gli argomenti prima separatamente e poi insieme:

```
$ ps -e -f
$ ps -ef
```

Infine, visualizzate lo spazio disponibile su disco, utilizzando `df` con la sintassi lunga degli argomenti:

```
$ df –human-readable
```

In questo libro utilizzerete questo genere di comandi Linux nei vostri script.

Gli elementi di uno script bash

In questo paragrafo, scoprirete i blocchi costitutivi di uno script bash. Utilizzerete i commenti per documentare ciò che fa uno script, direte a Linux di usare un determinato interprete per eseguire lo script e formatterete i vostri script per migliorarne la leggibilità. La shell bash non ha una guida ufficiale per lo stile, ma vi suggeriamo di attenervi alla *Shell Style Guide* di Google (`https://google.github.io/styleguide/shellguide.html`), che delinea le *best practice* da seguire quando si sviluppa codice bash. Se lavorate in un team di penetration tester e avete un repository di codice per exploit, l'uso di buone pratiche di stile nella scrittura di codice aiuterà anche il vostro team.

La riga shebang

Ogni script deve iniziare con la *riga shebang*, una sequenza di caratteri che inizia con il cancelletto e il punto esclamativo (`#!`), seguita dal percorso completo dell'interprete di script. Il Listato 1.4 mostra un esempio di riga shebang per un tipico script bash.

Listato 1.4 Una riga shebang per bash.

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

L'interprete bash, in genere, si trova in `/bin/bash`. Se invece lo script fosse in Python o Ruby, la vostra riga shebang dovrà includere il percorso completo all'interprete Python o Ruby. A volte incontrerete script bash che utilizzano una riga shebang come questa:

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

Potreste voler usare questa riga shebang perché è più portatile di quella del Listato 1.4. Alcune distribuzioni Linux collocano l'interprete bash in posizioni diverse del sistema e questa riga shebang tenterà di trovare quella posizione. Questo approccio potrebbe essere particolarmente utile nel penetration testing, in cui potreste non conoscere la posizione dell'interprete bash sulla macchina target. Per semplicità, tuttavia, in tutto il libro useremo la versione shebang del Listato 1.4.
La riga shebang può anche accettare argomenti opzionali, per cambiare il modo in cui lo script deve essere eseguito. Per esempio, potreste passare l'argomento speciale `-x`, in questo modo:

```
#!/bin/bash -x
```

Questa opzione mostra tutti i comandi e i loro argomenti mentre vengono eseguiti sul terminale. È utile per il debug degli script mentre li sviluppate.
Un altro esempio di argomento opzionale è `-r`:

```
#!/bin/bash -r
```

Questa opzione crea una *shell bash limitata*, che blocca determinati comandi potenzialmente pericolosi che potrebbero, per esempio, far accedere a determinate directory, modificare variabili d'ambiente sensibili o tentare di disattivare la shell limitata dall'interno dello script.
Per specificare un argomento all'interno della riga shebang è necessario modificare lo script, ma è anche possibile passare argomenti all'interprete bash utilizzando questa sintassi:

```
$ bash -r myscript.sh
```

Non fa differenza se passate gli argomenti all'interprete bash sulla riga di comando o sulla riga shebang. L'opzione della riga di comando è solo un modo più semplice per attivare diverse modalità.

Commenti

I *commenti* sono parti di uno script che l'interprete bash non tratta come codice e che possono migliorare la leggibilità di un programma. Immaginate di preparare un lungo script e di dovervi tornare a lavorare qualche anno dopo. Se non avete scritto commenti per spiegare le vostre scelte, potrebbe essere piuttosto difficile ricordare lo scopo di ogni sezione.
I commenti, nella shell bash, iniziano con un cancelletto (#), come illustrato nel Listato 1.5.

Listato 1.5 Un commento in uno script bash.

```
#!/bin/bash

# Questo è il mio primo script.
```

A parte la riga shebang, ogni riga che inizia con un cancelletto è considerata un commento. Se provaste a scrivere due volte la riga shebang, la shell bash considererebbe la seconda come un commento.
Per scrivere un commento su più righe, dovete anteporre a ogni riga il carattere cancelletto, come vediamo nel Listato 1.6.

Listato 1.6 Un commento su più righe.

```
#!/bin/bash

# Questo è il mio primo script!
# Fare script per la shell bash è divertente
```

Oltre a documentare la logica di uno script, i commenti possono fornire metadati per indicare l'autore, la versione dello script, chi contattare in caso di problemi e molto altro ancora. Questi commenti, di solito, appaiono nella parte superiore dello script, sotto la riga shebang.

Comandi

Gli script possono essere brevi, di due sole righe: la riga shebang e un comando Linux. Prepariamo un piccolo script che mostri `Hello World!` sul terminale. Aprite l'editor di testo e inserite quanto segue:

```
#!/bin/bash

echo "Hello World!"
```

In questo esempio, utilizziamo l'istruzione shebang per specificare l'interprete scelto, bash. Quindi utilizziamo il comando `echo` per mostrare sullo schermo la stringa `Hello World!`.

Esecuzione

Per eseguire lo script, salvate il file come `helloworld.sh`, aprite il terminale e andate alla directory in cui avete collocato lo script. Se avete salvato il file nella vostra directory home, dovete eseguire i comandi presentati nel Listato 1.7.

Listato 1.7 Esecuzione di uno script dalla directory home.

```
$ cd ~
$ chmod u+x helloworld.sh
$ ./helloworld.sh

Hello World!
```

Usiamo il comando `cd` per cambiare directory. La tilde (`~`) rappresenta la directory home dell'utente corrente. Poi, usiamo `chmod` per impostare le autorizzazioni eseguibili (`u+x`) per l'utente proprietario del file (in questo caso, noi). Eseguiamo lo script usando la notazione punto-barra (`./`) seguita dal nome dello script. Il punto (`.`) rappresenta la directory corrente, quindi stiamo essenzialmente chiedendo alla shell bash di eseguire il file `helloworld.sh` che si trova nella directory di lavoro corrente.
Potete eseguire uno script bash anche con la seguente sintassi:

```
$ bash helloworld.sh
```

Poiché abbiamo specificato il comando `bash`, lo script verrà eseguito utilizzando l'interprete bash e non richiederà una riga shebang. Inoltre, se utilizziamo il comando `bash`, lo script non deve essere impostato con un'autorizzazione all'esecuzione (`+x`). Nei prossimi capitoli, esamineremo più in dettaglio il modello delle autorizzazioni ed esploreremo la sua importanza nel contesto della ricerca di configurazioni errate nel penetration testing.

Debug

Gli errori si verificheranno inevitabilmente nello sviluppare script bash. Fortunatamente, il debug degli script è piuttosto intuitivo. Un modo semplice per controllare gli errori in anticipo consiste nell'usare il parametro `-n` quando si esegue uno script:

```
$ bash -n script.sh
```

Questo parametro legge i comandi presenti nello script ma senza eseguirli, quindi mostra sullo schermo eventuali errori di sintassi. Potete considerare `-n` come un metodo per verificare la validità della sintassi che avete usato.
Potete impiegare anche il parametro `-x`, per attivare la modalità dettagliata, che vi consente di vedere i comandi in esecuzione e vi aiuta a risolvere i problemi mentre lo script viene eseguito in tempo reale:

```
$ bash -x script.sh
```

Se volete avviare il debug da un punto specifico dello script, includete nello script il comando `set` (Listato 1.8).

Listato 1.8 Utilizzo di set per il debug di uno script.

```
#!/bin/bash
set -x

--snip--

set +x
```

Potete considerare `set` come una "valvola" che attiva e disattiva una determinata opzione. In questo esempio, il primo comando imposta la modalità di debug (`set -x`), mentre l'ultimo comando (`set +x`) la disattiva. Utilizzando `set`, potete evitare di generare una quantità enorme di rumore sul terminale quando il vostro script è grande ma contiene un'area problematica ben delimitata, da sottoporre a test.

Sintassi di base

A questo punto, avete scritto uno script di due righe che mostra sullo schermo il messaggio `Hello World!`. Avete anche imparato a eseguire e fare il debug di uno script. Ora imparerete un po' di sintassi della shell bash, in modo da poter realizzare script più corposi. Gli script bash più semplici sono semplicemente elenchi di comandi Linux raccolti in un file. Per esempio, potreste realizzare uno script che crea risorse su un sistema e poi mostra informazioni su queste risorse (Listato 1.9).

Listato 1.9 Uno script bash che elenca il contenuto di una directory.

```
#!/bin/bash

# Questo script crea una directory, crea un file
# in tale directory e poi elenca il contenuto di tale directory.

mkdir mydirectory
touch mydirectory/myfile
ls -l mydirectory
```

In questo esempio, utilizziamo `mkdir` per creare la directory `mydirectory`. Quindi, utilizziamo il comando `touch` per creare il file `myfile` all'interno di tale directory. Infine, eseguiamo il comando `ls -l` per elencare il contenuto di `mydirectory`.
L'output dello script ha il seguente aspetto

```
--snip--
-rw-r--r-- 1 user user 0 Feb 16 13:37 myfile
```

Tuttavia, questa strategia riga per riga può essere migliorata in diversi modi. Innanzitutto, quando viene eseguito un comando, la shell bash attende che termini prima di passare alla riga successiva. Se si include un comando con una lunga durata (per esempio di download di un file o di copia di grandi quantità di file), i comandi rimanenti non verranno eseguiti finché il comando in corso non sarà completato. Inoltre, dobbiamo ancora implementare dei controlli per verificare che tutti i comandi siano stati eseguiti correttamente. Dovrete scrivere programmi più intelligenti per ridurre gli errori runtime. Scrivere programmi sofisticati richiede l'uso di funzionalità come variabili, condizioni, cicli e test. Per esempio, che cosa succederebbe se volessimo modificare questo script in modo che controlli che ci sia abbastanza spazio sul disco prima di tentare di creare nuovi file e directory? O se volessimo controllare se le operazioni di creazione della directory e del file siano effettivamente riuscite? Questo paragrafo e il Capitolo 2 presentano gli elementi sintattici di cui avrete bisogno per svolgere queste attività.

Variabili

Ogni linguaggio per script usa delle variabili. Le *variabili* sono nomi che assegniamo a certe posizioni di memoria e che contengono un valore, agendo quindi come segnaposto o etichette. Alle variabili possiamo assegnare un valore direttamente oppure possiamo eseguire dei comandi bash e memorizzare nelle variabili il loro output, da usare poi per vari scopi.
Se avete già utilizzato dei linguaggi di programmazione, saprete che le variabili possono essere di vari tipi: numeri interi, stringhe e array. Nella shell bash, le variabili non sono tipizzate; sono tutte stringhe di caratteri. Ciononostante, la shell bash consente di creare array, di accedere agli elementi degli array e di eseguire operazioni aritmetiche, purché il valore della variabile sia costituito solo da cifre.
Per la denominazione delle variabili della shell bash occorre considerare le seguenti regole.

- I nomi possono includere caratteri alfanumerici.
- Non possono iniziare con un numero.
- Possono contenere un carattere di sottolineatura (_).
- Non possono contenere spazi.

Assegnamento e accesso alle variabili

Proviamo a eseguire un assegnamento a una variabile. Aprite un terminale e digitate quanto segue direttamente al prompt dei comandi:

```
$ book="black hat bash"
```

Stiamo creando la variabile denominata `book` e, usando il segno di uguale (`=`), le assegniamo il valore `black hat bash`. Ora possiamo usare questa variabile in un comando. Nell'esempio che segue, usiamo il comando `echo` per mostrare sullo schermo il contenuto di tale variabile:

```
$ echo "This book's name is ${book}"

This book's name is black hat bash
```

Siamo stati in grado di mostrare il contenuto della variabile usando la sintassi `${book}` all'interno di un comando `echo`. Questo ha l'effetto di espandere la variabile `book`, fornendo il suo valore. Potete espandere una variabile anche usando solo il simbolo del dollaro (`$`) seguito dal nome della variabile:

```
$ echo "This book's name is $book"
```

Utilizzando la sintassi `${}` il codice diventa meno soggetto a interpretazioni errate e aiuta chi legge a capire quando inizia e finisce il nome di una variabile.
Potete anche assegnare a una variabile l'output di un comando, usando la sintassi di sostituzione dei comandi `$()`, specificando fra parentesi il comando desiderato. Utilizzerete spesso questa sintassi nella programmazione bash. Provate a eseguire i comandi contenuti nel Listato 1.10.

Listato 1.10 Assegnamento dell'output del comando a una variabile.

```
$ root_directory=$(ls -ld /)
$ echo "${root_directory}"

drwxr-xr-x 1 user user 0 Feb 13 20:12 /
```

Qui assegniamo alla variabile `root_directory` il valore del comando `ls -ld /` e poi utilizziamo `echo` per mostrare l'output del comando. In questo output, potete vedere che siamo stati in grado di ottenere i metadati della directory radice (`/`), come il suo tipo e le autorizzazioni, le dimensioni, i proprietari (utenti e gruppi) e il timestamp dell'ultima modifica.
Fate attenzione: non dovete lasciare spazi vuoti attorno al simbolo di assegnamento (`=`) quando create una variabile:

```
book = "questo assegnamento non funziona "
```

La precedente sintassi di assegnamento delle variabili non è valida.

Annullamento dell'assegnamento delle variabili

Per annullare un precedente assegnamento a una variabile si utilizza il comando `unset`, come illustrato nel Listato 1.11.

Listato 1.11 Annullamento dell'assegnamento di una variabile.

```
$ book="Black Hat Bash"
$ unset book
$ echo "${book}"
```

Se si eseguono questi comandi nel terminale, l'esecuzione del comando `echo` non produrrà alcun output.

Ambito (visibilità) delle variabili

Le variabili *globali* sono disponibili nell'intero programma. Ma nella shell bash l'ambito d'uso (*scope*) e quindi la "visibilità" delle variabili può essere limitato in modo che esse risultino accessibili solo da un determinato blocco di codice. Queste variabili *locali* vanno dichiarate usando la parola chiave `local`. Lo script del Listato 1.12 mostra come funzionano le variabili locali e globali.

Listato 1.12 Accesso alle variabili globali e locali (Local_scope_variable.sh).

```
#!/bin/bash

PUBLISHER="No Starch Press"

print_name(){
    local name
    name="Black Hat Bash"
    echo "${name} by ${PUBLISHER}"
}

print_name

echo "Variable ${name} will not be printed because it is a local variable."
```

Assegniamo il valore `No Starch Press` alla variabile `PUBLISHER` e poi creiamo la funzione `print_name()` (esamineremo le funzioni nel prossimo capitolo). All'interno della funzione, dichiariamo la variabile locale `name` e le assegniamo il valore `Black Hat Bash`. Quindi richiamiamo `print_name()` e proviamo ad accedere alla variabile `name` come parte di una frase che vogliamo mostrare usando `echo`.
Il comando `echo` alla fine del file script produrrà una variabile vuota, poiché la variabile `name` è locale e limitata alla funzione `print_name()`, il che significa che nulla al di fuori di tale funzione può accedervi. Quindi, la funzione si chiuderà senza restituire alcun valore.

> **NOTA**
>
> Gli script di questo capitolo sono disponibili su `https://github.com/dolevf/Black-Hat-Bash/blob/master/ch01`.

Salvate questo script, ricordandovi di impostare le autorizzazioni di esecuzione tramite `chmod`, ed eseguitelo utilizzando il seguente comando:

```
$ ./local_scope_variable.sh

Black Hat Bash by No Starch Press

Variable  will not be printed here because it is a local variable
```

Come potete vedere, la variabile locale `name` non viene visualizzata, fuori dalla funzione.

Operatori aritmetici

Gli *operatori aritmetici* consentono di eseguire operazioni matematiche sui numeri interi. La Tabella 1.1 mostra alcuni degli operatori aritmetici disponibili. Per l'elenco completo, vedere `https://tldp.org/LDP/abs/html/ops.html`.

Tabella 1.1 Gli operatori aritmetici.

Operatore	Descrizione
+	Addizione
-	Sottrazione
*	Moltiplicazione
/	Divisione
%	Modulo
+=	Incremento di una costante
-=	Decremento di una costante

Potete eseguire queste operazioni aritmetiche in diversi modi: utilizzando il comando `let`, utilizzando la sintassi delle doppie parentesi `$((espressione))`, oppure utilizzando il comando `expr`. Consideriamo un esempio di ciascun metodo.
Nel Listato 1.13 eseguiamo un'operazione di moltiplicazione utilizzando il comando `let`.

Listato 1.13 Operazione aritmetica con let.

```
$ let result="4 * 5"
$ echo ${result}

20
```

Questo comando prende un nome di variabile ed esegue un calcolo aritmetico per determinarne il valore.
Nel Listato 1.14, eseguiamo un'altra operazione di moltiplicazione utilizzando la sintassi delle doppie parentesi.

Listato 1.14 Operazione aritmetica con la sintassi delle doppie parentesi.

```
$ result=$((5 * 5))
$ echo ${result}

25
```

In questo caso, eseguiamo il calcolo posto fra le doppie parentesi.
Infine, nel Listato 1.15, eseguiamo un'operazione di addizione usando il comando `expr`.

Listato 1.15 Valutazione di un'espressione con expr.

```
$ result=$(expr 5 + 505)
$ echo ${result}

510
```

Il comando `expr` valuta le espressioni, che non necessariamente sono operazioni aritmetiche; per esempio, potreste usarlo per calcolare la lunghezza di una stringa. Usate `man expr` per saperne di più sulle funzionalità di `expr`.

Array

La shell bash consente di creare array monodimensionali. Un *array* è una raccolta di elementi cui è associato un indice. Potete accedere a questi elementi utilizzando il loro indice, che inizia da zero. Negli script bash, potete utilizzare gli array ogni volta che è necessario scorrere più stringhe ed eseguire gli stessi comandi su ciascuna di esse.
Il Listato 1.16 mostra come creare un array nella shell bash. Salvate questo codice nel file `array.sh` ed eseguitelo.

Listato 1.16 Creazione e accesso a un array.

```
#!/bin/bash

# Imposta l'array
IP_ADDRESSES=(192.168.1.1 192.168.1.2 192.168.1.3)

# Mostra tutti gli elementi dell'array
echo "${IP_ADDRESSES[*]}"

# Mostra solo il primo elemento dell'array
echo "${IP_ADDRESSES[0]}"
```

Questo script usa l'array `IP_ADDRESSES` che contiene tre indirizzi IP (*Internet Protocol*). Il primo comando `echo` mostra tutti gli elementi contenuti nell'array passando `[*]` alla variabile `IP_ADDRESSES`, che contiene i valori dell'array. L'asterisco (*) rappresenta ogni elemento dell'array. Infine, un altro comando `echo` mostra solo il primo elemento nell'array, specificando l'indice `0`.
L'esecuzione di questo script dovrebbe produrre il seguente output:

```
$ chmod u+x array.sh
$ ./array.sh

192.168.1.1 192.168.1.2 192.168.1.3
192.168.1.1
```

Come potete vedere, siamo riusciti a far sì che la shell bash mostrasse tutti gli elementi contenuti nell'array, e poi solo il primo elemento.
È anche possibile eliminare elementi da un array. Il Listato 1.17 elimina dall'array l'elemento `192.168.1.2`.

Listato 1.17 Eliminazione di elementi da un array.

```
IP_ADDRESSES=(192.168.1.1 192.168.1.2 192.168.1.3)

unset IP_ADDRESSES[1]
```

Potete anche sostituire uno dei valori con un altro valore. Il seguente codice sostituisce `192.168.1.1` con `192.168.1.10`:

```
IP_ADDRESSES[0]="192.168.1.10"
```

Gli array sono particolarmente utili quando è necessario scorrere più valori ed eseguire azioni su di essi, per esempio quando vi è un elenco di indirizzi IP da analizzare (o un elenco di indirizzi e-mail cui inviare un'e-mail di phishing).

Stream

Gli *stream* sono file che fungono da canali di comunicazione fra un programma e il suo ambiente. Quando interagite con un programma (che sia un'utility Linux come `ls` o `mkdir` o un programma che avete scritto voi stessi), state interagendo con uno o più stream. La shell bash ha tre stream di dati standard, come mostra la Tabella 1.2.

Tabella 1.2 Gli stream.

Nome stream	Descrizione	Numero descrittore del file
Input standard (stdin)	Dati che entrano in un programma come input	0
Standard output (stdout)	Dati in uscita da un programma	1
Standard error (stderr)	Errori in uscita da un programma	2

Finora abbiamo eseguito alcuni comandi dal terminale e abbiamo scritto ed eseguito uno script semplice. L'output che ha generato è stato inviato allo *stream di output standard* (*stdout*) o, in altre parole, alla schermata del terminale.
Gli script possono anche ricevere un input. Quando uno script è progettato per ricevere input, lo legge dallo *stream di input standard* (*stdin*). Infine, gli script possono visualizzare messaggi di errore sullo schermo, a causa di un bug o di un errore di sintassi nei comandi. Questi messaggi vengono inviati allo *stream di errore standard* (*stderr*).
Per illustrare il comportamento degli stream, useremo il comando `mkdir` per creare alcune directory e poi useremo `ls` per elencare il contenuto della directory corrente. Aprite il terminale e lanciate il seguente comando:

```
$ mkdir directory1 directory2 directory1
mkdir: cannot create directory 'directory1': File exists
```

```
$ ls -l
total 1
drwxr-xr-x 1 user user   0 Feb 17 09:45 directory1
drwxr-xr-x 1 user user   0 Feb 17 09:45 directory2
```

Notate che `mkdir` genera un errore. Questo perché passiamo il nome della directory `directory1` per due volte sulla riga di comando. Quindi, quando viene eseguito il comando `mkdir`, crea `directory1` e `directory2`, quindi fallisce sul terzo argomento perché, a quel punto, la `directory1` esiste già. Questi tipi di errori vengono inviati allo stream di errore standard. Poi eseguiamo `ls -l`, che elenca semplicemente le directory. Il risultato del comando `ls` ha esito positivo e senza errori, quindi viene inviato allo stream di output standard. Vi eserciterete a lavorare con lo stream di input standard quando introdurremo la redirezione, più avanti in questo stesso capitolo.

Operatori di controllo

Gli *operatori di controllo*, nella shell bash, sono elementi che eseguono una funzione di controllo sugli elementi delle istruzioni. La Tabella 1.3 ne fornisce una panoramica.

Tabella 1.3 Gli operatori di controllo di bash.

Operatore	Descrizione
`&`	Pone un comando in background.
`&&`	AND logico. Il secondo comando nell'espressione verrà eseguito solo se il primo comando risulta vero.
`( e )`	Parentesi utilizzate per il raggruppamento dei comandi.
`;`	Utilizzato come terminatore di una lista. Un comando che segue il terminatore verrà eseguito dopo che è terminato il comando precedente, indipendentemente dal fatto che venga valutato come vero o meno.
`;;`	Termina un'istruzione `case`.
`|`	Reindirizza l'output di un comando all'input per un altro comando.
`||`	OR logico. Il secondo comando verrà eseguito anche se il primo comando risulta falso.

Vediamo alcuni di questi operatori di controllo in azione. L'operatore `&` pone un comando in background. Se avete un elenco di comandi da eseguire, come vediamo nel Listato 1.18, l'invio del primo comando in background permetterà alla shell bash di procedere alla riga successiva, anche se il comando precedente non ha terminato il proprio lavoro.

Listato 1.18 Un comando viene posto in background, in modo che l'esecuzione possa procedere alla riga successiva.

```
#!/bin/bash

# Questo script pone in background il comando sleep.
echo "Sleeping for 10 seconds..."
sleep 10 &  ❶
```

```
# Crea un file
echo "Creating the file test123"
touch test123

# Elimina un file
echo "Deleting the file test123"
rm test123
```

Spesso i comandi di lunga durata vengono posti in background per evitare che gli script si blocchino ❶. Imparerete a porre i comandi in background quando parleremo del controllo dei job, nel Capitolo 2.
L'operatore `&&` consente di eseguire un'operazione AND fra due comandi. Nell'esempio che segue, il file `test123` verrà creato solo se il primo comando ha esito positivo:

```
touch test && touch test123
```

L'operatore `()` consente di raggruppare i comandi, in modo che agiscano come un'unica unità quando dobbiamo redirigerli insieme:

```
(ls; ps)
```

In genere, ciò risulta utile quando è necessario redirigere i risultati di più comandi a uno stream, come vedremo quando, fra poco, parleremo degli operatori di redirezione.
L'operatore ; consente di eseguire più comandi indipendentemente dal loro stato di uscita:

```
ls; ps; whoami
```

Di conseguenza, ogni comando verrà eseguito dopo il precedente, non appena esso termina.
L'operatore `||` consente di concatenare i comandi utilizzando un'operazione OR:

```
lzl || echo "the lzl command failed"
```

In questo esempio, il comando `echo` verrà eseguito solo se il primo comando fallisce.

Operatori di redirezione

I tre stream standard di cui abbiamo parlato possono essere rediretti da un programma a un altro. La *redirezione* consiste nel prendere l'output di un comando o script e utilizzarlo come input per un altro comando o script o scritto su file. La Tabella 1.4 descrive gli operatori di redirezione disponibili.
Facciamo pratica con gli operatori di redirezione per vedere come funzionano con gli stream standard. L'operatore > reindirizza lo stream di output standard su un file. Qualsiasi comando che precede questo carattere invierà il proprio output al file specificato. Provate a lanciare il seguente comando direttamente nel terminale:

```
$ echo "Hello World!" > output.txt
```

Tabella 1.4 Gli operatori di redirezione.

Operatore	Descrizione
>	Reindirizza lo stdout su un file.
>>	Reindirizza lo stdout su un file aggiungendolo al contenuto esistente.
&> o >&	Reindirizza lo stdout e lo stderr su un file.
&>>	Reindirizza lo stdout e lo stderr su un file aggiungendoli al contenuto esistente.
<	Reindirizza l'input a un comando.
<<	Chiamato *here document* o *heredoc*, reindirizza più righe di input a un comando.
|	Reindirizza l'output di un comando all'input di un altro comando.

Qui redirigiamo lo stream di output standard al file `output.txt`. Per vedere il contenuto di `output.txt`, eseguite semplicemente quanto segue:

```
$ cat output.txt

Hello World!
```

Successivamente, utilizzeremo l'operatore `>>` per aggiungere nuovi contenuti alla fine dello stesso file (Listato 1.19).

Listato 1.19 Aggiunta di nuovi contenuti a un file.

```
$ echo "Goodbye!" >> output.txt
$ cat output.txt

Hello World!
Goodbye!
```

Se avessimo utilizzato `>` invece di `>>`, il contenuto di `output.txt` sarebbe stato completamente sovrascritto con il testo `Goodbye!`.
Per redirigere su un file sia lo stream di output standard sia lo stream di errore standard, usate l'operatore `&>`. Questo è utile quando non volete inviare alcun output allo schermo, ma salvare tutto su un file log (magari per una successiva analisi):

```
$ ls -l / &> stdout_and_stderr.txt
```

Per aggiungere a un file entrambi gli stream di output standard e errore standard, utilizzate l'operatore `&>>`.
Come fare per inviare lo stream di output standard su un file e lo stream di errore standard a un altro? Potete farlo utilizzando i numeri dei descrittori di file degli stream:

```
$ ls -l / 1> stdout.txt 2> stderr.txt
```

A volte può essere utile redirigere lo stream di errore standard su un file, come abbiamo fatto qui, in modo da poter registrare qualsiasi errore che si verifica durante l'esecuzione. Il prossimo esempio esegue un comando inesistente, `lzl`. Questo dovrebbe generare degli errori della shell bash, che saranno scritti nel file `error.txt`:

```
$ lzl 2> error.txt
$ cat error.txt

bash: lzl: command not found
```

Notate che l'errore non viene visualizzato sullo schermo, perché bash invia l'errore al file. Ora, usiamo lo stream di input standard. Eseguiamo nella shell il comando riportato nel Listato 1.20 per fornire il contenuto di `output.txt` come input per il comando `cat`.

Listato 1.20 Utilizzo di un file come input di un comando.

```
$ cat < output.txt

Hello World!
Goodbye!
```

E se a un comando volessimo redirigere più righe? Può aiutarci la redirezione del documento (`<<`), Listato 1.21.

Listato 1.21 Redirezione di un documento.

```
$ cat << EOF
  Black Hat Bash
  by No Starch Press
EOF

Black Hat Bash
by No Starch Press
```

In questo esempio, passiamo più righe come input a un comando. L'`EOF` in questo esempio funge da delimitatore, contrassegnando i punti di inizio e fine dell'input. Qui la *redirezione di un documento* tratta l'input come se fosse un file, preservando i fine riga e gli spazi.
L'operatore *pipe* (`|`) reindirizza l'output di un comando e lo usa come input di un altro comando. Per esempio, potremmo eseguire il comando `ls` sulla directory root e poi usare un altro comando per estrarre dati dall'output prodotto da `ls`, come mostrato nel Listato 1.22.

Listato 1.22 Inoltro dell'output del comando a un altro comando.

```
$ ls -l / | grep "bin"

lrwxrwxrwx   1 root root            7 Mar 10 08:43 bin -> usr/bin
lrwxrwxrwx   1 root root            8 Mar 10 08:43 sbin -> usr/sbin
```

Utilizziamo `ls` per produrre il contenuto della directory radice nello stream di output standard, quindi utilizziamo una pipe per inviare l'output come input al comando `grep`, che ne estrae tutte le righe che contengono la parola `bin`.

Argomenti posizionali

Gli script bash possono accettare *argomenti posizionali* (chiamati anche *parametri*) passati sulla riga di comando. Gli argomenti sono particolarmente utili, per esempio, quando occorre sviluppare un programma che modifichi il proprio comportamento in base all'input passatogli da un altro programma o dall'utente. Gli argomenti possono anche modificare le funzionalità dello script, come il formato di output e quanto dovrà essere dettagliato durante l'esecuzione.
Per esempio, immaginate di sviluppare un exploit e di inviarlo ad alcuni colleghi, ognuno dei quali lo utilizzerà su un indirizzo IP diverso. Invece di realizzare uno script e chiedere all'utente di specificare le informazioni relative alla propria rete, potete scriverlo in modo che prenda come argomento un indirizzo IP e poi operi su questo input, per evitare di dover far modificare il codice sorgente per ogni singolo caso.
Uno script bash può accedere agli argomenti passati sulla riga di comando usando le variabili `$1`, `$2` e così via. Il numero rappresenta l'ordine in cui è stato specificato l'argomento. Per illustrare la cosa, lo script del Listato 1.23 accetta un argomento (un indirizzo IP o un nome di dominio) ed esegue su di esso un test ping usando l'utility `ping`. Salvate questo file come `ping_with_arguments.sh`.

Listato 1.23 Uno script che accetta input dalla riga di comando (ping_with_arguments.sh).

```
#!/bin/bash

# Questo script esegue un ping sull'indirizzo fornito come argomento.
SCRIPT_NAME="${0}"
TARGET="${1}"

echo "Running the script ${SCRIPT_NAME}..."
echo "Pinging the target: ${TARGET}..."
ping "${TARGET}"
```

Questo script assegna il primo argomento posizionale alla variabile `TARGET`. Notate, inoltre, che l'argomento `${0}` viene assegnato alla variabile `SCRIPT_NAME`. Questo "argomento" contiene il nome dello script (in questo caso, `ping_with_arguments.sh`).
Per eseguire questo script, utilizzate i comandi contenuti nel Listato 1.24.

Listato 1.24 Passaggio di argomenti a uno script.

```
$ chmod u+x ping_with_arguments.sh
$ ./ping_with_arguments.sh nostarch.com

Running the script ping_with_arguments.sh...
Pinging the target nostarch.com . ..
PING nostarch.com (104.20.120.46) 56(84) bytes of data.

64 bytes from 104.20.120.46 (104.20.120.46): icmp_seq=1 ttl=57 time=6.89 ms
64 bytes from 104.20.120.46 (104.20.120.46): icmp_seq=2 ttl=57 time=4.16 ms
--snip--
```

Questo script eseguirà un comando `ping` sul dominio `nostarch.com` passatogli sulla riga di comando. Il valore è assegnato alla variabile `$1`; se passassimo un secondo argomento, verrebbe assegnato alla seconda variabile, `$2`. Usate Ctrl-C per uscire da questo script, poiché il `ping` potrebbe essere eseguito indefinitamente su alcuni sistemi operativi.
E se volessimo accedere a tutti gli argomenti? Potete farlo usando la variabile `$@`. Inoltre, usando `$#`, potete ottenere il numero totale di argomenti passati. Il Listato 1.25 ne illustra il funzionamento.

Listato 1.25 Recupero di tutti gli argomenti e del numero totale di argomenti.

```
#!/bin/bash

echo "The arguments are: $@"
echo "The total number of arguments is: $#"
```

Salvate questo script nel file `show_args.sh` ed eseguitelo come segue:

```
$ chmod u+x show_args.sh
$ ./show_args.sh "hello" "world"

The arguments are: hello world
The total number of arguments is: 2
```

La Tabella 1.5 riassume le variabili relative agli argomenti posizionali.

Tabella 1.5 Le variabili speciali correlate agli argomenti posizionali.

Variabile	Descrizione
`$0`	Il nome del file dello script.
`$1, $2, $3, …`	Gli argomenti posizionali.
`$#`	Il numero di argomenti posizionali passati.
`$*`	Tutti gli argomenti posizionali.
`$@`	Tutti gli argomenti posizionali, dove ogni argomento è quotato singolarmente.

Quando uno script usa `"$*"` con le virgolette, la shell bash espanderà gli argomenti in una singola parola. Il seguente esempio raggruppa gli argomenti in una parola:

```
$ ./script.sh "1" "2" "3"
1 2 3
```

Quando uno script utilizza `"$@"` (sempre includendo le virgolette), espande gli argomenti in parole separate:

```
$ ./script.sh "1" "2" "3"
1
2
3
```

Nella maggior parte dei casi, si preferisce usare `"$@"`, in modo che ogni argomento venga trattato come una singola parola.
Lo script seguente mostra come utilizzare queste variabili speciali in un ciclo `for`:

```
#!/bin/bash
# Cambiate "$@" con "$*" per osservarne il comportamento.
for args in "$@"; do
    echo "${args}"
done
```

Richiedere l'input

Alcuni script bash non accettano argomenti durante l'esecuzione. Tuttavia, potrebbero dover chiedere informazioni all'utente in modo interattivo e far sì che la risposta venga usata a runtime. In questi casi, possiamo usare il comando `read`. Spesso si vedono applicazioni che utilizzano *prompt di input* quando tentano di installare software, chiedendo all'utente di rispondere *yes* per procedere o *no* per annullare l'operazione.
Nello script bash del Listato 1.26, chiediamo all'utente di specificare il suo nome e cognome e poi li mostriamo sullo stream di output standard.

Listato 1.26 Richiesta di input all'utente (Input_prompting.sh).

```
#!/bin/bash

# Chiede l'input all'utente e lo assegna alle variabili
echo "What is your first name?"
read -r firstname

echo "What is your last name?"
read -r lastname

echo "Your first name is ${firstname} and your last name is ${lastname}"
```

Salvate ed eseguite questo script come `input_prompting.sh`:

```
$ chmod u+x input_prompting.sh
$ ./input_prompting.sh

What is your first name?
John

What is your last name?
Doe

Your first name is John and your last name is Doe
```

Notate che vi viene chiesto di inserire delle informazioni che verranno poi mostrate a schermo.

Codici di uscita

I comandi bash restituiscono dei *codici di uscita*, che indicano se l'esecuzione del comando è riuscita o meno. Tali codici rientrano nell'intervallo `0–255`, dove `0` significa successo, `1` significa fallimento, `126` significa che il comando è stato trovato ma non è eseguibile e `127` significa che il comando non è stato trovato. Il significato di qualsiasi altro numero dipende dal comando utilizzato e dalla sua logica.

Controllo dei codici di uscita

Per provare a usare i codici di uscita, salvate lo script del Listato 1.27 nel file `exit_codes.sh` ed eseguitelo.

Listato 1.27 Utilizzo dei codici di uscita per determinare il successo di un comando.

```
#!/bin/bash

# Sperimentare con i codici di uscita
ls -l > /dev/null
echo "The exit code of the ls command was: $?"

lzl 2> /dev/null
echo "The exit code of the non-existing lzl command was: $?"
```

Utilizziamo la variabile speciale `$?` con il comando `echo` per restituire i codici di uscita dei comandi eseguiti, `ls` e `lzl`. Redirigiamo anche i loro stream di output e di errore standard al file `/dev/null`, uno speciale "file" e "dispositivo" che getta via tutti i dati inviatigli. Quando volete silenziare dei comandi, potete redirigere il loro output su di esso. Dovreste ottenere un output simile al seguente:

```
$ ./exit_codes.sh

The exit code of the ls command was: 0
The exit code of the non-existing lzl command was: 127
```

Otteniamo due codici di uscita distinti, uno per ogni comando. Il primo comando restituisce `0` (successo) e il secondo restituisce `127` (comando non trovato).

> **ATTENZIONE**
>
> Usate `/dev/null` con cautela. Potreste perdervi errori importanti se scegliete di redirigergli l'output. In caso di dubbi, redirigete gli stream standard (output standard ed errore standard) su un file log dedicato.

Per capire perché potreste voler usare i codici di uscita, immaginate di provare a scaricare da Internet un file da 1 GB usando la shell bash. Potrebbe essere saggio controllare se il file esiste già sul file system, nel caso in cui qualcuno abbia già eseguito lo script e abbia già scaricato il file. Inoltre, potreste voler controllare di avere abbastanza spazio libero sul disco prima di tentare il download. Eseguendo i comandi e osservando i codici di uscita che essi restituiscono, potete decidere se procedere con il download o meno.

Impostazione del codice di uscita di uno script

È anche possibile scegliere il codice di uscita di uno script utilizzando il comando `exit` seguito dal codice, come mostrato nel Listato 1.28.

Listato 1.28 Impostazione del codice di uscita di uno script.

```
#!/bin/bash

# Imposta il codice di uscita dello script: 223

echo "Exiting with exit code: 223"
exit 223
```

Salvate questo script come `set_exit_code.sh` ed eseguitelo dalla riga di comando. Quindi usate la variabile speciale `$?` per vedere il codice di uscita che restituisce:

```
$ chmod u+x set_exit_code.sh
$ ./set_exit_code.sh
Exiting with exit code: 223

echo $?
223
```

Potete utilizzare la variabile `$?` per controllare il codice di uscita restituito non solo da uno script ma anche dai singoli comandi:

```
$ ps -ef
$ echo $?

0
```

I codici di uscita sono importanti: possono essere utilizzati in una serie di script che si richiamano fra loro o all'interno dello stesso script, per controllare il flusso logico dell'esecuzione del codice.

Esercizio 1 – Registrare il nome e la data

Realizzate uno script che faccia quanto segue.

1. Accetta due argomenti dalla riga di comando e li assegna alle variabili. Il primo argomento deve essere il vostro nome e il secondo il vostro cognome.
2. Crea un nuovo file, denominato `output.txt`.
3. Scrive la data corrente in `output.txt` utilizzando il comando `date`. Punti bonus se riuscite a far sì che il comando `date` produca la data nel formato *GG-MM-AAAA;* usate `man date` per scoprire come funziona.
4. Scrive il nome e il cognome in `output.txt`.

5. Crea una copia di backup di `output.txt`, denominata `backup.txt`, utilizzando il comando `cp`. Utilizzate `man cp` se non siete sicuri della sintassi del comando.
6. Invia il contenuto del file `output.txt` nello stream di output standard.

Potete trovare una possibile soluzione, `exercise_solution.sh`, nel repository GitHub del libro.

Riepilogo

In questo capitolo, avete eseguito nel terminale alcuni semplici comandi Linux e avete utilizzato `man` per scoprire le opzioni dei comandi. Avete anche imparato a passare argomenti agli script e a eseguire una sequenza di comandi dall'interno degli script. Abbiamo trattato i fondamenti della shell bash, per iniziare a scrivere semplici programmi che utilizzano variabili, array, reindirizzamenti, codici di uscita e argomenti. Avete anche imparato a chiedere all'utente di immettere informazioni arbitrarie e a utilizzarle come parte del flusso di esecuzione dello script.