

1

Getting started

7

Introducing Unix	8
Discovering the shell	10
Understanding commands	12
Navigating directories	14
Dealing with wildcards	16
Recognizing metacharacters	18
Quoting phrases	20
Getting help	22
Summary	24

2

Managing files

25

Creating folders	26
Arranging files	28
Adding links	30
Examining properties	32
Comparing files	34
Finding files	36
Compressing files	38
Making backups	40
Summary	42

3

Handling text

43

Reading and writing	44
Redirecting output	46
Seeking strings	48
Sorting order	50
Arranging columns	52
Matching expressions	54
Editing text	56
Inserting text	58
Summary	60

4

Editing commands

61

Amending characters	62
Changing lines	64
Completing commands	66
Adjusting characters	68
Inserting text	70
Repeating history	72
Fixing commands	74
Expanding history	76
Summary	78

5

Customizing environment

79

Switching users	80
Setting permissions	82
Adding colors	84
Creating aliases	86
Setting options	88
Modifying variables	90
Changing prompts	92
Adjusting paths	94
Summary	96

6

Controlling behavior

97

Disabling defaults	98
Formatting output	100
Reading input	102
Substituting commands	104
Managing jobs	106
Killing processes	108
Communicating routines	110
Relating shells	112
Summary	114

7

Performing operations

115

Storing values	116
Filling arrays	118
Handling strings	120
Doing arithmetic	122
Assigning values	124
Comparing values	126
Assessing logic	128
Matching patterns	130
Summary	132

8

Directing flow

133

Examining conditions	134
Providing alternatives	136
Testing cases	138
Iterating for	140
Selecting options	142
Looping while	144
Looping until	146
Breaking out	148
Summary	150

9

Employing functions

151

Creating scripts	152
Displaying variables	154
Inputting values	156
Providing options	158
Restricting scope	160
Repeating calls	162
Locating bugs	164
Randomizing numbers	166
Summary	168

10

Handy reference

169

Special characters	170
Commands A-D	172
Commands D-F	174
Commands G-L	176
Commands L-S	178
Commands S-U	180
Commands U-Z	182
Date formats	184
Shell variables	186

Index

187

Preface

The creation of this book has been for me, Mike McGrath, an exciting opportunity to demonstrate the powerful command-line shell functionality available in almost any modern Unix-family operating system. I sincerely hope you enjoy discovering the exciting possibilities of the command-line and have as much fun with it as I did in writing this book.

In order to clarify script code listed in the steps given I have adopted certain colorization conventions. Interpreter directives and comments are colored green, shell components are blue, literal string and numeric values are black, user-specified variable and function names are red. Additionally a colored icon and a file name appears in the margin alongside the script code to readily identify each particular script:

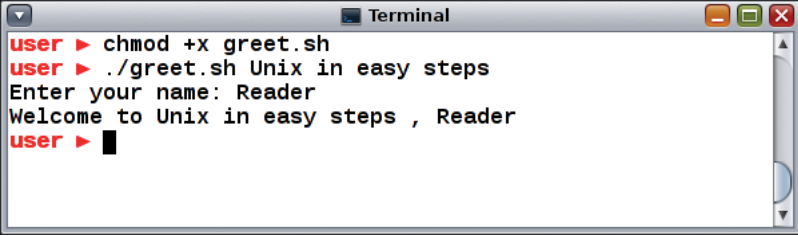


```
#!/bin/bash
# A Script To Greet The User.

echo -n 'Please enter your name: '
read username
echo "Welcome to $@ , $username "
```

greet.sh

The screenshots that accompany each example illustrate the actual output produced by precisely executing the commands listed in the easy steps:



```
user > chmod +x greet.sh
user > ./greet.sh Unix in easy steps
Enter your name: Reader
Welcome to Unix in easy steps , Reader
user > █
```

For convenience I have placed source code files from the examples featured in this book into a single ZIP archive. You can obtain the complete archive by following these three easy steps:

- 1 Open a web browser and navigate to www.ineasysteps.com then select the menu entitled “Free Resources” and choose the “Downloads” item
- 2 Next find “Unix in easy steps” in the list then click on the hyperlink entitled “All Code Examples” to download the archive
- 3 Now extract the contents to any convenient location, such as your home directory

1

Getting started

Welcome to the exciting world of Unix. This chapter introduces the Bash command interpreter shell and demonstrates essential basic commands.

- 8** Introducing Unix
- 10** Discovering the shell
- 12** Understanding commands
- 14** Navigating directories
- 16** Dealing wildcards
- 18** Recognizing metacharacters
- 20** Quoting phrases
- 22** Getting help
- 24** Summary

Introducing Unix



“Multics” (MULTiplexed Information and Computing Service) was the forerunner to “Unics” (UNiplexed Information and Computing Service) – a.k.a. “Unix”.

Unix is a multi-user, multi-tasking computer operating system developed at AT&T’s Bell Labs. Way back in the mid 1960s Bell Labs, the Massachusetts Institute of Technology and General Electric co-developed a mainframe multi-tasking operating system named “Multics”. The Multics operating system became large and complex so Bell Labs withdrew from the project. A group from Bell Labs, notably Brian Kernighan and Dennis Ritchie, wanted to create something with many of the same multi-tasking capabilities but simpler to use. They named it “Unix” as a joke, saying “whatever Multics is many of, Unix is just one of”.

In 1973 Kernighan and Ritchie created the “C” high-level programming language and rewrote Unix in their new language. This was a huge leap from the assembly code used in all other operating system development at that time. The migration from assembly code to the C language produced much more portable software that was easily ported to other computing platforms. AT&T made the Unix source code available to universities, which created a great amount of academic interest, particularly at the University of California, Berkeley. Their Computer Science Research Group added new capabilities then released their version of Unix as “BSD” (Berkeley Software Distribution):

- BSD Unix proved very popular and was adopted by many commercial startups including Sun Microsystems who used it as the basis for SunOS. Sun later created the Solaris Unix operating system and were acquired by the Oracle Corporation in 2010 - since then that system has been known as Oracle Solaris.
- BSD Unix was adopted for the development of the Darwin operating system – providing the core set of components upon which Apple’s OS X and iOS are based.
- BSD Unix also proved interesting to a 21-year old student at the University of Helsinki. He released his own lightweight Unix variant specifically for PCs. His name was Linus Torvalds and he named his operating system “Linux” – (LINUs’ uniX). Many variations of this popular release have subsequently been developed including Red Hat, Ubuntu, and the Raspbian OS for the Raspberry Pi device.



Only Oracle Solaris, Apple OS X, HP-UX and IBM’s AIX systems are fully recognized as Unix – all others are more properly referred to as “Unix-like” systems.

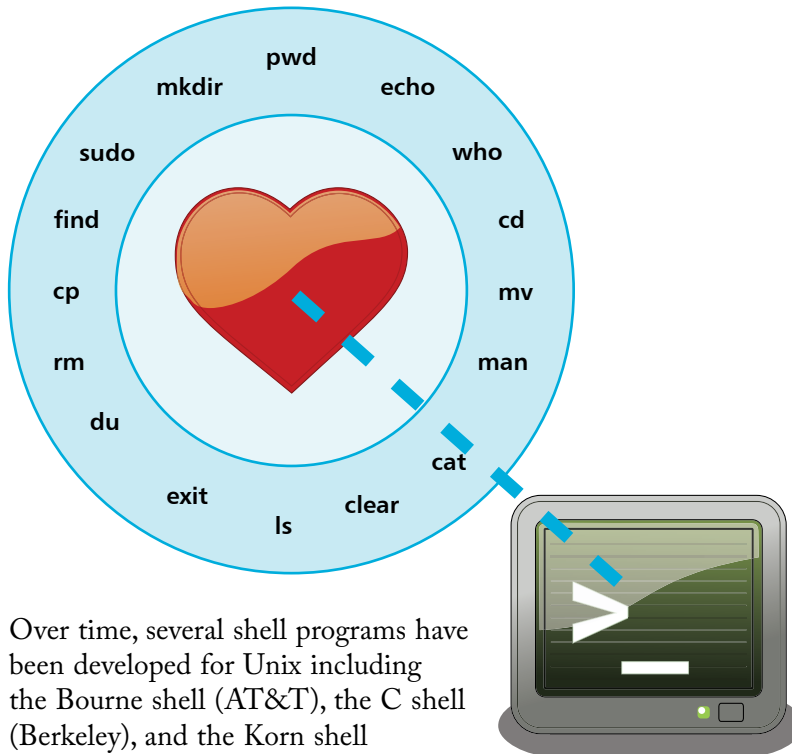
...cont'd

Meeting the kernel

At the heart of every variant of Unix is the “kernel” – a program that gets loaded into memory at boot-up time and manages the system until shutdown. The kernel controls hardware devices, schedules tasks, and manages memory. All other programs reside on the hard disk drive and get loaded into memory by the kernel. The kernel executes them and cleans up the system on completion.

Meeting the shell

The “shell” is a utility program that starts up when you log on to a Unix system. It allows the user to interact with the kernel by interpreting commands that are typed at the command line or read from a script file.



Over time, several shell programs have been developed for Unix including the Bourne shell (AT&T), the C shell (Berkeley), and the Korn shell (a superset of the Bourne shell).

The most popular shell today is “Bash” – the **B**ourne **a**gain **s**hell. Bash is an enhanced Bourne shell and is the default shell on Oracle Solaris, Linux and Mac OS X operating systems. This book demonstrates how to use the Bash shell to interact with the kernel from the command line and from scripts.



The shell program allows the user to communicate with the kernel at the heart of their Unix system.



The New icon pictured above indicates a new or enhanced feature introduced with the latest version of Bash.



Bash is part of the “GNU” Unix-like operating system – find details and download instructions online at gnu.org/software/bash.



Unix is case-sensitive so the commands **MUST** be capitalized exactly as listed. For example, the **ps** command must use only lowercase letters.

Discovering the shell

When you initially log in to a Unix system, or open a new terminal window, a command prompt appears indicating that a shell process has been started for you automatically. This shell will typically be the Bash shell program.

The name of the shell in use can be seen in information about the current terminal process by issuing a **ps \$\$** command. The output from this command should confirm Bash as the current shell under its **COMMAND** heading. If another shell is listed you can switch to the Bash shell simply by issuing a **bash** command if it is available. In the event that the Bash program is not already available it must be installed by you or the system administrator.

Once you have confirmed that Bash is the current shell you can see its version information by issuing a **bash --version** command:

- 1 Launch a Terminal window then at the prompt exactly type **ps \$\$** and hit Return to discover the current shell

```

user@host:~$ ps $$
  PID TT          S   TIME COMMAND
 3585 pts/1      S    0:00 bash
user@host:~$ █
  
```

- 2 If Bash is not the current **COMMAND** shell listed, type **bash** at the prompt then hit Return to switch to Bash

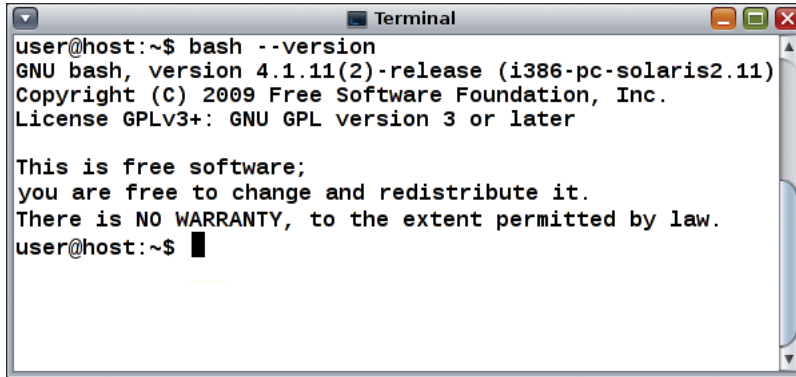
- 3 At the prompt, exactly type **ps \$\$** once more and hit Return to discover if Bash is now the current shell

```

host% ps $$
  PID TT          S   TIME COMMAND
 3617 pts/1      S    0:00 csh
host% bash
user@host:~$ ps $$
  PID TT          S   TIME COMMAND
 3622 pts/1      S    0:00 bash
user@host:~$ █
  
```


...cont'd

- 4 Once you have confirmed you are indeed using the Bash shell, at the prompt exactly type **bash --version** then hit Return to discover the current Bash version



```
user@host:~$ bash --version
GNU bash, version 4.1.11(2)-release (i386-pc-solaris2.11)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later

This is free software;
you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
user@host:~$
```

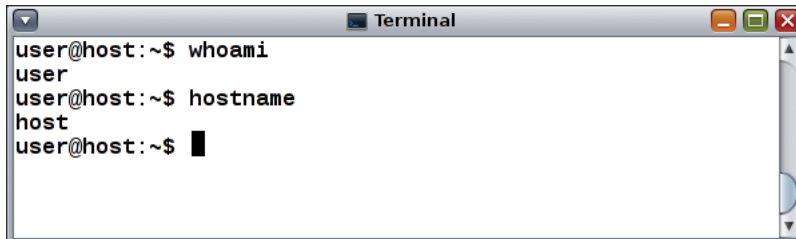


You can also use the command **which bash** to detect Bash. This should respond with the directory location of the Bash program if present on the system.

- 5 Now, issue a **clear** command (or press Ctrl + L keys) to clear the terminal screen back to a command prompt

Notice that the default Bash command prompt comprises several pieces of information including the user's name (shown here as **user**) and the host system name (shown here as **host**). These details can also be displayed using the **whoami** and **hostname** commands:

- 6 Type **whoami** then hit Return to see your user name, then type **hostname** and hit Return to see your computer name



```
user@host:~$ whoami
user
user@host:~$ hostname
host
user@host:~$
```



The **whoami** and **hostname** commands are mostly useful when the command prompt has been customized to hide the user name and host name.

- 7 Finally issue an **exit** command (or press Ctrl + D keys) to end the session and thereby close the Terminal window



Each Unix program can accept standard input and can produce standard output and standard error messages.



You can also use the built-in command **hash** to see a list of your recently issued program commands and the number of times executed (hits).

Understanding commands

When the user hits the Return key after typing a command at a shell prompt it adds a final invisible newline character. This denotes the end of the command and indicates to the shell that it should then attempt to interpret that command. The Bash interpreter first reads the command line as “standard input” (stdin) and splits it into separate words broken by spaces or tabs. Each of these words is known as a “token”. The interpreter next examines the first token to see if it is one of the shell’s “built-in” commands or an executable program located on the file system.

When the first token is recognized as a built-in shell command, the interpreter executes that command otherwise it searches through the directories on a specified path to find a program of that name. The interpreter will then execute a recognized built-in command or recognized program and display any result in the Terminal as “standard output” (stdout). Where neither is found, the interpreter will display an error message in the Terminal as “standard error” (stderr).

The Bash **type** command can be used to determine whether a token is recognized as a built-in shell command, or the location of a recognized program, or display a message if none can be found:

- 1 At a command prompt, type **type clear** then hit Return to see the location of the **clear** program on the filesystem
- 2 Next, type **type exit** then hit Return to discover that **exit** is in fact a built-in shell command
- 3 Now, type **type nosuch** then hit Return to see this token cannot be found to match a built-in command or program name

```

Terminal
user@host:~$ type clear
clear is /usr/bin/clear
user@host:~$ type exit
exit is a shell builtin
user@host:~$ type nosuch
bash: type: nosuch: not found
user@host:~$
  
```

...cont'd

The Bash built-in **echo** command simply reads all following tokens from standard input then prints them as standard output - unless they are recognized as a command “option”. Many built-in commands and programs accept one or more options that specify how they should be executed. Typically, an option consists of a dash followed by a letter. For example, the **echo** command accepts a **-n** option that denotes it should omit the newline character that it automatically prints after other output:

- 4 At a prompt, type **echo** followed by some text then hit Return to see that text printed with an added newline
- 5 Now, type **echo -n** followed by some text then hit Return to see that text printed without an added newline

```
Terminal
user@host:~$ echo Unix in easy steps...
Unix in easy steps...
user@host:~$ echo -n Unix in easy steps...
Unix in easy steps...user@host:~$
```

In addition to the built-in shell commands the Bash shell also contains a number of built-in shell variables. These are named “containers” that each store a piece of information and their names use all uppercase characters. To access the information stored within a variable its name must be prefixed with a **\$** dollar sign:

- 6 At a prompt, type **echo \$SHELL** to see the location of the Bash interpreter program on the filesystem
- 7 Now, type **echo \$BASH_VERSION** to see the version number of the Bash shell interpreter

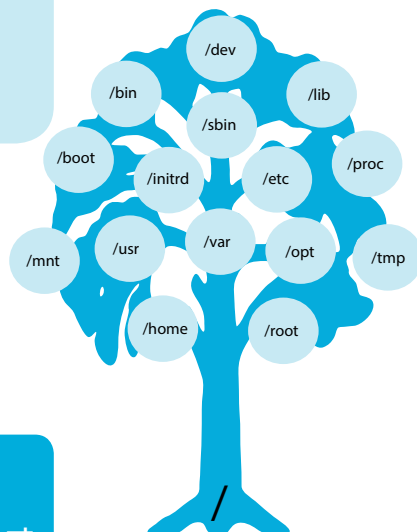
```
Terminal
user@host:~$ echo $SHELL
/usr/bin/bash
user@host:~$ echo $BASH_VERSION
4.1.11(2)-release
user@host:~$ █
```



You will often want to suppress the automatic newline with **echo -n** when printing a request for user input.



You can also use the command **echo \$PATH** to discover which directories the Bash shell searches when you issue any command.



Navigating directories

The Unix filesystem is arranged as a tree-like hierarchical structure of directories and files with the “root” directory at its base. The root directory is addressed simply as a forward-slash character `/`.

Sub-directories within the root directory are addressed by appending their directory name to the forward-slash. For example, the “home” directory has the address `/home`. Directories within those sub-directories are addressed by appending another forward-slash and their directory name. For example, a “user” directory within the “home” directory has the address `/home/user`.

Similarly, files in directories are addressed by appending another forward-slash and the filename, such as `/home/user/filename`.

This hierarchical address system can therefore easily describe the “absolute path” of any directory or file from the root base.

Additionally, contents of the current directory and its sub-directories can be addressed by name using their “relative path”. For example, a sub-directory named “user” within `/home` can be addressed from `/home` simply as `user`, and a file within that sub-directory can be addressed as `user/filename`.

When the user launches a Terminal window it conveniently sets the filesystem location to that user’s home directory. This is typically a directory bearing that user’s name located in `/home`. For example, for a user named “Carole” at `/home/Carole`. The user can ascertain their current location at any time with the `pwd` (print working directory) command and list the contents of that directory with the `ls` command. Contents of sub-directories can be listed simply by specifying their address to the `ls` command:



As Unix is case-sensitive the names of directories and files in addresses must be correctly capitalized.

- 1 Launch a Terminal and type `pwd` at the prompt then hit Return to see the absolute path address of your location
- 2 Next, type `ls` then hit Return to see the contents of the current directory
- 3 Now, specify the relative address of an immediate child sub-directory to the `ls` command to see nested directories – for example, type `ls Documents` then hit Return
- 4 Then, specify the relative address of a nested sub-directory to the `ls` command to see that directory’s contents – for example, type `ls Documents/Pictures` then hit Return

...cont'd

```
Terminal
user@host:~$ pwd
/home/user
user@host:~$ ls
core      Desktop    Documents  Downloads  Public
user@host:~$ ls Documents
Music     Pictures   Scripts    Templates  Videos
user@host:~$ ls Documents/Pictures
vette.jpg viper.jpg
user@host:~$
```



The **Documents** sub-directory here could also be addressed by its absolute path of **/home/user/Documents**.

Repeatedly addressing contents of a sub-directory by path can become tedious. It is more convenient to change location into that sub-directory so its contents can easily be addressed by their name alone. The directory to move into can be specified to the **cd** command as either an absolute or relative path address. Additionally, the user's home directory can be addressed using a tilde **~** alias and the parent of the current directory can be addressed using a **..** alias. By default, the Bash prompt string displays the current directory just before the **\$** sign:

- 5 Type **cd Documents/Pictures** then hit Return to see the prompt change to display the nested sub-directory location
- 6 Next, type **pwd** then hit Return to confirm the absolute path address of that new location
- 7 Now, type **cd ..** then hit Return to move to the parent directory and see the prompt string change once more
- 8 Finally, type **cd ~** then hit Return to go back to your home directory and see the prompt string change again



You can also use the command **cd -** to return to the previous directory you were located in.

```
Terminal
user@host:~$ cd Documents/Pictures
user@host:~/Documents/Pictures$ pwd
/home/user/Documents/Pictures
user@host:~/Documents/Pictures$ cd ..
user@host:~/Documents$ cd ~
user@host:~$ █
```



If you wish to delete a directory remember that it may contain hidden files – use the **ls -a** command to check.

Dealing wildcards

The **ls** command, introduced on the previous page, will list all files and folders in the current or specified directory, except special hidden files whose names begin with a **.** period character. Typically, these are system files such as a **.bashrc** hidden file in the user's home directory containing the shell configuration details. Hidden files can be included in the list displayed by the **ls** command by adding a **-a** option, so the command becomes **ls -a**.

Optionally, a filename pattern can be supplied to the **ls** command so it will list only filenames matching the specified pattern. Special “wildcard” characters, described in the table below, can be used to specify the filename pattern to be matched:

Wildcard:	Matches:
?	Any single character
*	Any string of characters
[set]	Any character in <i>set</i>
[!set]	Any character not in <i>set</i>

The **?** wildcard is used to specify a pattern that matches filenames where only one single character may be unknown. For example, where the **ls** command lists **file.a**, **file.b**, and **file.exe** the command **ls file.?** would list only **file.a** and **file.b** – not **file.exe**.

More usefully, the ***** wildcard is used to specify a pattern that matches filenames where multiple characters may be unknown. For example, where **ls** lists **img.png**, **pic.png**, and **pic.jpg** the command **ls *.png** would list only **img.png** and **pic.png** – not **img.jpg**.

The **[set]** wildcard construct is used to specify a pattern that matches a list or a range of specified characters. For example, where the **ls** command lists **doc.a**, **doc.b**, **doc.c**, and **doc.d** the command **ls doc.[ac]** would list only **doc.a** and **doc.c** – as this pattern specifies a list of two possible extensions to be matched. In the same directory the command **ls doc.[a-c]** would, however, list **doc.a**, **doc.b**, and **doc.c** – as the pattern specifies a range of three possible extensions to be matched. Placing an exclamation mark at the start of a set pattern lists files not matched. For example, here the command **ls doc[!a-c]** would list only **doc.d**.



You can include a hyphen in the set pattern by placing it first or last in the list within the square brackets.

...cont'd

In executing commands containing wildcards, the shell first expands the wildcard matches and substitutes them as a list of “arguments” to the command. So the command **ls doc.[a-c]** might, in effect, become **ls doc.a doc.b doc.c** before the list gets printed. This is apparent in the error message that gets displayed when no matches are found. For example, **ls non*** might produce the error message **non*: No such file or directory** – as **non*** is the argument:

- 1 Type **ls** at the prompt then hit Return to see all unhidden files in the current working directory
- 2 Next, type **ls doc.?** then hit Return to see all files named “doc” that have a single-letter file extension
- 3 Now, type **ls *.c** then hit Return to see all files of any name that have a “.c” file extension
- 4 Type **ls *.*[a-c]** then hit Return to see all files of any name that have a “.a”, “.b” or “.c” file extension
- 5 Now, type **ls *.*[!a-c]** then hit Return to see all files of any name that do not have a “.a”, “.b” or “.c” file extension
- 6 Finally, type **ls non*** then hit Return to see an error message reporting no matches

```
Terminal
user@host:~/Docs$ ls
doc.a doc.b doc.c doc.d txt.a txt.b txt.c txt.d
user@host:~/Docs$ ls doc.?
doc.a doc.b doc.c doc.d
user@host:~/Docs$ ls *.c
doc.c txt.c
user@host:~/Docs$ ls *.*[a-c]
doc.a doc.b doc.c txt.a txt.b txt.c
user@host:~/Docs$ ls *.*[!a-c]
doc.d txt.d
user@host:~/Docs$ ls non*
non*: No such file or directory
user@host:~/Docs$
```



The process of pattern matching with wildcards demonstrated here is commonly known as “globbing” – a reference to global wildcard expansion.



Wildcards can also be used for pathname expansion when specifying addresses – for example **ls ~/D***.



There must be no spaces within the braces or between the braces and each specified prefix and suffix.



Bash version 4 introduced zero-padded brace expansion so that `echo {001..3}` produces 001 002 003.

Recognizing metacharacters

Just as the special wildcard characters `? * []` can be used to perform pathname expansion, plain strings can be expanded using `{ }` brace characters. These may contain a comma-separated list of substrings that can be appended to a specified prefix, or prepended to a specified suffix, or both, to generate a list of expanded strings. The brace expansions can also be nested for complex expansion. Additionally, brace expansion can produce a sequence of letters or numbers by specifying a range separated by `..` between the braces:

- 1 At a prompt, type `echo b{ad,oy}` then hit Return to see two expanded strings – appended to the specified prefix
- 2 Next, type `echo {ge,fi}t` then hit Return to see two expanded strings – prepended to the specified suffix
- 3 Now, type `echo s{i,a,o}ng` then hit Return to see four expanded strings – both appended and prepended
- 4 Type `echo s{tr{i,o},a,u}ng` then hit Return to see four complex expanded strings – appended and prepended
- 5 Next, type `echo {a..z}` then hit Return to see an expanded letter sequence of the lowercase alphabet
- 6 Finally, type `echo {1..20}` then hit Return to see an expanded numeric sequence from 1 to 20

```

Terminal
user@host:~$ echo b{ad,oy}
bad boy
user@host:~$ echo {ge,fi}t
get fit
user@host:~$ echo s{i,o,a,u}ng
sing song sang sung
user@host:~$ echo s{tr{i,o},a,u}ng
string strong sang sung
user@host:~$ echo {a..z}
a b c d e f g h i j k l m n o p q r s t u v w x y z
user@host:~$ echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
user@host:~$ █

```


...cont'd

The wildcards `? *` `[]` and braces `{ }` are just some examples of “metacharacters” that have special meaning to the bash shell. The table below lists all metacharacters that have the special meaning described when used in commands at a shell prompt only – the same characters can have other meanings when used in other situations, such as in arithmetical expressions.

Metacharacter:	Meaning:
<code>~</code>	Home directory
<code>`</code>	Command substitution (old style)
<code>#</code>	Comment
<code>\$</code>	Variable expression
<code>&</code>	Background job
<code>*</code>	String wildcard
<code>(</code>	Start of subshell
<code>)</code>	End of subshell
<code>\</code>	Escape next character
<code> </code>	Pipe
<code>[</code>	Start of wildcard set
<code>]</code>	End of wildcard set
<code>{</code>	Start of command block
<code>}</code>	End of command block
<code>;</code>	Pipeline command separator
<code>'</code>	Quote mark (strong)
<code>"</code>	Quote mark (weak)
<code><</code>	Redirect input
<code>></code>	Redirect output
<code>/</code>	Pathname address separator
<code>?</code>	Single-character wildcard
<code>!</code>	Pipeline logical NOT



Some of the metacharacters in this table have been introduced already but others are described later in this book.



Notice that the semi-colon `;` character allows two commands to be issued on the same line. For example, type **`echo {a..z} ; echo {1..9}`** then hit Return.



Always enclose phrases you want to use literally within single quotes to avoid interpretation.

Quoting phrases

The metacharacters that have special meaning to the Bash shell can be used literally, without applying their special meaning, by enclosing them within a pair of `' '` single quote characters to form a quoted phrase. For example, to include the name of a shell variable in a phrase without interpreting its value:

- 1 At a prompt, type **echo Processed By: \$SHELL** then hit Return to see the shell variable get interpreted in output
- 2 Now, type **echo 'Processed By: \$SHELL'** then hit Return to see the shell variable printed literally in output

```

Terminal
user@host:~$ echo Processed By: $SHELL
Processed By: /usr/bin/bash
user@host:~$ echo 'Processed By: $SHELL'
Processed By: $SHELL
user@host:~$ █
  
```

Alternatively, the significance of the leading `$` metacharacter of a shell variable can be ignored if preceded by a `\` backslash character to “escape” it from recognition as having special meaning:

- 3 At a prompt type **echo Processed By: \$SHELL** then hit Return to see the shell variable get interpreted in output
- 4 Now, type **echo Processed By: \SHELL** then hit Return to see the shell variable printed literally in output

```

Terminal
user@host:~$ echo Processed By: $SHELL
Processed By: /usr/bin/bash
user@host:~$ echo Processed By: \SHELL
Processed By: $SHELL
user@host:~$ █
  
```



The newline `\n` and tab `\t` sequences can be included in phrases if preceded by a backslash. For example, **echo \nNEWLINE \tTAB**.

...cont'd

It is necessary to precede a single quote character with a \ backslash when it is used as an apostrophe, so it is not interpreted as an incomplete quoted phrase. An incomplete quoted phrase or a \ backslash at the end of a line allows a command to continue on the next line as they escape the newline when you hit Return:

- 5 At a prompt, type **echo It\'s escaped** then hit Return to see the apostrophe appear in output
- 6 Next, type **echo Continued ** then hit Return, type **text written along ** then hit Return, and type **several lines** then hit Return to see the continued phrase in output



Notice that the shell prompt string changes to a > to indicate it is awaiting further input.

```
Terminal
user@host:~$ echo It\'s escaped
It's escaped
user@host:~$ echo Continued \
> text written along \
> several lines
Continued text written along several lines
user@host:~$
```

Double quote marks " " are regarded as weak by the Bash shell as they do allow the interpretation of shell variables they enclose. They can, however, be useful to print out a quoted string if the entire string (and its double quotes) are enclosed in single quotes:

- 7 Type **echo "Interpreted With \$SHELL"** then hit Return to see the shell variable get interpreted in unquoted output
- 8 Type **echo "'Interpreted With \$SHELL'"** then hit Return to see the shell variable printed literally in quoted output



You could alternatively escape double quote characters with a backslash to print them in output. For example, **echo \"With \\$SHELL\"**.

```
Terminal
user@host:~$ echo "Interpreted With $SHELL"
Interpreted With /usr/bin/bash
user@host:~$ echo "'Interpreted With $SHELL'"
'Interpreted With $SHELL'
user@host:~$
```



The `|` character is a “pipe” that allows output to be redirected – here output is sent to the **more** command. Pipelines are described in detail on page 46.

Getting help

Bash includes an online help system for its built-in commands. Information on all its built-in commands can be displayed using the **help** command and **help | more** can be used to display just one screen at a time. A command name can be specified to discover information about that particular command:

- 1 At a prompt, type **help | more** then hit Return to see all built-in bash commands and their options

```

user@host:~$ help | more
GNU bash, version 4.1.11(2)-release (i386-pc-solaris2.11)
These shell commands are defined internally. Type 'help'

job_spec [&]                                history [-c] [-d
(( expression ))                            if COMMANDS; the
. filename [arguments]                     jobs [-lnprs] [j
:                                           kill [-s sigspec
[ arg... ]                                let arg [arg ...
[[ expression ]]                           local [option] n
alias [-p] [name[=value] ... ]            logout [n]
bg [job_spec ...]                          mapfile [-n coun
bind [-lpvsPVS] [-m keymap] [-f file]>    popd [-n] [+N |
break [n]                                  printf [-v var]
builtin [shell-builtin [arg ...]]          pushd [-n] [+N |
caller [expr]                              pwd [-LP]
case WORD in [PATTERN [| PATTERN]...]>    read [-ers] [-a
cd [-L|-P] [dir]                          readarray [-n co
--More--

```

- 2 Hit Return to scroll down the screen one line at a time or type **q** and hit Return to quit help and return to a prompt
- 3 Now, type **help echo** and hit Return to display information about the Bash shell built-in **echo** command



Bash version 4 introduced two new help options; **help -d** displays a short description and **help -m** displays information in a manual page-like format.

```

user@host:~$ help echo
echo: echo [-neE] [arg ...]
      Write arguments to the standard output.

      Display the ARGs on the standard output
      followed by a newline.

Options:
  -n          do not append a newline

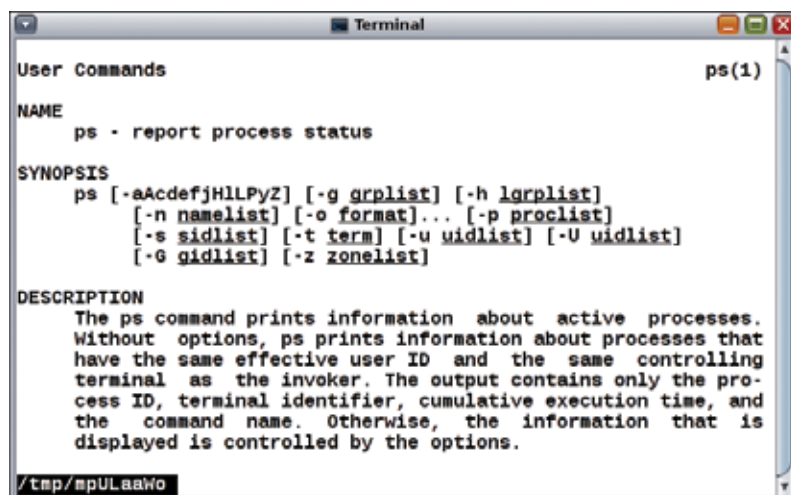
Exit Status:
  Returns success unless a write error occurs.
user@host:~$

```

...cont'd

Information about all commands, both shell built-in commands and those other commands that are actually programs located on the filesystem, can be found on any Unix-based operating system in the famous Manual Pages. The name of any command can be specified to the **man** command to display the manual page describing that command and its options. Alternatively, a **-f** option can be used to display the location and brief description:

- 4 At a prompt, type **man ps** then hit Return to see the manual page for the **ps** command automatically paginated



```
Terminal
User Commands                                ps(1)
NAME
ps - report process status

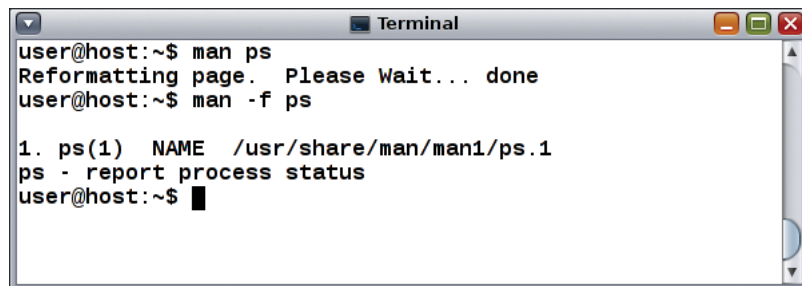
SYNOPSIS
ps [-aAcdefjHlLPyZ] [-g grplist] [-h lgrplist]
  [-n namelist] [-o format]... [-p proclist]
  [-s sidlist] [-t term] [-u uidlist] [-U uidlist]
  [-G gidlist] [-z zonelist]

DESCRIPTION
The ps command prints information about active processes.
Without options, ps prints information about processes that
have the same effective user ID and the same controlling
terminal as the invoker. The output contains only the pro-
cess ID, terminal identifier, cumulative execution time, and
the command name. Otherwise, the information that is
displayed is controlled by the options.

/tmp/mpULaawo
```

- 5 Hit Return to scroll down the screen one line at a time or type **q** and hit Return to return to a prompt

- 6 Now, type **man -f ps** and hit Return to display the filesystem location and description of the **ps** command



```
Terminal
user@host:~$ man ps
Reformatting page. Please Wait... done
user@host:~$ man -f ps

1. ps(1) NAME /usr/share/man/man1/ps.1
ps - report process status
user@host:~$ █
```



You can use the **type** command, described on page 12, to discover whether a command is a shell built-in or its filesystem location.



You can also use the **info** command as an alternative to **man**.

Summary

- Bash is a command interpreter shell that enables the user to interact with the kernel of any Unix-based operating system
- The command **ps \$\$_** displays the current process information and can be used to confirm Bash as the current shell
- User and host names can be displayed with the **whoami** and **hostname** commands
- A Terminal window can be cleared using the **clear** command and closed using the **exit** command
- The **type** command can be used to determine whether a token is a built-in shell command or a recognized program
- Standard input can be printed on standard output using the shell built-in **echo** command
- Shell variables **\$SHELL** and **\$BASH_VERSION** store the filesystem location and version number of the Bash program
- The **pwd** command displays the current working directory address and the **ls** command can be used to list its contents
- Absolute and relative addresses, or **~** and **..** aliases, can be specified to the **cd** command to change directory location
- Wildcards **?**, *****, and **[]** can be used to specify filename patterns to match a single character, a string, or a set
- Brace expansion combines each item in a comma-separated list within **{ }** characters to a specified outer prefix and suffix
- Brace expansion can also produce a sequence of letters or numbers from a range separated by **..** within **{ }** characters
- Wildcards **?**, *****, **[]** and braces **{ }** are just some examples of metacharacters that have special meaning to the Bash shell
- Enclosing with single quotes **' '** or prefixing with a backslash **** allows metacharacters to be displayed literally
- Surrounding phrases with weak **" "** double quote characters allows the shell to perform interpretation
- Online help can be found for any command using the **man** or **info** commands and for built-ins using the **help** command