

1

Getting Started

7

Introducing Bash	8
Discovering the Shell	9
Installing Linux on Windows	10
Initializing the Distro	12
Understanding Commands	14
Navigating the File System	16
Dealing Wildcards	18
Recognizing Metacharacters	20
Quoting Phrases	22
Getting Help	24
Summary	26

2

Managing Files

27

Creating Folders	28
Arranging Files	30
Adding Links	32
Examining Properties	34
Comparing Files	36
Finding Files	38
Compressing Files	40
Making Backups	42
Summary	44

3

Handling Text

45

Reading and Writing	46
Redirecting Output	48
Seeking Strings	50
Sorting Order	52
Arranging Columns	54
Matching Expressions	56
Editing Text	58
Inserting Text	60
Summary	62

4

Editing Commands**63**

Amending Characters	64
Changing Lines	66
Completing Commands	68
Adjusting Characters	70
Inserting Text	72
Repeating History	74
Fixing Commands	76
Expanding History	78
Summary	80

5

Customizing Environment**81**

Switching Users	82
Setting Permissions	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 with Bash. 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:

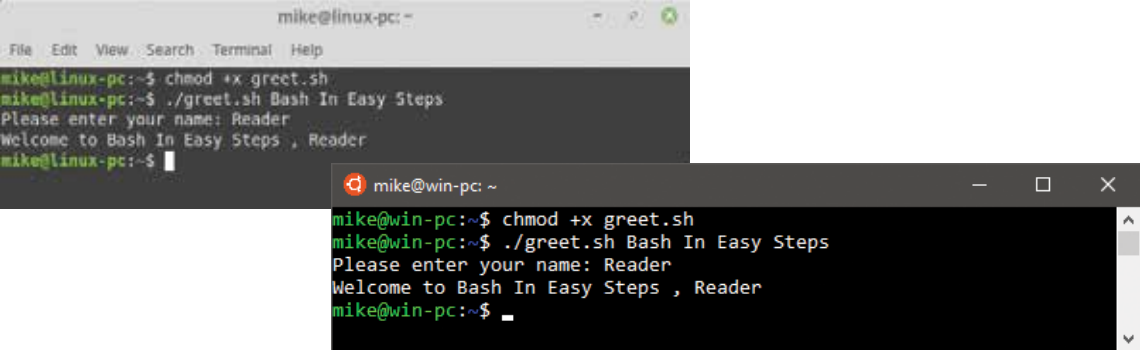


```
#!/bin/bash
# A Script to Greet the User.

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:



```
mike@linux-pc: ~
File Edit View Search Terminal Help
mike@linux-pc:~$ chmod +x greet.sh
mike@linux-pc:~$ ./greet.sh Bash In Easy Steps
Please enter your name: Reader
Welcome to Bash In Easy Steps , Reader
mike@linux-pc:~$

mike@win-pc: ~
mike@win-pc:~$ chmod +x greet.sh
mike@win-pc:~$ ./greet.sh Bash In Easy Steps
Please enter your name: Reader
Welcome to Bash In Easy Steps , Reader
mike@win-pc:~$
```

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 Browse to www.ineasysteps.com then navigate to [Free Resources](#) and choose the [Downloads](#) section
- 2 Find [Bash 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

*This chapter introduces the
Bash command interpreter
shell and demonstrates
essential basic commands.*

- 8** Introducing Bash
- 9** Discovering the Shell
- 10** Installing Linux on Windows
- 12** Initializing the Distro
- 14** Understanding Commands
- 16** Navigating the File System
- 18** Dealing Wildcards
- 20** Recognizing Metacharacters
- 22** Quoting Phrases
- 24** Getting Help
- 26** Summary



GNU Bash logo

Introducing Bash

Computer operating systems include a command-line interpreter that allows the user to communicate directly with the system by typing text commands at a waiting prompt. The command-line interpreter is a “shell” facility that will process the input command and produce an appropriate output response.

An early shell facility was created for the Unix operating system by Stephen Bourne at Bell Labs way back in 1979. This early “Bourne shell” (**sh**) proved to be very popular as it was both a command-line interpreter and a scripting language that supported most features needed to produce structured programs.

Ten years later, in 1989, a free software replacement for the Bourne shell was created by Brian Fox for the GNU Project. Recognizing its ancestry, the replacement shell was named “Bash” – an acronym for the phrase **B**ourne-**a**gain **s**hell. The Bash command syntax is a superset of that in the Bourne shell, but incorporates many extensions that are lacking in the Bourne shell. Bash can efficiently process commands typed at a prompt and execute shell program scripts that have been saved as text files.

Today Bash is the default shell for most Linux operating systems, for Apple’s macOS operating system (formerly OS X), and for the Solaris Unix operating system.

The Bash command-line processor typically runs in a text window. This is similar to the Command Prompt window found in the Windows operating system in which users can type commands to be interpreted by the Windows Command Processor.

Microsoft has recognized the power and popularity of Bash by introducing support for “WSL” (**W**indows **S**ubsystem for **L**inux) in the Windows 10 operating system. This allows users to run a Linux environment directly on Windows so they can issue Bash commands at a shell prompt, create and execute Bash shell scripts, and run Linux command-line applications.

This book describes and demonstrates how to utilize the power of Bash with examples that can be run natively in a “Terminal” text window on a Linux operating system and within a (WSL) Linux environment on the Windows 10 operating system.



The GNU Project is a free-software mass collaboration project. You can discover more at gnu.org/gnu/thegnuproject.en.html

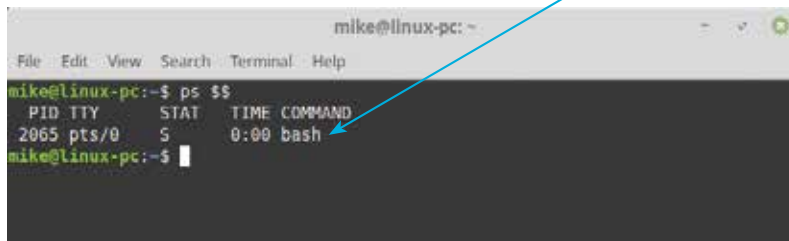
Discovering the Shell

When you open a new Terminal window on a Linux operating system, a command prompt appears indicating that a shell process has been started for you automatically. This shell will typically be the Bash shell facility.

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

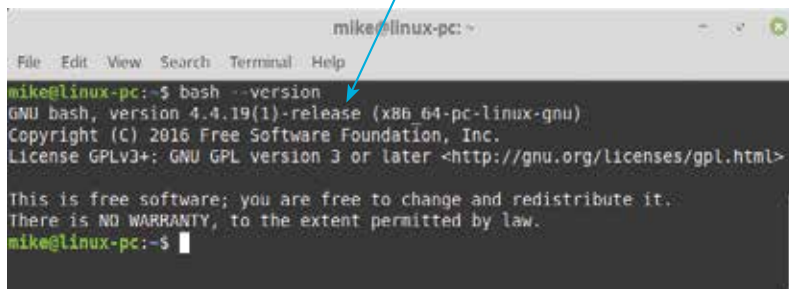


```
mike@linux-pc: ~  
File Edit View Search Terminal Help  
mike@linux-pc:~$ ps $$  
PID TTY STAT TIME COMMAND  
2065 pts/0 S 0:00 bash  
mike@linux-pc:~$
```

A screenshot of a Linux terminal window. The title bar says "mike@linux-pc: ~". The menu bar includes "File", "Edit", "View", "Search", "Terminal", and "Help". The user has entered the command "ps \$\$" and the output shows a single process with PID 2065, TTY pts/0, status S, and command bash. A blue arrow points from the instruction in step 1 to the "bash" command in the output.

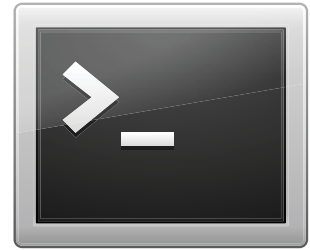
- 2 Next, type a **clear** command and hit **Return** (or press **Ctrl + L** keys) to clear the Terminal window to a prompt

- 3 Now, exactly type **bash --version** then hit **Return** to discover the current Bash version



```
mike@linux-pc: ~  
File Edit View Search Terminal Help  
mike@linux-pc:~$ bash --version  
GNU bash, version 4.4.19(1)-release (x86_64-pc-linux-gnu)  
Copyright (C) 2016 Free Software Foundation, Inc.  
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>  
  
This is free software; you are free to change and redistribute it.  
There is NO WARRANTY, to the extent permitted by law.  
mike@linux-pc:~$
```

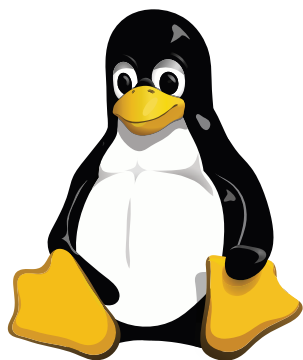
A screenshot of a Linux terminal window. The title bar says "mike@linux-pc: ~". The menu bar includes "File", "Edit", "View", "Search", "Terminal", and "Help". The user has entered the command "bash --version" and the output shows the GNU bash version 4.4.19(1)-release for x86_64-pc-linux-gnu, along with copyright and license information. A blue arrow points from the instruction in step 3 to the "bash --version" command.



Linux Terminal



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



This happy penguin is “Tux” – the friendly mascot of the Linux operating system.



You will need your PC to have an internet connection to download a Linux distro.



You cannot install Ubuntu Linux on the Windows 10 S version of the operating system.

Installing Linux on Windows

The Windows Subsystem for Linux (WSL) allows you to install a variety of Linux distributions (“distros”) from the Windows Store. Before any distro can be installed you must, however, first enable the optional WSL feature on your Windows 10 system:

- 1 Right-click the “Windows PowerShell” item on the Start menu, then choose **Run as Administrator** – to open a PowerShell window with Administrator privileges
- 2 In PowerShell, precisely type this command:
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux



- 3 Hit **Return** to execute the command, then type **Y** and hit Return once more – to restart the operating system
- 4 Next, click the **Microsoft Store** item on the Start menu, then search for “WSL” to find the available Linux distros



...cont'd

- 5 Select your preferred Linux distro, such as the **Ubuntu** distro that is used to demonstrate WSL in this book
- 6 Click the **Get** button to download the chosen Ubuntu distro onto your PC



If installation fails with error **0x8007007e** your system doesn't support Linux from the store – ensure you are running Windows 10 build 16215 or later; the Windows Subsystem for Linux is enabled; and that you have restarted your PC.

- 7 Open the Start menu, then click the new **Ubuntu** item to open a Terminal window



- 8 Wait while the distro's files are decompressed and installed – this takes a while, but only happens once

- 9 Finally, enter a username and password of your choice to set up a Linux user account

```
mike@win-pc: ~  
Installing, this may take a few minutes...  
Please create a default UNIX user account. The username does not need to  
match your Windows username.  
For more information visit: https://aka.ms/wslusers  
Enter new UNIX username: mike  
Enter new UNIX password:  
Retype new UNIX password:  
passwd: password updated successfully  
Installation successful!  
To run a command as administrator (user "root"), use "sudo <command>".  
See "man sudo_root" for details.  
  
mike@win-pc:~$
```



Record your username and password in a safe place where you can easily find them – you will need them to perform some Bash shell operations later.



Advanced Package Tool

Initializing the Distro

Following installation of a Linux distro in WSL, as described on page 11, it is recommended that you update the distro.

The Linux operating system consists of separate “packages” that are indexed in a catalog, which can be updated by the system’s package manager. In the Ubuntu distro the package manager is the **Advanced Package Tool (apt)**. This package manager can update the package catalog with the command **apt update** and can upgrade the packages with the command **apt upgrade**.

Changes to the system require administrator privileges so each of these commands must therefore be preceded by a **sudo** command:

- 1 Click Start, **Ubuntu** to open a Terminal window
- 2 At the prompt, type this command, then hit **Return**
sudo apt update
- 3 Now, enter the password you chose for your Linux account, then hit **Return** to update the catalog



You can discover more about the **sudo** command on page 82.

```
mike@win-pc: ~$ sudo apt update
[sudo] password for mike:
Hit:1 http://archive.ubuntu.com/ubuntu bionic InRelease
Hit:2 http://archive.ubuntu.com/ubuntu bionic-updates InRelease
Hit:3 http://archive.ubuntu.com/ubuntu bionic-backports InRelease
Get:4 http://security.ubuntu.com/ubuntu bionic-security InRelease [83.2 kB]
Fetched 83.2 kB in 7s (12.3 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
17 packages can be upgraded. Run 'apt list --upgradable' to see them.
mike@win-pc: ~$
```

- 4 Type a **clear** command and hit **Return** (or press **Ctrl + L** keys) to clear the Terminal window to a prompt
- 5 At the prompt, type this command, then hit **Return**
sudo apt upgrade
- 6 Read the summary of available upgrades, then type **Y** and hit **Return** to continue

...cont'd

```
mike@win-pc -
mike@win-pc:~$ sudo apt upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
The following packages will be upgraded:
  apt apt-utils base-files cloud-init console-setup console-setup-linux
  keyboard-configuration libapt-inst2.0 libapt-pkg5.0 liblxc-common liblxc1
  libmspack0 libxml2 lshw open-iscsi python3-distupgrade
  ubuntu-release-upgrader-core
17 upgraded, 0 newly installed, 0 to remove and 0 not upgraded.
Need to get 6200 kB of archives.
After this operation, 38.9 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y_
```



For further help with installation of Linux in the WSL you should refer to the Troubleshooting page online at docs.microsoft.com/en-us/windows/wsl/troubleshooting

7

See the available replacement packages now get downloaded and unpacked to upgrade your system

```
mike@win-pc -
Preparing to unpack .../04-open-iscsi_2.0.874-5ubuntu2.1_amd64.deb ...
Unpacking open-iscsi (2.0.874-5ubuntu2.1) over (2.0.874-5ubuntu2) ...
Preparing to unpack .../05-libxml2_2.9.4+dfsg1-6.1ubuntu1.2_amd64.deb ...
Unpacking libxml2:amd64 (2.9.4+dfsg1-6.1ubuntu1.2) over (2.9.4+dfsg1-6.1ubuntu1) ...
Preparing to unpack .../06-lshw_02.10-0.1ubuntu6.18.04.1_amd64.deb ...
Unpacking lshw (02.10-0.1ubuntu6.18.04.1) over (02.10-0.1ubuntu6) ...
Preparing to unpack .../07-ubuntu-release-upgrader-core_153a18.04.24_all.deb ...
Unpacking ubuntu-release-upgrader-core (1:18.04.24) over (1:18.04.21) ...
Preparing to unpack .../08-python3-distupgrade_153a18.04.24_all.deb ...
Unpacking python3-distupgrade (1:18.04.24) over (1:18.04.21) ...
Preparing to unpack .../09-liblxc-common_3.0.1-0ubuntu1-18.04.2_amd64.deb ...
Unpacking liblxc-common (3.0.1-0ubuntu1-18.04.2) over (3.0.1-0ubuntu1-18.04.1) ...
Preparing to unpack .../10-liblxc1_3.0.1-0ubuntu1-18.04.2_amd64.deb ...
Unpacking liblxc1 (3.0.1-0ubuntu1-18.04.2) over (3.0.1-0ubuntu1-18.04.1) ...
Preparing to unpack .../11-libmspack0_0.6-3ubuntu0.1_amd64.deb ...
Unpacking libmspack0:amd64 (0.6-3ubuntu0.1) over (0.6-3) ...
Preparing to unpack .../12-cloud-init_18.3-9-g2e62cb8a-0ubuntu1-18.04.2_all.deb ...
Progress: [ 64%] [#####]
```



8

Type a **clear** command and hit **Return** (or press **Ctrl + L** keys) to clear the Terminal window to a prompt

9

Now, exactly type **bash --version** then hit **Return** to discover the current Bash version – identical to that on the Linux system illustrated on page 9

```
mike@win-pc -
mike@win-pc:~$ bash --version
GNU bash, version 4.4.19(1)-release (x86_64-pc-linux-gnu)
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>

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

The Bash versions are identical on this Windows 10 PC and the Linux PC, so the vast majority of examples in this book can run on either PC. Most screenshots illustrate Bash in WSL on Windows but would appear identical in a Linux Terminal. There are, however, a few examples that are illustrated in a Linux Terminal as these require a graphical Linux interface or a multi-user Linux environment.



Each 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 to 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 the command **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

```
mike@win-pc: ~
mike@win-pc:~$ type clear
clear is /usr/bin/clear
mike@win-pc:~$ type exit
exit is a shell builtin
mike@win-pc:~$ type nosuch
-bash: type: nosuch: not found
mike@win-pc:~$
```

...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 an **-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

```
mike@win-pc: ~  
mike@win-pc:~$ echo Bash in easy steps  
Bash in easy steps  
mike@win-pc:~$ echo -n Bash in easy steps  
Bash in easy stepsmike@win-pc:~$
```

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, enter **echo \$SHELL** to see the location of the Bash interpreter program on the filesystem
- 7 Now, enter **echo \$BASH_VERSION** to see the version number of the Bash shell interpreter

```
mike@win-pc: ~  
mike@win-pc:~$ echo $SHELL  
/bin/bash  
mike@win-pc:~$ echo $BASH_VERSION  
4.4.19(1)-release  
mike@win-pc:~$
```



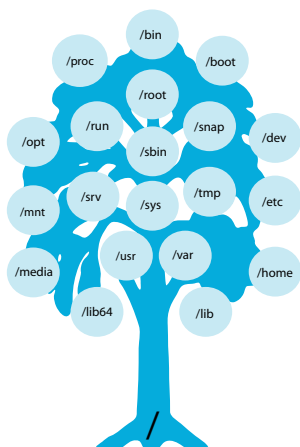
You will often want to suppress the automatic newline with **echo -n** when printing a request for user input.



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

Navigating the File System

The Linux filesystem is arranged in a tree-like hierarchy of directories and files, with the “root” directory at its base:



- The root directory is simply addressed by a / forward slash.
- Sub-directories in the root directory are addressed by appending their directory name to the forward slash. For example, the “home” directory has the address **/home**.
- Directories in sub-directories are addressed by appending another forward slash and their name. For example, a “user” directory in the “home” directory has the address **/home/user**.
- 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 you launch a Terminal window you are located in your home directory – a directory bearing your username that is located in **/home**, such as **/home/mike**. You can see your 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:

- 1 Launch a Terminal, then enter **pwd** at the prompt to see the absolute path address of your current location
- 2 Next, enter **ls /** to see the contents of the root directory



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



Here, listed directories are colored blue, executable files are colored green, and other writables are colored black-on-green.

```
mike@win-pc: ~
mike@win-pc:~$ pwd
/home/mike
mike@win-pc:~$ ls /
bin  dev  home  lib  media  opt  root  sbin  srv  tmp  var
boot  etc  init  lib64  mnt  proc  run  snap  sys  usr
```


...cont'd

- Now, enter **ls /bin** to see the many executable binary files you can run from the command line

```
mike@win-pc:/home$ ls /bin
bash             chgrp             getfacl           mkfs.btrfs        ntfsmove          setupcon          ulockmgr_server
btrfs            chmod             grep              mknod             ntfsrecover       sh               umount
btrfs-debug-tree chown             gunzip            mktemp            ntfssecaudit      sh.distrib       uname
btrfs-find-root chvt              grxex             more              ntfsstruncate     sleep            uncompress
btrfs-image      cp                gzip              mount             ntfsusermap       ss               unicode_start
btrfs-map-logical cpio              hostname          mountpoint         ntfswipe           static-sh        vdir
btrfs-select-super dash              ip                mt                ntfswipe           stty             wdiff
btrfs-zero-log   date             journalctl        nt-gnu            ntfsxattr          su               which
btrfsck          dd               kbd_mode          mv                ntfsxattr          sync             whiptail
btrfstuna        df               kill              nano              ntfsxattr          systemdctl       wslpath
bunzip2          dmesg            kmod              nc                ntfsxattr          systemd          ypserv
busybox          dnsdomainname    less              nc.openbsd        ntfsxattr          systemd-ask-password  zcat
bzip             dircat           lessecho          netcat            ntfsxattr          systemd-escape    zdiff
bzip2            domainname       lessfile          netstat           ntfsxattr          systemd-hwdb      zegrep
bzdiff           dumpkeys         lesskey           networkctl        ntfsxattr          systemd-inhibit    zfgrep
bzegrep          echo             lesspipe          nlsdomainname     ntfsxattr          systemd-machine-id-setup  zforce
bzexe            ed               ln                ntfs-3g           ntfsxattr          systemd-notify     zgrep
bzfgrep          egrep            loadkeys          ntfs-3g.probe     ntfsxattr          systemd-sysusers   zless
bzgrep           false            login             ntfscluster       ntfsxattr          systemd-tmpfiles    zmore
bzlib2           fgconsole        loginsctl         ntfsctl           ntfsxattr          tar               znew
bzlib2recover    fgrep            ls                ntfsctl           ntfsxattr          run-parts          touch
bzless           findant          lsblk             ntfsctl           ntfsxattr          sed               true
bzmore           fsck.btrfs       lsmod             ntfsctl           ntfsxattr          setfont           udevadm
cat              fuser            lsmod             ntfsctl           ntfsxattr          setfont           udevadm
chacl            fusermount       mkdir             ntfsctl           ntfsxattr          setfont           udevadm
```

You can change location into a directory using the **cd** command. Additionally, your home directory can be addressed using a tilde **~** alias, and the parent of the current directory can be addressed using a **..** alias. The Bash prompt string typically displays the current directory just before the **\$** sign:

- Enter **cd /bin** to see the prompt change to display the sub-directory location
- Enter **cd ..** then hit **Return** to move to the parent directory, and see the prompt string change once more
- Finally, enter **cd ~** to return to your home directory, and see the prompt string change again

```
mike@win-pc: ~$ cd /bin
mike@win-pc:/bin$ cd ..
mike@win-pc:/:$ cd ~
mike@win-pc:~$
```



Here, all listed files are executable but those colored white-on-red set the user ID when run, and those colored light blue are (symbolic) links to the actual location. You can learn more about links on pages 32-33.



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



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

Dealing Wildcards

The **ls** command, introduced on pages 16-17, will list all files and folders in the current or specified directory except special hidden files whose names begin with a . period (full stop) character. Typically these are system files, such as a **.bashrc** hidden file in your home directory containing the shell configuration details. Hidden files can be included in the list displayed by the **ls** command by adding an **-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 **pic.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 a current `/home/Documents` directory
- 2 Next, enter `ls doc.?` to see all files named “doc” that have a single-letter file extension
- 3 Now, enter `ls *.c` to see all files of any name that have a “.c” file extension
- 4 Enter `ls *.[a-c]` to see all files of any name that have a “.a”, “.b” or “.c” file extension
- 5 Now, enter `ls *.[!a-c]` to see all files of any name that do not have a “.a”, “.b” or “.c” file extension
- 6 Finally, enter `ls non*` to see an error message reporting that no matches have been found



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

```
mike@win-pc: ~/Documents
mike@win-pc:~/Documents$ ls
doc.a doc.b doc.c doc.d txt.a txt.b txt.c txt.d
mike@win-pc:~/Documents$ ls doc.?
doc.a doc.b doc.c doc.d
mike@win-pc:~/Documents$ ls *.c
doc.c txt.c
mike@win-pc:~/Documents$ ls *.[a-c]
doc.a doc.b doc.c txt.a txt.b txt.c
mike@win-pc:~/Documents$ ls *.[!a-c]
doc.d txt.d
mike@win-pc:~/Documents$ ls non*
ls: cannot access 'non*': No such file or directory
mike@win-pc:~/Documents$
```



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

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:



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

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



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

```
mike@win-pc: ~
mike@win-pc:~$ echo b{ad,oy}
bad boy
mike@win-pc:~$ echo {ge,fi}t
get fit
mike@win-pc:~$ echo s{i,a,o,u}ng
sing sang song sung
mike@win-pc:~$ echo s{tr{i,o},a,u}ng
string strong sang sung
mike@win-pc:~$ 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
mike@win-pc:~$ echo {1..20}
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
mike@win-pc:~$
```

...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 misinterpretation.



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

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, enter **echo 'Processed By: \$SHELL'** to see the shell variable printed literally in output

```
mike@win-pc: ~
mike@win-pc:~$ echo Processed By: $SHELL
Processed By: /bin/bash
mike@win-pc:~$ echo 'Processed By: $SHELL'
Processed By: $SHELL
mike@win-pc:~$
```

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, enter **echo Processed By: \ \$SHELL** to see the shell variable printed literally in output

```
mike@win-pc: ~
mike@win-pc:~$ echo Processed By: $SHELL
Processed By: /bin/bash
mike@win-pc:~$ echo Processed By: \ $SHELL
Processed By: $SHELL
mike@win-pc:~$
```

...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, enter **echo It\'s escaped** 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

```
mike@win-pc: ~  
mike@win-pc:~$ echo It\'s escaped  
It's escaped  
mike@win-pc:~$ echo Continued \  
> text written along \  
> several lines  
Continued text written along several lines  
mike@win-pc:~$ _
```



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

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 Now, enter **echo '"Interpreted With \$SHELL"'** to see the shell variable printed literally in quoted output

```
mike@win-pc: ~  
mike@win-pc:~$ echo "Interpreted With $SHELL"  
Interpreted With /bin/bash  
mike@win-pc:~$ echo '"Interpreted With $SHELL"'  
"Interpreted With $SHELL"  
mike@win-pc:~$ _
```



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

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:



The **|** character is a “pipe” that allows output to be redirected – here output is sent to the **more** command. Pipelines are described in more detail on pages 48-49.

1

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

```
mike@win-pc: ~
mike@win-pc:~$ help | more
GNU bash, version 4.4.19(1)-release (x86_64-pc-linux-gnu)
These shell commands are defined internally.
job_spec [&]                history [-c] [-d offset] [n] or>
(( expression ))           if COMMANDS; then COMMANDS; [ e>
. filename [arguments]     jobs [-lnprs] [jobspec ...] or >
:                           kill [-s sigspec | -n signum | >
[ arg... ]                 let arg [arg ...]
[[ expression ]]           local [option] name[=value] ...
alias [-p] [name[=value] ... ] logout [n]
bg [job_spec ...]          mapfile [-d delim] [-n count] [>
bind [-lpsvPSVX] [-m keymap] [-f>
break [n]                  popd [-n] [+N | -N]
builtin [shell-builtin [arg ...]>
caller [expr]              printf [-v var] format [argumen>
case WORD in [PATTERN] [PATTERN>
cd [-L] [-P [-e]] [-@]] [dir] pushd [-n] [+N | -N | dir]
--More--                  pwd [-LP]
                           read [-ers] [-a array] [-d deli>
                           readarray [-n count] [-O origin>
```

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 shell prompt

3

Now, enter **help echo** to display information about the Bash shell built-in **echo** command

```
mike@win-pc: ~
mike@win-pc:~$ help echo
echo: echo [-neE] [arg ...]
Write arguments to the standard output.

Display the ARGs, separated by a single space character and followed by a
newline, on the standard output.

Options:
-n          do not append a newline
-e          enable interpretation of the following backslash escapes
-E          explicitly suppress interpretation of backslash escapes
```



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

...cont'd

Information about all commands, both shell built-in commands and those other commands that are actually programs located on the file system, can be found on any Linux 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, an **-f** option can be used to display a brief description of a command:

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

```
mike@win-pc: ~
PS(1)                                User Commands                                PS(1)
NAME
    ps - report a snapshot of the current processes.
SYNOPSIS
    ps [options]
DESCRIPTION
    ps displays information about a selection of the active
    processes.  If you want a repetitive update of the
    selection and the displayed information, use top(1)
    instead.

    This version of ps accepts several kinds of options:

Manual page ps(1) line 1 (press h for help or q to quit)
```

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

- 6 Now, enter **man -f ps** to display the “what is” description of the **ps** command

```
mike@win-pc: ~
mike@win-pc:~$ man ps
mike@win-pc:~$ man -f ps
ps (1)
mike@win-pc:~$ - report a snapshot of the current processes.
mike@win-pc:~$ _
```



You can use the **type** command, described on page 14, to discover whether a command is a shell built-in or its file system 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 a Linux operating system.
- The command **ps \$\$_** displays the current process information and can be used to confirm Bash as the current shell.
- Usernames and hostnames 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.