Chapter 1. bash Basics

The Bourne Again Shell

The Bourne Again shell (named in punning tribute to Steve Bourne's shell) was created for use in the GNU project.

Generically speaking, a shell is any user interface to the UNIX operating system, i.e., any program that takes input from the user, translates it into instructions that the operating system can understand, and conveys the operating system's output back to the user.

Interactive Shell Use

When you use the shell interactively, you engage in a login session that begins when you log in and ends when you type exit or logout or press CTRL-D. During a login session, you type in command lines to the shell; these are lines of text ending in RETURN that you type into your terminal or workstation.

Commands, Arguments, and Options

Shell command lines consist of one or more words, which are separated on a command line by blanks or TABs. The first word on the line is the command. The rest (if any) are arguments (also called parameters) to the command, which are names of things on which the command will act.

E.G.

```
$ ls myvm_key.pem
myvm_key.pem

$ ls *
agent.jar anaconda-ks.cfg  myvm_key.pem  nats-server-v2.9.20-linux-
amd64.zip  raaje01.tar
$ free -m
                     total        used        free      shared
buff/cache   available
Mem:            2786         288        2035          16         462
2322
Swap:           2047           0        2047

$ ls -l
total 255032
-rw-r--r--. 1 root root   1370647 Jul 19 18:42 agent.jar
-rw-------. 1 root root      1108 Jul 19 16:34 anaconda-ks.cfg
drwxr-xr-x. 2 root root       123 Jul 19 17:00 k8s_manifest
-rw-r--r--. 1 root root      2498 Jul 24 09:54 myvm_key.pem
drwxr-xr-x. 2 root root        57 Jul 24 19:05 nats-server-v2.9.20-
linux-amd64
-rw-r--r--. 1 root root   5117831 Jul 14 01:43 nats-server-v2.9.20-
linux-amd64.zip
-rw-r--r--. 1 root root 254648832 Jul 19 17:06 raaje01.tar
```

Filenames, Wildcards, and Pathname Expansion

Filenames are so important in UNIX that the shell provides a built-in way to specify the pattern of a set of filenames without having to know all the names themselves. You can use special characters, called wildcards, in filenames to turn them into patterns.

Basic wildcards

Wildcard	Matches
?	Any single character
*	Any string of characters
[set]	Any character in set
[! set]	Any character not in set

Using the * wildcard

Expression	Yields
fr*	frank fred
*ed	ed fred
b* bob	bob
e	darlene dave ed fred google
r	darlene frank fred
*	bob darlene dave ed frank fred google
d*e	darlene dave
g*	google

Using the set construct wildcards

Expression	Yields
[abc]	a, b, or c
[.,;]	Period, comma, or semicolon
[-_]	Dash or underscore
[a-c]	a, b, or c
[a-z]	All lowercase letters
[!0-9]	All non-digits
[0-9!]	All digits and exclamation point
[a-zA-Z]	All lower- and uppercase letters
[a-zA-Z0-9_-]	All letters, all digits, underscore, and dash

Brace Expansion

```
$ echo b{ed,olt,ar}s,
beds, bolts, bars,
$ echo b{ar{d,n,k},ed}s.
bards. barns. barks. beds.
$ echo b{ar{d,n,k},ed}s
bards barns barks beds
$ echo {2..5}
2 3 4 5
$ echo {d..h}
```

```
d e f g h
$ ls -l *.{c,p,t}*
-rw-------. 1 root root      1108 Jul 19 16:34 anaconda-ks.cfg
-rw-r--r--. 1 root root      2498 Jul 24 09:54 myvm_key.pem
-rw-r--r--. 1 root root 254648832 Jul 19 17:06 raaje01.tar
```

Input and Output

Standard I/O:

A single way of accepting input called standard input,
a single way of producing output called standard output.
a single way of producing error messages called standard error output, usually shortened to standard error.

Popular data filtering utilities

cat : Copy input to output
grep: Search for strings in the input
sort: Sort lines in the input
cut: Extract columns from input, some older BSD-derived systems don't have *cut*, but
You can use awk instead. Whenever you see a command of the form: `cut -f N -d C filename`, use this instead.

```
$awk -F C '{print $ N}' filename.
```

sed: Perform editing operations on input.
tr : Translate characters in the input to other characters.
Awk :

I/O Redirection

You redirect standard input so that it comes from a file. The notation
$ command < filename does this; it sets things up so that command takes standard input from a file instead of from a terminal.

```
$ cat < list.txt
total 255036
-rw-r--r--. 1 root root   1370647 Jul 19 18:42 agent.jar
-rw-------. 1 root root      1108 Jul 19 16:34 anaconda-ks.cfg
drwxr-xr-x. 2 root root       123 Jul 19 17:00 k8s_manifest
-rw-r--r--. 1 root root         0 Oct 21 01:24 list.txt
-rw-r--r--. 1 root root      2498 Jul 24 09:54 myvm_key.pem
drwxr-xr-x. 2 root root        57 Jul 24 19:05 nats-server-v2.9.20-
linux-amd64
-rw-r--r--. 1 root root   5117831 Jul 14 01:43 nats-server-v2.9.20-
linux-amd64.zip
-rw-r--r--. 1 root root      1108 Oct 21 01:20 out.file
-rw-r--r--. 1 root root 254648832 Jul 19 17:06 raaje01.tar
```

Similarly, *$ command > filename* causes the command's standard output to be redirected to the named file.

```
$ date > now
$ cat now
Sat Oct 21 01:28:03 IST 2023

$ cat < file1 > file2
```

This would be like

```
$ cp file1 file2.

Adding error output to standard output
$ command > logs 2>&1
this sends both standard output & Standard error to logs
```

Pipelines

It is also possible to redirect the output of a command into the standard input of another command instead of a file. The construct that does this is called the pipe, notated as |.

```
$ cut -d: -f1 < /etc/passwd | sort
$ cut -d: -f1 < /etc/passwd | sort | wc -l
```

Background Jobs

If you want to run a command that does not require user input and you want to do other things while the command is running, put an ampersand (&) after the command. This is called running the command in the background, and a command that runs in this way is called a background job; by contrast, a job run the normal way is called a foreground job.

E.G.

```
$ sleep 5 &
[1] 2349
$
[1]+  Done                    sleep 5
```

Spinning up multiple parallel processes

```
$ for i in `seq 5`
> do
> sleep 5 &
> done
[1] 2352
[2] 2353
[3] 2354
[4] 2355
[5] 2356
$
[1]   Done                    sleep 5
[2]   Done                    sleep 5
[3]   Done                    sleep 5
[4]   Done                    sleep 5
[5]   Done                    sleep 5
$
```

Special Characters

Character	Meaning
~	Home directory
`	Command substitution (archaic)
#	Comment
$	Variable expression
&	Background job
*	String wildcard
(	Start subshell
)	End subshell
\	Quote next character
|	Pipe
[	Start character-set wildcard
]	End character-set wildcard
{	Start command block
}	End command block
;	Shell command separator
'	Strong quote
<">	Weak quote
<	Input redirect
>	Output redirect
/	Pathname directory separator
?	Single-character wildcard
!	Pipeline logical NOT

Quoting

Sometimes you will want to use special characters literally, i.e., without their special meanings. This is called *quoting.*

E.G.

```
$ my="Sandip"
$ echo $my
Sandip
$ echo "My name is $my"   # Soft Quote
My name is Sandip
$ echo ' My name is $my'   # Hard Quote
My name is $my
```

Backslash-Escaping

Another way to change the meaning of a character is to precede it with a backslash (\). This is called backslash-escaping the character. In most cases, when you backslash-escape a character, you quote it.

E.G.

```
$ echo 2 * 3 > 5 is a valid inequality.

$
```

```
No output!
$ echo "2 * 3 > 5 is a valid inequality."
2 * 3 > 5 is a valid inequality.
$ echo 2 \* 3 \> 5 is a valid inequality.
2 * 3 > 5 is a valid inequality.
$ echo This is Sandip's shell
> ^C
$ echo This is Sandip\'s shell
This is Sandip's shell
$ echo "This is Sandip's shell"
This is Sandip's shell
```

Continuing Lines

A related issue is how to continue the text of a command beyond a single line on your terminal or workstation window. The answer is conceptually simple: just quote the RETURN key. After all, RETURN is just another character.

```
$ echo Line one \
> Line two \
> Line three \
> Last Line
Line one Line two Line three Last Line
```

Quoting Quotation Marks

```
$ echo \"2 \* 3 \> 5\" is a valid inequality.
```

produces the following output:

```
"2 * 3 > 5" is a valid inequality.
```

Control Keys

Control keys—those that you type by holding down the CONTROL (or CTRL) key and hitting another key—are another type of special character. These normally don't print anything on your screen, but the operating system interprets a few of them as special commands.

Control Key	stty Name	Function Description
CTRL-C	intr	Stop current command
CTRL-D	eof	End of input
CTRL-\	quit	Stop current command, if CTRL-C doesn't work
CTRL-S	stop	Halt output to screen
CTRL-Q		Restart output to screen
DEL or CTRL-?	erase	Erase last character
CTRL-U	kill	Erase entire command line
CTRL-Z	susp	Suspend current command

The resulting output will include this information:

```
intr = ^c; quit = ^|; erase = DEL; kill = ^u; eof = ^d; eol swtch =
^`; susp = ^z; dsusp <undef>;
```

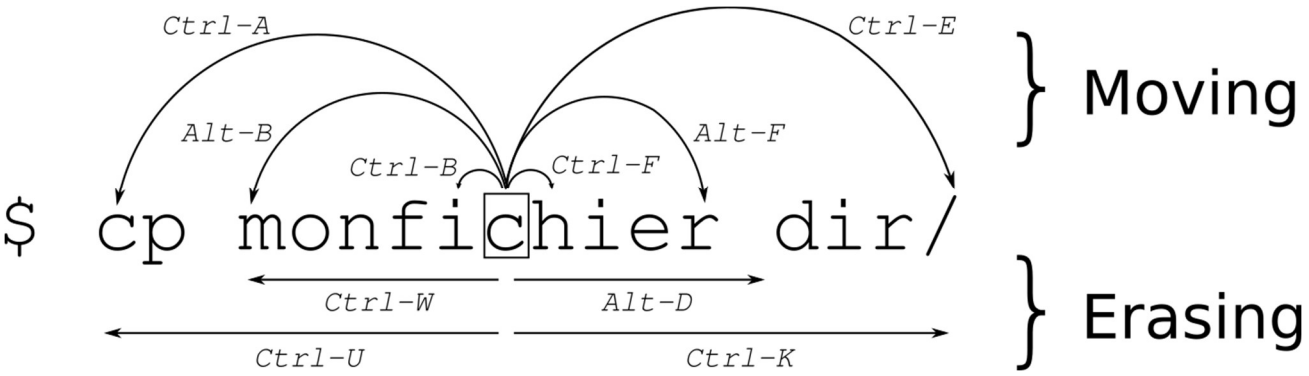
Chapter 2. Command-Line Editing

It's always possible to make mistakes when you type at a computer keyboard, but perhaps even more so when you are using a UNIX shell.

Word, Line Commands

bash initially starts interactively with emacs-mode as the default (unless you have started bash with the **-noediting** option. There are two ways to enter either editing mode while in the shell. First, you can use the set command:

```
$ set -o emacs
or:
$ set -o vi
```



Source

Moving

command	description
ctrl + a	Goto BEGINNING of command line
ctrl + e	Goto END of command line
ctrl + b	move back one character
ctrl + f	move forward one character
alt + f	move cursor FORWARD one word
alt + b	move cursor BACK one word
ctrl + xx	Toggle between the start of line and current cursor position
ctrl +] + x	Where x is any character, moves the cursor forward to the next occurrence of x
alt + ctrl +] + x	Where x is any character, moves the cursor backwards to the previous occurrence of x

Edit/Other

command	description
ctrl + d	Delete the character under the cursor
ctrl + h	Delete the previous character before cursor
ctrl + u	Clear all / cut BEFORE cursor
ctrl + k	Clear all / cut AFTER cursor
ctrl + w	delete the word BEFORE the cursor
alt + d	delete the word FROM the cursor
ctrl + y	paste (if you used a previous command to delete)
ctrl + i	command completion like Tab
ctrl + l	Clear the screen (same as clear command)
ctrl + c	kill whatever is running
ctrl + d	Exit shell (same as exit command when cursor line is empty)
ctrl + z	Place current process in background
ctrl + _	Undo
ctrl + x ctrl + u	Undo the last changes. ctrl+ _ does the same
ctrl + t	Swap the last two characters before the cursor
esc + t	Swap last two words before the cursor
alt + t	swap current word with previous
esc + .	NO DESCIPRITION IN REF
esc + _	NO DESCIPRITION IN REF
alt + [Backspace]	delete PREVIOUS word
alt + <	Move to the first line in the history
alt + >	Move to the end of the input history, i.e., the line currently being entered
alt + ?	display the file/folder names in the current path as help
alt + *	print all the file/folder names in the current path as parameter
alt + .	print the LAST ARGUMENT (ie "vim file1.txt file2.txt" will yield "file2.txt")
alt + c	capitalize the first character to end of word starting at cursor (whole word if cursor is at the beginning of word)
alt + u	make uppercase from cursor to end of word
alt + l	make lowercase from cursor to end of word
alt + n	
alt + p	Non-incremental reverse search of history.
alt + r	Undo all changes to the line
alt + ctl + e	Expand command line.
~[TAB][TAB]	List all users
$[TAB][TAB]	List all system variables
@[TAB][TAB]	List all entries in your /etc/hosts file
[TAB]	Auto complete
cd -	change to PREVIOUS working directory

History

command	description
ctrl + r	Search backward starting at the current line and moving 'up' through the history as necessary
crtl + s	Search forward starting at the current line and moving 'down' through the history as necessary
ctrl + p	Fetch the previous command from the history list, moving back in the list (same as up arrow)
ctrl + n	Fetch the next command from the history list, moving forward in the list (same as down arrow)
ctrl + o	Execute the command found via Ctrl+r or Ctrl+s
ctrl + g	Escape from history searching mode
!!	Run PREVIOUS command (ie sudo !!)
!vi	Run PREVIOUS command that BEGINS with vi
!vi:p	Print previously run command that BEGINS with vi
!n	Execute nth command in history
!$	Last argument of last command
!^	First argument of last command
^abc^xyz	Replace first occurance of abc with xyz in last command and execute it

Textual Completion

One of the most powerful (and typically underused) features of emacs-mode is its *textual completion* facility.

command	description
TAB	Attempt to perform general completion of the text
ESC-?	List the possible completions
ESC-/	Attempt filename completion
CTRL-X /	List the possible filename completions
ESC-~	Attempt username completion
CTRL-X ~	List the possible username completions
ESC-$	Attempt variable completion
CTRL-X $	List the possible variable completions
ESC-@	Attempt hostname completion
CTRL-X @	List the possible hostname completions
ESC-!	Attempt command completion
CTRL-X !	List the possible command completions
ESC-TAB	Attempt completion from previous commands in the history list

MISC: Several miscellaneous commands complete *emacs* editing mode.

command	description
CTRL-J	Same as RETURN
CTRL-L	Clears the screen, placing the current line at the top of the screen
CTRL-M	Same as RETURN
CTRL-O	Same as RETURN, then display next line in command history
CTRL-T	Transpose two characters on either side of point and move point forward by one
CTRL-U	Kills the line from the beginning to point
CTRL-V	Quoted insert
CTRL-[	Same as ESC (most keyboards)
ESC-C	Capitalize word after point
ESC-U	Change word after point to all capital letters
ESC-L	Change word after point to all lowercase letters
ESC-.	Insert last word in previous command line after point
ESC-_	Same as ESC-.

VI

To Start VI

Command	Effect
vi filename	edit *filename* starting at line 1
vi +n filename	edit *filename* beginning at line n
vi +filename	edit *filename* beginning at the last line
vi -r filename	recover *filename* after a system crash
vi +/patter filename	edit *filename* starting at the first line containing **pattern**

VI Command Mode vs. Insert Mode

Insert mode is the mode to be in when inserting text into the file. Command mode is the mode to be in when giving commands which will move the cursor, delete text, copy and paste, save the file etc.

Command	Insert Text
i	before cursor
a	after cursor
A	at the end of the line
o	open a line below the current line
O	open a line above the current line

r	replace the current character
R	replace characters until <ESC>, overwrite

To move the cursor:

You must be in Command Mode to use commands that move the cursor. Each of these commands can be preceded with a Repeat Factor.
Examples:
8j will move the cursor down 8 lines.
3w will move the cursor 3 words to the right.

Command	**Moves the cursor**
SPACE, l (el), or right arrow	space to the right
h or left arrow	space to the left
j or down arrow	down one line
k or up arrow	up one line
w	word to the right
b	word to the left
$	end of the line
0 (zero)	beginning of the line
e	end of the word to the right
-	beginning of previous line
)	end of the sentence
(	beginning of the sentence
}	end of paragraph
{	beginning of paragraph

To Delete Text

Command	**Action**
d0	delete to beginning of line
dw	delete to end of word
d3w	delete to end of third word
db	delete to beginning of word
dW	delete to end of blank delimited word
dB	delete to beginning of blank delimited word
dd	delete current line

5dd	delete 5 lines starting with the current line
dL	delete through the last line on the screen
dH	delete through the first line on the screen
d)	delete through the end of the sentence
d(	delete through the beginning of the sentence
x	delete the current character
nx	delete the number of characters specified by n.
nX	delete n characters before the current character

Viewing Different Parts of the Work Buffer:

^Character means that you should hold down the Control key while striking the indicated character key.

Command	**Moves the cursor**
^D	forward one-half screenful
^U	backward one-half screenful
^F	forward one screenful
^B	backward one screenful
nG	to line n (Ex: 25G moves the cursor to line #25)
H	to the top of the screen
M	to the middle of the screen
L	to the bottom of the screen
^L	refresh the screen

Yanking (copy) and Putting (paste) Text:

Example: **3yy** will yank (copy) 3 lines

p will put the 3 lines just yanked on the line below the current cursor.

In the following list **M** is a Unit of Measure that you can precede with a Repeat Factor, n.

Command	Effect
yM	yank text specified by M
y3w	yank 3 words
nyy	yank **n** lines
Y	yank to the end of the line
P	put text above current line
p	put text below current line

Changing Text

Example: ***cw*** *allows you to change a word. The word may be replaced by as many word as needed. Stop the change by hitting < esc &gt.*
c3w allows you to change 3 words.

Ending an Editing Session

Command	Effect
:w	writes the contents of the work buffer to the file
:q	quit
:q!	quit without saving changes
ZZ	save and quit
:wq	save and quit
:w *filename*	saves to *filename* (allows you to change the name of the file)

Miscellaneous commands

Command	Effect
J	join the current line and the following line
:set number	number the lines on the screen (not actually added to file)
:set nonumber	turns off numbering of lines
:r *filename*	reads *filename* into the current file at the location of the cursor
:set showmode	displays INPUT MODE at the lower right hand corner of screen
~	change uppercase to lowercase and vice-versa