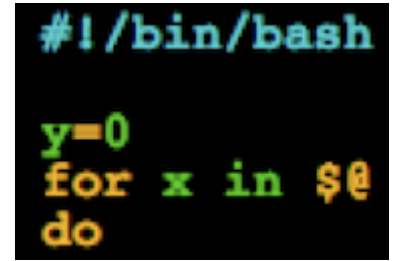


Introduction to Bash



```
#!/bin/bash  
  
y=0  
for x in $@  
do
```

It's a good idea to be "command-line" literate. Seriously a very good idea. But rather than just memorizing twenty commands, you should really get into a good shell, like Bash.

Overview

Bash, the Bourne-Again Shell, refers both to a particular Unix shell program and its associated scripting language. It is the default shell of the GNU Operating System (Linux) and Apple's OS X and is POSIX 1003.2 compliant. It is a powerful shell, and features, among other things:

- Command line editing
- Command history
- A directory stack (**pushd**, **popd**)
- Command substitution
- Special variables, like \$PPID
- Autocompletion
- In-process integer arithmetic: **\$((...))**
- In-process regexes
- Aliases
- Functions
- Arrays
- Expansions: tilde, brace, variable
- Substring awesomeness
- Conditional expressions
- Security (restricted shell mode)
- Job Control
- Timing
- Prompt customization

Bash, the Shell

What's a Shell?

A shell is a command interpreter. Commands can be executable files or built-ins. Commands can be bundled together into a *script* which a shell program executes. How the commands are packaged and wired together, using variables, functions, and control-flow operators makes up the shell's *scripting language*.

Examples of Shells

Organized by family:

- Bourne family: sh, ash, zsh, ksh, **bash**
- C family: csh, tcsh
- Perl family: perlsh, zoidberg
- Plan9 family: rc, es
- Secure/Restricted family: ibsh, rssh, scponly
- Microsoft family: cmd.exe, Windows PowerShell

Also see [Wikipedia's Unix shell page](#) and their [Shell comparison page](#).

Using Bash

Like all shells, bash can be run interactively or non-interactively. An example interactive session:

```
ray@siouxsie:~$ x=4
ray@siouxsie:~$ echo x
x
ray@siouxsie:~$ echo $x
4
ray@siouxsie:~$ echo "hello"
hello
ray@siouxsie:~$ echo $PS1
\u@\h:\w\$
ray@siouxsie:~$ PS1="\w\$ "
~$ mkdir test
~$ cd test
~/test$ ls
~/test$ cat > message
here is some
text
~/test$ cat message
here is some
```

```

text
~/test$ ls
message
~/test$ ls -l
total 1
-rw-rw---- 1 ray ray 18 Nov 12 22:42 message
~/test$ ls -la
total 9
drwxrwx--- 2 ray ray 2048 Nov 12 22:42 .
drwxr-xr-x 52 ray ray 6144 Nov 12 22:42 ..
-rw-rw---- 1 ray ray 18 Nov 12 22:42 message
~/test$ cp message new
~/test$ ls
message new
~/test$ du
4 .
~/test$ df
Filesystem            1k-blocks      Used Available Use% Mounted on
/dev/hda6              5044156    3518624   1269300   74% /
varrun                 517076         88    516988    1% /var/run
varlock                517076          0    517076    0% /var/lock
udev                  517076        116    516960    1% /dev
devshm                 517076          0    517076    0% /dev/shm
/dev/hda1              1011928         20    960504    1% /rescue
AFS                    9000000          0   9000000    0% /afs
~/test$ rm message
~/test$ help kill
kill: kill [-s sigspec | -n signum | -sigspec] pid | jobspec ... or kill -l [sigspec]
    Send the processes named by PID (or JOBSPEC) the signal SIGSPEC.  If
    SIGSPEC is not present, then SIGTERM is assumed.  An argument of '-l'
    lists the signal names; if arguments follow '-l' they are assumed to
    be signal numbers for which names should be listed.  Kill is a shell
    builtin for two reasons: it allows job IDs to be used instead of
    process IDs, and, if you have reached the limit on processes that
    you can create, you don't have to start a process to kill another one.
~/test$ y = 5
-bash: y: command not found
~/test$ y=5
~/test$ echo x+y
x+y
~/test$ echo $((x+y))
9
~/test$ jobs
~/test$ printf '%d + %d = %8d\n' 9 4 13
9 + 4 =          13
~/test$ seq 4 10 3
~/test$ seq 1 5
1
2
3
4
5
~/test$ for w in $(seq 1 5); do echo $w; done
1
2
3
4
5
~/test$ cat > stuff
first line
second line
third line
fourth line
fourth line
~/test$

```

```
~/test$ sort < stuff
first line
fourth line
fourth line
second line
third line
~/test$ grep t stuff
first line
fourth line
fourth line
~/test$ grep line stuff | sort | uniq
first line
fourth line
second line
third line
~/test$
```

And here are some example scripts:

hello.sh

```
echo Hello World
```

triple.sh

```
for ((c=1; c<=100; c++)); do
    for ((b=1; b<=c; b++)); do
        for ((a=1; a<=b; a++)); do
            if [[ $(( $a * $a + $b * $b )) == $(( $c * $c )) ]]; then
                echo $a $b $c
            fi
        done
    done
done
```

fib.sh

```
# Shows fibonaoacci numbers up to the first command line argument ($1)
a=0
b=1
while (( $b < $1 )); do
    echo "$b"
    olda=$a
    a=$b
    b=$(( $olda + $b ))
done
```

Good to know

Make these snippets of knowledge second nature:

- The word separators (metacharacters) are these seven: `(,), <, >, ;, &, |`
- No spaces on either side of the `"=`" in variable assignments, e.g. `x=4`.

- Expansion does not occur in single quotes but it does in double quotes.
- Get the value of a variable: `$x` or `${x}`.
- Get the result of command execution: ``command args`` or `$(command args)`.

Built-in commands

Remember a command is either an executable file or is built into the shell. These are the builtins (as of Bash 4.2):

:	command	eval	jobs	read	times
.	compgen	exec	kill	readarray	trap
[complete	exit	let	readonly	type
alias	comptopt	export	local	return	typeset
bg	continue	fc	logout	set	ulimit
bind	declare	fg	mapfile	shift	umask
break	dircs	getopts	pushd	shopt	unalias
builtin	disown	hash	popd	source	unset
caller	echo	help	pwd	suspend	wait
cd	enable	history	printf	test	

What about the non-builtin commands?

Aren't **cat**, **cp**, **mv**, **rm**, **ls**, **mkdir**, **less**, **find**, **grep**, **sed**, **cut**, **ps**, **chmod**, and friends part of Bash? No! These commands live in their own executable files. In fact **bash** is itself a command just like them.

There are many places to get information on these commands, including:

- [This command reference at SS64.com](#)
- [Wikipedia's List of Unix Utilities](#)

Using the commandline like a pro

At a minimum you should know these keyboard shortcuts:

Ctrl+A	Cursor to beginning of line
Ctrl+E	Cursor end of line

Ctrl+K	Delete to end of line
Ctrl+_	Undo
Up and down arrows	Previous/next command in history
Left and right arrows	Previous/next character on current line
Tab	Autocompletion
Ctrl+R	Search the history
Ctrl+L	Clear the screen
!!	Repeat last command
!__	Repeat last command beginning with __
Ctrl+C	Interrupt currently running process

But there are *dozens more*. See [this cool reference to Bash keyboard shortcuts at SS64.com](http://ss64.com/bash/keyboard-shortcuts.html) or see the chapter on [command line editing](#) in the Bash Reference Manual.

Special Files

- If invoked as an interactive login shell
 - First **/etc/profile** is executed
 - Then the first file found in the order **~/.bash_profile**, **~/.bash_login**, **~/.profile** is executed
 - Then you get to interact
 - **~/.bash_logout** is executed on logout
- If invoked as a non-login shell
 - **~/.bashrc** is executed first, then you get to interact

Exercise: Many people's ~/.bash_profile contains only the line
if [-f ~/.bashrc]; then . ~/.bashrc; fi.
Why?

Reserved Words

These can't be variables:

```
!           time
[[          ]]      {          }
```

```
if          then      elif      else      fi
case        esac
select      in
while       until     for        do         done
function
```

Command Syntax and Execution

Parses the tokens into simple and compound commands (see Shell Commands). Performs the various shell expansions (see Shell Expansions), breaking the expanded tokens into lists of filenames (see Filename Expansion) and commands and arguments. Performs any necessary redirections (see Redirections) and removes the redirection operators and their operands from the argument list. Executes the command (see Executing Commands). Optionally waits for the command to complete and collects its exit status (see Exit Status).

Bash executes scripts as follows:

1. Tokenizes the input into words and operators, respecting quoting, separating by metacharacters, and applying aliases
2. Parses the tokens into commands. Then for each command:
 - a. Performs expansions
 - b. Figures out all the redirections
 - c. Executes the command
 - d. If not an asynchronous command, waits for it to finish then records its exit status.

For details see Chapter 3 in the Reference Manual.