Bash
page 1 of 2

1 Command-Line Editing

print controls

\	line continuation
in	echo “1”2’"\ ⏎ "3"
out	1 2 3
\t	horizontal tab
\r	carriage return
\n	new line
\b	back space
\\	backslash
\"	double quote
#	comment

write to files

echo “string to write’» ./filename.txt

variables

echo a=string echo b=string2

Control Keys

Use **trap** *to handle signals, which are issued like:*

ctrl-c	stop current command
ctrl-d	end of input
ctrl-\	quit
ctrl-s	halt output to screen
ctrl-q	restart output to screen
ctrl-?	erase last character
ctrl-u	erase entire command line
ctrl-z	suspend current command

Hot Keys (“bindings”)

bind *ties key sequences to certain activities. System-wide bindings are in* `/etc/inputrc` *and user-specific in* `~/.inputrc`. *These customizations control the command-line interpreter, which is named* **readline**.

```
bind '"\C-t": "test"'        # bind string test
bind -x '"\C-t": "date"'     # bind executable
bind -l                      # list bindable f'ns
bind -P                      # list all bindings
```

2 History Expansion

History expansion offers advanced techniques, but it is often better to use emacs or vi-style command editing instead.

Event Designators

!	start hist substitution
!!	last command
!<int>	<int>th command
!-<int>	<int>th previous cmd
!<str>	most recent cmd with <str>
!?<str>?	most recent cmd ∋ <str>
^<s1>^<s2>	repeat match with subs

Word Designators

“Word designators” and “modifiers” must be separated from “event designator” using a colon (:).

0	zero-th word in a line
<int>	<int>th word in a line
^	first argument in a line
$	last arg in a line
%	match of most recent search
<int>-<int>	range of words
*	all words but the first
<int>*	equivalent to <int>-$
<int>-	words from <int> to penult.

Modifiers

h	remove pathname, leaving head
r	remove trailing suffix
e	remove all but suffix
t	remove leading path (tail remains)
p	only print resulting command
q	quote the substituted words
x	ibid, breaking into words
<sed>	<sed> substitution (s/old/new/)

Examples

```
!2048:p          # print 2048th cmd
!!:0 newarg      # last cmd, new args
```

3 Emacs Mode

The default command-line editor; change with:

```
set -o vi        # . . . to vi editor
```

Character Movement / Editing

ctrl-b	back one char
ctrl-f	forward one char
del	delete one char back
ctrl-d	delete one char forward

Word Movement / Editing

esc-b	move one word back
esc-f	move one word forward
esc-del	kill one word back
esc-ctrl-h	kill one word back
esc-d	kill one word forward
ctrl-y	yank last item killed

Line and History Commands

ctrl-a	move to line begin
ctrl-f	move to end of line
ctrl-k	kill to end of line
ctrl-p	move up line
ctrl-n	move to next line
ctrl-r	search backward
esc-<	move to 1st hist line
esc->	move to last hist line
esc-?	expand all choices

Completion Commmands

<tab>	general text completion
esc-?	expand all choices
esc-/	filename completion
ctrl-x	list filename possibilities
esc-~	username completion
ctrl-x ~	list username possibilities
esc-$	attempt variable completion
ctrl-x $	list possible variables
esc-@	attempt hostname completion
ctrl-x @	list hostname possibilities
esc-!	attempt command completion
ctrl-x !	list possible commands
esc-<tab>	attempt completion from hist

4 Environment

Environment Files

Several files control shell settings:

/etc/profile	system settings
~/.bash_profile	user settings, read @ login
~/.bash_login	synonym for ibid
~/.profile	synonym for ibid
~/.bashrc	read @ subshell init
~/.bash_logout	read @ shell logout

Given the difference of when they are read, it is considered best practice to minimize the contents of `.bash_profile` *to commands that write to console; instead, putting most definitions / aliases / etc into* `.bashrc`

Aliases

```
alias name=command        # alias syntax
```

Inspiration:

```
alias cdprj='cd ~/Projects/'   # alias syntax
alias ls='ls -l'               # recursion is ok!
```

Options

Options for **set**; *set/unset like:*
`set -o <opt>;    set +o <opt>`

allexport	noglob (f)	monitor	nounset
Braceexp'nd	Histexp'nd	noexec	verbose
notify (b)	history	Physical	vi
noClobber	hashall	privileged	xtrace
errexit	ignoreeof	posix	-
emacs	keyword	onecmd (t)	

Options for **shopt**; *set/unset like*:
`shopt -s<arg>;    shopt -u<arg>`

cdable_vars	dotglob	histverify	nullglob
cdspell	execfail	hostc'mplete	pr'mptvars
checkhash	exp'd_aliases	int._c'mnts	shift_vrb'se
ch'ckw'nsize	histappend	lithist	sourcepath
cmdhist		mailwarn	

Environment Variables

export *promotes a simple variable into an environment variable, to be inherited by all subshells.*

*	FIGNORE	LC_COLL'G	PS2
@	GL'BLIGN'R	LC_M'SGS	PS3
#	GROUPS	LINENO	PS4
_	IFS	MACHTYPE	PPID
?	HISTCMD	MAIL	**PWD**
_	HISTCTL.	MAILCH'CK	RANDOM
$	HISTFILE	MAILPATH	REPLY
!	HISTF.SIZE	OLDPWD	SECONDS
0	**HOME**	**OPTARG**	**SHELL**
BASH	HOSTFILE	OPTERR	**SHELLOPTS**
BASH_ENV	HOSTNAME	**OPTIND**	SHLVL
BASH_VS'N	HOSTTYPE	OSTYPE	TIMEF'MT
CDPATH	IGNOREEOF	PATH	TMOUT
DIRSTACK	**INPUTRC**	PIPESTAT'	UID
EUID	LANG	P'MPT_C'MD	auto_resume
FCEDIT	LC_ALL	**PS1**	histchars

Directory Search Path

`CDPATH` *gives shell a list of places to look for the location of the argument to* **cd**, *eg:*
CDPATH=:~/Projects/
Alternatively, run **shopt -o cdable_vars** *and then define “cdable” aliases a la:*

```
cs=~/Projects/cheatsheets/   # set normal var
cd cs                        # Δs dir accordingly
```

Prompt String

Prompt can be customized to indicate current directory, date, shell name, and other information. ∃ *4 different prompt variables:* PS1, PS2, PS3, PS4. *For example (more here), set* PS1 *to a string like:*

```
\u@\h \w                  # user, host, & WD
\!|$ str|\$?>             # cmd #, content of str
\e[0;34m\u@\h \w> \e[m    # light blue
[‘myfunc‘]>               # run shell fn
[\$(script.sh)]>          # run shell script
```

5 Types

Similarly-named commands are first taken to be aliases, then keywords, functions, “built-ins”, and finally scripts, in that order. Override this precedence with: **builtin** *or* **command**, *or by disabling higher-precedent commands with* **enable -n**. *Use* **declare** [**afFirx**] *to set a type explicitly, otherwise type string is assumed. Use* **type** [**afptP**] *to ascertain typing information about a variable.*

Variables

Subshells inherit environment variables only; all others remain local to invoking shell. Variables are assumed global unless declared otherwise.

```
myvar="12345"        # basic assignment
source myvar         # promotn to envmt
local mylocvar       # local to a func
echo $myvar          # regurgitate
echo "$myvar"        # more correct!
echo "${myvar}"      # the above does this
```

Functions

```
function myfunc1     # syntax 1
{                    #
```

```
  ...                      # statements
}                          #

myfunc ( )                 # syntax 2
declare -f                 # list all funcs
```

Invoke a function like a command, passing args right afterward. When doing so, positional params ($1, . . .) will reflect args passed to the function.

Strings

"String expansion" is the dereferencing of a variable (assumed a string), per **echo $<var>** *syntax above. "String operators" afford handling of strings, including default values and error messages:*

- ${<var>:-<word>} . . . <var> ? <var> : <word>
- ${<var>:=<word>} . . . above, plus set var=word
- ${<var>:?<msg>} . . . <var> ? <var> : abort
- ${<var>:+<word>} . . . <var> ? <word> : NULL
- ${<var>:<offset>} . . . substring expansion
- ${<var>:<offset>:<len>} . . . ibid

A class of string operators use "pattern matching", which allows for quick (though syntactically obscure) manipulation of string contents:

- ${<var>#<ptrn>} . . . delete shortest match
- ${<var>##<ptrn>} . . . delete longest match
- ${<var>%<ptrn>} . . . delete shortest match
- ${<var>%%<ptrn>} . . . delete longest match
- ${<var>/<ptrn>/<str>} . . . sub 1st <p> with <s>
- ${<var>//<ptrn>/<str>} . . . sub <u>all</u> <p> with <s>

Common pattern-matching idioms:

```
$(path##/*/ )              # only filename
$(path#/*/)                # strip 1st dir
$(path)                    # full path & file
$(path%.*)                 # strip last extension
$(path%%.*)                # strip all .* extens's
```

Output using **echo** *or* **printf** *(advanced), eg:*

```
echo -en "hello\t world"  # note: echo optns
printf "|%10s|\n" hello   # works like C versn
```

Numerics

"Numeric expansion" is indicated by $((<expr>)) *syntax, where* <expr> *allows a wide range of simple mathmatical expressions.*

```
$(( 365 - $(date +%j) ))    # wks to new year ☺
[ \( 2 -gt 2\) && \( 4 -le 1 \) ]     # good
[ $(( (3 > 2) && (4 <= 1) )) = 1 ]  # better
(( (3 > 2) && (4 <= 1) ))             # best
```

Arrays

```
names[2]=alice                  # indexed assign
names=([2]=alice [0]=bob)       # compound asgn
names=(bob '' alice)            # ibid
declare -a myarr                # empty array
for i in "${names[@]}"          # @ → "all"
"${!names[@]}"                  # print all indices
"${#names[@]}"                  # array length
```

Command Substitution

```
$(<command> <arg>*)  # syntax
```

"Command substitution" expands the results of a called command into a string, eg:

```
$(ls $HOME)              # contents of ~/
cd $(DIR_STACK%% *)      # what popd does
```

6 Flow Control

Executed commands run in a subshell; called functions block and run in the same shell. Use **return <int>** *to exit immediately encapsulating function, or* **exit <int>** *to exit script. Loops admit the usual* **break** *syntax.*

Conditions

Success or failure of a command, via its exit status, is sufficient to implement a condition (<cond>) *below. though* **test** *is often used instead. Form logical combinations of* <cond>*s using ORs* (||) *and ANDs* (&&)*, or* **test***'s* **-o**, **-a** *flags, respectively. Negate phrases with exclamation* (**!**)*:*

```
if [<cond>] && [<cond>]  # ANDed <cond>s
if [<cond> -a <cond>]       # sole ANDed test
if [ -x "$1"-a ! -d "$1"]    # idiom: is executable
```

Test Command

test *evaluates an expression. The results, in the form of <u>exit statuses</u>, are ubiquitously used in flow-control statements, as a means of implementing a more general condition* (<cond>)*. Sadly, an exit status of 0 is a "success"; 1 or other is a "failure". [. . .] is a shorthand for* **test***, and the following options control the evaluation:*

-b	is block dev	-c	is char dev
-c	is dir	-e	exists
-f	is regular	-g	setfid set
-G	owned by grp	-k	sticky bit set
-L	is sym link	-n	str non-null
-O	owned by usr	-p	is pipe
-r	is readable	-s	is non-empty
-S	is a socket	-t *n*	*n* points to term
-u	setuid bit	-w	writeable
-x	executable		

Can also use comparison operators =, !=, <, >, <=, >=, ==, *the first four of which can be used with strings or numerics; or exclusively numeric comparison operators* **-lt**, **-le**, **-eq**, **-ge**, **-gt**, **-ne**, *as well as* +, −, ∗, /, %, <<, >>, $, |, ~, !, ^ . *Warning: use double quotes for* **[-n "$<var>"]** *tests, as empty strings otherwise still succeed!*

If / Else

```
if <cond>              # <cond> is often a  test
then                   # idiom: append to prev line
   <statemt>*
elif <cond>            # optional as usual
   <statemt>*
else                   # optional as well
   <statemt>*
fi                     # requires new line
```

For

```
for <i> in <list>    # <i> used only in loop
do                   # requires n.l. (or for ; do)
   <statemt>*        # can now use <i>
done                 # requires new line
```

Eg, iterate PATH *or files in* . *using:*

```
IFS=: ; for dir in $PATH; do ls -ld $dir; done
for f in $(ls -1); do
```

Newer bash versions afford a "numeric for," which approximates traditional for *loops:*

```
for (( <init> ; <end> ; <update> )); do . . .
for (( i=1     ; i<=12  ; i++          )); do . . .
```

Case

```
case <expr> in         # <expr> is a string
   <ptrn> )            # <ptrn> is another string
      <stmt>* ;;       # note double ; to end
   <ptrn> )            # as many cases as needed
      <stmt>* ;;       #
   <ptrn>|<ptrn> )     # log'ly ORed <ptrn>s
      <stmt>* ;;       #
   * )                 # optional "catch-all" case
      <stmt>* ;;       #
esac                   # on its own line
```

Select

A higher-level interface for implementing menus. Alternatively, can elicit input more manually using lower-level **read** *invocations. Note: idiomatically, user-prompts write to STDERR.*

```
select <sel> in <list>   # options from <list>
do                       # on own line
   <stmt>*               # can now use <sel>
done                     # on own line
```

While / Until

```
while <cond> ; do  # <cond> tests exit status
   <stmt>*         #
done               #

until <cmd>; do    # do while <cmd> fails
   <stmt>*         #
done               #

echo "$var" | while IFS= read -r ln ; do . . . ; done
```

Eg, expand on previous PATH *iteration:*

```
path=$PATH ; while [ $path ] ;
do ls -ld ${path%%:*}; path=${path#*:} ; done
```

Eg, process command-line args without getopts:

```
while [ -n "$(echo $1 | grep '-')"] ;
do . . . ; shift ; done
```

7 Command-line Args

Positional Parameters

"Positional parameters" ($1, $2, . . .) hold passed parameters; while # holds # of params and ∗, @ *hold all params (in a string, array, respectively). For manual parsing,* **shift** *pops the top of the positional parameter stack. Use* **getops** *for complex parsing (eg, multiple options, not all required).*

getopts

```
getopts <options_list> <option>
```

getopts *assigns each successive command-line argument to* <option>. <options_list> *describes available options: assign a letter for each, separated by colons, with an optional first colon to override default* **getopts** *error msg. If an option has an argument,* **getopts** *will store the current one in* OPTARG.

```
while getopts ":a:b:C"  opt ; do
   case $opt in
      a ) <stmt>
      \? ) echo "usage"; exit 1 ;;
esac ; done
```

eval

eval *affords dynamic creation of shell commands. Eg, programmatically create a pipeline by storing each step into a string, then* **eval***ing it:*

```
eval $convert $file $scale $border > $outfile
```

8 Input / Output Processing

Examples / Idioms

```
"$@"> logfile > 2>&1 &   # bg job; err, out to log
"$@"| tee logfile &       # ibid (synonym)
```

Synopsis

|	pipe	>	std out
<	std in	>>	append out
>|	force out	*n*>|	force to *n*
<>	in & out	*n*<>	in, out to *n*
<<	"here doc"	*n*>	out to FD *n*
<*n*	FD in *n*	*n*>>	FD append *n*
n>&	dup out	*n*<&	dup in, FD *n*
n>&*m*	*n* follows *m*	*n*<&*m*	*n* follows *m*
&>	out & err	<&-	close std in
>&-	close s-out	*n*>&-	close *n*-out