Standard

[March 24, 2018](https://azizsaya.wordpress.com/2018/03/24/korn-shell-cheat-sheet/) [Leave a comment](https://azizsaya.wordpress.com/2018/03/24/korn-shell-cheat-sheet/#respond) [Uncategorized](https://azizsaya.wordpress.com/category/uncategorized/)

###KornShell###

1. Derivative of Bourne and C Shells  
2. Developed by David Korn from Bell Labs during the early 1980s  
3. Popular releases:  
a. ksh88  
b. ksh93  
4. Supports scripting capabilities:  
a. loops – iteration  
b. conditions – branching – if then else – else if  
c. variables – active  
d. command substitution – 2 declarations – “, $()  
e. string manipulation  
f. regular expressions  
g. arrays – lists  
h. functions  
5. Will process virtually ALL Bourne scripts, unmodified  
6. Supports in-line editing of commands (like BASH)  
7. Provides a shared (by all instances) command history:  
a. ~/.kshrc\_history  
8. Compatible with BASH scripts  
9. Tab completion  
10. Arrays – 2 types  
a. Indexed – varname[0]  
b. Associative – varname[string]  
11. Functions – 2 declarations  
a. Korn – function name  
b. POSIX – function\_name()  
12. Supports the creation and reference of dynamic libraries  
13. Provides numerous built-ins (internal commands)  
a. test  
b. for, if, the, do, while, until, continue, else…  
14. Supports 8-bit range for reporting errors:  
a. 0-255  
Note: Default successful exit status: 0

###Shel Basics###  
Tasks:  
1. Create new users on both systems, who default to ksh  
a. Use ‘yast’ on SUSE Linux  
b. Use ‘useradd’ or GUI on RedHat Linux  
Note: ‘useradd -s /bin/ksh -d /home/linuxcbtkorn linuxcbtkorn’ auto-creates user’s $HOME directory  
c. Assign password to user on RedHat system: ‘passwd linuxcbtkorn’

2. SSH into both hosts as ‘linuxcbtkorn’  
3. Reveal the current shell:  
a. echo $SHELL  
b. echo $$ – returns the PID of the current shell – use ‘ps’ to determine

Note: KornShell reads the following files upon invocation:  
a. ~/.profile  
b. /etc/profile

4. Reveal $PWD in shell prompt and in general  
a. echo $PS1 – returns the current prompt configuration  
b. echo $PWD – stores the current working directory  
c. echo $OLDPWD – stores the previously visited directory

5. Reveal other useful internal variables  
a. echo $HISTFILE – returns the filename used to house the user’s history  
b. echo $HISTCMD – returns the number of items in the $HISTFILE  
c. echo $HISTSIZE – returns the number of commands to store in the $HISTFILE  
d. echo $SECONDS – returns number of seconds since shell invocation

6. Explore Redirection  
a. ‘<‘ – controls input to program – STDIN  
b. ‘>’ – controls output to a file – STDOUT – This will clobber the target file.  
c. ‘>>’ – controls output to a file using append mode – STDOUT – This will NOT clobber the target file.

8. Explore Pipes – Connects output stream of one program, to the input stream of another program  
a. cat testksh\_history.txt | grep ‘wc’

9. Explore ‘test’ command  
a. ‘test -e numbers.txt’ – returns ‘0’ or TRUE if file ‘numbers.txt’ exists  
b. ‘test -d Documents/’ – returns ‘0’ or TRUE if directory ‘Documents/’ exists  
Note: use ‘echo $?’ to return the exit status of the most recent command  
c. ‘man test’ – use to determine possible tests  
d. ‘test -b /dev/sda2’ – returns TRUE if device is type ‘block’

###Command Chaining, Processing & Substitution###

Features:  
1. Sequential execution (;) – executes all commands regardless of prior commands  
2. Logical AND (&&) – executes subsequent command if prior command exits 0 – builds contingencies between commands  
3. Logical OR (||) – executes subsequent command if prior command exits non-zero

Tasks:  
1. Command chain a few commands  
a. ls ; pwd  
b. clear ; ls -l ; pwd ; echo $?

2. clear && ls -l

3. clear || ls -l

Command Substitution:  
Features:  
1. Permits capture of external command for usage in a variable and/or for analysis

2 Methods:  
a. `pwd` – backticks  
b. $(pwd)

Tasks:  
1. var1=`pwd` ; echo $var1  
2. var1=$(pwd) ; echo $var1  
3. var1=$(pwd) && echo $var1 – this echoes $var1 only if $var1 has been set with ‘pwd’

###Variables###  
Features:  
1. Stores changing/dynamic information

Key Built-in Variables:  
1. Positional Parameters  
a. $0 – calling program  
b. $1 .. $n – variables other than the calling program  
c. $@, $\* – list every positional parameter as one string

2. $? – last returned exit status (error)  
a. exit status values range from 0-255 (8-bit)

3. $$ – returns the PID of the current shell

4. $! – returns the PID of the last backgrounded command

Variable Definition:

Tasks:  
1. typeset varname=value  
a. typeset var1=”Hello World”; OR typeset var1=’Hello World’;  
b. echo $var1  
Note: Variables do NOT persist across shells, unless, the variable is constructed at startup of the shell instance: ~/.profile and/or /etc/profile  
c. unset var1  
Note: Use the leading ‘$’ only when referencing variables; not when setting/unsetting them.  
Note: Variables are dynamically set. There is no need to unset, before updating or resetting a variable  
2. varname=value – ensure that there are no non-printing characters in ‘value’  
3. varname=”value” OR varname=’value’  
4. echo $varname – prints the contents of the variable  
5. print “$varname” – prints the contents of the variable  
6. var1=Hello; var2=World; var3=”$var1 $var2″ ; echo $?  
7. first=Dean && last=Davis && fullname=”$first $last” && echo $?

###Prompt Configuration###  
Features:  
1. Customizable using variables  
2. System-wide (/etc/profile) or per-user (~/.profile) configuration  
3. Multiple prompts:  
a. PS1 – primary prompt  
b. PS2 – secondary prompt – returned when a command is incomplete – defaults ‘>’  
c. PS3 – select menus  
d. PS4 – debugging/traces on commands

Tasks:  
1. export a simple PS1 prompt on the RedHat box  
a. export PS1=’${PWD}’  
b. export PS1=’${PWD} ‘  
c. export PS1=’${PWD}> ‘

2. Mimic the ksh prompt on SUSE Linux  
a. export PS1=’${USER}@${HOST}:${PWD}> ‘  
b. ~/.profile – HOST=`hostname -s`;  
c. export PS1=’${USER}@${HOST}:${PWD}:${SECONDS}> ‘  
Note: ‘$’ is the default prompt for a non-privileged (non-root) user

3. Enable ‘vi’ style command history navigation in the shell  
a. set -o vi – enables command history like ‘vi’  
b. enable in: ~/.profile to persist across logins

4. Set ‘PS2’ to return a useful prompt other than ‘>’  
a. export PS2=’incomplete command> ‘

###Functions###  
Features:  
1. Used to contain code  
2. Can be referenced when necessary  
3. Accept parameters for processing  
4. Can return exit status and useful information  
5. Allows you to run a ksh script within a ksh script

Function Definition Methods – 2 Types  
1. function function\_name {  
command1..commandn  
}

2. POSIX compliant  
function\_name() {  
command1..commandn

}

Note: Call the function by simply referencing the name of the function in your script (after function definition)

Tasks:  
1. Create a ksh-default function type to return the current Unix Epoch Date

#!/usr/bin/ksh  
function epochdate {  
echo `date +%s`;

}

chmod +x function\_test.ksh – to permit execution

Note: Inside a function, the $0 var is the name of the function  
Note: Outside of a function, the $0 var is the name of the calling process

###Looping###  
Features:  
1. Facilitates iteration

###Supported Loops###  
1. For  
2. While  
3. Until

Keywords that can be applied to all loops:  
1. continue  
2. break

###For Loop###  
Features:  
1. Iterates through/over a list of items  
Note: Typically, for loops iterate over lists that are built using command substitution

Tasks:  
1. Create a for loop, which iterates over the output from ‘ls -A’

for i in $(ls -A); do  
echo $i;  
done

2. Create loop to iterate through ‘numbers.txt’ file

3. Create loop to rename files with new suffix and prefix

###While Loop###  
Features:  
1. Iterates based on conditional test, which is usually true  
2. Permits precise control over the number of iterations

while [[ $num -lt 10 ]]; do  
print “$num”;  
(( num += 1 ));  
done

###Until Loop###  
Features:  
1. Similar to a While loop, however, performs at least 1 iteration

Task:  
1. Create an Until loop which tests user input

###Conditions & Comparisons###  
Features:  
1. Facilitates string and numeric conditions  
2. Branching

Comparison of numbers:  
1. -eq – equal to  
2. -ne – not equal to  
3. -gt – greater than  
4. -lt – less than  
5. -le – less than or equal to  
6. -ge – greater than or equal to

Comparison of strings:  
1. = – equal to  
2. != – not equal to

Task:  
1. Create conditions based on numerical comparisons  
a. if the current value of $i -qt 32 print something  
b. if the current value of $i -ge && -le 64 do something

2. Create condtion based on string comparison  
a. if the current value of $i = forloop\_test.ksh then do something

3. Include ‘else’ to branch and handle non-matches

4. Include ‘elif’ to perform an additonal test

###Error Handling###  
Features:  
1. Facilitates the handling of errors

Task:  
1. Write a script, which opens a filename supplied as a positional parameter  
a. Trap missing filename, report error, then exit gracefully  
b. Test that the file exists on the file system  
c. Ensure that the script will only process ‘ASCII text’  
d. Convert processing of file to a function and call from within main script

###Backup Script###  
Features:  
1. Ability to backup key files & directories

Task:  
1. Write a script, which backs up various files and directories

Note: The shebang header is unnecessary if you call the script from ‘ksh’  
a. Ensure that the script is called with 1 positional parameter referencing the input file, which includes the list of files and/or directories to archive

b. loop through the contents of the input file  
c. while looping, backup/archive each line item to destination file  
d. ensure that the destination backup file name is unique  
3. Facilitate appending of multiple items (files and/or directories)  
Note: Use ‘tar’ without compression to append sequentially. At the end, compress.

###Korn Shell – Arrays###  
Features:  
1. 2 Supported types of Arrays  
a. Indexed  
b. Associative

Indexed Arrays:  
1. They use integers as indices, typically indexed at ‘0’  
2. Reference list element using the array variable name and the index number

Define Indexed Arrays:  
1. set -A daysofweek values…  
Note: ‘set -A’ first nulls previous entries in the array and then populates  
Note: ‘set +A’ appends element(s) to an existing array

a. set -A daysofweek Monday Tuesday Wednesday Thursday Friday  
b. echo “${daysofweek[0]}”  
c. set -A daysofweek Tuesday  
d. echo “${daysofweek[\*]}” – dumps all array elements to STDOUT  
e. echo “${daysofweek[@]}” – dumps all array elements to STDOUT  
d. echo “${#daysofweek[\*]}” – dumps all array elements to STDOUT

2. daysofweek2[0]=”Sunday”;  
a. daysofweek2[1]=”Monday”;

3. set -A lsoutput `ls -A`; – creates indexed array based on command substitution

Note: use ‘unset lsoutput’ to nullify the array

###Loop through Indexed Array###  
set -A lsoutput $(ls -A);  
for j in ${lsoutput[@]; do  
print “LSOUTPUT: $j”;  
done

j=0;  
while [[ $j -lt 10 ]]; do  
print “WHILE LOOP: ${lsoutput[i]}”;  
(( j += 1 ));  
done

###Associative Arrays###  
Features:  
1. Uses strings as indices  
2. maps one-to-one or one-to-many relationships

Task:  
1. Define an associative array to map car manufacturers to models

carman[Acura]=”TSX”;  
print “${carman[Acura]}”;

###Regular Expressions###  
Features:  
1. Facilitates the searching of text using arbitrary characters  
2. Primarily implemented via:  
a. shell commands such as: ls  
b. grep  
c. sed  
d. awk  
e. conditional testing/string comparison within KornShell scripts

RegEx Basics:  
Reserved characters include:  
1. ?(pattern) – matches 0 or 1 time  
2. \*(pattern) – matches 0 or more times  
3. +(pattern) – matches 1 or more times  
4. @(pattern) – matches one time  
5. !(pattern) – matches ALL but the pattern

Tasks:  
1. Use ‘ls’ with simple regular expressions  
a. ls -l test\* – returns all files with ‘test’ as a prefix with 0 or more additional chars  
b. ls -l [uw]\* – character-class search for files beginning with ‘u’ OR ‘w’  
c. ls -l [0-9]\* – character-class search for files beginning with numeric values

Grep:  
Features:  
1. Line-based RegExes

Tasks:  
1. Use grep to search for various text  
a. grep ‘500’ numbers.txt  
b. grep ‘^5’ numbers.txt – returns ALL lines starting with the number ‘5’  
c. grep ‘5$’ numbers.txt – returns ALL lines ending with the number ‘5’  
d. grep ‘[a-z][0-9]’ numbers.txt – returns ALL lines with lower-case alpha, terminating with one or more digits

SED:  
Features:  
1. Stream editing (textual changes) on-the-fly to STDOUT

Tasks:  
1. Use sed to modify the output of ‘ls’ using RegExes  
a. ls -A | sed ‘s/korn/corn/’  
b. sed -ne ‘/[a-z][0-9]/p’ – searches for matching alphanumeric lines and prints them  
c. sed -ne ‘/[a-z][0-9]/Ip’ – searches for matching alphanumeric lines (case-insensitive) and prints them

Awk:  
Features:  
1. Field extraction

Tasks:  
1. Print various lines from numbers.txt  
a. awk ‘{ print $1 }’ numbers.txt – prints the ENTIRE line of EACH line  
b. awk ‘/[a-z][0-9]/ { print $0 }’ numbers.txt – searches using RegExes and prints the entire line of matching rows

###Korn on Solaris###  
Features:  
1. Similar to Korn on Linux

Tasks:  
1. Connect to Solaris box and create new user, which defaults to ‘ksh’  
a. ssh to Solaris box  
b. useradd -d /export/home/ucbtkorn -s /usr/bin/ksh ucbtkorn  
c. mkdir /export/home/ucbtkorn && chown ucbtkorn /export/home/ucbtkorn  
d. passwd ucbtkorn

2. Customize the ‘ksh’ environment for the new user  
a. export PS1=’${USER}@${HOST}:${PWD}> ‘  
b. export HOST=’linuxcbtsol1’;

3. Explore ‘ksh’ Sun package info  
a. pkgchk -lp /usr/bin/ksh  
b. pkginfo -l SUNWcsu  
c. pkgchk -v SUNWcsu

4. Explore aliases – facilitates referencing commands using various names/options  
a. alias ls=’ls -l’

Note: Solaris ‘ksh’ defaults the history items to: ~/.sh\_history  
Note: ~/.sh\_history is shared by: sh, ksh and bash

###Case###  
Feature:  
1. Simplifies if then else blocks, by requiring less syntax

Syntax:  
case expression in  
pattern1 )  
command(s);;  
pattern2 )  
command(s);;  
pattern3 )  
command(s);;  
\*)  
command(s);;  
esac

Task:  
1. Use case to test positional parameter $1  
2. Create an array based on a list of files and loop through using case to test

###Job Control###  
Features:  
1. Ability to background and foreground jobs

Tasks:  
1. Launch job and background it  
a. mutt – CTRL-Z (backgrounds)  
Note: CTRL-Z places the application in a frozen state

b. jobs -l – enumerates the currently running jobs

Note: By default, jobs/programs are tied to the parent TTY

2. Foreground the jobs  
a. fg %N (job number)  
fg %1 OR fg %2  
fg %+ – resumes the most recently backgrounded job to the fore

Note: Sometimes ‘kill’ will terminate a backgrounded job, whilst otherwise you need to foreground it first

3. Background a non-interactive application  
a. seq 1000000000 > 1billion.txt & – backgrounds the job

4. Disown a job, which basically separates it(the job) from a TTY  
a. disown %N (job\_id)

###TypeSetting Variables###  
Features:  
1. Facilitates setting of types of values that variables can store – i.e. string, integer, array, etc.  
2. Formats variables: case (upper or lower), left-justify, right-justify, truncation, etc.  
3. Localizes function variables – scoping  
4. Enumerates currently defined variables – ‘typeset’  
5. Ability to set read-only variables – i.e. ‘UID’

Tasks:  
1. Dump all variables  
a. typeset – dumps ALL variables, including type  
b. typeset -x – dumps ALL exported shell variables  
c. typeset -i – dumps ALL integer variables

2. Write a simple script using typeset, showing variable scoping

Note: If ‘typeset’ is NOT used within a function to localize variables, the variable(s) that are set in the function, when called, will overrite its global counterpart

3. Convert vars to upper and lower case  
a. typeset -u – UPPERCASE  
b. typeset -l – lowercase

4. Truncate text  
a. typeset -L4 – extracts the first 4 chars from the left of the variable  
b. typeset -R4 – extracts the first 4 chars from the right of the variable

Note: typesetting of variables is dynamic. The most recent instance of a typset variable is used

Note: You may combine options such as: lower|upper with left|right justification

5. Define an integer variable  
a. typeset -i[n](optional base of the number)  
b. typeset -i loopcount=$2;

6. Define read-only variable  
a. typeset -r varreadonly1=”Test Read Only”;

###Input Validation###  
Features:  
1. The ability to confirm that input meets expectations

Tasks:  
1. Check that only 1 positional parameter exists  
a. Use test to ensure at least 1 positional parameter  
b. Use compound test to ensure ONLY 1 positional parameter

2. Ensure that positional parameter 1 is a value between: 1 & 50 to drive case statement