BASH Shell Scripting

Alexander B. Pacheco

User Services Consultant
LSU HPC & LONI
sys-help@loni.org

HPC Training Fall 2012
Louisiana State University
Baton Rouge
September 26, 2012

What is a SHELL

- The command line interface is the primary interface to Linux/Unix operating systems.
- Shells are how command-line interfaces are implemented in Linux/Unix.
- Each shell has varying capabilities and features and the user should choose the shell that best suits their needs.
- The shell is simply an application running on top of the kernel and provides a powerful interface to the system.

`sh` : Bourne Shell

- Developed by Stephen Bourne at AT&T Bell Labs

`csh` : C Shell

- Developed by Bill Joy at University of California, Berkeley

`ksh` : Korn Shell

- Developed by David Korn at AT&T Bell Labs
- backward-compatible with the Bourne shell and includes many features of the C shell

`bash` : Bourne Again Shell

- Developed by Brian Fox for the GNU Project as a free software replacement for the Bourne shell (sh).
- Default Shell on Linux and Mac OSX
- The name is also descriptive of what it did, bashing together the features of sh, csh and ksh

`tcsh` : TENEX C Shell

- Developed by Ken Greer at Carnegie Mellon University
- It is essentially the C shell with programmable command line completion, command-line editing, and a few other features.

Software	sh	csh	tcsh	ksh	bash
Programming Language	✓	✓	✓	✓	✓
Shell Variables	✓	✓	✓	✓	✓
Command alias	✗	✓	✓	✓	✓
Command history	✗	✓	✓	✓	✓
Filename completion	✗	★	✓	★	✓
Command line editing	✗	✗	✓	★	✓
Job control	✗	✓	✓	✓	✓

✓ : Yes

✗ : No

★ : Yes, not set by default

Ref : `http://www.cis.rit.edu/class/simg211/unixintro/Shell.html`

`cat` : Show contents of a file

- `cat filename`

`cd` : Change Directory

- `cd Tutorials`

`cp` : Copy a file

- `cp file1 file2`

`mv` : Move or rename a file

- `mv file1 file2`

`ls` : List files in a directory

- `ls Tutorials`
- Options to `ls`
  - -l show long listing format
  - -a show hidden files
  - -r reverse order while sorting
  - -t show modification times

`mkdir` : Create a directory

- `mkdir dir1`

`rm` : Remove a file

- `rm file1 file2`
- Options to `rm`
  - -i interactive
  - -r remove files recursively used to delete directories and its contents
  - -f force, ignore nonexistent files

`rmdir` : Remove a directory

- `rmdir dir1`

`file` : Determine file type

`more` : Display a file one page at a time

`less` : Same as `more` but allow scrolling

`man` : Access Manual for given application

`vi` : Edit a file using VI/VIM

`emacs` : Edit a file using Emacs

`wc` : Count words, lines and characters in a file

`awk` : File processing and report generating

- `awk '{print $1}' file1`

`grep` : Find lines in a file

- `grep alias .bashrc`

`sed` : Stream Editor

- `sed 's/home/HOME/g' .bashrc`

`find` : Find a file

`ln` : Link a file to another file

- `ln -s file1 file2`

`top` : Produces an ordered list of running processes

`ps` : Displays statistics on running processes

`scp` : secure copy a file/directory between two machines

- `scp username@host1:/path/to/file1 username@host2:/path/to/file2`

`sftp` : connect to another machine using secure ftp

`export` : export variables to your `PATH` (`sh,ksh & bash` only)

- `export PATH=/home/apacheco/bin:${PATH}`

`setenv` : equivalent of `export` for `csh & tcsh`

- `setenv LD_LIBRARY_PATH /home/apacheco/lib:${LD_LIBRARY_PATH}`

`alias` : enables replacement of a word by another string

- `sh/ksh/bash: alias ll="ls -l"`
- `csh/tcsh: alias rm "rm -i"`

`set` : manipulate environment variables

- `set -o emacs`

`echo` : print to screen or standard output

- `echo $LD_INCLUDE_PATH`

`date` : display or set date and time

`&` : run a job in background

`CNTRL-Z` : suspend a running job

`CNTRL-C` : Kill a running job

`jobs` : Show list of background jobs

`fg` : run a suspended job in foreground

`bg` : run a suspended job in background

`wait` : wait until all backgrounded jobs have completed

`kill` : kill a running job, need to provide process id

To learn more about these commands, type `man` *`command`* on the command prompt

- The two most commonly used editors on Linux/Unix systems are:
  1. `vi`
  2. `emacs`
- `vi` is installed by default on Linux/Unix systems and has only a command line interface (CLI).
- `emacs` has both a CLI and a graphical user interface (GUI).
- If `emacs` GUI is installed then use `emacs -nw` to open file in console.
- Other editors that you may come across on *nix systems
  1. `kate`: default editor for KDE.
  2. `gedit`: default text editor for GNOME desktop environment.
  3. `gvim`: GUI version of `vim`
  4. `pico`: console based plain text editor
  5. `nano`: GNU.org clone of `pico`
  6. `kwrite`: editor by KDE.

Cursor Movement	vi	emacs
move left	`h`	`C-b`
move down	`j`	`C-n`
move up	`k`	`C-p`
move right	`l`	`C-f`
jump to beginning of line	`^`	`C-a`
jump to end of line	`$`	`C-e`
goto line `n`	`nG`	`M-x goto-line [RET] n`
goto top of file	`1G`	`M-<`
goto end of file	`G`	`M->`
move one page up	`C-u`	`M-v`
move one page down	`C-d`	`C-v`

C : Control Key

M : Meta or ESCAPE (ESC) Key

[RET] : Enter Key

Insert/Appending Text

	vi
insert at cursor	`i`
insert at beginning of line	`I`
append after cursor	`a`
append at end of line	`A`
newline after cursor in insert mode	`o`
newline before cursor in insert mode	`O`
append at end of line	`ea`
exit insert mode	`ESC`

- `emacs` has only one mode unlike `vi` which has insert and command mode

File Editing	vi	emacs
save file	`:w`	`C-x C-s`
save file and exit	`:wq, ZZ`	
quit	`:q`	`C-x C-c`
quit without saving	`:q!`	
delete a line	`dd`	`C-a C-k`
delete *n* lines	*n*`dd`	`C-a M-`*n*` C-k`
paste deleted line after cursor	`p`	`C-y`
paste before cursor	`P`	
undo edit	`u`	`C-_`
delete from cursor to end of line	`D`	`C-k`
search forward for *patt*	`\`*patt*	`C-s` *patt*
search backward for *patt*	`?`*patt*	`C-r` *patt*
search again forward (backward)	`n`	`C-s(r)`

File Editing (contd)	vi	emacs
replace a character	`r`	
join next line to current	`J`	
change a line	`cc`	
change a word	`cw`	
change to end of line	`c$`	
delete a character	`x`	`C-d`
delete a word	`dw`	`M-d`
edit/open file *file*	`:e file`	`C-x C-f file`
insert file *file*	`:r file`	`C-x i file`
split window horizontally	`:split or C-ws`	`C-x 2`
split window vertically	`:vsplit or C-wv`	`C-x 3`
switch windows	`C-ww`	`C-x o`

- To change a line or word in `emacs`, use `C-`*`spacebar`* and navigate to end of word or line to select text and then delete using `C-w`

- Do a google search for more detailed cheatsheets

```
vi https://www.google.com/search?q=vi+cheatsheet
emacs https://www.google.com/search?q=emacs+cheatsheet
```

- When you login to a *NIX computer, shell scripts are automatically loaded depending on your default `shell`
- `sh,ksh`
  1. `/etc/profile`
  2. `$HOME/.profile`
- `bash`
  1. `/etc/profile, login terminal only`
  2. `/etc/bashrc or /etc/bash/bashrc`
  3. `$HOME/.bash_profile, login terminal only`
  4. `$HOME/.bashrc`
- `csh,tcsh`
  1. `/etc/csh.cshrc`
  2. `$HOME/.tcshrc`
  3. `$HOME/.cshrc if .tcshrc is not present`
- The `.bashrc, .tcshrc, .cshrc, .bash_profile` are script files where users can define their own aliases, environment variables, modify paths etc.
- e.g. the `alias rm="rm -i"` command will modify all `rm` commands that you type as `rm -i`

.bashrc

```
# .bashrc

# Source global definitions
if [ -f /etc/bashrc ]; then
        . /etc/bashrc
fi

# User specific aliases and functions
alias c="clear"
alias rm="/bin/rm -i"
alias psu="ps -u apacheco"
alias em="emacs -nw"
alias ll="ls -lF"
alias la="ls -al"
export PATH=/home/apacheco/bin:${PATH}
export g09root=/home/apacheco/Software/Gaussian09
export GAUSS_SCRDIR=/home/apacheco/Software/scratch
source $g09root/g09/bsd/g09.profile

export TEXINPUTS=.:/usr/share/texmf//:/home/apacheco/LaTeX//:${TEXINPUTS}
export BIBINPUTS=.:/home/apacheco/TeX//:${BIBINPUTS}
```

.tcshrc

```
# .tcshrc

# User specific aliases and functions
alias c clear
alias rm "/bin/rm -i"
alias psu "ps -u apacheco"
alias em "emacs -nw"
alias ll "ls -lF"
alias la "ls -al"
setenv PATH "/home/apacheco/bin:${PATH}"
setenv g09root "/home/apacheco/Software/Gaussian09"
setenv GAUSS_SCRDIR "/home/apacheco/Software/scratch"
source $g09root/g09/bsd/g09.login

setenv TEXINPUTS ".:/usr/share/texmf//:/home/apacheco/LaTeX//:${TEXINPUTS}"
setenv BIBINPUTS ".:/home/apacheco/TeX//:${BIBINPUTS}"
```

What is a Scripting Language?

- A **scripting language** or **script language** is a *programming language* that supports the writing of **scripts**.
- **Scripting Languages** provide a higher level of abstraction than standard programming languages.
- Compared to programming languages, scripting languages do not distinguish between data types: integers, real values, strings, etc.
- Scripting Languages tend to be good for automating the execution of other programs.
  - analyzing data
  - running daily backups
- They are also good for writing a program that is going to be used only once and then discarded.

What is a script?

- A **script** is a program written for a software environment that automate the execution of tasks which could alternatively be executed one-by-one by a human operator.
- The majority of script programs are “quick and dirty”, where the main goal is to get the program written quickly.

Three things to do to write and execute a script

1. Write a script
   - A shell script is a file that contains ASCII text.
   - Create a file, `hello.sh` with the following lines

```
#!/bin/bash
# My First Script
echo "Hello World!"
```

2. Set permissions

```
apacheco@apacheco:~/Tutorials/BASH/scripts> chmod 755 hello.sh
```

   - Why did we do this? Wait a couple of slides.

3. Execute the script

```
apacheco@apacheco:~/Tutorials/BASH/scripts> ./hello.sh
Hello World!
```

- My First Script

```
#!/bin/bash
# My First Script
echo "Hello World!"
```

- The first line is called the "SheBang" line. It tells the OS which interpreter to use. In the current example, bash
- Other options are:
  - `sh  :  #!/bin/sh`
  - `ksh :  #!/bin/ksh`
  - `csh :  #!/bin/csh`
  - `tcsh:  #!/bin/tcsh`
- The second line is a comment. All comments begin with "#".
- The third line tells the OS to print "Hello World!" to the screen.

#:	starts a comment.
$:	indicates the name of a variable.
\:	escape character to display next character literally.
{ }:	used to enclose name of variable.
;	Command separator [semicolon]. Permits putting two or more commands on the same line.
;;	Terminator in a case option [double semicolon].
.	"dot" command [period]. Equivalent to source. This is a bash builtin.
$?	exit status variable.
$$	process ID variable.
[ ]	test expression
[[ ]]	test expression, more flexible than [ ]
$[ ], (( ))	integer expansion.
||, &&, !	Logical OR, AND and NOT

- In *NIX OS's, you have three types of file permissions
  1. read (r)
  2. write (w)
  3. execute (x)
- for three types of users
  1. user
  2. group
  3. world i.e. everyone else who has access to the system
- Read carries a weight of 4
- Write carries a weight of 2
- Execute carries a weight of 1
- `chmod` is a *NIX command to change permissions on a file
- In the above example `chmod 755 hello.sh` implies
  - the user (you) have read, write and execute permission
  - members of your group have read and execute permission
  - everyone else aka world has read and write permission

```
apacheco@apacheco:~/Tutorials/BASH/scripts> ls -l hello.sh
-rwxr-xr-x  1 apacheco  staff  52 Sep 17 10:52 hello.sh
```

- Instead of using numerical permissions you can also use symobolic mode

u/g/o or a — user/group/world or all i.e. ugo

+/- — Add/remove permission

r/w/x — read/write/execute

- Give everyone execute permission:

```
chmod a+x hello.sh
```

```
chmod ugo+x hello.sh
```

- Remove group and world read & write permission:

```
chmod go-rw hello.sh
```

- The basis I/O statements are `echo` for displaying output to screen and `read` for reading input from screen/keyboard/prompt

```
apacheco@apacheco:~/Tutorials/BASH/scripts> cat helloname.sh
#!/bin/bash

# My Second Script

echo Please Enter your name:
read name
echo Hello $name
apacheco@apacheco:~/Tutorials/BASH/scripts> chmod 755 helloname.sh
apacheco@apacheco:~/Tutorials/BASH/scripts> ./helloname.sh
Please Enter your name:
Alex Pacheco
Hello Alex Pacheco
```

- The `read` statement takes all characters typed until the `enter` key is pressed and stores them into a variable.
- In the above example, the name that you enter in stored in the variable `name`.
- The `echo` statement can print multiple arguments. By default, `echo` eliminates redundant whitespace (multiple spaces and tabs) and replaces it with a single whitespace between arguments.

```
apacheco@apacheco:~/Tutorials/BASH/scripts> ./helloname.sh
Please Enter your name:
Alex       Pacheco
Hello Alex Pacheco
```

- To include redundant whitespace, enclose the arguments within double quotes

```
apacheco@apacheco:~/Tutorials/BASH/scripts> cat helloname.sh
#!/bin/bash

# My Second Script

echo Please Enter your name:
read name
echo "Hello $name"
apacheco@apacheco:~/Tutorials/BASH/scripts> ./helloname.sh
Please Enter your name:
Alex     Pacheco
Hello Alex     Pacheco
```

- Double Quotation " "
  - Enclosed string is expanded ("$", "/" and "`")
  - Example: `echo "$myvar"` prints the value of `myvar`
- Single Quotation ' '
  - Enclosed string is read literally
  - Example: `echo '$myvar'` prints `$myvar`
- Back Quotation ` `
  - Enclosed string is executed as a command
  - Example: `echo `pwd`` prints the output of the `pwd` command i.e. print working directory

```
apacheco@apacheco:~/Tutorials/BASH/scripts> cat quotes.sh
#!/bin/bash

HI=Hello

echo HI             # displays HI
echo $HI            # displays Hello
echo \$HI           # displays $HI
echo "$HI"          # displays Hello
echo '$HI'          # displays $HI
echo "$HIAlex"      # displays nothing
echo "${HI}Alex"    # displays HelloAlex
echo `pwd`          # displays working directory
apacheco@apacheco:~/Tutorials/BASH/scripts> ./quotes.sh
HI
Hello
$HI
Hello
$HI

HelloAlex
/home/apacheco/Tutorials/BASH/scripts
apacheco@apacheco:~/Tutorials/BASH/scripts>
```

- There are three file descriptors for I/O streams
  1. STDIN: Standard Input
  2. STDOUT: Standard Output
  3. STDERR: Standard Error
- 1 represents STDOUT and 2 represents STDERR
- I/O redirection allows users to connect applications
  - `<` : connects a file to STDIN of an application
  - `>` : connects STDOUT of an application to a file
  - `>>` : connects STDOUT of an application by appending to a file
  - `|` : connects the STDOUT of an application to STDIN of another application.
- Examples:
  1. write STDOUT to file: `ls -l > ls-l.out`
  2. write STDERR to file: `ls -l 2> ls-l.err`
  3. write STDOUT to STDERR: `ls -l 1>&2`
  4. write STDERR to STDOUT: `ls -l 2>&1`
  5. send STDOUT as STDIN: `ls -l | wc -l`

- Similar to any programming language such C, C++, Fortran, You can use `variables` in shell scripting languages.
- The only difference is that you do not have to declare the type of variables.
- A variable in **bash** (or any scripting language such as **sh,ksh,csh** or **tcsh**) can contain a number, character or a string of characters.
- You do not need to declare a variable, just assigning a value to its reference will create it.

```
apacheco@apacheco:~/Tutorials/BASH/scripts> cat hellovariable.sh
#!/bin/bash

# Hello World script using a variable
STR="Hello World!"
echo $STR
apacheco@apacheco:~/Tutorials/BASH/scripts> ./hellovariable.sh
Hello World!
```

- By Convention, variables are often named using all uppercase letters
  - `PATH, LD_LIBRARY_PATH, LD_INCLUDE_PATH, TEXINPUTS, etc`
- Rules for Variable Names
  1. Variable names must start with a letter or underscore
  2. Number can be used anywhere else
  3. DO NOT USE special characters such as `@, #, %, $`
  4. Case sensitive
  5. Examples
     - Allowed: VARIABLE, VAR1234able, var_name, _VAR
     - Not Allowed: 1VARIABLE, %NAME, $myvar, VAR@NAME
- Assigning value to a variable
- `sh, ksh, bash`
  1. shell variable: `variablename=value`
  2. environmental variable: `export variablename=value`
  3. NOTE: THERE IS NO SPACE ON EITHER SIDE OF =

- `csh, tcsh`
  1. shell variable: `set variablename = value`
  2. environmental variable: `setenv variablename value`
  3. NOTE: space on either side of = is allowed for the `set` command
  4. NOTE: There is no = in the `setenv` command
- All variables are stored in memory as strings and converted to numbers when needed
- You can carry out numeric operations on variables
- Arithmetic operations in `bash` can be done within the `$((···))` or `$[···]` commands
  - ★ Add two numbers: `$((1+2))`
  - ★ Multiply two numbers: `$[$a*$b]`
  - ★ You can also use the `let` command: `let c=$a-$b`
- In `tcsh`,
  - ★ Add two numbers: `@ x = 1 + 2`
  - ★ Divide two numbers: `@ x = $a / $b`

Exercise

Write a script to add/subtract/multiply/divide two numbers.

```
apacheco@apacheco:~/Tutorials/BASH/scripts> cat dosum.sh
#!/bin/bash

FIVE=5
SEVEN=7
echo "5 + 7 = " $FIVE + $SEVEN
echo "5 + 7 = " $(($FIVE + $SEVEN))
let SUM=$FIVE+$SEVEN
echo "sum of 5 & 7 is " $SUM
exit
apacheco@apacheco:~/Tutorials/BASH/scripts> ./dosum.sh
5 + 7 = 5 + 7
5 + 7 =  12
sum of 5 & 7 is  12
apacheco@apacheco:~/Tutorials/BASH/scripts> cat dosum.csh
#!/bin/tcsh

set FIVE=5
set SEVEN=7
echo "5 + 7 = " $FIVE + $SEVEN
@ SUM = $FIVE + $SEVEN
echo "sum of 5 & 7 is " $SUM
exit
apacheco@apacheco:~/Tutorials/BASH/scripts> ./dosum.csh
5 + 7 =  5 + 7
sum of 5 & 7 is  12
```

```
apacheco@apacheco:~/Tutorials/BASH/scripts> cat backups.sh
#!/bin/bash

BACKUPDIR=$(pwd)
OF=$BACKUPDIR/$(date +%Y-%m-%d).tgz
tar -czf ${OF} ./*sh
apacheco@apacheco:~/Tutorials/BASH/scripts> ./backups.sh
apacheco@apacheco:~/Tutorials/BASH/scripts> ls *gz
2012-09-18.tgz
```

- `bash` supports one-dimensional arrays.
- Array elements may be initialized with the `variable[xx]` notation

  ```
  variable[xx]=1
  ```
- Initialize an array during declaration

  ```
  name=(firstname 'last name')
  ```
- reference an element `i` of an array `name`

  ```
  ${name[i]}
  ```
- print the whole array

  ```
  ${name[@]}
  ```
- print length of array

  ```
  ${#name[@]}
  ```
- print length of element `i` of array `name`

  ```
  ${#name[i]}
  ```

  Note: `${#name}` prints the length of the first element of the array

- Add an element to an existing array

```
name=("title" "${name[@]}")
```

- copy an array `name` to an array `user`

```
user=("${name[@]}")
```

- concatenate two arrays

```
nameuser=("${name[@]}" "${user[@]}")
```

- delete an entire array

```
unset name
```

- remove an element `i` from an array

```
unset name[i]
```

- Similar to C/C++, the first array index is zero (0)

Exercise

1. Write a script to read your first and last name to an array.
2. Add your salutation and suffix to the array.
3. Drop either the salutation or suffix.
4. Print the array after each of the three steps above.

```
apacheco@apacheco:~/Tutorials/BASH/scripts> cat name.sh
#!/bin/bash

echo "Print your first and last name"
read firstname lastname

name=($firstname $lastname)

echo "Hello " ${name[@]}

echo "Enter your salutation"
read title

echo "Enter your suffix"
read suffix

name=($title "${name[@]}" $suffix)
echo "Hello " ${name[@]}

unset name[2]
echo "Hello " ${name[@]}

apacheco@apacheco:~/Tutorials/BASH/scripts> ./name.sh
Print your first and last name
Alex Pacheco
Hello  Alex Pacheco
Enter your salutation
Dr.
Enter your suffix
(the one and only)
Hello  Dr. Alex Pacheco (the one and only)
Hello  Dr. Alex (the one and only)
```

- Similar to programming languages, `bash` (and other shell scripting languages) can also take command line arguments
  - `./scriptname arg1 arg2 arg3 arg4 ...`
  - `$0,$1,$2,$3, etc`: positional parameters corresponding to `./scriptname,arg1,arg2,arg3,arg4,...` respectively
  - `$#`: number of command line arguments
  - `$*`: all of the positional parameters, seen as a single word
  - `$@`: same as `$*` but each parameter is a quoted string.
  - `shift N`: shift positional parameters from `N+1` to `$#` are renamed to variable names from `$1` to `$# - N + 1`
- In `csh,tcsh`
  - an array `argv` contains the list of arguments with `argv[0]` set to name of script.
  - `#argv` is the number of arguments i.e. length of `argv` array.

```
apacheco@apacheco:~/Tutorials/BASH/scripts> cat shift.sh
#!/bin/bash

USAGE="USAGE: $0 arg1 arg2 arg3 arg4"

if [[ "$#" -ne 4 ]]; then
   echo $USAGE
   exit
fi

echo "Number of Arguments: " $#
echo "List of Arguments: " $@
echo "Name of script that you are running: " $0
echo "Command You Entered:" $0 $*

while [ "$#"  -gt 0 ]; do
  echo "Argument List is: " $@
  echo "Number of Arguments: " $#
  shift
done
apacheco@apacheco:~/Tutorials/BASH/scripts> ./shift.sh arg1 arg2 arg3 arg4
Number of Arguments:  4
List of Arguments:  arg1 arg2 arg3 arg4
Name of script that you are running:  ./shift.sh
Command You Entered: ./shift.sh arg1 arg2 arg3 arg4
Argument List is:  arg1 arg2 arg3 arg4
Number of Arguments:  4
Argument List is:  arg2 arg3 arg4
Number of Arguments:  3
Argument List is:  arg3 arg4
Number of Arguments:  2
Argument List is:  arg4
Number of Arguments:  1
```

- Shell Scripting Languages execute commands in sequence similar to programming languages such as C, Fortran, etc.
- Control constructs can change the sequential order of commands.
- Control constructs available in `bash` and `tcsh` are
  1. Conditionals: `if`
  2. Loops: `for, while, until`
  3. Switches: `case`

- An `if/then` construct tests whether the exit status of a list of commands is 0, and if so, executes one or more commands.

bash: if construct

```
if [ condition1 ]; then
    some commands
elif [ condition2 ]; then
    some commands
else
    some commands
fi
```

tcsh: if construct

```
if ( condition1 ) then
    some commands
else if ( condition2 ) then
    some commands
else
    some commands
endif
```

- Note the space between *condition* and "[" "]"
- `bash` is very strict about spaces.
- `tcsh` commands are not so strict about spaces.
- `tcsh` uses the `if-then-else if-else-endif` similar to Fortran.

File Test Operators

-e	: file exists	`if [ -e .bashrc ]`
-f	: file is a regular file	`if [ -f .bashrc ]`
-d	: file is a directory	`if [ -d /home ]`
-s	: file is not zero size	`if [ -s .bashrc ]`

Logical Operators

!	: NOT	`if [ !-e .bashrc ]`
&&	: AND	`if [ -f .bashrc ] && [ -s .bashrc]`
||	: OR	`if [ -f .bashrc ] || [ -f .bash_profile]`

Integer Comparison

-eq : equal to	`[ 1 -eq 2 ]`
-ne : not equal to	`[ "$a" -ne "$b" ]`
-gt : greater than	`[ "$a" -gt "$b" ]`
-ge : greater than or equal to	`[ 1 -ge "$b" ]`
-lt : less than	`[ "$a" -lt 2 ]`
-le : less than or equal to	`[ "$a" -le "$b" ]`

String Comparison

== : equal to	`[ "$a" == "$b" ]`
!= : not equal to	`[ "$a" != "$b" ]`
-z : string is null	`[ -z "$a" ]`
-n : string in not null	`[ -n "$b" ]`

```
apacheco@apacheco:~/Tutorials/BASH/scripts> cat backups2.sh
#!/bin/bash

OF=$(date +%Y-%m-%d).tgz

if [ -e "$OF" ]; then
  echo "You have already created a backup today"
  echo `ls -ltr $OF`
else
  tar -czf ${OF} ./*sh
fi
apacheco@apacheco:~/Tutorials/BASH/scripts> ls
2012-09-18.tgz  backups.csh  dosum.sh      hello.sh           name.sh    shift.sh
backups2.sh     backups.sh   helloname.sh  hellovariable.sh  quotes.sh  tmp
apacheco@apacheco:~/Tutorials/BASH/scripts> ./backups2.sh
apacheco@apacheco:~/Tutorials/BASH/scripts> ./backups2.sh
You have already created a backup today
-rw-r--r-- 1 apacheco users 1168 Sep 24 13:16 2012-09-24.tgz
apacheco@apacheco:~/Tutorials/BASH/scripts>
```

- Condition tests using the `if/then` may be nested

```
a=3
if [ "$a" -gt 0 ]; then
  if [ "$a" -lt 5 ]; then
    echo "The value of \"a\" lies somewhere between 0 and 5"
  fi
fi
```

- This is same as

```
if [[ "$a" -gt 0 &&  "$a" -lt 5 ]]; then
  echo "The value of \"a\" lies somewhere between 0 and 5"
fi
OR
if [ "$a" -gt 0 ] && [ "$a" -lt 5 ]; then
  echo "The value of \"a\" lies somewhere between 0 and 5"
fi
```

- A *loop* is a block of code that iterates a list of commands as long as the *loop control condition* is true.
- Loop constructs available in `bash`: `for`, `while` and `until`
- Loop constructs available in `tcsh`: `foreach` and `while`

`for/foreach` loop

- The `for` loop is the basic looping construct in `bash`

```
for arg in list
do
     some commands
done
```

- the `for` and `do` lines can be written on the same line: `for` *arg* in *list*; `do`
- `bash for` loops can also use C style syntax

```
for ((EXP1; EXP2; EXP3 )); do
     some commands
done
```

- The `foreach` loop is the basic looping construct in `tcsh`

```
foreach arg (list)
          some commands
end
```

`while` loop

- The `while` construct tests for a condition at the top of a loop, and keeps looping as long as that condition is true (returns a 0 exit status).
- In contrast to a `for` loop, a `while` loop finds use in situations where the number of loop repetitions is not known beforehand.
- `bash`

```
while [ condition ]
do
        some commands
done
```

- `tcsh`

```
while ( condition )
        some commands
end
```

`until` loop

- The `until` construct tests for a condition at the top of a loop, and keeps looping as long as that condition is false (opposite of `while` loop).

```
until [ condition is true ]
do
        some commands
done
```

- `for, while & until` loops can nested. To exit from the loop use the `break` command

```
apacheco:~/Tutorials/BASH/scripts> cat nestedloops.sh
#!/bin/bash

## Example of Nested loops

echo "Nested for loops"
for a in $(seq 1 5) ; do
  echo "Value of a in outer loop:" $a
  for b in `seq 1 2 5` ; do
    c=$(($a*$b))
    if [ $c -lt 10 ]; then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 10"
      break
    fi
  done
done
echo "========================"
echo
echo "Nested for and while loops"
for ((a=1;a<=5;a++)); do
  echo "Value of a in outer loop:" $a
  b=1
  while [ $b -le 5 ]; do
    c=$(($a*$b))
    if [ $c -lt 5 ]; then
      echo "a * b = $a * $b = $c"
    else
      echo "$a * $b > 5"
      break
    fi
    let b+=2
  done
done
echo "========================"
```

```
apacheco:~/Tutorials/BASH/scripts> ./nestedloops.sh
Nested for loops
Value of a in outer loop: 1
a * b = 1 * 1 = 1
a * b = 1 * 3 = 3
a * b = 1 * 5 = 5
Value of a in outer loop: 2
a * b = 2 * 1 = 2
a * b = 2 * 3 = 6
2 * 5 > 10
Value of a in outer loop: 3
a * b = 3 * 1 = 3
a * b = 3 * 3 = 9
3 * 5 > 10
Value of a in outer loop: 4
a * b = 4 * 1 = 4
4 * 3 > 10
Value of a in outer loop: 5
a * b = 5 * 1 = 5
5 * 3 > 10
========================

Nested for and while loops
Value of a in outer loop: 1
a * b = 1 * 1 = 1
a * b = 1 * 3 = 3
1 * 5 > 5
Value of a in outer loop: 2
a * b = 2 * 1 = 2
2 * 3 > 5
Value of a in outer loop: 3
a * b = 3 * 1 = 3
3 * 3 > 5
Value of a in outer loop: 4
a * b = 4 * 1 = 4
4 * 3 > 5
Value of a in outer loop: 5
5 * 1 > 5
========================
```

- The `case` and `select` constructs are technically not loops, since they do not iterate the execution of a code block.
- Like loops, however, they direct program flow according to conditions at the top or bottom of the block.

`case` construct

```
case "$variable" in
        "$condition1")
        some command
        ;;
        "$condition2")
        some other commands
        ;;
esac
```

`select` construct

```
select variable [in list]
do
        command
        break
done
```

- `tcsh` has the `switch` construct

`switch` construct

```
switch (arg list)
case "$variable"
        some command
        breaksw
end
```

Problem Description

- I have to run more than one serial job.
- I don't want to submit multiple job using the serial queue
- How do I submit *one* job which can run multiple serial jobs?

Solution

- Write a script which will log into all unique nodes and run your serial jobs in background.
- Easy said than done
- What do you need to know?
  1. Shell Scripting
  2. How to run a job in background
  3. Know what the `wait` command does

```
[apacheco@eric2 traininglab]$ cat checknodes.sh
#!/bin/bash
#
#PBS -q checkpt
#PBS -l nodes=4:ppn=4
#PBS -l walltime=00:10:00
#PBS -V
#PBS -o nodetest.out
#PBS -e nodetest.err
#PBS -N testing
#

export WORK_DIR=$PBS_O_WORKDIR
export NPROCS=`wc -l $PBS_NODEFILE |gawk '//{print $1}'`
NODES=(`cat "$PBS_NODEFILE"` )
UNODES=(`uniq "$PBS_NODEFILE"` )

echo "Nodes Available: " ${NODES[@]}
echo "Unique Nodes Available: " ${UNODES[@]}

echo "Get Hostnames for all processes"
i=0
for nodes in "${NODES[@]}"; do
  ssh -n $nodes 'echo $HOSTNAME '$i' ' &
  let i=i+1
done
wait

echo "Get Hostnames for all unique nodes"
i=0
NPROCS=`uniq $PBS_NODEFILE | wc -l |gawk '//{print $1}'`
let NPROCS-=1
while [ $i -le $NPROCS ] ; do
  ssh -n ${UNODES[$i]} 'echo $HOSTNAME '$i' '
  let i=i+1
done
```

```
[apacheco@eric2 traininglab]$ qsub checknodes.sh
[apacheco@eric2 traininglab]$ cat nodetest.out
---------------------------------------
Running PBS prologue script
---------------------------------------
User and Job Data:
---------------------------------------
Job ID:    422409.eric2
Username:  apacheco
Group:     loniadmin
Date:      25-Sep-2012 11:01
Node:      eric010 (3053)
---------------------------------------
PBS has allocated the following nodes:

eric010
eric012
eric013
eric026

A total of 16 processors on 4 nodes allocated
---------------------------------------------
Check nodes and clean them of stray processes
---------------------------------------------
Checking node eric010 11:01:52
Checking node eric012 11:01:54
Checking node eric013 11:01:56
Checking node eric026 11:01:57
Done clearing all the allocated nodes
-----------------------------------------------------
Concluding PBS prologue script - 25-Sep-2012 11:01:57
-----------------------------------------------------
Nodes Available:  eric010 eric010 eric010 eric010 eric012 eric012 eric012 eric012 eric013 eric013 eric013 eric01
eric026 eric026
Unique Nodes Available:  eric010 eric012 eric013 eric026
Get Hostnames for all processes
```

```
eric010 3
eric012 5
eric010 1
eric012 6
eric012 4
eric013 10
eric010 2
eric012 7
eric013 8
eric013 9
eric026 15
eric013 11
eric010 0
eric026 13
eric026 12
eric026 14
Get Hostnames for all unique nodes
eric010 0
eric012 1
eric013 2
eric026 3
------------------------------------------------------
Running PBS epilogue script    - 25-Sep-2012 11:02:00
------------------------------------------------------
Checking node eric010 (MS)
Checking node eric026 ok
Checking node eric013 ok
Checking node eric012 ok
Checking node eric010 ok
------------------------------------------------------
Concluding PBS epilogue script - 25-Sep-2012 11:02:06
------------------------------------------------------
Exit Status:
Job ID:          422409.eric2
Username:        apacheco
Group:           loniadmin
```

```
Job Name:        testing
Session Id:      3052
Resource Limits: ncpus=1,nodes=4:ppn=4,walltime=00:10:00
Resources Used:  cput=00:00:00,mem=5260kb,vmem=129028kb,walltime=00:00:01
Queue Used:      checkpt
Account String:  loni_loniadmin1
Node:            eric010
Process id:      4101
------------------------------------------------------
[apacheco@eric2 traininglab]$ cat nodetest.err
```

- A regular expression (regex) is a method of representing a string matching pattern.
- Regular expressions enable strings that match a particular pattern within textual data records to be located and modified and they are often used within utility programs and programming languages that manipulate textual data.
- Regular expressions are extremely powerful.
- Supporting Software and Tools
  1. Command Line Tools: grep, egrep, sed
  2. Editors: ed, vi, emacs
  3. Languages: awk, perl, python, php, ruby, tcl, java, javascript, .NET

Shell regex

- ? : match any single character.
- * : match zero or more characters.
- [ ] : match list of characters in the list specified
- [!] : match characters not in the list specified
- ^ : match at begining of line
- $: match at end of line
- [^] : match characters not in the list specified

- `grep` is a Unix utility that searches through either information piped to it or files in the current directory.
- `egrep` is extended grep, same as `grep -E`
- Use `zgrep` for compressed files.
- Usage: `grep <options> <search pattern> <files>`
- Commonly used options

  -i : ignore case during search
  -r : search recursively
  -v : invert match i.e. match everything except pattern
  -l : list files that match pattern
  -L : list files that do not match pattern
  -n : prefix each line of output with the line number within its input file.

```
apacheco@apacheco:~/Tutorials/BASH/scripts> egrep -i sum *
dosum.csh:@ SUM = $FIVE + $SEVEN
dosum.csh:echo "sum of 5 & 7 is " $SUM
dosum.sh:let SUM=$FIVE+$SEVEN
dosum.sh:echo "sum of 5 & 7 is " $SUM
apacheco@apacheco:~/Tutorials/BASH/scripts> egrep -il sum *
dosum.csh
dosum.sh
apacheco@apacheco:~/Tutorials/BASH/scripts> cd ../
apacheco@apacheco:~/Tutorials/BASH> egrep -inR 'backupdir' *
Bash-Scripting-Fall-2012.tex:1084:BACKUPDIR=$(pwd)
Bash-Scripting-Fall-2012.tex:1085:OF=$BACKUPDIR/$(date +%Y-%m-%d).tgz
scripts/backups.sh:3:BACKUPDIR=${HOME}
scripts/backups.sh:4:OF=$BACKUPDIR/$(date +%Y-%m-%d).tgz
scripts/backups.csh:3:set BACKUPDIR=`pwd`
scripts/backups.csh:4:set OF = $BACKUPDIR/`date +%Y-%m-%d`.tgz
```

- The Awk text-processing language is useful for such tasks as:
  - Tallying information from text files and creating reports from the results.
  - Adding additional functions to text editors like "vi".
  - Translating files from one format to another.
  - Creating small databases.
  - Performing mathematical operations on files of numeric data.
- Awk has two faces:
  - it is a utility for performing simple text-processing tasks, and
  - it is a programming language for performing complex text-processing tasks.
- Simplest form of using awk
  - **awk** *search pattern* {`program actions`}
  - Most command action: `print`
  - Print file dosum.sh: `awk '{print $0}' dosum.sh`
  - Print line matching bash in all files in current directory:
    `awk '/bash/{print $0}' *.sh`
- **awk** supports the if conditional and for loops

```
awk '{ if (NR > 0){print "File not empty"}}' hello.sh
awk '{for (i=1;i<=NF;i++){print $i}}' name.sh
ls *.sh | awk -F. '{print $1}'
```

NR≡Number of records; NF≡Number of fields (or columns)

- awk one-liners: `http://www.pement.org/awk/awk1line.txt`

- sed ("stream editor") is Unix utility for parsing and transforming text files.
- sed is line-oriented, it operates one line at a time and allows regular expression matching and substitution.
- The most commonly used feature of sed is the 's' (substitution command)
  - echo Auburn Tigers | sed 's/Auburn/LSU/g'
  - Add the `-e` to carry out multiple matches.
  - echo LSU Tigers | sed -e 's/LSU/LaTech/g' -e 's/Tigers/Bulldogs/g'
  - insert a blank line above and below the lines that match regex: `sed '/regex/{x;p;x;G;}'`
  - delete all blank lines in a file: `sed '/^$/d'`
  - delete lines n through m in file: `sed 'n,md'`
  - delete lines matching pattern regex: `sed '/regex/d'`
  - print only lines which match regular expression: `sed -n '/regex/p'`
  - print section of file between two regex: `sed -n '/regex1/,/regex2/p'`
  - print section of file from regex to enf of file: `sed -n '/regex1/,$p'`
- sed one-liners: `http://sed.sourceforge.net/sed1line.txt`

- BASH Programming `http://tldp.org/HOWTO/Bash-Prog-Intro-HOWTO.html`
- Advanced Bash-Scripting Guide `http://tldp.org/LDP/abs/html/`
- Regular Expressions `http://www.grymoire.com/Unix/Regular.html`
- AWK Programming `http://www.grymoire.com/Unix/Awk.html`
- awk one-liners: `http://www.pement.org/awk/awk1line.txt`
- sed `http://www.grymoire.com/Unix/Sed.html`
- sed one-liners: `http://sed.sourceforge.net/sed1line.txt`
- CSH Programming `http://www.grymoire.com/Unix/Csh.html`
- csh Programming Considered Harmful
  `http://www.faqs.org/faqs/unix-faq/shell/csh-whynot/`
- Wiki Books `http://en.wikibooks.org/wiki/Subject:Computing`

- User's Guide
  - HPC: `http://www.hpc.lsu.edu/help`
  - LONI: `https://docs.loni.org`
- Contact us
  - Email ticket system: sys-help@loni.org
  - Telephone Help Desk: 225-578-0900
  - Instant Messenger (AIM, Yahoo Messenger, Google Talk)
    - Add "lsuhpchelp"