# Basic Shell Scripting

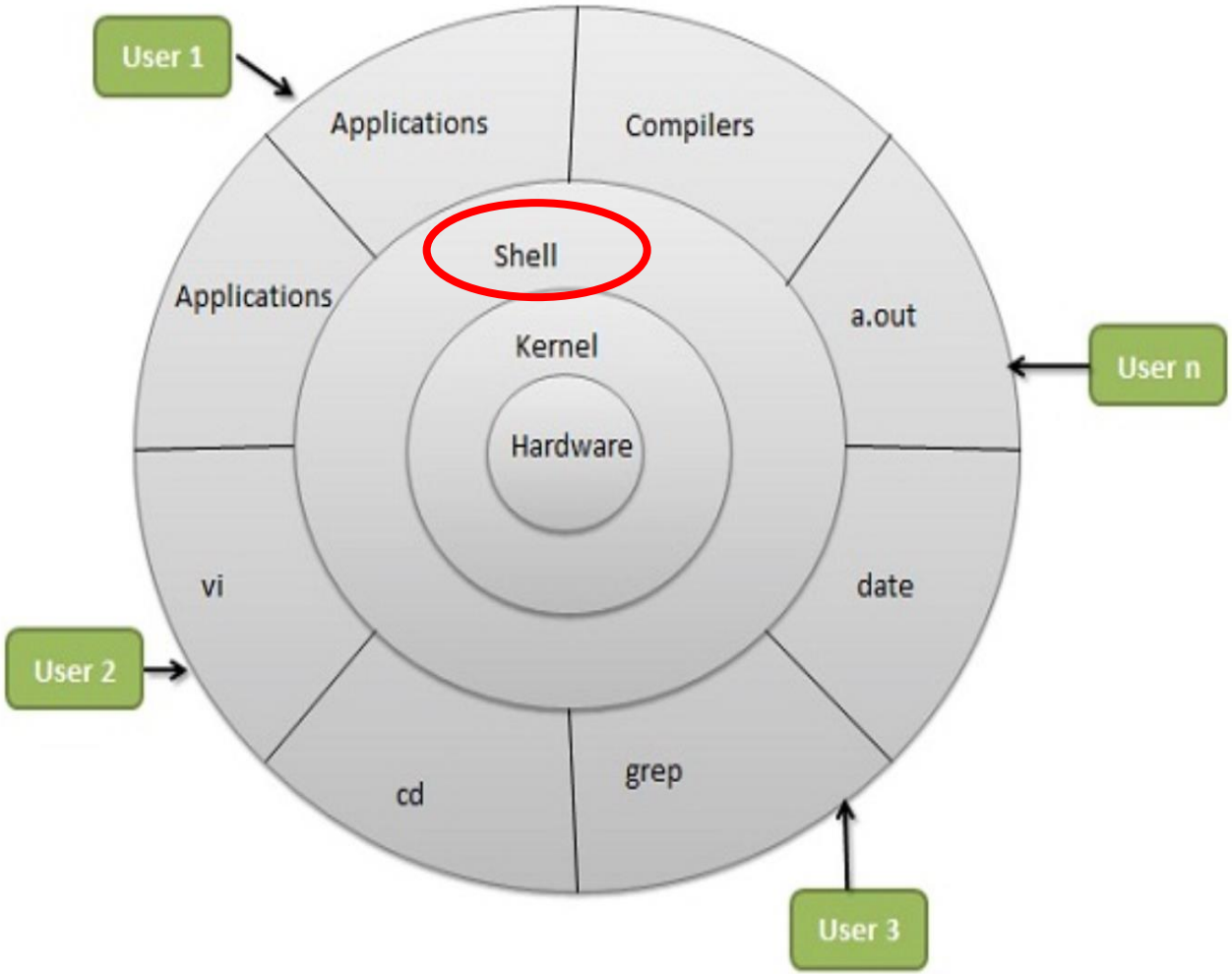
Feng Chen
HPC User Services
LSU HPC & LONI
sys-help@loni.org

Louisiana State University
Baton Rouge
September 25, 2019

# Outline

- Introduction to Linux Shell
- Shell Scripting Basics
  - Variables/Special Characters
  - Arithmetic Operations
  - Arrays
- Beyond Basic Shell Scripting
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

# Linux System Architecture



# What is a Linux Shell

- An application running on top of the kernel and provides a command line interface to the system
  - Process user's commands, gather input from user and execute programs
- Types of shell with varied features
  - sh
    - the original Bourne shell.
  - ksh
    - one of the three: Public domain ksh (pdksh), AT&T ksh or mksh
  - bash
    - the GNU Bourne-again shell. It is mostly Bourne-compatible, mostly POSIX-compatible, and has other useful extensions. It is the default on most Linux systems.
  - csh
    - BSD introduced the C shell, which sometimes resembles slightly the C programming language.
  - tcsh
    - csh with more features. csh and tcsh shells are NOT Bourne-compatible.

# Shell Comparison

| Software | sh | csh | ksh | bash | tcsh |
|---|---|---|---|---|---|
| Programming language | y | y | y | y | y |
| Shell variables | y | y | y | y | y |
| Command alias | n | y | y | y | y |
| Command history | n | y | y | y | y |
| Filename autocompletion | n | y* | y* | y | y |
| Command line editing | n | n | y* | y | y |
| Job control | n | y | y | y | y |

*: not by default

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

# What can you do with a shell?

- Check the current shell you are using
  - `echo $0`
- List available shells on the system
  - `cat /etc/shells`
- Change to another shell
  - `csh`
- Date
  - `date`
- wget: get online files
  - wget https://ftp.gnu.org/gnu/gcc/gcc-7.1.0/gcc-7.1.0.tar.gz
- Compile and run applications
  - `gcc hello.c –o hello`
  - `./hello`
- What we need to learn today?
  - Automation of an entire script of commands!
  - Use the shell script to run jobs – Write job scripts

# Shell Scripting

- Script: a program written for a software environment to automate execution of tasks
  - A series of shell commands put together in a file
  - When the script is executed, those commands will be executed one line at a time automatically
  - Shell script is interpreted, not compiled.

- The majority of script programs are “quick and dirty”, where the main goal is to get the program written quickly
  - May not be as efficient as programs written in C and Fortran

# When NOT to use Shell Scripting…

- Selected situations:
  - Resource-intensive tasks, especially where speed is a factor (sorting, hashing, recursion [2] ...)
  - Procedures involving heavy-duty math operations, especially floating point arithmetic, arbitrary precision calculations, or complex numbers (use C++ or FORTRAN instead)
  - Complex applications, where structured programming is a necessity (type-checking of variables, function prototypes, etc.)
  - Extensive file operations required (Bash is limited to serial file access, and that only in a particularly clumsy and inefficient line-by-line fashion.)
  - Need native support for multi-dimensional arrays, data structures, such as linked lists or trees
  - Need to use libraries or interface with legacy code

# Script Example (~/.bashrc)

```
# .bashrc

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

# User specific aliases and functions
export PATH=$HOME/packages/bin:$PATH
export LD_LIBRARY_PATH=$HOME/packages/lib:$LD_LIBRARY_PATH
alias qsubI="qsub -I -X -l nodes=1:ppn=20 -l walltime=01:00:00 –A
my_allocation"
alias lh="ls -altrh"
```

# Hello World

```
#!/bin/bash
# A script example
echo "Hello World!" # print something
```

1. #!: "Shebang" line to instruct which interpreter to use. In the current example, bash. For tcsh, it would be: #!/bin/tcsh
2. All comments begin with "#".
3. Print "Hello World!" to the screen.

```
[fchen14@mike1 shelltut]$ ./hello_world.sh # using default /bin/bash
Hello World!
[fchen14@mike1 shelltut]$ bash hello_world.sh # using bash to run the script
Hello World!
```

# Interactive and non-interactive shells

- An interactive shell is one started without non-option arguments, unless -s is specified, without specifying the -c option, and whose input and error output are both connected to terminals or one started with the -i option.
  - *The user can interact with the shell from the terminal.*
  - e.g., open an interactive shell by typing bash or ssh from the terminal
- A shell running a script is always a non-interactive shell.
  - All the same, the script can still access its tty. It is even possible to emulate an interactive shell in a script.

- Test whether you are using an interactive shell using `$-` (prints The current set of options in your current shell.)

```
[fchen14@philip1 shelltut]$ echo $-
himBH
[fchen14@philip1 shelltut]$ cat checkshell.sh
#!/bin/bash
# read value # you can still interact with the script
echo $-
[fchen14@philip1 shelltut]$ ./checkshell.sh
hB
```

# Subshell

- Definition:
  - A subshell is a child process launched by a shell (or shell script).
  - Just as your commands are interpreted at the command-line prompt, similarly does a script batch-process a list of commands.
  - Each shell script running is, in effect, a subprocess (child process) of the parent shell.
- Two typical examples of starting subshell:
  - Running a shell script launches a new process, a *subshell*.
  - Type "`bash`" from an interactive shell

# Outline

- Introduction to Linux Shell
- Shell Scripting Basics
  - Variables/Special Characters
  - Arithmetic Operations
- Beyond Basic Shell Scripting
  - Control flow
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

# Variables

- Variable names
  - Must start with a letter or underscore
  - Number can be used anywhere else
  - Do not use special characters such as `@,#,%,$`
  - Case sensitive
  - Allowed: `VARIABLE, VAR1234able, var_name, _VAR`
  - Not allowed: `1var, %name, $myvar, var@NAME, myvar-1`
- To reference a variable, prepend **`$`** to the name of the variable
- Example: `$PATH, $LD_LIBRARY_PATH, $myvar` etc.

# Global and Local Variables

- Two types of variables:
  - Global (Environmental) variables
    - Inherited by subshells (child process, see next slide)
    - provide a simple way to share configuration settings between multiple applications and processes in Linux
    - Using all uppercase letters by convention
    - Example: `PATH, LD_LIBRARY_PATH, DISPLAY` etc.
    - `printenv/env` list the current environmental variables in your system.
  - Local (shell) variables
    - Only visible to the current shell
    - Not inherited by subshells

# Editing Variables

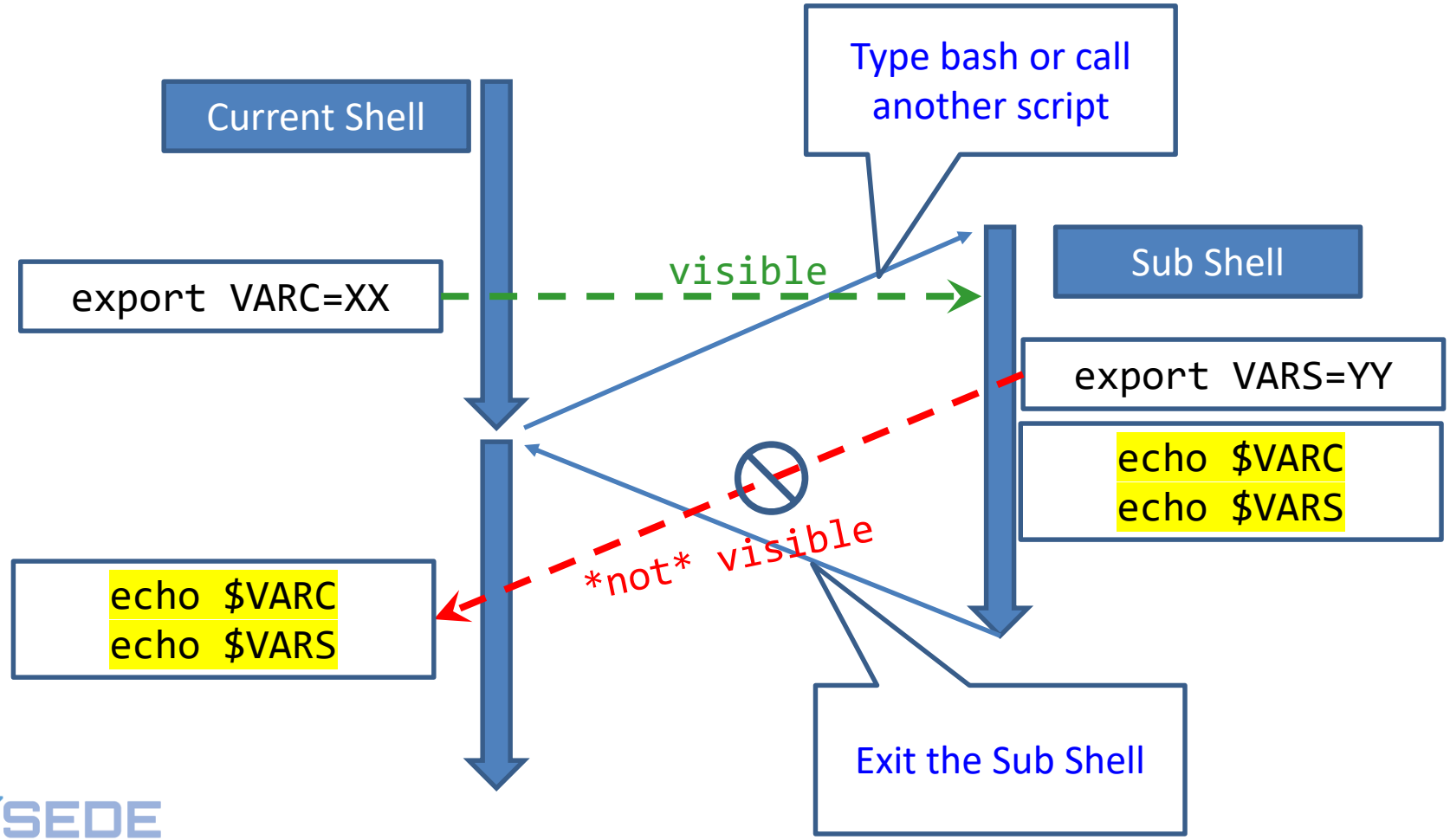
- Assign values to variables

| Type | sh/ksh/bash | csh/tcsh |
|---|---|---|
| Shell (local) | `name=value` | `set name=value` |
| Environment (global) | `export name=value` | `setenv name value` |

- Local (Shell) variables is only valid within the current shell, while environment variables are valid for all subsequently opened shells.
- Example: useful when running a script, where exported variables (global) at the terminal can be inherited within the script.

| With export | Without export |
|---|---|
| `$ export v1=one`<br>`$ bash`<br>`$ echo $v1`<br>`→one` | `$ v1=one`<br>`$ bash`<br>`$ echo $v1`<br>`→` |

# Global and Local Variables - who can see who?



# How to inherit the variables in the script?

- Using the *source* command, it has a synonym in dot "." (period)
  - Syntax:

```
. filename [arguments]
source filename [arguments]
```

  - The script does not need execute permission in this case. Commands are executed *in the current shell process*, so any changes made to your environment will be visible when the script finishes execution.
  - Executing will run the commands in a new shell process (subshell).

```
[fchen14@philip1 shelltut]$ cat source_var.sh
#!/bin/bash
export myvar="newvalue"
[fchen14@philip1 shelltut]$ bash source_var.sh
[fchen14@philip1 shelltut]$ echo $myvar

[fchen14@philip1 shelltut]$ source source_var.sh
[fchen14@philip1 shelltut]$ echo $myvar
newvalue
```

# List of Some Environment Variables

| | |
|---|---|
| PATH | A list of directory paths which will be searched when a command is issued |
| LD_LIBRARY_PATH | colon-separated set of directories where libraries should be searched for first |
| HOME | indicate where a user's home directory is located in the file system. |
| PWD | contains path to current working directory. |
| OLDPWD | contains path to previous working directory. |
| TERM | specifies the type of computer terminal or terminal emulator being used |
| SHELL | contains name of the running, interactive shell. |
| PS1 | default command prompt |
| PS2 | Secondary command prompt |
| HOSTNAME | The systems host name |
| USER | Current logged in user's name |
| DISPLAY | Network name of the X11 display to connect to, if available. |

# Quotations

- Single quotation
  - Enclosing characters in single quotes (') preserves the literal value of each character within the quotes. A single quote may not occur between single quotes, even when preceded by a backslash.
- Double quotation
  - Enclosing characters in double quotes (") preserves the literal value of all characters within the quotes, with the exception of ‘$’, ‘`’, ‘\’
- Back “quotation?”
  - Command substitution (``) allows the output of a command to replace the command itself, enclosed string is executed as a command, almost the same as $()

# Quotation - Examples

```
[fchen14@mike1 ~]$ str1='echo $USER'
[fchen14@mike1 ~]$ echo "$str1"
echo $USER
[fchen14@mike1 ~]$ str2="echo $USER"
[fchen14@mike1 ~]$ echo "$str2"
echo fchen14
[fchen14@mike1 ~]$ str3=`echo $USER`
[fchen14@mike1 ~]$ echo $str3
fchen14
[fchen14@mike1 ~]$ str3=$(echo $USER)
[fchen14@mike1 ~]$ echo "$str3"
fchen14
```

Always use double quotes around variable substitutions and command substitutions: `"$foo"`, `"${foo}"`

# Special Characters (1)

| | |
|---|---|
| # | Start a comment line. |
| $ | Indicate the name of a variable. |
| \ | Escape character to display next character literally |
| {} | Enclose name of variable |
| ; | Command separator. Permits putting two or more commands on the same line. |
| ;; | Terminator in a case option |
| . | "dot" command, equivalent to `source` (for bash only) |
| \| | Pipe: use the output of a command as the input of another one |
| > < | Redirections (`0<:` standard input; `1>:` standard out; `2>:` standard error) |

# Special Characters (2)

| | |
|---|---|
| `$?` | Exit status for the last command, 0 is success, failure otherwise |
| `$$` | Process ID variable. |
| `[]` | Test expression, eg. if condition |
| `[[ ]]` | Extended test expression, more flexible than [ ] |
| `$[], $(( ))` | Integer expansion |
| `\|\|, &&, !` | Logical OR, AND and NOT |
| | |

# Outline

- Introduction to Linux Shell
- Shell Scripting Basics
  - Variables/Special Characters
  - Arithmetic Operations
- Beyond Basic Shell Scripting
  - Arrays
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

# Integer Arithmetic Operations

| Operation | Operator |
|---|---|
| Addition | + |
| Subtraction | - |
| Multiplication | * |
| Division | / |
| Exponentiation | ** (bash only) |
| Modulo | % |

# Integer Arithmetic Operations

- `$((...))` or `$[...]` commands
  - `x=$((1+2))` `# Addition, suggested`
  - `echo $[$x*$x] # Multiplication, deprecated`
- `let` command:
  - `let c=$x+$x` `# no space`
  - `let c=x+x` `# you can omit the $ sign`
  - `let c="x + x"` `# can have space`
  - `let c+=1 or let --c # C-style increment operator`
- `expr` command:
  - `expr 10 / 2` (space required)

Note: Bash is picky about spaces!

# Floating-Point Arithmetic Operations

GNU basic calculator (bc) external calculator

- Add two numbers

```
echo "3.8 + 4.2" | bc
```

- Divide two numbers and print result with a precision of 5 digits:

```
echo "scale=5; 2/5" | bc
```

- Convert between decimal and binary numbers

```
echo "ibase=10; obase=2; 10" |bc
```

- Call bc directly:

```
bc <<< "scale=5; sqrt(2)"
```

# Outline

- Introduction to Linux Shell
- Shell Scripting Basics
  - Variables
  - Quotations
  - Arithmetic Operations
  - Arrays
- Beyond Basic Shell Scripting
  - Flow Control
  - Command Line Arguments
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

# Arrays Operations (1)

- Initialization

```
my_array=("Alice" "Bill" "Cox” "David")
my_array[0]="Alice";
my_array[1]="Bill”
```

- Bash supports one-dimensional arrays
  - Index starts at 0
  - No space around “=“
- Reference an element

```
${my_array[i]} # must include curly braces {}
```

- Print the whole array

```
${my_array[@]}
```

- Length of array

```
${#my_array[@]}
```

# Array Operations (2)

- Add an element to an existing array
  - `my_array=(first ${my_array[@]})`
  - `my_array=("${my_array[@]}" last)`
  - `my_array[4]=(“Nason”)`
- Copy the current array to a new array
  - `new_array=(${my_array[@]})`
- Concatenate two arrays
  - `two_arrays=(${my_array[@]} ${new_array[@]})`

# Array Operations (3)

- Delete the entire array
  - `unset my_array`
- Delete an element to an existing array
  - `unset my_array[0]`

# Outline

- Introduction to Linux Shell
- Shell Scripting Basics
  - Arrays
  - Arithmetic Operations
- Beyond Basic Shell Scripting
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

# Flow Control

- Shell scripting languages execute commands in sequence similar to programming languages such as C and Fortran
  - Control constructs can change the order of command execution
- Control constructs in bash
  - Conditionals:
    - if-then-else
    - Switches: case
  - Loops: for, while, until

# if statement

- if/then construct test whether the exit status of a list of commands is **0**, and if so, execute one or more commands

```
if [ condition ]; then
   Do something
elif [ condition 2 ] ; then
   Do something
else
   Do something else
fi
```

- Strict spaces between condition and the brackets (bash)
- `[[ condition ]]` `extended test` construct is the more versatile Bash version of `[ condition ]`, generally safer to use.

# File Operations

| Operation | bash |
|---|---|
| File exists | `if [ -e test ]` |
| File is a regular file | `if [ -f test]` |
| File is a directory | `if [ -d /home ]` |
| File is not zero size | `if [ -s test ]` |
| File has read permission | `if [ -r test ]` |
| File has write permission | `if [ -w test ]` |
| File has execute permission | `if [ -x test ]` |

# Integer Comparisons

| Operation | bash |
|---|---|
| Equal to | `if [ 1 -eq 2 ]` |
| Not equal to | `if [ $a -ne $b ]` |
| Greater than | `if [ $a -gt $b ]` |
| Greater than or equal to | `if [ 1 -ge $b ]` |
| Less than | `if [ $a -lt 2 ]` |
| Less than or equal to | `if [ $a -le $b ]` |

# String Comparisons

| Operation | bash |
|---|---|
| Equal to | `if [ $a == $b ]` |
| Not equal to | `if [ $a != $b ]` |
| Zero length or null | `if [ -z $a ]` |
| Non zero length | `if [ -n $a ]` |

# Logical Operators

| Operation | Example |
|---|---|
| ! (NOT) | `if [ ! –e test ]` |
| && (AND) | `if [ -f test] && [ -s test ]`<br>`if [[ -f test && -s test ]]`<br>`if ( -e test && ! –z test )` |
| \|\| (OR) | `if [ -f test1 ] \|\| [ -f test2 ]`<br>`if [[ -f test1 \|\| -f test2 ]]` |

# if condition examples

**Example 1:**

```
read input
if [ $input == "hello" ]; then
   echo hello;
else echo wrong ;
fi
```

**Example 2**

```
touch test.txt
if [ -e test.txt ]; then
   echo “file exist”
elif [ ! -s test.txt ]; then
   echo “file empty”;
fi
```

**What happens after**

```
echo “hello world” >> test.txt
```

# Loop Constructs

- A loop is a block of code that iterates a list of commands as long as the loop control condition stays true
- Loop constructs

  `for`, `while` and `until`

# for loop examples

Exmaple1:

```
for arg in `seq 1 4`
do
   echo $arg;
   touch test.$arg
done
```

How to delete test files using a loop?

```
rm test.[1-4]
```

Example 2:

```
for file in `ls /home/$USER`
do
   cat $file
done
```

# While Loop

- The `while` construct test for a condition at the top of a loop and keeps going as long as that condition is true.
- In contrast to a `for` loop, a `while` is used when loop repetitions is not known beforehand.

```
read counter
while [ $counter -ge 0 ]
do let counter--
   echo $counter
done
```

# Until Loop

- The `until` construct test a condition at the top of a loop, and stops looping when the condition is met (opposite of `while` loop)

```
read counter
until [ $counter -lt 0 ]
do let counter--
   echo $counter
done
```

# Switching Constructs - bash

- The `case` constructs are technically not loops since they do not iterate the execution of a code block

```
#!/bin/sh
echo "Please talk to me ..."
while :
do
  read INPUT_STRING
  case $INPUT_STRING in
    hello)
        echo "Hello yourself!"
        ;;
    bye)
        echo "See you again!"
        break
        ;;
    *)
        echo "Sorry, I don't understand"
        ;;
  esac
Done
echo "That's all folks!"
```

# Outline

- Introduction to Linux Shell
- Shell Scripting Basics
- Beyond Basic Shell Scripting
  - Arithmetic Operations
  - Arrays
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

# Functions

- A function is a code block that implements a set of operations. Code reuse by passing parameters,
  - Syntax:

```
function_name () {
    command...
}
```

- By default all variables are global.
- Modifying a variable in a function changes it in the whole script.
- Create a local variables using the local command, which is invisible outside the function

```
local var=value
local varName
```

# Pass Arguments to Bash Scripts

- Note the difference between the arguments passed to the script and the function.
- All parameters can be passed at runtime and accessed via `$1, $2, $3…, add {} when >=10`
- `$0:` the shell script name
- Array variable called `FUNCNAME` contains the names of all shell functions currently in the execution call stack.
- `$* or $@:` all parameters passed to a function
- `$#:` number of positional parameters passed to the function
- $?: exist code of last command
- $$: PID of current process

# Function example

```
#!/bin/bash

func_add () # define a simple function
{
    local x=$1    # 1st argument to the function
    local y=$2    # 2nd argument to the function
    result=$(( x + y ))
    # echo "result is: " $result
}

a=3;b=4
echo "a= $a, b= $b"
result="nothing"
echo "result before calling the function is: " $result
func_add $a $b # note this is arguments to the function
echo "result by passing function arguments is: " $result
func_add $1 $2 # note this is command line arguments
echo "result by passing command line arguments is: " $result
```

# Outline

- Introduction to Linux Shell
- Shell Scripting Basics
  - Variables/Special Characters
  - Arrays
  - Arithmetic Operations
- Beyond Basic Shell Scripting
  - Flow Control
  - Functions
- Advanced Text Processing Commands (grep, sed, awk)

# Advanced Text Processing Commands

- `grep`
- `sed`
- `awk`

# One slide about Regular Expression

- What are Regular Expressions (regex)?
  - They describe patterns in strings
  - These patterns can be used to modify strings
  - Invented by Stephen Cole Kleene
  - Idea of RegEx dates back to the 1950s
- Today, they come indifferent “flavors”
- PCRE, POSIX Basic & Extended RegEx, ECMA RegEx and loads more!
- Examples:

# Regex examples

- Anchors - `^ and $`
  - `^The` matches any string that starts with The
  - `end$` matches a string that ends with end
  - `^The end$` exact string match (starts and ends with The end)
  - `roar` matches any string that has the text roar in it
- Quantifiers - `* + ? and {}`
  - `abc*` matches a string that has ab followed by zero or more c
  - `abc+` matches a string that has ab followed by one or more c
  - `abc?` matches a string that has ab followed by zero or one c
  - `abc{2}` matches a string that has ab followed by 2 c
  - `abc{2,}` matches a string that has ab followed by 2 or more c
  - `abc{2,5}` matches a string that has ab followed by 2 up to 5 c
- OR operator - `| or []`
  - `a(b|c)` matches a string that has a followed by b or c
  - `a[bc]` same as previous

# grep & egrep

- **`grep:`** Unix utility that searches a pattern through either information piped to it or files.
- **`egrep:`** extended grep, same as `grep -E`
- **`zgrep:`** compressed files.

- Usage: `grep <options> <search pattern> <files>`
- Options:

| | |
|---|---|
| **`-i`** | ignore case during search |
| **`-r,-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. |
| **`-A num`** | print `num` lines of trailing context after matching lines. |
| **`-B num`** | print `num` lines of leading context before matching lines. |

# grep Examples

- Search files containing the word `bash` in current directory

```
grep bash *
```

- Search files NOT containing the word `bash` in current directory

```
grep -v bash *
```

- Repeat above search using a case insensitive pattern match and print line number that matches the search pattern

```
grep -in bash *
```

- Search files not matching certain name pattern

```
ls | grep -vi fun
```

# grep Examples

```
100  Thomas  Manager    Sales       $5,000
200  Jason   Developer  Technology  $5,500
300  Raj     Sysadmin   Technology  $7,000
500  Randy   Manager    Sales       $6,000
```

- grep OR

```
grep 'Man\|Sales' employee.txt
-> 100 Thomas Manager   Sales      $5,000
   300 Raj   Sysadmin   Technology $7,000
   500 Randy Manager    Sales      $6,000
```

- grep AND

```
grep -i 'sys.*Tech' employee.txt
-> 100300 Raj   Sysadmin  Technology $7,000
```

# sed

- "stream editor" to parse and transform information
  - information piped to it or from files
- line-oriented, operate one line at a time and allow regular expression matching and substitution.
- `s` substitution command

# `sed` commands and flags

| Flags | Operation | Command | Operation |
|---|---|---|---|
| `-e` | combine multiple commands | `s` | substitution |
| `-f` | read commands from file | `g` | global replacement |
| `-h` | print help info | `p` | print |
| `-n` | disable print | `i` | ignore case |
| `-V` | print version info | `d` | delete |
| `-r` | use extended regex | `G` | add newline |
| | | `w` | write to file |
| | | `x` | exchange pattern with hold buffer |
| | | `h` | copy pattern to hold buffer |
| | | `;` | separate commands |

# sed Examples

```
#!/bin/bash

# My First Script

echo "Hello World!”
```

# sed Examples (2)

- Delete blank lines from a file

```
sed '/^$/d' hello.sh

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

- Delete line `n` through `m` in a file

```
sed '2,4d' hello.sh

#!/bin/bash
echo "Hello World!"
```

# sed Examples (1)

- Add flag -e to carry out multiple matches.

```
cat hello.sh | sed -e 's/bash/tcsh/g' -e 's/First/Second/g'
#!/bin/tcsh
# My Second Script
echo "Hello World!"
```

- Alternate form

```
sed 's/bash/tcsh/g; s/First/Second/g' hello.sh

#!/bin/tcsh
# My Second Script
echo "Hello World!"
```

- The default delimiter is slash (/), can be changed

```
sed 's:/bin/bash:/bin/tcsh:g' hello.sh

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

# sed Examples (4)

- Replace-in-place with a backup file

```
sed –i.bak ’/First/Second/i’ hello.sh
```

- echo with sed

```
$ echo "shell scripting" | sed "s/[si]/?/g”
$ ?hell ?cr?pt?ng

$ echo "shell scripting 101" | sed "s/[0-9]/#/g”
$  shell scripting ###
```

# awk

- The `awk` text-processing language is useful for tasks such 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.

# How Does awk Work

- `awk` reads the file being processed line by line.
- The entire content of each line is split into columns with space or tab as the delimiter.
- `$0` Print the entire line
- `$1, $2, $3, ...` for each column (if exists)
- `NR` number of records (lines)
- `NF` number of fields or columns in the current line.
- By default the field delimiter is space or tab. To change the field delimiter use the `-F<delimiter>` command.

# awk Syntax

`awk pattern {action}`

`pattern` decides when `action` is performed

`Actions:`

- Most common action: `print`
- Print file dosum.sh:

```
awk '{print $0}' dosum.sh
```

- Print line matching files in all `.sh` files in current directory:

```
awk '/bash/{print $0}' *.sh
```

```
uptime
11:18am up 14 days 0:40, 5 users, load average: 0.15, 0.11,
0.17

uptime | awk '{print $0}'
11:18am up 14 days 0:40, 5 users, load average: 0.15, 0.11,
0.17

uptime | awk '{print $1,NF}'
11:18am 12

uptime | awk '{print NR}'
1

uptime | awk -F, '{print $1}'
11:18am up 14 days 0:40

for i in $(seq 1 3); do touch file${i}.dat ; done
for i in file* ; do
> prefix=$(echo $i | awk -F. '{print $1}')
> suffix=$(echo $i | awk -F. '{print $NF}')
> echo $prefix $suffix $i; done

file1 dat file1.dat
file2 dat file2.dat
file3 dat file3.dat
```

# Awk Examples

- Print list of files that are bash script files

```
awk '/^#\!\/bin\/bash/{print $0, FILENAME}' *
→   #!/bin/bash Fun1.sh
    #!/bin/bash fun_pam.sh
    #!/bin/bash hello.sh
    #!/bin/bash parm.sh
```

- Print extra lines below patterns

```
awk '/sh/{print;getline;print}' <hello.sh
#!/bin/bash
```

# Getting Help

- User Guides
  - LSU HPC: http://www.hpc.lsu.edu/docs/guides.php#hpc
  - LONI: http://www.hpc.lsu.edu/docs/guides.php#loni
- Documentation: http://www.hpc.lsu.edu/docs
- Archived tutorials: http://www.hpc.lsu.edu/training/archive/tutorials.php
- Contact us
  - Email ticket system: sys-help@loni.org
  - Telephone Help Desk: 225-578-0900

# Upcoming trainings in October

October 2,2019: Version Control using Git

October 9,2019: Introduction to Python