Basic Shell Scripting

Wei Feinstein

HPC User Services
LSU HPC & LON
sys-help@loni.org
February 2018
Outline

• Introduction to Linux Shell
• Shell Scripting Basics
  • Variables
  • Quotations
• Beyond Basic Shell Scripting
  – Arithmetic Operations
  – Arrays
  – Flow Control
  – Functions
• Advanced Text Processing Commands (grep, sed, awk)
Linux System Architecture
What is a Shell

- An application running on top of the kernel and provides a powerful interface to the system
- Process user’s commands, gather input from user and execute programs
- Types of shell with varied features
  - sh
  - csh
  - ksh
  - bash
  - tcsh
# Shell Comparison

<table>
<thead>
<tr>
<th>Software</th>
<th>sh</th>
<th>csh</th>
<th>ksh</th>
<th>bash</th>
<th>tcsh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Programming language</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Shell variables</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Command alias</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Command history</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Filename autocompletion</td>
<td>n</td>
<td>y*</td>
<td>y*</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Command line editing</td>
<td>n</td>
<td>n</td>
<td>y*</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>Job control</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>

*: not by default

http://www.cis.rit.edu/class/simg211/unixintro/Shell.html
What can you do with a shell?

- Check the current shell
  - `echo $SHELL`
- List available shells on the system
  - `cat /etc/shells`
- Change to another shell
  - `exec sh`
- Date and time
  - `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`
Outline

- Introduction to Linux Shell
- Shell Scripting Basics
  - Variables
  - Quotations
- Beyond Basic Shell Scripting
- Advanced Text Processing Commands (grep, sed, awk)
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

- 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
Script Example (~/.bashrc)

```bash
# .bashrc

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

# User specific aliases and functions
export PATH=$HOME/packages/eFindsite/bin:$PATH
export LD_LIBRARY_PATH=$HOME/packages/eFindsite/lib:$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"

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.
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

- Two types of variables:
  - Global variables (ENVIRONMENT variables)
  - Local variables (user defined variables)
Global Variables

- Environment variables provide a simple way to share configuration settings between multiple applications and processes in Linux
  - Using all uppercase letters
  - Example: PATH, LD_LIBRARY_PATH, DISPLAY etc.

- To reference a variable, prepend $ to the name of the variable
  - Example: $PATH, $LD_LIBRARY_PATH, $DISPLAY etc.

- `printenv/env` list the current environmental variables in your system.
## List of Some Environment Variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PATH</td>
<td>A list of directory paths which will be searched when a command is issued.</td>
</tr>
<tr>
<td>LD_LIBRARY_PATH</td>
<td>Colon-separated set of directories where libraries should be searched for first.</td>
</tr>
<tr>
<td>HOME</td>
<td>Indicate where a user's home directory is located in the file system.</td>
</tr>
<tr>
<td>PWD</td>
<td>Contains path to current working directory.</td>
</tr>
<tr>
<td>OLDPWD</td>
<td>Contains path to previous working directory.</td>
</tr>
<tr>
<td>TERM</td>
<td>Specifies the type of computer terminal or terminal emulator being used.</td>
</tr>
<tr>
<td>SHELL</td>
<td>Contains name of the running, interactive shell.</td>
</tr>
<tr>
<td>PS1</td>
<td>Default command prompt</td>
</tr>
<tr>
<td>PS2</td>
<td>Secondary command prompt</td>
</tr>
<tr>
<td>HOSTNAME</td>
<td>The system's host name</td>
</tr>
<tr>
<td>USER</td>
<td>Current logged in user's name</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>Network name of the X11 display to connect to, if available.</td>
</tr>
</tbody>
</table>
Editing Variables

- Assign values to variables

<table>
<thead>
<tr>
<th>Type</th>
<th>sh/ksh/bash</th>
<th>csh/tcsh</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shell (local)</td>
<td>name=value</td>
<td>set name=value</td>
</tr>
<tr>
<td>Environment (global)</td>
<td>export name=value</td>
<td>setenv name value</td>
</tr>
</tbody>
</table>

- 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.

<table>
<thead>
<tr>
<th>With export</th>
<th>Without export</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ export v1=one</td>
<td>$ v1=one</td>
</tr>
<tr>
<td>$ bash</td>
<td>$ bash</td>
</tr>
<tr>
<td>$ echo $v1</td>
<td>$ echo $v1</td>
</tr>
<tr>
<td>→one</td>
<td></td>
</tr>
</tbody>
</table>
Quotations

- Single quotation
  - Enclosed string is read literally
- Double quotation
  - Enclosed string is expanded
- Back quotation
  - Enclose string is executed as a command
Quotation - Examples

```bash
str1='echo $USER'

str1='echo $USER'

str2="echo $USER"

str3=`echo $USER`
```

Basic Shell Scripting
## Special Characters (1)

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Start a comment line.</td>
</tr>
<tr>
<td>$</td>
<td>Indicate the name of a variable.</td>
</tr>
<tr>
<td>\</td>
<td>Escape character to display next character literally</td>
</tr>
<tr>
<td>{}</td>
<td>Enclose name of variable</td>
</tr>
<tr>
<td>;</td>
<td>Command separator. Permits putting two or more commands on the same line.</td>
</tr>
<tr>
<td>;;</td>
<td>Terminator in a case option</td>
</tr>
<tr>
<td>.</td>
<td>“dot” command, equivalent to <code>source</code> (for bash only)</td>
</tr>
</tbody>
</table>
## Special Characters (2)

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$?</td>
<td>Exit status variable.</td>
</tr>
<tr>
<td>$$</td>
<td>Process ID variable.</td>
</tr>
<tr>
<td>[]</td>
<td>Test expression, eg. if condition</td>
</tr>
<tr>
<td>[[]]</td>
<td>Test expression, more flexible than [ ]</td>
</tr>
<tr>
<td>$[ ], $(())</td>
<td>Integer expansion</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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)
## Integer Arithmetic Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
</tr>
<tr>
<td>Subtraction</td>
<td>−</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>** (bash only)</td>
</tr>
<tr>
<td>Modulo</td>
<td>%</td>
</tr>
</tbody>
</table>
Integer Arithmetic Operations

- $((...))$ or $[...]$ commands
  - Addition: x = $((1+2))$
  - Multiplication: echo $[$x*$x$]
- let command: let c=$x + $x
- expr command: expr 10 / 2 (space required)
- C-style increment operators:
  - let c+=1 or let c--
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 btw decimal and binary numbers
  
  ```
  echo “ibase=10; obase=2; 10” | bc
  ```

- Call bc directly:
  
  ```
  bc <<< “scale=5; 2/5”
  ```
Outline

• Introduction to Linux Shell
• Shell Scripting Basics
• Beyond Basic Shell Scripting
  – Arithmetic Operations
  – Arrays
  – Flow Control
  – Command Line Arguments
  – Functions
• Advanced Text Processing Commands (grep, sed, awk)
Arrays Operations (1)

- Initialization
  ```
  declare -a my_array
  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]}
  ```
- 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 an array name to an array user
  • `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
• Beyond Basic Shell Scripting
  – Arithmetic Operations
  – Arrays
  – 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
  – Loops: for, while, until
  – Switches: case
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

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

• Strict spaces between condition and the brackets (bash)
## File Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>bash</th>
</tr>
</thead>
<tbody>
<tr>
<td>File exists</td>
<td>if [ -e test ]</td>
</tr>
<tr>
<td>File is a regular file</td>
<td>if [ -f test ]</td>
</tr>
<tr>
<td>File is a directory</td>
<td>if [ -d /home ]</td>
</tr>
<tr>
<td>File is not zero size</td>
<td>if [ -s test ]</td>
</tr>
<tr>
<td>File has read permission</td>
<td>if [ -r test ]</td>
</tr>
<tr>
<td>File has write permission</td>
<td>if [ -w test ]</td>
</tr>
<tr>
<td>File has execute permission</td>
<td>if [ -x test ]</td>
</tr>
</tbody>
</table>
## Integer Comparisons

<table>
<thead>
<tr>
<th>Operation</th>
<th>bash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal to</td>
<td>if [ 1 -eq 2 ]</td>
</tr>
<tr>
<td>Not equal to</td>
<td>if [ $a -ne $b ]</td>
</tr>
<tr>
<td>Greater than</td>
<td>if [ $a -gt $b ]</td>
</tr>
<tr>
<td>Greater than or equal to</td>
<td>if [ 1 -ge $b ]</td>
</tr>
<tr>
<td>Less than</td>
<td>if [ $a -lt 2 ]</td>
</tr>
<tr>
<td>Less than or equal to</td>
<td>if [ $a -le $b ]</td>
</tr>
</tbody>
</table>
## String Comparisons

<table>
<thead>
<tr>
<th>Operation</th>
<th>bash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equal to</td>
<td><code>if [ $a == $b ]</code></td>
</tr>
<tr>
<td>Not equal to</td>
<td><code>if [ $a != $b ]</code></td>
</tr>
<tr>
<td>Zero length or null</td>
<td><code>if [ -z $a ]</code></td>
</tr>
<tr>
<td>Non zero length</td>
<td><code>if [ -n $a ]</code></td>
</tr>
</tbody>
</table>

**Basic Shell Scripting**
## Logical Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>! (NOT)</td>
<td>if [ ! -e test ]</td>
</tr>
<tr>
<td>&amp;&amp; (AND)</td>
<td>if [ -f test ] &amp;&amp; [ -s test ]</td>
</tr>
<tr>
<td></td>
<td>if [[ -f test &amp;&amp; -s test ]]</td>
</tr>
<tr>
<td></td>
<td>if ( -e test &amp;&amp; ! -z test )</td>
</tr>
<tr>
<td></td>
<td>(OR)</td>
</tr>
<tr>
<td></td>
<td>if [[ -f test1</td>
</tr>
</tbody>
</table>
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 tests 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.

```bash
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)

```bash
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

```bash
#!/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.
- By default all variables are global.
- Modifying a variable in a function changes it in the whole script.
- Create a local variable using the `local` command, which is invisible outside the function:
  
  ```
  local var=value
  local varName
  ```
Functions example

#!/bin/bash
fun1(){
    local x_local=10
    x_global=100
}
x_global=10
echo "\nnglobal initial x_global = $x_global"
fun1
echo "\nlocal x_local = $x_local"
echo "\nnglobal final x_global = $x_global\n"

$sh fun1.sh
    global initial x_global = 10
    local x_local =
    global final x_global = 100
Pass Arguments to Bash Scripts

- All parameters can be passed at runtime and accessed via $1, $2, $3...
- $0: the shell script name
- $* or $@: all parameters passed to a function
- $#: number of positional parameters passed to the function
- $?: exist code of last cmd
- $$: PID of current process
- Array variable called FUNCNAME contains the names of all shell functions currently in the execution call stack.
Parameter example

#!/bin/bash
a=$1
b=$2
fun_mul(){
    fun_mul=$((a*b))
    echo ${FUNCNAME[0]}
}
echo "There are $# params $1 $2 passed in"
fun_mul
echo "\nProduct of $1 and $2 is $fun_mul\n"
echo "exit code= $? processID= $$ param= $$*
"

$ sh fun_param.sh 3 5
There are 2 params 3 5 passed in
fun_mul
Product of 3 and 5 is 15
exit code=0 processID=21459  param=3 5
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)
Advanced Text Processing Commands

- grep
- sed
- awk
grep & egrep

- **grep**: Unix utility that searches 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

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Title</th>
<th>Department</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Thomas</td>
<td>Manager</td>
<td>Sales</td>
<td>$5,000</td>
</tr>
<tr>
<td>200</td>
<td>Jason</td>
<td>Developer</td>
<td>Technology</td>
<td>$5,500</td>
</tr>
<tr>
<td>300</td>
<td>Raj</td>
<td>Sysadmin</td>
<td>Technology</td>
<td>$7,000</td>
</tr>
<tr>
<td>500</td>
<td>Randy</td>
<td>Manager</td>
<td>Sales</td>
<td>$6,000</td>
</tr>
</tbody>
</table>

- 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**

<table>
<thead>
<tr>
<th>Flags</th>
<th>Operation</th>
<th>Command</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-e</td>
<td>combine multiple commands</td>
<td>s</td>
<td>substitution</td>
</tr>
<tr>
<td>-f</td>
<td>read commands from file</td>
<td>g</td>
<td>global replacement</td>
</tr>
<tr>
<td>-h</td>
<td>print help info</td>
<td>p</td>
<td>print</td>
</tr>
<tr>
<td>-n</td>
<td>disable print</td>
<td>i</td>
<td>ignore case</td>
</tr>
<tr>
<td>-V</td>
<td>print version info</td>
<td>d</td>
<td>delete</td>
</tr>
<tr>
<td>-r</td>
<td>use extended regex</td>
<td>G</td>
<td>add newline</td>
</tr>
<tr>
<td></td>
<td></td>
<td>w</td>
<td>write to file</td>
</tr>
<tr>
<td></td>
<td></td>
<td>x</td>
<td>exchange pattern with hold buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>h</td>
<td>copy pattern to hold buffer</td>
</tr>
<tr>
<td></td>
<td></td>
<td>;</td>
<td>separate commands</td>
</tr>
</tbody>
</table>
sed Examples

#!/bin/bash

# My First Script

echo "Hello World!"
sed Examples (1)

• Add flag -e to carry out multiple matches.

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

• Alternate form

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

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

```bash
sed 's:/bin/bash:/bin/tcsh:g' hello.sh
```
```bash
#!/bin/tcsh
# My First Script
echo "Hello World!"
```
sed Examples (2)

- Delete blank lines from a file
  
  ```bash
  sed '/^$/d' hello.sh
  
  #!/bin/bash
  # My First Script
  echo "Hello World!"
  ```

- Delete line \n through \m in a file
  
  ```bash
  sed '2,4d' hello.sh
  
  #!/bin/bash
  echo "Hello World!"
  ```
sed Examples (3)

- Insert a blank line below every line matches pattern

```bash
sed '/First/G' hello.sh
#!/bin/bash
# My First Script
echo "Hello World!"
```

- Insert a blank line above and below every line matches pattern

```bash
sed '/First/{x;p;x;G}' hello.sh
#!/bin/bash
# My First Script
echo "Hello World!"
```
sed Examples (4)

- Replace-in-place with a backup file
  ```bash
  sed -i.bak '/First/Second/i' hello.sh
  ```

- echo with sed
  ```bash
  $ echo "shell scripting" | sed "s/[si]#/g"
  $ hellocriptng

  $ 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, use.
• `NR` #records (lines)
• `NF` #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

```bash
awk '/^#!\!/bin/\!/bash/{print $0, FILENAME}' *
```

```bash
#!/bin/bash Fun1.sh
#!/bin/bash fun_pam.sh
#!/bin/bash hello.sh
#!/bin/bash parm.sh
```

- Print extra lines below patterns

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

- **User Guides**
  - LSU HPC: [http://www.hpc.lsu.edu/docs/guides.php#hpc](http://www.hpc.lsu.edu/docs/guides.php#hpc)
  - LONI: [http://www.hpc.lsu.edu/docs/guides.php#loni](http://www.hpc.lsu.edu/docs/guides.php#loni)

- **Documentation**: [http://www.hpc.lsu.edu/docs](http://www.hpc.lsu.edu/docs)

- **Archived tutorials**: [http://www.hpc.lsu.edu/training/archive/tutorials.php](http://www.hpc.lsu.edu/training/archive/tutorials.php)

- **Contact us**
  - Email ticket system: [sys-help@loni.org](mailto:sys-help@loni.org)
  - Telephone Help Desk: 225-578-0900
Upcoming trainings

March 07, 2018: Hands-On Practice Session
March 14, 2018: Introduction to R
March 21, 2018: Parallel Computing with Matlab
April 04, 2018: Data Visualization in R
April 11, 2018: Introduction to Python
April 18, 2018: Deep Learning Software