Shell Scripting – Part 2

Le Yan/Alex Pacheco

HPC User Services @ LSU

Shell Scripting

- Part 1 (Feb 11th)
  - Simple topics such as creating and executing simple shell scripts, arithmetic operations, flow control, command line arguments and functions.
- Part 2 (today)
  - Advanced topics such as regular expressions and text processing tools (grep, sed, awk etc.)

Outline

- Regular Expressions (RegEx)
- grep
- sed
- awk
- File manipulation

Wildcards

- The Unix shell recognizes wildcards (a very limited form of regular expressions) used with filename substitution

  ? : match any single character.
  * : match zero or more characters.
  [ ] : match list of characters in the list specified
  [!] : match characters not in the list specified

```
ls *
cp [a-z]* lower_case/
cp [!a-z]* upper_digit/
```

Regular Expressions

- A regular expression (regex) is a extremely powerful method of using a pattern to describe a set of strings.
- Regular expressions enable strings that match a particular pattern within textual data records to be located and modified.
- Regular expressions are constructed analogously to arithmetic expressions by using various operators to combine smaller expressions
- Regular expressions are often used within utility programs that manipulate textual data.
  - Command line tools: `grep`, `egrep`, `sed`
  - Editors: `vi`, `emacs`
  - Languages: `awk`, `perl`, `python`, `php`, `ruby`, `tcl`, `java`, `javascript`, `.NET`

Bracket Expression

- `[ ]`
  - Matches a single character that is contained within the brackets. For example, `[abc]` matches "a", "b", or "c". `[a-z]` specifies a range which matches any lowercase letter from "a" to "z".
  - These forms can be mixed: `[abcx-z]` matches "a", "b", "c", "x", "y", or "z", as does `[a-cx-z]`.
- `[^ ]`
  - Matches a single character that is not contained within the brackets. For example, `[^abc]` matches any character other than "a", "b", or "c". `[^a-z]` matches any single character that is not a lowercase letter from "a" to "z".

Metacharacters

.	Any character except \n		
^	Start of a line		
$	End of a line		
\s	Any whitespace	\S	Any non-whitespace
\d	Any digit	\D	Any non-digit
\w	Any word	\W	Any non-word
\b	Any word boundary	\B	Anything except a word boundary

Special characters such as `.`, `^`, `$` need to be escaped by using "\" if the literals are to be matched

Regex Examples

Pattern	Matches	Does not match
`line`	`Feline animals` `Two linebackers`	`Three LINEs` `FeLine animals`
`^line`	`line 1 has eight characters` `linebackers`	`feline animals` `two lines of text`
`\bline\b`	`There are one line of text`	`Three lines of text` `The file has only one line.`

`l.ne`	`line`
	`lane`
	`l$ne`	

`\$\d\.\d\d`	`$3.88`	

Quantifiers

- Allow users to specify the number of occurrence

*	Zero or more
+	One or more
?	Zero or one
{n}	Exactly n times
{n,}	At least n times
{n,m}	At least n, but at most m times

Regex Examples

Pattern	Matches
`ca?t`	`ct` `cat`

`ca*t`	`ct`
	`cat`
	`caat`

`ca+t`	`cat`
	`caat`
	`caaat`

`ca{3}t`	`caaat`

`\d{3}-?\d{3}-?\d{4}`	`2255787844`
	`000-000-0000`
	`291-1938183`
	`573384-2333`

Alternation

- Use "|" to match either one thing or another
  - Example: `pork|chicken` will match `pork` or `chicken`

Grouping And Capturing Matches

- Use parentheses ( ) to
  - Group atoms into larger unit
  - Capture matches for later use

Pattern	Matches

`(ca?t){2,3}`	`ctct`
	`ctctct`
	`catcat`
	`catcatcat`

`(\d*\.)?\d+`	Integers or decimals

A Great Learning Source

- http://www.regexr.com/

Outline

- Regular Expressions (RegEx)
- grep
- sed
- awk
- File manipulation

grep

- `grep` is a Unix utility that searches through either information piped to it or files.
- `egrep` is extended grep (extended regular expressions), same as `grep -E`
- Use `zgrep` for compressed files.
- Usage: `grep <options> <search pattern> <files>`
- Commonly used 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 (1)

- Search files that contain the word node in the examples directory

```
egrep node *

checknodes.pbs:#PBS -o nodetest.out
checknodes.pbs:#PBS -e nodetest.err
checknodes.pbs:for nodes in "${NODES[@]}"; do
checknodes.pbs: ssh -n $nodes 'echo $HOSTNAME '$i' ' &
checknodes.pbs:echo "Get Hostnames for all unique nodes"
```

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

```
egrep -in node *

checknodes.pbs:20:NODES=(`cat "$PBS_NODEFILE"` )
checknodes.pbs:21:UNODES=(`uniq "$PBS_NODEFILE"` )
checknodes.pbs:23:echo "Nodes Available: " ${NODES[@]}
checknodes.pbs:24:echo "Unique Nodes Available: " ${UNODES[@]}
checknodes.pbs:28:for nodes in "${NODES[@]}"; do
checknodes.pbs:29: ssh -n $nodes 'echo $HOSTNAME '$i' ' &
checknodes.pbs:34:echo "Get Hostnames for all unique nodes"
checknodes.pbs:39: ssh -n ${UNODES[$i]} 'echo $HOSTNAME '$i' '
```

grep Examples (2)

- Print files that contain the word "counter”

```
egrep -l counter *

factorial2.sh
factorial.csh
factorial.sh
```

- List all files that contain a comment line i.e. lines that begin with "#“

```
egrep -l “^#” *

backups.sh
checknodes.pbs
dooper1.sh
dooper.csh
dooper.sh
factorial2.sh
factorial3.sh
factorial.csh
factorial.sh
hello.sh
name.csh
name.sh
```

grep Examples (3)

- List all files that are bash or csh scripts i.e. contain a line that end in bash or csh

```
egrep -l "bash$|csh$" *

backups.sh
checknodes.pbs
dooper1.sh
dooper.csh
dooper.sh
factorial2.sh
factorial3.sh
factorial.csh
factorial.sh
hello.sh
name.csh
name.sh
nestedloops.csh
nestedloops.sh
quotes.csh
quotes.sh
shift10.sh
```

Outline

- Regular Expressions (RegEx)
- grep
- sed
- awk
- File manipulation

sed

- sed ("stream editor") is Unix utility for parsing and transforming text files.
  - Also works for either information piped to it or files
- sed is line-oriented - it operates one line at a time and allows regular expression matching and substitution.
- sed has several commands, the most commonly used command and sometime the only one learned is the substituion command, *s*

```
echo day | sed 's/day/night/'

day
```

List of 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 (1)

- Double space a file

```
sed G hello.sh

#!/bin/bash


# My First Script


echo "Hello World!"
```

- Triple space a file sed ’G;G’

sed Examples (2)

- Add the -e to carry out multiple matches.

```
cat hello.sh | sed -e 's/bash/tcsh/g' -e 's/First/First tcsh/g'

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

- Alternate form

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

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

- The default delimiter is slash (/), but you can change it to whatever you want which is useful when you want to replace path names

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

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

sed Examples (3)

- If you do not use an alternate delimiter, use backslash (\) to escape the slash character in your pattern

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

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

- If you enter all your `sed` commands in a file, say `sedscript`, you can use the `-f` flag to `sed` to read those commands

```
cat sedscript

s/bash/tcsh/g

sed -f sedscript hello.sh

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

sed Examples (4)

- sed can also 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!"
```

- Insert a blank line above every line which matches *pattern*

```
sed '/First/{x;p;x}' hello.sh

#!/bin/bash


# My First Script

echo "Hello World!"
```

sed Examples (5)

- Insert a blank line below every line which matches *pattern*

```
sed '/First/G' hello.sh

#!/bin/bash

# My First Script


echo "Hello World!"
```

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

```
sed '/First/{x;p;x;G}' hello.sh

#!/bin/bash


# My First Script


echo "Hello World!"
```

sed Examples (6)

- Print only lines which match *pattern* (emulates `grep`)

```
sed -n '/echo/p' hello.sh

echo "Hello World!"
```

- Print only lines which do NOT match *pattern* (emulates `grep -v`)

```
sed -n '/echo/!p' hello.sh

#!/bin/bash
# My First Script
```

- Print current line number to standard output

```
sed -n '/echo/ =' quotes.sh

5
6
7
8
9
10
11
12
13
```

sed example (7)

- If you want to make substitution in place, i.e. in the file, then use the `-i` command. If you append a suffix to `-i`, then the original file will be backed up as *`filename.suffix`*.

```
cat hello1.sh

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

sed -i.bak -e 's/bash/tcsh/g' -e 's/First/First tcsh/g' hello1.sh

cat hello1.sh

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

cat hello1.sh.bak

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

sed Examples (8)

- Print section of file between *pattern1* and *pattern2*

```
cat nh3-drc.out | sed -n '/START OF DRC CALCULATION/,/END OF ONEELECTRON INTEGRALS/p'

START OF DRC CALCULATION
************************
------------------------------------------------------------------------------
TIME MODE Q P KINETIC POTENTIAL TOTAL
FS BOHR*SQRT(AMU) BOHR*SQRT(AMU)/FS E ENERGY ENERGY
0.0000 L 1 1.007997 0.052824 0.00159 -56.52247 -56.52087
L 2 0.000000 0.000000
L 3 -0.000004 0.000000
L 4 0.000000 0.000000
L 5 0.000005 0.000001
L 6 -0.138966 -0.014065
------------------------------------------------------------------------------
CARTESIAN COORDINATES (BOHR) VELOCITY (BOHR/FS)
------------------------------------------------------------------------------
7.0 0.00000 0.00000 0.00000 0.00000 0.00000 -0.00616
1.0 -0.92275 1.59824 0.00000 0.00000 0.00000 0.02851
1.0 -0.92275 -1.59824 0.00000 0.00000 0.00000 0.02851
1.0 1.84549 0.00000 0.00000 0.00000 0.00000 0.02851
------------------------------------------------------------------------------
----------------------
GRADIENT OF THE ENERGY
----------------------
UNITS ARE HARTREE/BOHR E'X E'Y E'Z
1 NITROGEN 0.000042455 0.000000188 0.000000000
2 HYDROGEN 0.012826176 -0.022240529 0.000000000
3 HYDROGEN 0.012826249 0.022240446 0.000000000
4 HYDROGEN -0.025694880 -0.000000105 0.000000000
...... END OF ONE-ELECTRON INTEGRALS ......
```

sed Examples (9)

- Print section of file from *pattern* to end of file

```
cat h2o-opt-freq.nwo | sed -n '/CITATION/,$p'

CITATION
--------
Please use the following citation when publishing results
obtained with NWChem:
E. J. Bylaska, W. A. de Jong, N. Govind, K. Kowalski, T. P. Straatsma,
M. Valiev, D. Wang, E. Apra, T. L. Windus, J. Hammond, P. Nichols,
S. Hirata, M. T. Hackler, Y. Zhao, P.-D. Fan, R. J. Harrison,
M. Dupuis, D. M. A. Smith, J. Nieplocha, V. Tipparaju, M. Krishnan,
Q. Wu, T. Van Voorhis, A. A. Auer, M. Nooijen,
E. Brown, G. Cisneros, G. I. Fann, H. Fruchtl, J. Garza, K. Hirao,
R. Kendall, J. A. Nichols, K. Tsemekhman, K. Wolinski, J. Anchell,
D. Bernholdt, P. Borowski, T. Clark, D. Clerc, H. Dachsel, M. Deegan,
K. Dyall, D. Elwood, E. Glendening, M. Gutowski, A. Hess, J. Jaffe,
B. Johnson, J. Ju, R. Kobayashi, R. Kutteh, Z. Lin, R. Littlefield,
X. Long, B. Meng, T. Nakajima, S. Niu, L. Pollack, M. Rosing,
G. Sandrone, M. Stave, H. Taylor, G. Thomas, J. van Lenthe, A. Wong,
and Z. Zhang,
"NWChem, A Computational Chemistry Package for Parallel Computers,
Version 5.1" (2007),
Pacific Northwest National Laboratory,
Richland, Washington 99352-0999, USA.
Total times cpu: 3.4s wall: 18.5s
```

Grouping And Capturing Matches

- Use parentheses ( ) to
  - Group atoms into larger unit
  - Capture matches for later use
    - & represent the matched *pattern*
    - "\1" is the first remembered *pattern*, and "\2" is the second remembered *pattern*, and so on
    - sed can remember up to 9 patterns

```
echo abcd123 | sed –r 's/([a-z]*)[^a-z]*/\1/'

abcd
```

sed One-liners

- sed one-liners:
  http://sed.sourceforge.net/sed1line.txt

Outline

- Regular Expressions (RegEx)
- grep
- sed
- awk
- File manipulation

awk

- 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.
- awk comes in three variations
  - awk : Original AWK by A. Aho, B. W. Kernighnan and P. Weinberger from AT&T
  - nawk : New AWK, also from AT&T
  - gawk : GNU AWK, all Linux distributions come with gawk. In some distros, awk is a symbolic link to gawk.

awk Syntax

- Simplest form of using `awk`
  - `awk pattern {action}`
    - `pattern` decides when `action` is performed
  - Most common action: `print`
  - Print file dosum.sh: `awk '{print $0}' dosum.sh`
  - Print line matching bash in all `.sh` files in current directory: `awk '/bash/{print $0}' *.sh`

awk Patterns

`BEGIN`	special pattern which is not tested against input. Action will be performed before reading input.
`END`	special pattern which is not tested against input. Action will be performed after reading all input.
`/regular expression/`	the associated regular expression is matched to each input line that is read
`relational expression`	used with the if, while relational operators
`&&`	logical AND operator used as pattern1 && pattern2. Execute action if pattern1 and pattern2 are true
`||`	logical OR operator used as pattern1 || pattern2. Execute action if either pattern1 or pattern2 is true
`!`	logical NOT operator used as !pattern. Execute action if pattern is not matched
`?:`	Used as pattern1 ? pattern2 : pattern3. If pattern1 is true use pattern2 for testing else use pattern3
`pattern1, pattern2`	Range pattern, match all records starting with record that matches pattern1 continuing until a record has been reached that matches pattern2

Awk Examples

- Print list of files that are csh script files

```
awk '/^#\!\/bin\/tcsh/{print FILENAME}' *

dooper.csh
factorial.csh
hello1.sh
name.csh
nestedloops.csh
quotes.csh
shift.csh
```

- Print contents of hello.sh that lie between two patterns

```
awk '/^#\!\/bin\/bash/,/echo/{print $0}' hello.sh

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

How awk Works

- 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. The delimiter can be changed as will be seen in the next few slides.
- To print the entire line, use `$0`.
- The intrinsic variable NR contains the number of records (lines) read.
- The intrinsic variable NF contains the number of fields or columns in the current line.

Changing Field Delimiter

- By default the field delimiter is space or tab. To change the field delimiter use the `-F<delimiter>` command.

```
uptime

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

uptime | awk '{print $1,NF}'

11:19am 0.17

uptime | awk -F: '{print $1,NF}'

11 0.12, 0.10, 0.16

for i in $(seq 1 10); do touch file${i}.dat ; done
ls file*

file10.dat file2.dat file4.dat file6.dat file8.dat
file1.dat file3.dat file5.dat file7.dat file9.dat

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

file10 dat file10.dat
file1 dat file1.dat
file2 dat file2.dat
file3 dat file3.dat
file4 dat file4.dat
file5 dat file5.dat
file6 dat file6.dat
file7 dat file7.dat
file8 dat file8.dat
file9 dat file9.dat
```

Formatting Output (1)

- Printing an expression is the most common action in the `awk` statement. If formatted output is required, use the `printf` format.
- The `print` command puts an explicit newline character at the end while the `printf` command does not.

```
echo hello 0.2485 5 | awk '{printf "%s %f %d %05d\n",$1,$2,$3,$3}'

hello 0.248500 5 00005
```

Formatting Output (2)

- Format specifiers are similar to the C-programming language

%d,%i	decimal number
%e,%E	floating point number of the form [-]d.dddddd.e[]dd. The %E format uses E instead of e.
%f	floating point number of the form [-]ddd.dddddd
%g,%G	Use %e or %f conversion with nonsignificant zeros truncated. The %G format uses %E instead of %e
%s	character string

Formatting Output (3)

- Format specifiers have additional parameter which may lie between the % and the control Letter

`0`	A leading 0 (zero) acts as a flag, that indicates output should be padded with zeroes instead of spaces.
`width`	The field should be padded to this width. The field is normally padded with spaces. If the 0 flag has been used, it is padded with zeroes.
`.prec`	A number that specifies the precision to use when printing.

- String constants supported by awk

`\\`	Literal backslash
`\n`	newline
`\r`	carriage-return
`\t`	horizontal tab
`\v`	vertical tab

Arithmetic Operations (1)

- awk has in-built support for arithmetic operations

Operator	Operation	Operator	Operation
+	Addition	++	Autoincrement
-	Subtraction	--	Autodecrement
*	Multiplication	+=	Add to
/	Division	-=	Subtract from
**	Exponentiation	*=	Multiple with
%	Modulo	/=	Divide by

```
echo | awk '{print 10%3}'
1
echo | awk '{a=10;print a/=5}'
2
```

Arithmetic Operations (2)

- awk also supports trignometric functions such as `sin(expr)` and `cos(expr)` where `expr` is in radians and `atan2(y/x)` where `y/x` is in radians
- Other Arithmetic operations supported are
  - `exp(expr)` : The exponential function
  - `int(expr)` : Truncates to an integer
  - `log(expr)` : The natural Logarithm function
  - `sqrt(expr)` : The square root function
  - `rand()` : Returns a random number N between 0 and 1 such that `0 N < 1`
  - `srand(expr)` : Uses expr as a new seed for random number generator. If expr is not provided, time of day is used.

awk Variables

- Like all progamming languages, awk supports the use of variables. Like shell, variable types do not have to be defined.
- awk variables can be user defined or could be one of the columns of the file being processed.
- Unlike Shell, awk variables are referenced as is i.e. no $ prepended to variable name (except for the special variables such as $1, $2 etc).

```
awk '{print $1}' hello.sh

#!/bin/bash
#
echo

awk '{col=$1;print col,$2}' hello.sh

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

Conditionals and Loops (1)

- awk supports
  - `if ... else if .. else` conditionals.
  - `while` and `for` loops
- They work similar to that in C-programming
- Supported operators: ==, !=, >, >=, <, <=, ~ (string matches), !~ (string does not match)

```
awk '{if (NR > 0 ){print NR,":", $0}}' hello.sh

1 : #!/bin/bash
2 :
3 : # My First Script
4 :
5 : echo "Hello World!"
```

Conditionals and Loops (2)

- The `for` command can be used for processing the various columns of each line

```
cat << EOF | awk '{for (i=1;i<=NF;i++){if (i==1){a=$i}else if (i==NF){print a}else{a+=$i}}}'

1 2 3 4 5 6
7 8 9 10
EOF
15
24

echo $(seq 1 10) | awk 'BEGIN{a=6}{for (i=1;i<=NF;i++){a+=$i}}END {print a}'

61
```

awk One-liners

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

The awk Programming Language

- awk can also be used as a programming language.
- The first line in awk scripts is the shebang line (#!) which indicates the location of the awk binary.
- To support scripting, awk has several built-in variables, which can also be used in one line commands
  - ARGC : number of command line arguments
  - ARGV : array of command line arguments
  - FILENAME : name of current input file
  - FS : field separator
  - OFS : output field separator
  - ORS : output record separator, default is newline
- awk permits the use of arrays
- awk has built-in functions to aid writing of scripts
- GNU awk also supports user defined function

Outline

- Regular Expressions (RegEx)
- grep
- sed
- awk
- File manipulation

Cut

- Linux command `cut` is used for text processing to extract portion of text from a file by selecting columns.
- Usage: `cut <options> <filename>`
- Common Options:
  - `-c list` : The `list` specifies character positions.
  - `-f list` : select only these fields.
  - `-d delim` : Use `delim` as the field delimiter character instead of the tab character.
- List is made up of one range, or many ranges separated by commas
  - N : N^{th} byte, character or field. count begins from 1
  - N- : N^{th} byte, character or field to end of line
  - N-M : N^{th} to M^{th} (included) byte, character or field
  - -M : from first to M^{th} (included) byte, character or field

Cut Example

```
uptime

14:17pm up 14 days 3:39, 5 users, load average: 0.51, 0.22, 0.20

uptime | cut -c-8

14:17pm

uptime | cut -c14-20

14 days

uptime | cut -d":" -f4

0.41, 0.22, 0.20
```

Paste

- The `paste` utility concatenates the corresponding lines of the given input files, replacing all but the last file's newline characters with a single tab character, and writes the resulting lines to standard output.
- If end-of-file is reached on an input file while other input files still contain data, the file is treated as if it were an endless source of empty lines.
- Usage: `paste <option> <files>`
- Common Options
  - `-d` delimiters specifies a list of delimiters to be used instead of tabs for separating consecutive values on a single line. Each delimiter is used in turn; when the list has been exhausted, paste begins again at the first delimiter.
  - `-s` causes paste to append the data in serial rather than in parallel; that is, in a horizontal rather than vertical fashion.

Paste Example

```
> cat names.txt
Mark Smith
Bobby Brown
Sue Miller
Jenny Igotit

> cat numbers.txt
555-1234
555-9876
555-6743
867-5309

> paste names.txt numbers.txt
Mark Smith 555-1234
Bobby Brown 555-9876
Sue Miller 555-6743
Jenny Igotit 867-5309
```

Join (1)

- `join` is a command in Unix-like operating systems that merges the lines of two sorted text files based on the presence of a common field.
- The `join` command takes as input two text files and a number of options.
- If no command-line argument is given, this command looks for a pair of lines from the two files having the same first field (a sequence of characters that are different from space), and outputs a line composed of the first field followed by the rest of the two lines.
- The program arguments specify which character to be used in place of space to separate the fields of the line, which field to use when looking for matching lines, and whether to output lines that do not match. The output can be stored to another file rather than printing using redirection.
- Usage: `join <options> <FILE1> <FILE2>`

Join (2)

- Common options:
  - `-a FILENUM` : also print unpairable lines from file `FILENUM`, where `FILENUM` is 1 or 2, corresponding to FILE1 or FILE2
  - `-e EMPTY` : replace missing input fields with EMPTY
  - `-i` : ignore differences in case when comparing fields
  - `-1 FIELD` : join on this `FIELD` of file 1
  - `-2 FIELD` : join on this `FIELD` of file 2
  - `-j FIELD` : equivalent to `'-1 FIELD -2 FIELD'`
  - `-t CHAR` : use CHAR as input and output field separator

```
cat file1

george jim
mary john

cat file2

albert martha
george sophie

join file1 file2

george jim sophie
```

Split

- `split` is a Unix utility most commonly used to split a file into two or more smaller files.
- Usage: `split <options> <file to be split> <name>`
- Common Options:
  - `-a suffix_length` : Use `suffix_length` letters to form the suffix of the file name.
  - `-b byte_count[k|m]` : Create smaller files `byte_count` bytes in length. If "`k`" is appended to the number, the file is split into `byte_count` kilobyte pieces. If "`m`" is appended to the number, the file is split into `byte_count` megabyte pieces.
  - `-l n` : Create smaller files n lines in length.
  - `-d`: Use numerical suffix instead of alphabetic
- The default behavior of `split` is to generate output files of a fixed size, default 1000 lines.
- The files are named by appending aa, ab, ac, etc. to output filename.
- If output filename (<name>) is not given, the default filename of x is used, for example, `xaa`, `xab`, etc

csplit

- The `csplit` command in Unix is a utility that is used to split a file into two or more smaller files determined by context lines.
- Usage: `csplit <options> <file> <args>`
- Common Options:
  - `-f prefix` : Give created files names beginning with prefix. The default is "xx".
  - `-k` : Do not remove output files on errors.
  - `-s` : Do not write the size of each output file to standard output as it is created.
  - `-n number` : Use number of decimal digits after the prefix to form the file name. The default is 2.
- The args operands may be a combination of the following patterns:
  - `/regexp/[[+|-]offset]` : Create a file containing the input from the current line to (but not including) the next line matching the given basic regular expression. An optional offset from the line that matched may be specified.
  - `%regexp%[[+|-]offset]` : Same as above but a file is not created for the output.
  - `line_no` : Create containing the input from the current line to (but not including) the specified line number.
  - `{num}` : Repeat the previous pattern the specified number of times. If it follows a line number pattern, a new file will be created for each line_no lines, num times. The first line of the file is line number 1 for historic reasons.

Split and csplit Examples

- Example: Run a multi-step job using Gaussian 09, for example, geometry optimization followed by frequency analysis of water molecule.
- Problem: Some visualization packages like molden cannot visualize such multi-step jobs. Each job needs to visualized separately.
- Solution: Split the single output file into two files, one for the optimization calculation and the other for frequency calculation.
- Source Files see
  - `script/day2/csplit/h2oopt-freq.log` on Philip
- Example: `split -l 1442 h2o-opt-freq.log`
- Example: `csplit h2o-opt-freq.log "/Normal termination of Gaussian 09/+1"`

Further Reading

- 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

Next Tutorial – Numerical Libraries

- If the following fits you, then you might want come
  - I am developing a code targeted for HPC platforms
  - I want my code to run fast
  - I do not want to reinvent the wheels
- Date: March 11th, 2015

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
- Online courses: http://moodle.hpc.lsu.edu
- Contact us
  - Email ticket system: sys-help@loni.org
  - Telephone Help Desk: 225-578-0900
  - Instant Messenger (AIM, Yahoo Messenger, Google Talk)
    - Add “lsuhpchelp”

Questions?