Seneca | SCHOOL OF INFORMATION AND COMMUNICATIONS TECHNOLOGY

Unix Scripting

Lecturer: Shahdad Shariatmadari

June 2020

What we have learned…

- Introduction to Shell Scripting
  - Categories of variables
  - Conditional Statements
  - Loops
- stdin, stdout, stderr Redirection and piping
- File descriptor
- Filtering
  - Simple filter commands: head, tail, cut, sort, wc
  - grep utility
- Multiple commands : () vs {}

Agenda

- Regular Expressions
  - Grep
  - Sed
  - Awk

What are Regular Expressions?

- Regular expressions are special characters which help search data, matching complex patterns.
- If you recall, grep helps us to search for string in a file
  - `grep friend poem.txt`
  - It will match any string having friend as substring in it, like *friends*, *friendly*, *friendship*
- Sometimes we want to search for strings that follow a complex ***pattern*** instead of specific string
  - Like postal codes, email address

Regular Expressions

- In order to work with patterns, as opposed to specific strings, ***grep*** uses ***regular expressions***
- Regular expression (regex), is a sequence of characters (**literal** and **special**) that defines a search ***pattern***.
  - Regular expressions are shortened as 'regexp' or 'regex'.
- Some of the commonly used commands with Regular expressions are `awk, sed` and `grep`.

Regular Expressions

- With grep, you can use them to search for patterns.
  - Syntax: `grep "REGEX" filename`

Regular expression characters

Symbol	Descriptions
.	replaces any character
^	matches start of string
$	matches end of string
*	matches up zero or more times the preceding character
\	Represent special characters
()	Groups regular expressions
?	Matches up exactly one character

Regular expression characters

Symbol	Descriptions
[0-9]	Range of digits between 0 to 9
[a-z]	Range of alphabet from a to z
[A-Z]	Range of alphabet from A to Z

Square brackets

- [] Brackets define a character class that will match any single character within the brackets
  - e.g [0-9]
  - `grep [Ll]inux testfile`
    - would match both ***Linux*** and ***linux***
- Hyphen can be used for defining a range of characters
  - e.g [a-z0-9]
- The caret sign at the beginning of the list means exclusion
  - ([^a] = not a)

Example

- grep “[0-9]” test.txt
  - It returns lines with any digits
- grep “[A-Z]” test.txt
  - It returns lines with any alphabet A~Z

Matching any character: Period

- The period operator "**.** " matches any single character except a newline
- Example:
  - `grep c.t testfile`
    - would match cat, cet, cut, but not can or ct

Example

- grep “s.k” test.txt
  - It returns stack, snack, smack
- grep “s[nm].k” test.txt
  - It returns snack, smack

Using Anchors: ^ (caret) and $

- To constrain our search space to the beginning or end of lines or words, we can use **anchors**
- Anchors are special characters which specify where in line a match must occur to be valid
  - ^ matches the pattern at the beginning of line
  - $ matches the pattern at the end of line
- Examples
  - `grep ^hello testfile`
    - will find the lines that start with hello
  - `grep sky$ testfile`
    - will find lines that end with the word sky
  - `grep ^$ testfile`
    - Find blank lines

Grep : Regular expression

- `grep a data.txt`
  - Search for content containing letter ‘a’ in the data.txt
- `grep ^a data.txt`
  - Search for content that STARTS with ‘a’
  - '**^**' matches the start of a string.
- `grep a$ data.txt`
  - Search for content that ENDS with ‘a’
  - ‘**$**' matches the end of a string.

Repeating pattern Zero or more time

- * An asterisk matches zero or multiple occurrences of the part of regular expression directly preceding it
  - `grep "ab*c" testfile`
    - would match abc, abbc, abbbc, ac, aac, and abcc
    - * control the number of occurrences of nearest character, "b". So, "b*" means "b" can be repeated 0 or more
- If the preceding character is the period operator, the star operator matches anything
  - `grep ".*" testfile`

Special characters

- What if you want to find all lines containing the dollar character '$'
- The solution is to 'escape' the symbol, so you would use
  - `grep '\$' a_file`
  - Don’t use grep –E with this expression

Using Anchors: \< and \>

- To constrain our search space to the beginning or end of words, we can use the following anchors:
  - \<
  - \>
- **grep "\<free" testfile**
  - matches the words beginning with free, like freedom
- **grep "ick\>" testfile**
  - matches the words ending with ick like brick, click

Activity:Create Sample.txt file with following data

- This is that
- One two three
- Leader and leadership
- Led and leder
- Test 1
- 123
- Best circle
- `this is back`
- `zap` is my new tool
- People 123 is here
- Here is the code of ‘C$’
- 1230$

- Write some grep command to:
  - Search for “123”
  - Search for “L”
  - Search for $

Question

- How to search for those lines which contain ‘`’ (back-tick character)
- How to search for those lines which contain single quote character?

Using awk

- The `awk` program can be a handy text manipulator – performing jobs such as parsing, filtering, and easy formatting of text.
- A text pattern scanning and processing language, created by **A**ho, **W**einberger & **K**ernighan (hence the name).

Typical use of Awk

- Text processing,
- Producing formatted text reports,
- Performing arithmetic operations,
- Performing string operations, and many more

Awk

- AWK sees each line as being made up of a number of fields, each being separated by a 'field separator‘ (by default, it is space).
- Within awk, the first field is referred to as $1, the second as $2, etc. and the whole line is called $0.

General syntax

- `awk 'pattern {action}' filename`
- **Patterns:**
  - You can use a regular expression, enclosed within slashes, as a pattern.
  - The ~ operator tests whether a field or variable matches a regular expression
  - The !~ operator tests for no match.
  - You can perform both numeric and string comparisons using relational operators
  - You can combine any of the patterns using the Boolean operators || (OR) and && (AND).

awk - Actions

- The action portion of an awk command causes awk to take that action when it matches a pattern.
- When you do not specify an action, awk performs the default action, which is the print command (explicitly represented as **{print}**). This action copies the record (normally a line) from the input to standard output.
- When you follow a print command with arguments, awk displays only the arguments you specify.
  - These arguments can be variables or string constants.
- Unless you separate items in a print command with commas, awk concatenates them.
  - Commas cause awk to separate the items with the output field separator
- You can include several actions on one line by separating them with semicolons.

awk - Variables

- In addition to supporting user variables, awk maintains program variables.
- You can use both user and program variables in the **pattern** and **action** portions of an awk command.

Variable	**Meaning**
$0	The current record (as a single variable)
$1–$n	Fields in the current record
FILENAME	Name of the current input file (null for standard input)
FS	Input field separator (default: SPACE or TAB)
NF	Number of fields in the current record
NR	Record number of the current record
OFS	Output field separator (default: SPACE)
ORS	Output record separator (default: NEWLINE)
RS	Input record separator (default: NEWLINE)

awk – Example 1

- Because the pattern is missing, awk selects all lines of input.
- When used without any arguments the print command displays each selected line in its entirety.
- This command copies the input to standard output.

$ awk '{ print }' cars

```
plym    fury    77    73    2500
chevy   nova    79    60    3000
ford    mustang 65    45    17000
volvo   gl      78    102   9850
ford    ltd     83    15    10500
Chevy   nova    80    50    3500
fiat    600     65    115   450
honda   accord  81    30    6000
ford    thundbd 84    10    17000
toyota  tercel  82    180   750
chevy   impala  65    85    1550
ford    bronco  83    25    9525
```

awk – Example 2

- This example has a pattern but no explicit action.
- The slashes indicate that chevy is a regular expression.
- In this case awk selects from the input just those lines that contain the string chevy (lowercase).
- When you do not specify an action, awk assumes the action is print. The following command copies to standard output all lines from the input that contain the string chevy:

```
$ awk '/chevy/' cars
chevy   nova    79      60      3000
chevy   impala  65      85      1550
```

awk – Example 3

- These two examples select all lines from the file (they have no pattern).
- The braces enclose the action; you must always use braces to delimit the action so awk can distinguish it from a pattern.
- These examples display the third field ($3), and the first field ($1) of each selected line, with and without a separating space:

```
$ awk '{print $3, $1}' cars
77 plym
79 chevy
65 ford
78 Volvo
...
```

```
$ awk '{print $3 $1}' cars
77plym
79chevy
65ford
78Volvo
...
```

awk – Example 4

- This example, which includes both a pattern and an action, selects all lines that contain the string chevy and displays the third and first fields from the selected lines:

```
$ awk '/chevy/ {print $3, $1}' cars
79 chevy
65 chevy
```

awk – Example 5

- This example uses the matches operator (~) to select all lines that contain the letter h in the first field ($1), and because there is no explicit action, awk displays all the lines it selects.

```
$ awk '$1 ~ /h/' cars
chevy   nova    79      60      3000
Chevy   nova    80      50      3500
honda   accord  81      30      6000
chevy   impala  65      85      1550
```

awk – Example 6

- The caret (^) in a regular expression forces a match at the beginning of the line or, in this case, at the beginning of the first field, and because there is no explicit action, awk displays all the lines it selects.

```
$ awk '$1 ~ /^h/' cars
honda   accord  81      30      6000
```

awk – Example 7

- This example shows three roles a dollar sign can play within awk.
  - First, a dollar sign followed by a number identifies a field ($3).
  - Second, within a regular expression a dollar sign forces a match at the end of a line or field (5$).
  - Third, within a string a dollar sign represents itself.

```
$ awk '$3 ~ /5$/ {print $3, $1, "$" $5}' cars
65 ford $17000
65 fiat $450
65 chevy $1550
```

awk – Example 8

- Square brackets surround a character class definition.
- In this example, awk selects lines that have a second field that begins with t or m and displays the third and second fields, a dollar sign, and the fifth field.
- Because there is no comma between the “$” and the $5, awk does not put a SPACE between them in the output.

```
$ awk '$2 ~ /^[tm]/ {print $3, $2, "$" $5}' cars
65 mustang $17000
84 thundbd $17000
82 tercel $750
```

awk – Examples 9 & 10

- The equal-to relational operator (==) causes awk to perform a numeric comparison between the third field in each line and the number 83.
- This awk command takes the default action, print, on each line where the comparison is successful.

```
$ awk '$3 == 83' cars
ford    ltd      83      15      10500
ford    bronco   83      25      9525
```

- The next example finds all cars priced (5th field) at or less than $3,000.

```
$ awk '$5 <= 3000' cars
plym    fury     77      73      2500
chevy   nova     79      60      3000
fiat    600      65      115     450
toyota  tercel   82      180     750
chevy   impala   65      85      1550
```

awk – Example 11

- When both sides of a comparison operator are numeric, awk defaults to a numeric comparison. To force a string comparison, double quotes can be used.
- The following examples illustrate the effect of using double quotes.

```
$ awk '"2000" <= $5 && $5 < 9000' cars
plym    fury    77     73     2500
chevy   nova    79     60     3000
Chevy   nova    80     50     3500
fiat    600     65     115    450
honda   accord  81     30     6000
toyota  tercel  82     180    750

$ awk '2000 <= $5 && $5 < 9000' cars
plym    fury    77     73     2500
chevy   nova    79     60     3000
Chevy   nova    80     50     3500
honda   accord  81     30     6000
```

- **Notice that, for example, "2000" is less then "450" as a string, but 2000 is NOT less than 450 as a number**

Activity

- In the /etc/passwd, find all lines which have ‘9’ in their third field
- You noticed that in /etc/passwd file the “:” is the separator. So, you can tell AWK what to consider as separator using “-F” option:
  - Awk –F ”:” ‘{ print $1 } ‘ /etc/passwd

What does the following code do?

- `Awk –F”:” ‘ NR==1, NR==10 { print $1 } ‘ /etc/passwd`
  - ***NR*** *is used to display specific line of data*

Activity

- Download the file from the BB->courseDocument->week4->sampleFile and copy it into the matrix:
  - scp file.txt your account@matrix.senecac.on.ca:file.txt
- Try to practice some of the examples (3 of them) in previous slides

Advanced AWK (*for reading*)

- AWK provides support for loops (both 'for' and 'while') and for branching (using 'if').
- AWK can have an optional `BEGIN{}` section of commands that are done before processing any content of the file, then the main {} section works on each line of the file, and finally there is an optional `END{}` section of actions that happen after the file reading has finished:
  - `BEGIN { … initialization awk commands …}`
  - **`{awk commands for each line of the file}`**
  - `END { …. finalization awk commands …}`
- **Example:**
  - `awk 'BEGIN {print "REPORT TITLE"} $1 ~ /re/ {execution} END {print "END OF REPORT"}' filename`

Sed (Stream Editor)

- The stream editor `SED` is a useful text parsing and manipulation utility handy for doing transformations on files or streams of data.
- `SED` performs basic text transformations on an input stream (a file or input from a pipeline) in a
- single pass through the stream, so it is very efficient. However, it is sed's ability to filter text in a pipeline which particularly distinguishes it from other types of editor.

sed

- **Syntax:**

  `sed [-n] 'address instruction' filename`

- **address**
  - can use a line number, to select a specific line (for example: 5)
  - can specify a range of line numbers (for example: 5,7)
  - can specify a regular expression to select all lines that match (e.g /^happy[0-9]/)
    - Note: when using regular expressions, you must delimit them with a forward-slash"/"
  - default address (if none is specified) will match every line

- **instruction**
  - p - print line(s) that match the address (usually used with -n option)
  - d - delete line(s) that match the address
  - q - quit processing at the first line that matches the address
  - s - substitute text to replace a matched regular expression, similar to vi substitution

Examples

- `$ sed '/chevy/ p' cars`
- `$ sed -n '3,6 p' cars`
- `$ sed '5 q' cars`
- `$ sed '/[0-9][0-9][0-9]$/ q' cars`

Example

- `sed -n '3,6 p' cars` - display only lines 3 through 6
- `sed '5 d' cars` - display all lines except the 5th
- `sed '5,8 d' cars` - display all lines except the 5th through 8th
- `sed '5 q' cars` - display first 5 lines then quit, same as `head -5 cars`
- `sed -n '/chevy/ p' cars` - display only matching lines, same as `grep 'chevy' cars`
- `sed '/chevy/ d' cars` - delete all matching lines, same as `grep -v 'chevy' cars`
- `sed '/chevy/ q' cars` - display to first line matching regular expression
- `sed 's/[0-9]/*/' cars` - substitute first digit on each line with an asterisk
- `sed 's/[0-9]/*/g' cars` - substitute every digit on each line with an asterisk
- `sed '5,8 s/[0-9]/*/' cars` - substitute only on lines 5 to 8
- `sed 's/[0-9][0-9]*/*** & ***/' cars` - surround first number on each line with asterisks

What does the following command do?

- **$ sed 's/^./\t&/' cars**
  - **Sed 's/mainPattern/substitutePattern/' filename** is used for substitution of data in a file
  - The regular expression in the following instruction (^.) matches one character at the beginning of every line that is not empty.
  - The replacement string (between the second and third slashes) contains a backslash escape sequence that represents a TABcharacter (\t) followed by an ampersand (&).
    - The ampersand (&) takes on the value of what the regular expression matched.
  - This type of substitution is useful for indenting a file to create a left margin

SED Examples

- `sed 's/t/T/' cars`
  - Replace t with T
- `sed 's/t/T/g' cars`
  - Replace all T
- How do you physically replace all 't's into 'T'
  - `sed -i 's/t/T/g' test.txt`
    - Change it in-place!

More examples

- sed ‘s/./*/g’ test.txt
  - Replace all characters with ‘*’
- sed ‘s/\w/*/g’ test.txt
  - Replace all characters with ‘*’
- sed ‘s/[0-9]/*/g’ test.txt
  - Replace all digits with ‘*’

Activity

- Using AWK or SED :
  - Display all users in /etc/passwd which
    - Their username starts with ‘p’
  - Display all users in /etc/passwd which
    - Their username ends with ‘p’