Usage Guidelines

Education and Research

Do not forward this document to any non-Infosys mail ID. Forwarding this document to a non-Infosys mail ID may lead to disciplinary action against you, including termination of employment.

Contents of this material cannot be used in any other internal or external document without explicit permission from E&R@infosys.com.

UNIX sed & awk

Education & Research

Confidential Information

Education and Research

- *This Document is confidential to Infosys Limited. This document contains information and data that Infosys considers confidential and proprietary (“Confidential Information”).*
- *Confidential Information includes, but is not limited to, the following:*
  - *Corporate and Infrastructure information about Infosys;*
  - *Infosys’ project management and quality processes;*
  - *Project experiences provided included as illustrative case studies.*
  - *< Please list any/all other that is relevant>*
- *Any disclosure of Confidential Information to, or use of it by a third party, will be damaging to Infosys.*
- *Ownership of all Infosys Confidential Information, no matter in what media it resides, remains with Infosys.*
- *Confidential information in this document shall not be disclosed, duplicated or used – in whole or in part – for any purpose other than _(Please insert as applicable)__ without specific written permission of an authorized representative of Infosys.*
- *<Please include this point if applicable> This document also contains third party confidential and proprietary information. Such third party information has been included by Infosys after receiving due written permissions and authorizations from the party/ies. Such third party confidential and proprietary information shall not be disclosed, duplicated or used – in whole or in part – for any purpose other than _(Please insert as applicable)__ without specific written permission of an authorized representative of Infosys.*

Course Objectives

- To introduce regular expressions
- To introduce sed tool
- To introduce awk programming language

Session Plan – Day 1

- To introduce wildcards and regular expressions
- To introduce sed
- Addressing mechanism in sed
- To explain basic sed commands for stream manipulation –insert, delete, append, substitute, print , quit and transform
- To explain input/output processing in sed

Session Plan – Day 2

- Recap of sed
- Pattern space and sed
- To explain branching commands and multiline processing
- Advantage and disadvantages of sed
- To explain awk features
- To introduce basic structure of awk, running awk scripts
- To explain how to read input from files/records
- To explain print statement, output separators and formatted output with printf statement
- To explain awk expressions – Arithmetic Expressions, string operators, Boolean expressions, conditional expressions

Session Plan – Day 3

- To introduce Built-in variables & dynamic/user defined variables
- To introduce control statements – if, for and while
- To introduce arrays
- To explain how to work with associative arrays, multidimensional arrays
- To introduce built-in functions in awk
- To explain user-defined functions
- Case study discussion

Day 1

- Regular Expressions
- sed features
- sed commands for string manipulation
- Input/out processing
- Branching commands and multiline processing

Regular Expressions

- What is it?
  - String of ordinary and meta characters which can be used to match more than one type of pattern
  - Used in grep, egrep, awk, vi, sed etc.
  - Some examples of metacharacters are
    - [ ], ^, $, { }, ., etc

- Regular expression is collection of Atom and Operator
  - Atom specifies the nature and position of search
  - Operator provides robust constructs for using atoms in a more advanced way

Regular Expressions (Contd…)

- Atoms
  - Character
    - This may contain any printable character (alpha-numeric or special)
  - . (dot)
    - represents any single character except newline
  - Range/Class
    - matches any one character from the set [ ]
  - Anchor
  - Back Reference

Regular Expressions (Contd…)

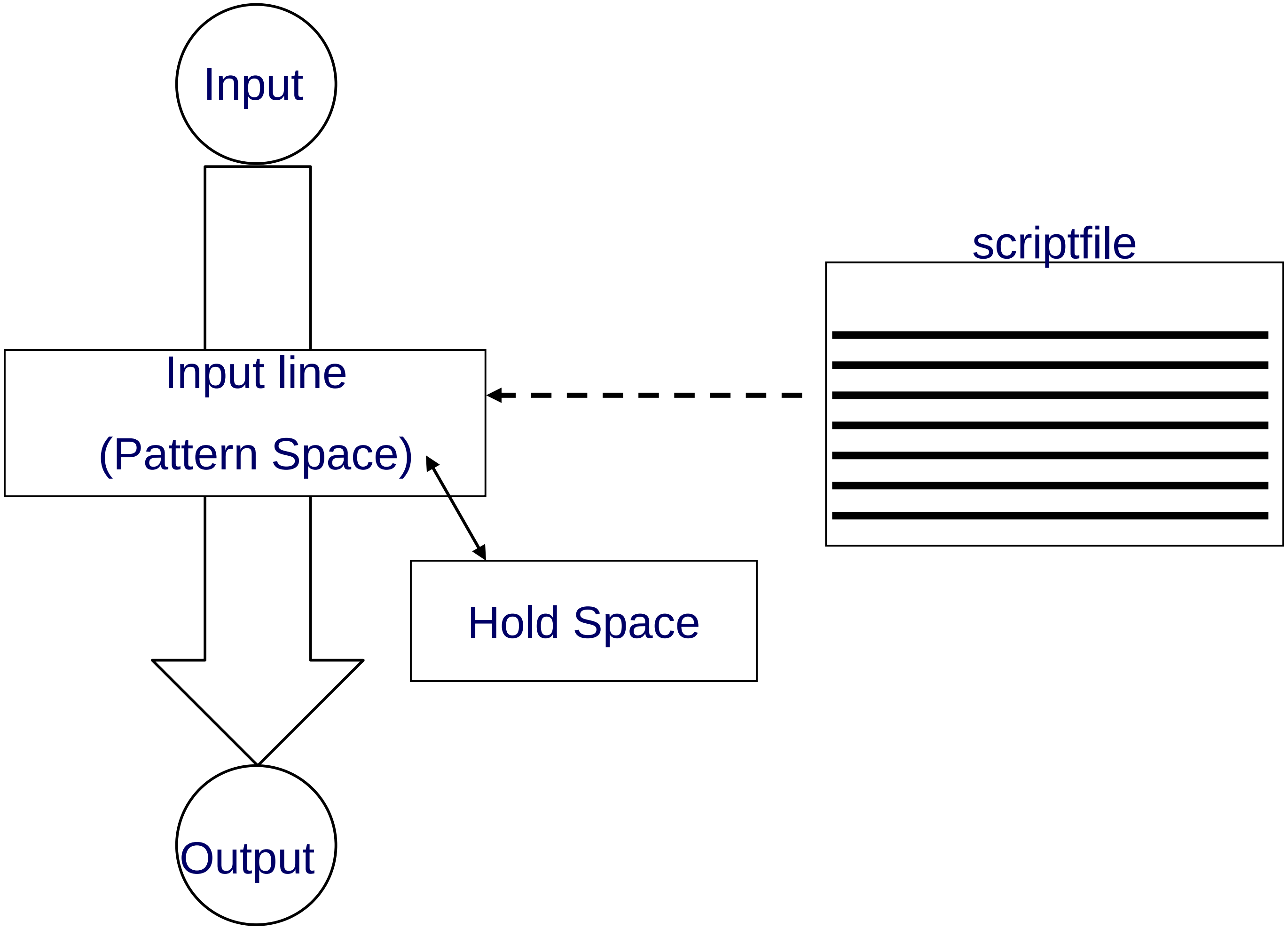
- Operators
  - Alternation |
  - Repetition \{m,n\}
  - Grouping ( )
  - Shorthand * + ?

  - * zero or more matches
  - + one or more matches
  - ? Zero or one match

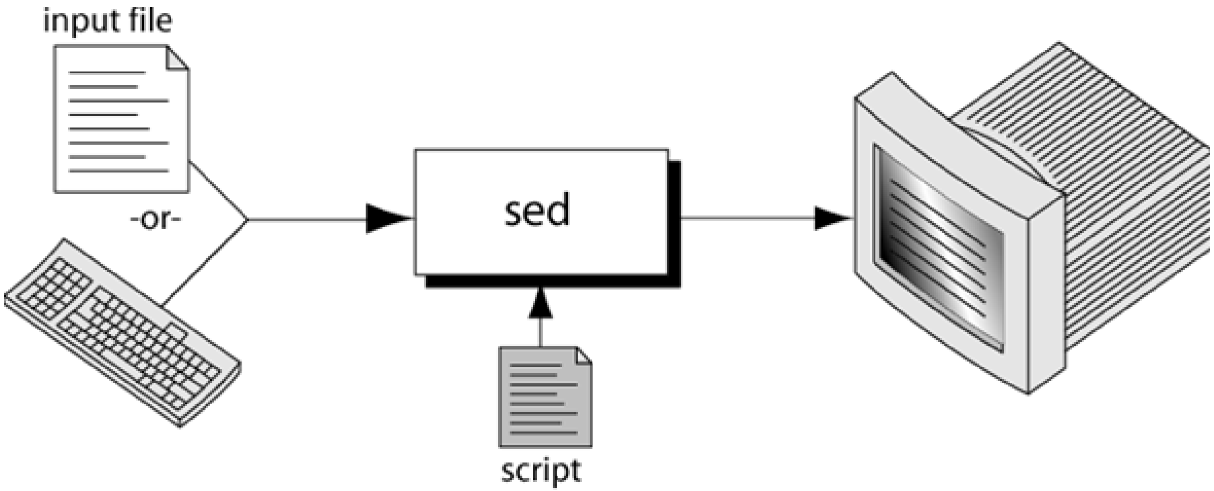
Introduction to sed

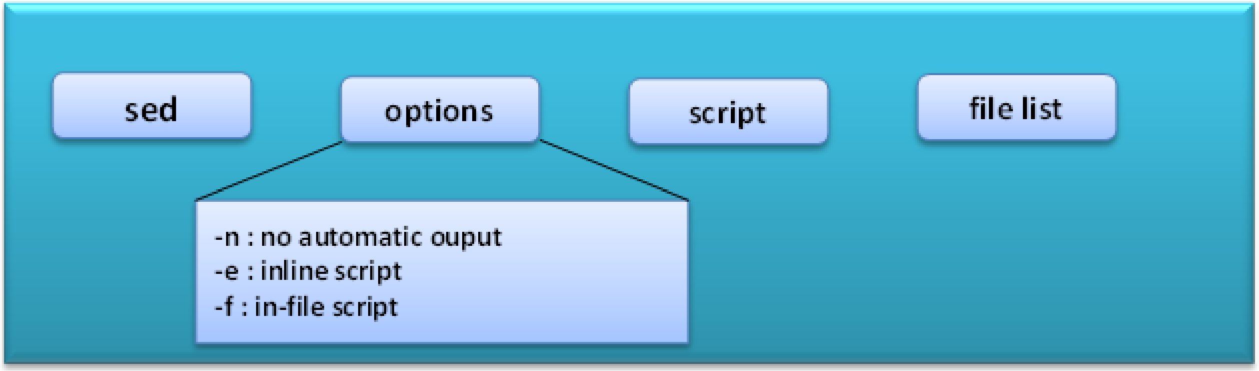
- sed stands for **s**tream **ed**itor
- A non-interactive, command-line editor
- Can be one-liner sed command or a script with multiple commands
- Supports all regular text editing operations
- Used to perform series of edit operations on same file(s)
- Automates text editing of multiple large files

Sed Architecture



Sed Architecture



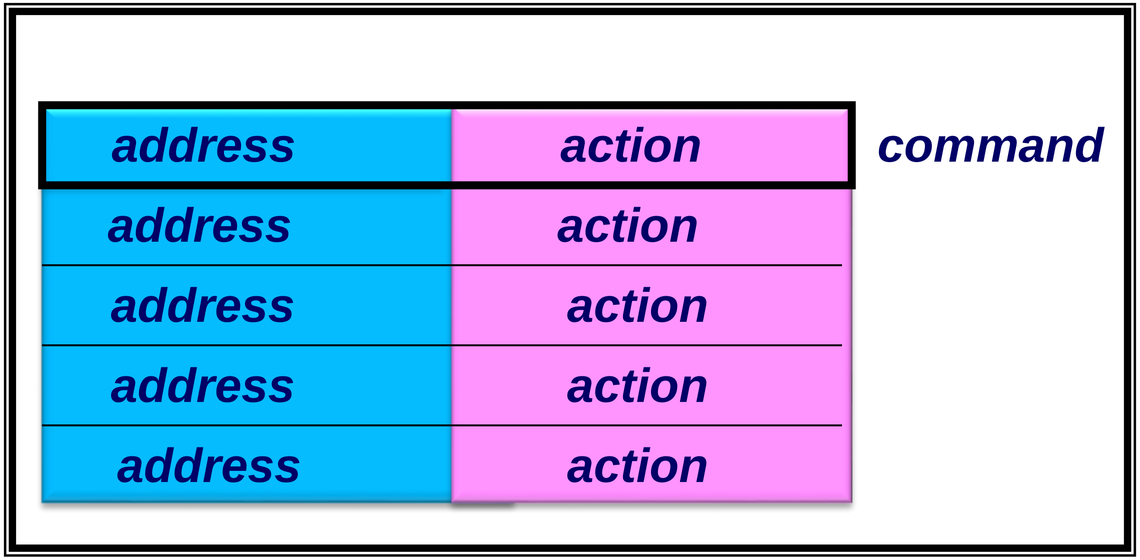


sed options script file list

-n : no automatic ouput
-e : inline script
-f : in-file script

Scripts

- A script is nothing more than a file of commands
- Each command consists of up to two *addresses* and an *action*, where the *address* can be a regular expression or line number.



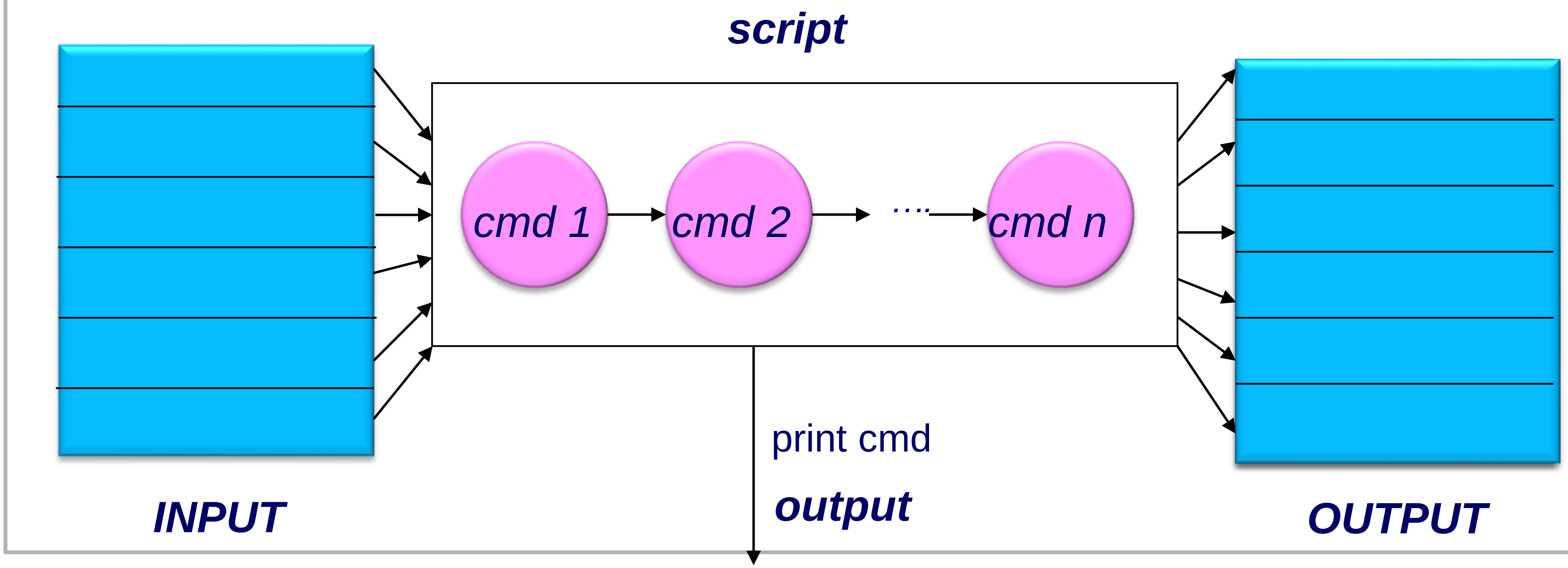
address	*action*
address	*action*
address	*action*
address	*action*
address	*action*

command

•script

Sed Flow of Control

- *sed* then reads the next line in the input file and restarts from the beginning of the script file
- All commands in the script file are compared to, and potentially act on, all lines in the input file



Addressing

- An address can be either a line number, a pattern enclosed in slashes (***Ipatternl***), or a "**$**" symbol (which refers the last line).
- A pattern is described using *regular expressions.*
  - If no sed address is specified, the command will be applied to all lines of the input file
  - If there is only one sed address, the command will be applied to any line which matches the address.
  - If two comma separated addresses are given, then the command operates on a range of lines between the first and second address, inclusively
  - The ! operator can be used to negate an address, ie; *address!command* causes command to be applied to all lines that do not match address

Syntax for sed addressing

- sed commands have the general form

[address[, address]][!]command [arguments]

- where,
  - address : either a line number, a pattern, or a “**$**”
  - ! : negate an address
  - command : can be valid sed command such as options s, a, i, c, d, p, y, q, etc.,
  - arguments : arguments such as input file

Example for sed addressing

- If no sed address is specified,
  - sed –n p file.txt
    - This will print all the lines from the file “file.txt”
- If there is only one sed address,
  - sed -n 2p file.txt
    - This will print only the second line from the from “file.txt”
- If two comma separated addresses are given,
  - sed -n 2,4p file.txt
    - This will print the lines from 2 to 5 from the file “file.txt”
- If ! operator is used,
  - sed -n '2!p' file.txt
    - This will print all the lines except second line from the file “file.txt”

How does sed works?

- *sed* copies each input line into a *pattern space*
  - If the address of the command matches the line in the *pattern space*, the command is applied to that line
  - If the command has no address, it is applied to each line as it enters *pattern space*
  - If a command changes the line in *pattern space*, subsequent commands operate on the modified line
- When all commands have been read, the line in *pattern space* is written to standard output and a new line is read into *pattern space*

Sed Commands

- Although sed contains many editing commands, we are only going to cover the following subset:

- ***`s` - substitute***
- ***`a` - append***
- ***`i` - insert***
- ***`c` - change***
- ***`d` - delete***
- ***`p` - print***
- ***`y` - transform***
- ***`q` - quit***

substitute command

- s/// is substitution command
- Used to replace strings matching the pattern specified.
- Syntax

[address]s/pattern/string/flags

- Address preceding the command is used to specify the range of input lines
  - ex-1: sed ‘2,4s/bat/xyz/’ input.txt
  - Replaces first occurrence of bat from line number 2 to 4(inclusive).
  - ex-2: sed ‘4s/bat/xyz/’ input.txt
  - Replaces first occurrence of bat from in line number 4.
- Address can be line numbers, keywords(patterns) or both

Example

- sed example
  - sed -e 's/sed/SED/i' input.txt(given in notes page)
    - This sample command will manipulate all occurrences of string sed in input.txt by SED.
    - s/// is a substitution command
  - sed -e ‘1,3s/sed/SED/i' input.txt(given in notes page)
    - Same as above but the command is operated only on range of line numbers specified
  - sed -e 's/old/new/' -e 's/fast/slow/' <in.txt >out.txt
    - It combines multiple commands

substitute command…

- Flags are used to modify the default behavior
  - /g for global replacement (all multiple occurrences)
  - /n for nth occurrence (by default substitution is done for first occurrence)
  - /p – prints the modified lines
    - Useful with sed's –n option to only print modified lines
  - /w to save to file
    - sed 's/bat/cat/w change.log' input.txt
    - Replaces all occurrences of 'bat' with 'cat' and writes the modified output into change.log

Example

- sed example
  - sed -e 's/sed/SED/i' input.txt(given in notes page)
    - This sample command will manipulate all occurrences of string sed in input.txt by SED.
    - s/// is a substitution command
  - sed -e ‘1,3s/sed/SED/i' input.txt(given in notes page)
    - Same as above but the command is operated only on range of line numbers specified
  - sed -e 's/old/new/' -e 's/fast/slow/' <in.txt >out.txt
    - It combines multiple commands

sed is not recursive

- Sed’s actions will be performed only on the incoming input(data) which has been read and pattern is matched.
- Generates modified output on that data and continues to read the rest of input lines.
- Here it will not scan the newly created output.
- The following will not cause any indefinite loop.
  - **sed 's/India/India defeats Pakistan in World cup final/g' in.txt**

append command

- append command used to add lines after the matched lines
- Syntax:

sed '[address] a new-line text' filename

- Examples
  - sed '1,10 a THIS IS NEW LINE' input.txt
    - Adds the new line after every line of first 10 lines
  - sed '/bat/ a THIS IS NEW LINE' input.txt
    - Adds the new line after every line that contains pattern ‘bat’
  - sed '$ a HEADING' input.txt
    - Adds text after last line

insert command

- insert command used to replace whole matching line
- Syntax:

sed '[address] i new-line text' filename

- Examples
  - sed '1,10 i THIS IS NEW LINE' input.txt
    - Adds the new line before every line of first 10 lines
  - sed '/bat/ i THIS IS NEW LINE' input.txt
    - Adds the new line before every line which has ‘bat’ pattern
  - sed '1 i HEADING' input.txt
    - Adds text before first line

change command

- Change Command is used to replace a line with new line
- Syntax:

sed '[address] c new-line text' filename

- Examples
  - sed '1,10 c THIS IS NEW LINE' input.txt
    - Suppress all lines from 1 to 10 and replaces with one new line
  - sed '/bat/ c\THIS IS NEW LINE' input.txt
    - Replaces every line containing 'bat' with new line

delete command

- Delete command deletes lines matching the pattern specified
- Complete line is deleted
- Use s/// command to delete part of line that matches
- Syntax

```
[address1[,address2]]d
```

  - Delete the addressed line(s) from the pattern space; line(s) not passed to standard output.
  - A new line of input is read and editing resumes with the first command of the script.

delete command…

- Examples
  - sed '1,10 d' input.txt
    - Deletes lines 1 to 10 and displays all other lines
  - sed '/bat/ d' input.txt
    - Deletes all lines containing 'bat' and displays all other lines
  - sed –e '1,10 !d' input.txt (! Operator is used to negate)
    - Negates the deletion and deletes all lines starting from line 11

print command

- The Print command (**p**) can be used to force the pattern space to be output, useful if the ***-n*** option has been specified
- Syntax:

[address1[,address2]]p

- Note: if the ***-n*** or ***#n*** option has not been specified, **p** will cause the line to be output twice!
- Examples:

  1,5p will display lines 1 through 5

  /^$/,$p will display the lines from the first blank line through the last line of the file

print command…

- Examples
  - sed –n '1,10 p' input.txt
    - Prints first 10 lines (similar to head command in unix)
  - sed -n '/bat/ p' input.txt
    - prints all lines containing 'bat'
  - sed –n '2 !p' input.txt (! Operator is used to negate)
    - Prints all lines except line 2

transform command

- transform command is used for character by character translation
- Syntax:

```
sed '[address] y/char-list/new-char-list/' filename
```

- Examples
  - sed '1,10 y/abc/xyz/' input.txt
    - Replaces all occurrences of 'a' , 'b' and 'c' with 'x' , 'y' and 'z' respectively.
  - sed '/bat/ y/abc/xyz/' input.txt
    - Replaces all occurrences of 'a', 'b' and 'c' with 'x' , 'y' and 'z' respectively in only those lines containing 'bat'.

quit command

- Quit causes **sed** to stop reading new input lines and stop sending them to standard output
- It takes at most a single line address
  - Once a line matching the address is reached, the script will be terminated
  - This can be used to save time when you only want to process some portion of the beginning of a file
- Syntax

[address1[,address2]]q

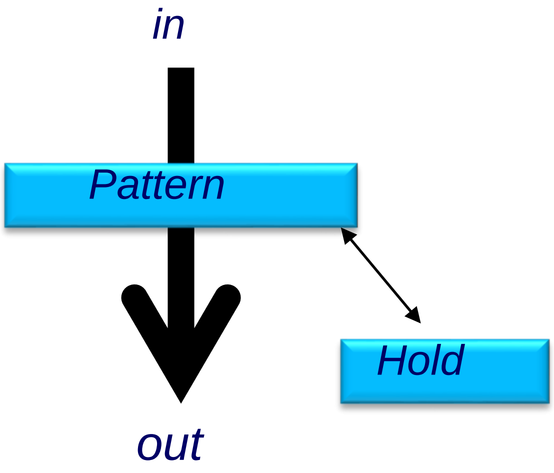
quit command

- Examples
  - to print the first 100 lines of a file (like *head*) use:
    - **`sed '100q' filename`**
      - sed will, by default, send the first 100 lines of *filename* to standard output and then quit processing

Pattern and Hold spaces

- **Pattern space**: Workspace or temporary buffer where a single line of input is held while the editing commands are applied
- **Hold space**: Secondary temporary buffer for temporary storage only

`h, H, g, G, x`



Pattern space

- sed reads each line into pattern space and performs operation on the current pattern space
- Ex:
  - sed '1,10 s/bat/cat/' input.txt
    - Each line from 1 to 10 are read into pattern space and substitution operation is performed
  - sed '1,10 {s/bat/cat/; s/bat/mat/} input.txt
    - Each line read into pattern space. Then the 2 s/// commands are performed on pattern space
    - In the above example, each line from 1 to 10 are read
    - first occurrence of 'bat' is replaced by 'cat'
    - Again the second s/// is operated on updated pattern space. So second occurrence is replaced by mat

Working with pattern space…

- n(next command) updates the current pattern space with content of next line.
- Ex:
  - sed -n '/bat/{n;p}' input.txt
    - prints next line after a line containing 'bat'
  - sed '/bat/ {n; y/aeiou/AEIOU/}' input.txt
    - Each line containing 'bat' is read into pattern space.
    - n(next) command replaces the current pattern space with next line
    - y(transform) command then transforms the new pattern space
    - In other words, transform command in this example is performed on line after a line containing 'bat'

Working with pattern space…

- N command appends the current pattern space with content of next line.
- While appending '\n' is added in between the lines
- Ex:
  - sed -n '/bat/{N;p}' input.txt
    - Searches for line with 'bat' and appends current line in pattern space with next line and prints the pattern space.
  - sed '/bat/ {N; y/aeiou/AEIOU/}' input.txt
    - Each line containing 'bat' is read into pattern space.
    - N command appends the current pattern space with next line
    - y(transform) command then transforms the updated pattern space
    - y command is applied on both lines appended in current pattern space

pattern space..holding & exchanging

- h command copies the current pattern space to the hold space(hold buffer)
- H command appends the current pattern space to the hold space(hold buffer)
- g command copies the current hold space to the pattern space
- G command appends the current hold space to the pattern space(hold buffer)
- x command swaps pattern space with hold space
- Ex:
  - sed -n '/me/{h;s/me/he/g;H;x;s/\n/:/;p}' input2.txt
  - Changes all occurrences of 'me' to 'he' and prints both original and changed line separated by colon

Advantages and Disadvantages

- Advantages
  - Regular expressions
  - Fast
  - Concise
- Disadvantages
  - Hard to remember text from one line to another
  - Not possible to go backward in the file
  - No facilities to manipulate numbers
  - Cumbersome syntax

Day 2

Introduction to AWK

Introduction to awk

- A data filtering tool and reporting generation tool
- Initially developed by A. Aho, B. W. Kernighan and P. Weinberger.
- It is data driven, hence convenient than procedural languages to operate easily on specific data.
- **awk** processes ***fields*** while **sed** only processes ***lines***
- An interpreter based language with extensive string handling functions

Advantages of awk over sed

- Convenient numeric processing
- Variables and control flow in the actions
- Convenient way of accessing fields within lines
- Flexible printing
- Built-in arithmetic and string functions
- C-like syntax

awk - Syntax

- The genral form is

awk option pattern { action } file(s)

  - awk '/sales/{print}' emp.dat

- Awk program/script contains a series of statements specifying the action to be take when a particular pattern is matched
  - Syntax

pattern { action }

AWK Program structure

- An **awk** program consists of:
  - An optional BEGIN segment
    - Executes prior to reading input
  - pattern - action pairs
    - Processing for input data
    - For each pattern matched, the corresponding action is taken
  - An optional END segment
    - Processing after end of input data

```
BEGIN {action}
pattern {action}
pattern {action}
  .
  .
  .
pattern { action}
END {action}
```

Running AWK Program

- There are several ways to run an Awk program

- Method1:

```
awk 'program' input_file(s)
```

  - program and input files are provided as command-line arguments

- Method2:

```
awk 'program'
```

  - program is a command-line argument; input is taken from standard input (yes, awk is a filter!)

- Method 3

```
awk -f program_file input_files
```

  - program is read from a file

How does awk works?

- Search a set of files for *patterns.*
- Perform specified *actions* upon lines or fields that contain instances of patterns.
- Does not alter input files.
- Process one input line at a time
- This is similar to **sed**

What is Pattern?

- Selector that determines whether *action* is to be executed *pattern* can be:
  - the special token **BEGIN** or **END**
  - regular expression (enclosed with //)
  - relational or string match expression
  - ! negates the match
  - arbitrary combination of the above using **&& ||**
  - Examples
    - ***/mysore/*** matches if the string “mysore” is in the record
    - **N > 0** matches if the condition is true
    - ***/mysore/ && (name == “Anderson")***
    - awk 'BEGIN { print “Marks"} $1==“Suresh“ {print“$2"} END{print“That’s all Suresh marks"}' std.dat

Actions

- *action* may include a list of one or more C like statements, as well as arithmetic and string expressions and assignments and multiple output streams.
- *action* is performed on every line that matches *pattern*.
  - If *pattern* is not provided, *action* is performed on every input line
  - If *action* is not provided, all matching lines are sent to standard output.
- Since *patterns* and *actions* are optional, *actions* must be enclosed in braces to distinguish them from *pattern*.

awk variables

- User Defined variables
  - Variables can be defined to hold values
  - No declaration is required
  - Can hold number or a string
- Positional variables
  - $(field operator) sign is used, refers to field in the input
  - For ex: $1 contains field 1, $2 contains field 2…..
  - When '$' is used with user defined variables, variable's value is treated as field number.
  - default field separator is single space character

Awk built-in variables

- Apart from user defined variables and positional variables, awk has some special built-in variables

Variable	Description
NF	***number of fields in current record***
NR	***number of records***
FS	***input field separator***
RS	***input record separator***
OFS	***output field separator***
ORS	***output record separator***

NF

- **NF** - Number of fields in record
  - Gives number of fields in record being processed

- Examples
  - $awk '{print NF}' emp.dat
    - This prints the number of fields in the current line

  - $awk '{print $NF}' emp.dat
    - This allows you to print the last field of any column

NR

- **NR -** Number of records
  - the variable whose value is the number of the current record or number of records processed

- Examples
  - awk '{print NR}' emp.dat
    - This prints no. of fields in the current line
  - awk -F"|" 'NR == 2, NR == 10 { print NR, $0 }' emp.dat
    - This prints records from 2 to 10 along with record numbers.

FS

- **FS** - Input field separator
  - The default value is " ". A character/regular expression can be assigned
  - awk –**Fc** option sets FS to the character c
  - Can also be changed in BEGIN
  - $0 is the entire line
  - $1 is the first field, $2 is the second field and so on…

- Examples
  - $ awk -F '|' '{print $1}' emp.dat
    - This has taken the field separator as “|” and prints the first field from the file emp.dat
  - $ awk 'BEGIN { FS="|" } /anil/ {print $1 " " $4}' emp.dat
    - This has taken the field separator as “|” and prints the fields 1 and 4 which matches the lines contains pattern anil.

RS

- **RS -** record separator
  - The default record separator is **newline**
  - By default, **awk** processes its input a line at a time.
  - Can be changed in **BEGIN** action

- Examples
  - awk 'BEGIN { RS = "|" } ; { print $0 }' emp.dat
    - This will print all the fields separated by “|” in a new line as a separate record.

OFS

- **OFS** - Output Field separator
  - used to separate field output when print is used. The default value is " “

- Examples

  - $ awk 'BEGIN { FS="|" ;OFS = ";" }{ print $1, $2 }' emp.dat
    - This will print the fields a,b,c and d separated by “:”

ORS

- **ORS**: Output record separator
  - The default output record separator is a newline
  - This is used to separate record output when print is used

- Examples
  - $ awk 'BEGIN { ORS = "\n\n" }{ print $1, $2 }' emp.dat
    - This prints the first two fields of all the records and prints the blank line after each record

Reading Input files

- **FIELDWIDTHS** variable is used to read fixed length data
- Used to read data files with fixed field sizes

- Example:
  - $ awk 'BEGIN{FIELDWIDTHS="3 2 5"};{print $1,":",$2}' field.txt
    - FIELDWIDTHS is set to string with each field sizes

Computing with awk

- **awk** variables take on numeric (floating point) or string values according to context.
- User-defined variables are *unadorned* (they need not be declared).
- By default, user-defined variables are initialized to the null string which has numerical value 0.
- Counting is easy to do with Awk

```
        $3 > 15 { emp = emp + 1}
        END { print emp, “employees worked more than 15
hrs” }
```

Selection in Awk

- Awk patterns are good for selecting specific lines from the input for further processing
  - Selection by Comparison
    - `$2 >= 5 { print }`
  - Selection by Computation

    `$2 * $3 > 50 { printf(“%6.2f for %s\n”,$2 * $3, $1) }`
  - Selection by Text Content
    - `$1 == "NYU”    or $2 ~ /NYU/`
  - Combinations of Patterns
    - `$2 >= 4 || $3 >= 20`
  - Selection by Line Number
    - `NR >= 10 && NR <= 20`

awk operators

- Arithmetic operators
- String concatenation
- Assignment & Increment operators
- Comparison operators
- Boolean operators
- Conditional expressions

awk operators

- Arithmetic operators
  - Unary plus [+] Ex: var1="1e2"; var2= var1; print var1
    - converts var1 to numeric and assigns to var2
  - Negation [-] Ex: var1="1e2"; var2= -var1; print var1
    - converts var1 to negative numeric and assigns to var2
  - var1 ^ var2 or var1 ** var2
    - var1 raised to power of var2
  - var1= var2 + var3 * var4 / var5;
- String concatenation
  - No specific operator to concatenate, string expressions are concatenated by writing together separated by space
  - var1="abc" ;var2="xyz";var3 = (var1 var2); print var3
  - Prints abcxyz

awk operators

- Assignment operator
  - var1="new"; var2="this is " var1 " value"; print var2
    - prints this is new value
- Increment/decrement operators
  - concatenation
  - No specific operator to concatenate
  - var1= "abc"; var2=var1"xyz";print var2
  - Prints abcxyz

awk operators

- Comparison Operators
  - <, <=, >, >=, ==, !=, ~, !~
  - Both operands should be numbers for numeric comparison, else string comparison is done.
  - x=25;y=100{print x < y }
    - Numeric comparison returns true. Output is 1
  - x="25";y=100{ print x < y }
    - string comparison, returns false. Output is 0
  - x="25";y="100"{print x < y }
    - string comparison, returns false as ascii value of 2 is not less then 1. Output is 0
  - x="25A";y="25a"{print x < y }
    - string comparison, returns true as ascii value of A is less than a. Output is 1

awk operators

- Boolean expressions
  - &&, || , !
- Conditional expression
  - Syntax: [condition ? true-block : false-block;]
  - var2=var1>0 ? var1 : -var1
  - If var1 is greater than zero, assigns var1 else assigns –var1
- Field Operator ($)
  - var1=1; print var1
    - Prints value of var1. output is 1
  - var1=1; print $var1 [$ is used]
    - Prints value of field 1.
    - $var1 is same as $1, if var1 is 1

Control statements

- If Statement.

```
if{
        statements;
}
else{
        statements;
}
```

Ifexample.awk

```
{
    pass_per = $3 / $2 * 100;
    if ( pass_per >= 50 )
    {
        print $1;
     }
}
```

```
{    pass_per = $3 / $2 * 100;
      if ( pass_per >= 50 ){
        pass++;
      }
      else{
       fail++;
      }
}
END { print "total no. of classes with
pass % >= 50:", pass;
          print "total no. of classes with
pass% < 50:",fail}
```

Control statements

- do-while

```
do
{
    block of statements
}while (condition)
```

- while

```
while (condition)
{
    block of statements
}
```

•while

```
# interest1 - compute compound interest
#   input: amount, rate, years
#   output: compound value at end of
each year
{     i = 1
      while (i <= $3) {
        printf("\t%.2f\n", $1 * (1 + $2) ^ i)
        i = i + 1
      }
}
```

For loop

- The syntax of for loop is similar to the one like C language
- Example

```
{
for (i = 1; i <= $3; i = i + 1)
     printf("\t%.2f\n", $1 * (1 + $2) ^ i)
}
```

Control statements

- break - breaks out of current innermost loop
- continue - skips current iteration and starts next iteration
- exit - stops processing immediately
  - If a END rule is specified it is executed

- Example:

  -
    ```
    NF != 3 { printf("Record %d do not contain req. no. of fields", FNR)
               next
             }
    ```

Arrays

- Set of related elements and can contain either number or strings
- Arrays are associative in nature, each value is associated with an index
- indices can be numbers or strings.
- Array declaration is not needed and size of array is not fixed
  - array[subscript] = value
    - array[1]=10
    - array[4]= 100
    - array["one"]=20

Arrays –if and for

- Scanning array element

```
if(4 in array)   # checks if index 4 exists in array
{
        print array[4]
}
```

- Scanning whole array

```
for(i in array)   # prints value for each index in the array
{
        print array[i]
}
```

Arrays –split, delete

- **split** function is used to split string into multiple array elements
- Syntax
  - size = split(string, array-name, separator);
  - s=split(var1, array1, " ")
  - splits var1 with space as delimiter and stores into array
  - returns last index or size. Indices are numeric and starts from 1
- **delete** function deletes one index-value pair from the array
  - delete array[2];
  - deletes element with index 2

awk Examples

- END { print NR }
  - Prints the number of records processed.
- NR == 10
- { print $NF }
- { field = $NF }

  END { print field }
- NF > 4
- $NF > 4
- { nf = nf + NF }

  END { print nf }

Functions

- Awk provide huge set of built-in functions – arithmetic functions, string functions, date functions so on
- string manipulation functions
  - index(in, find)
  - length([string])
  - match(string, regexp)
  - split(string, array [, fieldsep])
  - sub(regexp, replacement [, target])
  - gsub(regexp, replacement [, target])
  - substr(string, start [, length])
  - tolower(string)
  - toupper(string)

String functions

- **index(mainstring, findstring)**
  - Returns the position of first occurrence of 'findstring' in 'mainstring'
  - $ awk 'BEGIN { print index("abcdefgh", "cd") }'
  - prints 3
- **length([string])**
  - Returns length of the string
  - If string is not passed returns length of $0
  - awk 'BEGIN { print length(" abcdefgh ") }' , prints 8

String functions

- **sub(regexp, replacement [, target])**
  - Searches for longest match from start and replaces complete matched string with replacement string
  - Replaces only first match
  - awk ' BEGIN { var1="aabasdtxtzz";sub("b.*t","XYZ",var1); print var1 }'
  - prints aaXYZzz
  - The target should be a variable/array to store the modified value. If target is omitted, default target is $0
- **gsub(regexp, replacement [, target])**
  - Same as sub() with global replacement
  - All longest matches will be replaced

String functions

- **substr(string, start [, length])**
  - Extracts substring starting from **start** till end
  - If length is specified, extracts number of characters equal to length

String functions

- **match(source string, regular expression)**
  - This is used to match/find the regular expression within the source string
  - If it finds the match it sets two special variable which in turn indicate begin and end of regular expression.
  - RSTART:- indicates where the pattern starts
  - RLENGTH:- indicates the length of the pattern

User defined functions

- Function definitions can appear in between the awk rules
- Section of the script that performs a specific task
- Values can be passed to function so that it performs the task on these values
- Values passed to the functions are called arguments
- Return value is send back by the function
- Syntax

```
function name(parameter-list)
    {
        body-of-function
    }
```

User defined functions

Func_example.awk

```
function add(x)
{
        total= total + x ;
}
{
add($2);
}
END {print total}
```

- This script adds $2 of each record and prints after end of input

Summary

- Regular Expression
- Sed Tool
- Awk Programming

Education
and
Research
We enable you to leverage knowledge
anytime, anywhere!

Thank You