Tutorial 8: Sed, Awk

CS 104

Spring, 2024-25

TA: Hari
Credits: Kavya Gupta

Introduction to Sed

- **What Is Sed?**
  - **Sed** stands for '**Stream Editor**'.
  - It is a Unix utility that parses and changes text using a simple programming language.
  - It's used to quickly edit files on Unix systems by searching, adding, replacing, and deleting lines without opening the file in a text editor.
  - Sed is **non-interactive**, i.e., unlike editors like Notepad, we can't move a cursor in the file.
  - It goes through the file **LINE-BY-LINE** and uses pattern matching (**regex**) to identify targets.
- **Why Use Sed?**
  - Text Manipulation/Filtering
  - Efficient editing without opening the file
  - Automation
  - Batch Processing (such as modifying multiple files or performing consistent changes across a dataset)

Sed (and Awk) should already exist in your systems, except Windows. For Windows, WSL is needed.

Types of Commands in Sed

Some of them are:-

- Substitute (s)
- Print (p)
- Delete (d)
- Append (a)
- Insert (i)
- Replace (c)

```
parallels@itsmemario:~/Desktop$ sed 's/Pythagoras/Fibonacci/g' < input.txt
Fibonacci divided those who attended his lectures into two groups: the
Probationers (or listeners) and the Pythagoreans. After three years in
the first class, a listener could be initiated into the second class, to
whom were confided the main discoveries of the school. The Pythagoreans
were a closely knit brotherhood, holding all worldly goods in common and
bound by an oath not to reveal the founder's secrets. Legend has it that a
```

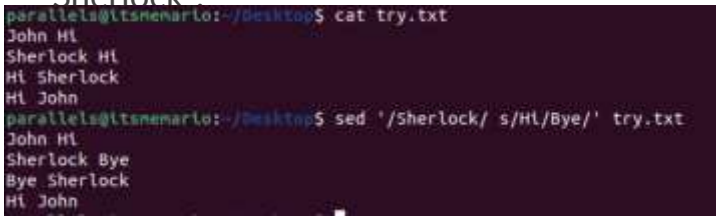
```
parallels@itsmemario:~/Desktop$ cat fibonacci.sed
s/Pythagoras/Fibonacci/g
parallels@itsmemario:~/Desktop$ sed -f fibonacci.sed < input.txt
Fibonacci divided those who attended his lectures into two groups: the
Probationers (or listeners) and the Pythagoreans. After three years in
the first class, a listener could be initiated into the second class, to
whom were confided the main discoveries of the school. The Pythagoreans
were a closely knit brotherhood, holding all worldly goods in common and
bound by an oath not to reveal the founder's secrets. Legend has it that a
```

All these commands can be executed in two ways:-

1) Inline Sed (enclosed within single/double quotes ‘...’/”...”)
2) External .sed file (using -f flag with sed)

Address in Sed Command

- A simple single command in Sed looks like `sed '[address(es)] <command>' <filename>`
- "address(es)" represent the line(s) of the input that sed will apply its "command" on.
- There are two ways to represent line address:-
  - Line Number: Starting from 1. `sed '<line number> <command>'`. Example: `sed '3 s/Hi/Bye/'` will replace "Hi" only on the 3rd line.
  - Pattern: `sed '/<pattern>/ <command>'`. Any line that **contains a string** matching the "pattern" is considered. Example: `sed '/Sherlock/ s/Hi/Bye/'` will replace "Hi" only in lines containing "Sherlock".



- `sed '2,7 s/Hi/Bye/'` will replace "Hi" from lines 2 to 7 (both included).
- `sed '[address(es)] <command>' < <filename>` is also fine.
- **Important: If no address is given then Sed applies the "command" to every line of input file!**

Substitute Command (s)

- Syntax: "`'[address(es)] s/<pattern>/<replacement>/[flags]'`"
- **Regex** is used for pattern matching.
- Replacement is the exact string you want to replace the string that matches the "pattern".
    - "flags" are optional. Possible flags are:-
        - A natural number. Say if our command looks like "s/pattern/replacement/100", then sed will replace **only that string** that matches the "pattern" 100th time **from start IN THAT LINE!**.
        - '**g**'. Consider **ALL** the strings that match the "pattern".
        - '**p**'. This is basically the print command. It prints the line (**again**) (in modified form) containing a string, sed has replaced.
        - '**I**'. Ensures case insensitivity in pattern matching.

```
parallels@itsmenario:~/Desktop$ echo "Virahanka founf the Fibonaccl Numbers first. Fibonaccl found them later." | sed 's/Fibonaccl/Pythagoras/2'
Virahanka founf the Fibonacci Numbers first. Pythagoras found them later.
```

```
parallels@itsmenario:~/Desktop$ echo "Virahanka founf the Fibonacci Numbers first. Fibonacci found them later." | sed 's/Fibonacci/Pythagoras/g'
Virahanka founf the Pythagoras Numbers first. Pythagoras found them later.
```

```
parallels@itsmenario:~/Desktop$ echo "(Hi)" | sed 's/\(Hi\)/**/'
(**)
```

```
parallels@itsmenario:~/Desktop$ echo "(Hi)" | sed -E 's/\(Hi\)/**/'
**
```

- If no flag is given, then sed takes **only the 1st string that matches in a particular line**.
- 's/pattern//[flags]' is equivalent to removing that string.
- **-E, -r** or **--regex-extended** flag can be used to use **extended regex** for pattern matching.
- Normal regex in sed **doesn't always** catch **( )** in pattern, they are sometimes used for **backreferencing** (explained later). To catch ( ) as a **normal characters always**, extended regex should be used.

Print (p) and Delete (d) Commands

```
parallels@itsmemario:~/Desktop$ cat try.txt
Hi John Watson
Hi Mary
Hi Sherlock Holmes
Hi Mycroft Holmes
Hi Hudson
Hi Molly
```

```
parallels@itsmemario:~/Desktop$ sed '/Holmes/ s/Hi/Bye/p' try.txt
Hi John Watson
Hi Mary
Bye Sherlock Holmes
Bye Sherlock Holmes
Bye Mycroft Holmes
Bye Mycroft Holmes
Hi Hudson
Hi Molly
```

```
parallels@itsmemario:~/Desktop$ sed -n '/Holmes/ s/Hi/Bye/p' try.txt
Bye Sherlock Holmes
Bye Mycroft Holmes
```

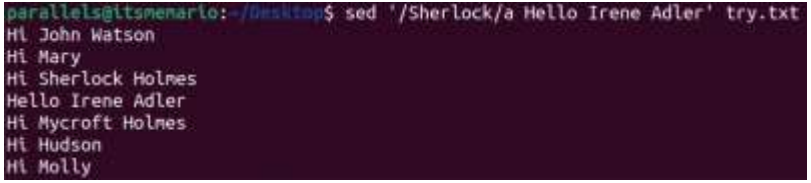
```
parallels@itsmemario:~/Desktop$ sed -n '/Holmes/ s/Hi/Bye/' try.txt
parallels@itsmemario:~/Desktop$
```

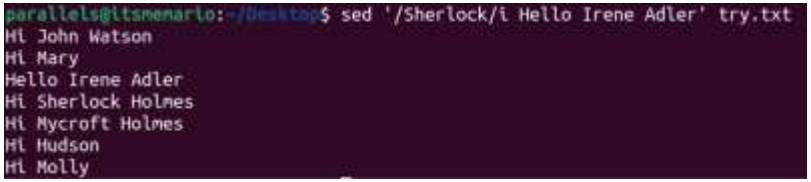
```
parallels@itsmemario:~/Desktop$ sed '/Holmes/d' try.txt
Hi John Watson
Hi Mary
Hi Hudson
Hi Molly
```

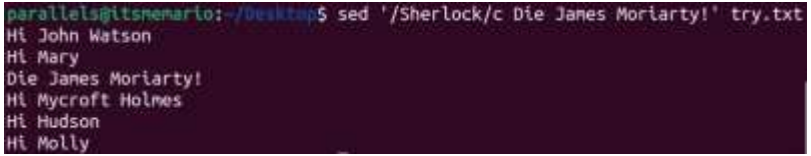
- Print (p) command:-
  - By default, Sed goes through every line of input file and that line is present as an “address” to that command, it will apply the command and print the modified version, otherwise it prints original version of that line.
  - With ‘p’ command, it will print the lines (specified at “address”) in their original/modified form **again!**
  - To **avoid the line getting printed twice**, we use the **-n flag**. It prevents the default behaviour of Sed of printing all the lines once by itself.
  - **‘p’ with -n** prints **only the line(s) specified in address(es)** in their original/modified form.
  - It can be given as a separate command or as a flag for ‘s’ command.
- Delete (d) command:-
  - Deletes the line(s) specified in the address(es).

Append (a), Insert (i), Replace (c) and Quit(q)

- Append (a) command:-
  - Syntax: `'[address(es)]a <string>'`
  - Adds the “string” as a new line **just after** the line(s) specified in the address(es).
- Insert (i) command:-
  - Syntax: `'[address(es)]i <string>'`
  - Adds the “string” as a new line **just before** the line(s) specified in the address(es).
- Replace (c) command:-
  - Syntax: `'[address(es)]c <string>'`
  - Replaces the line(s) specified in the address(es) with the “string” given.
- Quit (q) command:-
  - Syntax: `'[address]q'`
  - Executes till only the first line in the “address” then exits!





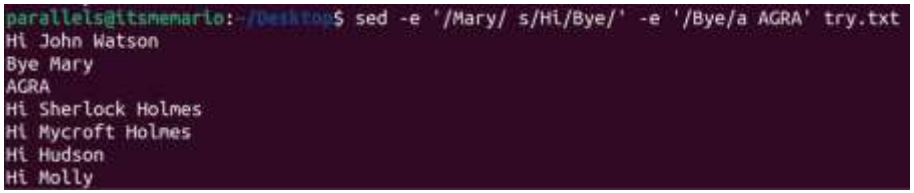


Multiple Commands Execution in Sed

We can achieve **sequential** execution of multiple commands by these 2 ways:-

- Using Semicolon (;):-
  - Syntax: `sed '[address(es)1] <command1>; [address(es)2] <command2>; …' filename`
- Using -e flag:-
  - Syntax: `sed -e '[address(es)1] <command1>' -e '[address(es)2] <command2>' … filename`

Output of one command serves as input for the next one.



- **-i flag** makes the changes “in-place”, i.e. whatever output comes for sed, will automatically be rewritten in the input file.

Backreferencing in Sed

- In Sed, backreferences are specified using a **backslash (\) followed by a single digit** (e.g., \1, \2, etc.).
- The part of the regular expression that you want to refer to is **enclosed in parentheses ( )**.
- These are used for the **Substitute (s)** command.
- By default, the “replacement” string couldn’t be related with the “pattern” string in any way, but backreferencing helps us achieve the same!

```
parallels@itsmemario:~/Desktop$ echo "I am Sherlock Holmes." | sed 's/I am \([A-Za-z]*\) \([A-Za-z]*\)/\1 \2 is my name/'
Sherlock Holmes is my name.
```

- Same in Extended Regex:-

```
parallels@itsmemario:~/Desktop$ echo "I am Sherlock Holmes." | sed -E 's/I am ([A-Za-z]*) ([A-Za-z]*)/\1 \2 is my name/'
Sherlock Holmes is my name.
```

- ‘\1’ contains “Sherlock” and ‘\2’ contains “Holmes”.
- Another Example:-

```
parallels@itsmemario:~/Desktop$ echo "22B1053:Kavya_Gupta:Hello" | sed -E 's/(\w*):(\w*):(\w*)/\2/'
Kavya_Gupta
```

- Remember from Regex in Unix… ‘\w’ is a shorthand form of [A-Za-z0-9_].

Introduction to Awk

- Its name comes from the initials of its designers: Aho, Weinberger, and Kernighan.
- Awk is a domain-specific language designed for text processing and typically used as a data extraction and reporting tool. Like sed and grep, it is a filter, and is a standard feature of most Unix-like operating systems. Awk. Paradigm. Scripting, procedural, data-driven.

Here are some examples of how Awk can be used:

Extract specific data from files:

Awk can be used to extract specific data from files, such as the first column of a file or the lines that contain a certain word.

Format data:

Awk can be used to format data, such as converting dates to a different format or adding commas to numbers.

Perform calculations:

Awk can be used to perform calculations on data, such as adding up the numbers in a column or finding the average of a set of numbers.

Write complex scripts:

Awk can be used to write complex scripts to automate tasks, such as generating reports or processing data.

- Like Sed, Awk goes through the file **LINE-BY-LINE**. But unlike Sed, Awk won't print anything by itself.

Fields, Records and Built-in Variables in Awk

- A record is a line of text and a field is a word within a line.
- Consecutive input fields in a record are separated by a delimiter known as **Field Separator (FS)**. This is also a built-in variable in Awk. It's default value is a single whitespace (' ').
- Consecutive input records are separated by a delimiter known as **Record Separator (RS)**. This is also a built-in variable in Awk. It's default value is the next-line character ('\n').
- We can specify the delimiter between consecutive output fields using **Output Field Separator (OFS)**.
- We can specify the delimiter between consecutive output records using **Output Record Separator (ORS).**
- **NF** specifies the total number of fields in the current record being processed.
- **NR = n** specifies that records 1 to n-1 have been processed and n^{th} one is being processed right now.
- The values of these built-in variables are in the "BEGIN" block of Awk. Alternatively we can set the value of **FS** from command line using **-F (-F fs)** or **--field-separator (--field-separator=fs) flag**.
- **$0** refers to the entire current record line. **$1** refers to the first field of the current record, **$2** the second one and so on… (**$NF refers to the last field in the current record.**)

```
parallels@itsmemario:~$ awk -F: '{print $1}' /etc/passwd
root
daemon
bin
sys
```

Structure of an Awk Script

- Like Sed, Awk can be written in two ways:-
  - Via command line within single quotes ('...'): `awk '<script>' <input filename>`
  - In an external file (.awk file, using **-f flag**): `awk -f <script_file.awk> <input_filename>`
    - **Use double quotes inside "script" or "script_file.awk"**
- BEGIN Block (optional):
  - The BEGIN block executes **once** before processing any input lines.
  - It's ideal for initialization tasks, such as **setting variables** or **printing headers**.
  - For example, you can use it to print a header for your AWK program:
    - `awk 'BEGIN{ print "==Employee Info==" }'`
- Pattern Block (There can be multiple of them):
  - The pattern block processes each input line based on specified conditions.
  - You define patterns (regular expressions) to match against input lines.
  - **If no pattern is mentioned, then Awk will check it against every line.**
  - If a line matches the pattern, the associated action(s) are executed.
  - For instance, to print lines containing the word "manager":
    - `awk '/manager/ { print $1, $NF }'`
- END Block (optional):
  - The END block executes **once** after processing all input lines.
  - It's useful for **final calculations**, summaries, or **printing closing messages**.
  - For example, to print a footer indicating the end of data:
    - `awk 'END{ print "==The End==" }'`

Printing in Awk

- There are two functions to print output in an Awk script:- 1) print, 2) printf

1) **print**:-
   a) Outputs data (pretty much like Python's print), with an additional newline character at the end by default.
   b) Syntax: `print item1, item2, ...` (See no braces used unlike Python)
   c) Example: `{ print $1, $3 }`
   d) Like in Python print is of form "print(…, sep=' ', end='\n')", for print in Awk, **OFS serves as the "sep" attribute** and **ORS as the "end" attribute**.
2) **printf**:-
   a) Pretty much like C's printf, no newline character appended in the end.
   b) It is used for formatted printing, allowing you to specify formatting options such as width, precision, and alignment.
   c) Syntax: `printf format, item1, item2, ...` (See no braces used unlike C)
   d) Example: `{ printf "%s\t%s\n", $1, $3 }`
   e) OFS, ORS has nothing to do with it.

```
#This is print.awk
BEGIN{
FS=":"
OFS=";;;"
}
/run/ {
print $1, $4
}
```

```
parallels@itsmenario:~/Desktop$ awk -f print.awk /etc/passwd
irc;;;39
systemd-network;;;102
systemd-resolve;;;103
systemd-timesync;;;105
sshd;;;65534
```

```
#This is printf.awk
BEGIN{
FS=":"
OFS=";;;"
}
/run/ {
printf "%s----%s\n", $1, $4
}
```

```
parallels@itsmenario:~/Desktop$ awk -f printf.awk /etc/passwd
irc----39
systemd-network----102
systemd-resolve----103
systemd-timesync----105
sshd----65534
uuidd----118
```

Variables and Arithmetic in Awk

```
{
    # Declare a variable
    name = "Sherlock Holmes"

    # Print the value of a variable
    print name

    # Perform a calculation on a variable
    age = 30
    print age + 1

    # Use a variable in an AWK statement
    if (age > 18)
        print "You are an adult."
    else
        print "You are not an adult."
}
```

```
parallels@itsmemario:~/Desktop$ echo "" | awk -f var_arithmetic.awk
Sherlock Holmes
31
You are an adult.
```

- In AWK, a variable is a named storage location that can be used to store data.
- Variables can be used to store any type of data, including numbers, strings, and arrays.
- Variables are declared by assigning a value to them.
- Variables can be used in AWK expressions and statements.
- AWK has a number of built-in variables, such as NR, NF, and FILENAME.
- User-defined variables can be created by assigning a value to them.
- Variables can be used in AWK expressions and statements.
- AWK expressions can be used to perform calculations on variables.
- AWK statements can be used to control the flow of an AWK program.
- Lot of mathematical built-in functions like: sin, cos, tan, sqrt etc. are available.

Conditionals and Loops in Awk

- Syntax of if-else (Pretty much like C/C++):-

```
if (condition) {
    # Actions to perform if condition is true
} else {
    # Actions to perform if condition is false
}
```

Example:-

```
{
    if ($1 > 10) {
        print "Value is greater than 10"
    } else {
        print "Value is not greater than 10"
    }
}
```

- Syntax of while loop (like C/C++):-

```
while (condition) {
    # Actions to perform repeatedly while condition is true
}
```

Example:-

```
{
    i = 1
    while (i <= NF) {
        print $i
        i++
    }
}
```

- Syntax of for loop (like C/C++):-

```
for (initialization; condition; increment/decrement) {
    # Actions to perform repeatedly until condition is false
}
```

```
{
    for (i = 1; i <= NF; i++) {
        print $i
    }
}
```

Arrays in Awk

```
my_array[1] = "apple"
my_array[2] = "banana"
```

```
fruit = my_array[1]
```

```
for (i in my_array) {
    print my_array[i]
}
```

```
matrix[1, 1] = 10
matrix[1, 2] = 20
```

```
fruit_prices["apple"] = 0.99
fruit_prices["banana"] = 1.25
```

- Arrays in Awk are associative arrays, meaning they use key-value pairs for indexing.
- No need to declare the array beforehand separately unlike C/C++.
- Syntax:
  - Initialisation => `array_name[index] = value`
  - Accessing => `value = array_name[index]`
- We can use for loop (in Python/C format) to go through all the elements in array:-
  - Syntax:-

```
for (index in array_name) {
    # Actions to perform for each index
}
```

- Length of an array/string:- `array_length = length(array_name)`
- We can have multi-dimensional arrays also.
  - `matrix[row_index, column_index] = value`