# DAY 4x/14 LEARNING ABOUT LINUX

## 4.X: MASTERING AWK FOR TEXT PROCESSING

**AWK**, named after its creators Aho, Weinberger, and Kernighan, stands as a powerful text processing tool within the Unix and Linux environments. Its utility in handling data extraction and reporting is unparalleled for those who master its syntax and operations. This section is dedicated to unraveling the intricacies of AWK programming, providing a solid foundation for both students and professionals aiming to enhance their shell scripting capabilities.

### FUNDAMENTALS OF AWK PROGRAMS

AWK programs can be executed directly from the command line or, more commonly, stored in a script file and invoked using the **-f** option. This versatility allows users to implement AWK commands for quick data manipulation or develop comprehensive AWK scripts for complex data processing tasks. It's standard practice to encapsulate command-line AWK programs in single quotes to prevent shell interpretation.

An AWK program operates by reading input lines sequentially, breaking each line into fields based on space or tab delimiters by default. This behavior can be customized using the **-F** flag to define alternative field separators, catering to a variety of data formats encountered in text processing tasks.

### THE STRUCTURE OF AN AWK PROGRAM

The essence of an **AWK** program lies in its pattern-action pairs, following the structure **[condition] [{ action }]**. These pairs dictate how AWK interacts with the input data, applying specific actions based on whether the input matches defined conditions. Conditions can be specified as **BEGIN** or **END** blocks for initializing settings before processing and for cleanup tasks after processing, respectively, or as expressions that utilize logical and relational operators alongside regular expressions.

AWK's action statements are reminiscent of C-language syntax, supporting control flow constructs such as **if-else**, **while**, and **for** loops. Notably, if no condition is provided, the corresponding action is applied to every input line. Similarly, omitting an action result in the default action of printing the line to standard output.

### ACCESSING DATA FIELDS IN AWK

AWK distinguishes itself with its straightforward access to individual data fields within an input line. Fields are accessible via the **$** symbol followed by the field number (e.g., **$1** for the first field), with **$0** representing the entire line. The built-in variable **NF** holds the number of fields in the current line, enabling dynamic data manipulation based on the structure of the input.

***Example**: A simple AWK script to prepend the number of fields to each line might look as follows:*

```
awk '{ print NF, $0 }' inputFile
```

### Special BEGIN and END Blocks

AWK's **BEGIN** and **END** constructs allow for setup and teardown operations, executed before reading any input and after processing all input, respectively. These blocks are instrumental in initializing variables or summarizing data after processing, enhancing AWK's flexibility in data handling.

***Example**: Utilizing BEGIN and END to frame the processing of data with custom messages:*

```
BEGIN { print "Processing Start" }

{ print $0 }
```

```
END { print "Processing Complete" }
```

**Regular Expressions and Control Structures**

AWK supports extended regular expressions for defining conditions, providing a powerful mechanism for pattern matching within the input data. This capability, combined with AWK's support for C-like control structures, equips users to develop sophisticated data processing logic with concise syntax.

**Example**: Filtering lines based on pattern matches and performing complex manipulations with control structures:

awk

/regexPattern/ { for (i = NF; i > 0; i--) print $i }

**Advanced Features: Variables and Built-in Functions**

AWK is not limited to simple text processing; it supports user-defined variables and a suite of built-in functions for mathematical calculations, string operations, and more. This feature set enables the development of complex scripts capable of tackling a wide range of text processing challenges.

***Example**: Leveraging AWK's built-in functions to analyze and transform data:*

```
{
   sum += $1
   print "Average so far: ", sum / NR
}
```

**Practical AWK Scripting**

To cement the concepts covered, let's examine a practical example that demonstrates AWK's capabilities in processing and analyzing textual data. Consider a scenario where we need to extract and report on certain data points from a structured log file. Through AWK, one can easily filter relevant lines, split data into fields, perform calculations, and format the output for reporting purposes.

**Conclusion**

AWK remains a potent tool for text processing within the Unix and Linux scripting arsenal. Its blend of simplicity for common tasks with the depth to handle complex processing makes it invaluable for anyone looking to automate and streamline their text processing workflows. Mastery of AWK opens the door to efficient data manipulation, enabling developers and system administrators alike to achieve sophisticated text processing objectives with minimal code.

## Homework TC 4x

### a) Homework 1: Basics of AWK

**Objective**: Familiarize yourself with the basic syntax and operations of AWK.

1. **Print Specific Fields**: Given a text file students.txt containing lines in the format "Name Age Grade", write an AWK script to print only the Name and Grade of each student.
2. **Summing Numbers**: Create an AWK script that sums the numbers in the first column of a given file and prints the total sum.

```
# Print Specific Fields
awk '{print $1, $3}' students.txt
# Summing Numbers
awk '{sum += $1} END {print sum}' numbers.txt
```

**b) Homework 2: Text Filtering and Processing**

**Objective**: Learn to use AWK for filtering text based on patterns and conditions.

1. **Pattern Matching**: Given a file logs.txt, write an AWK command to display all lines that contain the word "ERROR".
2. **Conditional Fields Printing**: For a file records.txt with records of the form "ID Name Score", write an AWK script that prints the Name of students whose Score is above 80.

```
# Pattern Matching
awk '/ERROR/' logs.txt
# Conditional Fields Printing
awk '$3 > 80 {print $2}' records.txt
```

**c) Homework 3: Data Transformation**

**Objective**: Use AWK to transform and manipulate data in text files.

1. **Field Rearrangement**: Create an AWK script that reads dates.txt containing dates in the format "YYYY-MM-DD" and transforms them to "DD/MM/YYYY".
2. **Selective Printing with BEGIN and END**: Using sales.txt, containing daily sales figures, write an AWK script that calculates and prints the total sales at the end and the text "Sales Report" at the beginning.

```
# Field Rearrangement
awk -F"-" '{print $3 "/" $2 "/" $1}' dates.txt
# Selective Printing with BEGIN and END
awk 'BEGIN {print "Sales Report"} {sum += $1} END {print "Total Sales:", sum}' sales.txt
```

**d) Homework 4: Advanced Data Analysis**

**Objective**: Perform complex data analyses using AWK's built-in functions and control structures.

1. **Average Calculation**: For a file grades.txt with student grades, write an AWK script that calculates and prints the average grade for each student.
2. **Min and Max Finder**: Given temperatures.txt with daily temperature records, develop an AWK script to find and print the minimum and maximum temperatures recorded.

```
# Average Calculation
awk '{sum=0; for(i=2; i<=NF; i++) sum+=$i; print $1, "Average:", sum/(NF-1)}' grades.txt
# Min and Max Finder
```

```
awk 'NR==1 {min=$1; max=$1} $1<min {min=$1} $1>max {max=$1} END {print "Min:", min, "Max:", max}' temperatures.txt
```

### e) Homework 5: Regular Expressions and Arrays

**Objective**: Master the use of regular expressions and arrays in AWK for sophisticated text processing tasks.

1. **Regex Filtering**: Using emails.txt, write an AWK script that filters and prints only the lines containing valid email addresses.
2. **Count Occurrences**: For a file words.txt, create an AWK script that counts and prints the number of occurrences of each word using an associative array.

```
# Regex Filtering

awk '/^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/' emails.txt

# Count Occurrences

awk '{for(i=1; i<=NF; i++) words[$i]++} END {for(word in words) print word, words[word]}' words.txt
```

### f) Homework 6: Real-world Data Processing Challenge

**Objective**: Apply AWK in a comprehensive real-world scenario involving multiple files and complex data processing requirements.

1. **Log Analysis**: Given a directory of log files from a web server, write an AWK script to analyze all files and generate a report showing the number of visits per day, the most visited pages, and the average size of requests.
2. **Data Joining**: Suppose you have two files, orders.txt (OrderID, Date, CustomerID) and customers.txt (CustomerID, Name, Email). Write an AWK script to merge these files on CustomerID and produce a report listing OrderID, Date, and Customer Name.

```
# Log Analysis

awk '{visits[$1]++; pages[$2]++; size+=$3} END {for(day in visits) print day, visits[day]; for(page in pages) print page, pages[page]; print "Average Size:", size/NR}' *.log

# Data Joining

awk 'NR==FNR {customers[$1]=$2; next} {print $1, $2, customers[$3]}' customers.txt orders.txt
```

**Submission Guidelines**

- Ensure your scripts are well-commented to explain your logic and approach.
- Test your scripts with sample data to verify correctness.
- Submit (in the forum) your AWK scripts along with a brief report describing your solution strategy for each task. Also export history and upload it as usual.