AWK: Complete Programming Guide

1. Basic awk Commands

What is awk?

Definition:[84] - Named after designers: Alfred V. Aho, Peter J. Weinberger, Brian W. Kernighan - Powerful programming language for pattern processing - Processes files line-by-line and takes actions on matched patterns - GNU version: gawk

Command Syntax

Basic syntax:[84]

```
gawk 'COMMANDS' FILE
gawk <OPTIONS> 'COMMANDS' <FILE(S)>
```

Using external command file:[84]

```
gawk <OPTIONS> -f COMMANDFILE <FILE(S)>
gawk -F: -f script.awk student.txt
```

With field separator option:[84]

```
gawk -F: 'COMMANDS' FILE
```

Records and Fields

Records:[84] - awk reads input file line-by-line - Each line is called a record

Fields:[84] - Each record is split into fields by a separator - Default field separator: space or tab - Specify separator with `-F` option - Current record: `$0` - Individual fields: `$1`, `$2`, `$3`, ...

Example:[84]

```
File: student.txt
Abhik::Das:UG:10MA20012:Algorithms,OS,Networks,ML:70,90,80

With gawk -F:
$0 = "Abhik::Das:UG:10MA20012:Algorithms,OS,Networks,ML:70,90,80"
$1 = "Abhik"
$2 = ""
$3 = "Das"
$4 = "UG"
$5 = "10MA20012"
$6 = "Algorithms,OS,Networks,ML"
$7 = "70,90,80"
```

---

2. BEGIN and END Sections with Pattern Matching

Program Structure

Complete awk program format:[84]

```
BEGIN { Initial actions }
PATTERN1 { Action1 }
PATTERN2 { Action2 }
...
PATTERNn { Actionn }
END { Final actions }
```

Execution flow:[84] - BEGIN section executes before any record is read - Records are processed one by one - For each record, only matching patterns execute - Actions taken in sequence as given - END section executes after all records - Empty pattern matches every record

BEGIN Section

Purpose:[84] - Execute before reading any file - Set initial values and print headers

Example:[84]

```
BEGIN {
    FS = ":"
    print "Reading the student database ..."
}
```

END Section

Purpose:[84] - Execute after all records processed - Print summary information

Example:[84]

```
END {
    print "That is all I have. Bye..."
}
```

Pattern Matching

Regular expression patterns:[84] - Enclosed with delimiters (usually `/`) - Any regular expression valid

Example - Pattern matching:[84]

```
BEGIN {
    FS = ":"
    nUG = 0
    nPG = 0
```

```
}
{
    if ($6 ~ /OS/) {
        if ($4 ~ /UG/) { nUG++ }
        else { nPG++ }
    }
}
END {
    print nUG " UG students have taken OS"
    print nPG " PG students have taken OS"
}
```

Output:

```
2 UG students have taken OS
2 PG students have taken OS
```

Pattern operators:[84] - `~` : Pattern match - `!~` : Pattern non-match

---

3. awk Variables

Built-in Variables

Field variables:[84] - `$0` : Current record (entire line) - `$1, $2, $3, ...` : Fields in current record

Record/File variables:[84] - `NR` : Number of current record (1, 2, 3, ...) - `NF` : Number of fields in current record - `RS` : Record Separator (default: newline) - `FS` : Field Separator - `FILENAME` : Name of current file (NULL if stdin)

Example - Using NR and NF:[84]

```
{
    print "Record " NR ": has " NF " fields"
}
```

User-Defined Variables

Automatic initialization:[84] - Not declared before use - Numeric variables: initialized to 0 - String variables: initialized to empty string - No fixed types

Type conversion:[84] - Strings in numerical context: converted to number (or 0 if not numeric) - Numbers in string context: converted to string - String comparison: lexicographic ordering - As numbers: $9 < 10$ - As strings: "9" > "10"

Example - Variable usage:[84]

```
{
    dept = substr($5, 3, 2)
    if (dept == "MA") {
        nst++
    }
}
```

---

4. awk Arrays

Indexed Arrays

Array declaration and usage:[84] - Arrays are indexed (1-based indexing) - Elements accessed as `Array[index]` - Can also store values while indexing

Example - Indexed array:[84]

```
BEGIN { FS = ":" }
{
    if ($4 ~ /UG/) {
        nUG++
        UG_OS[nUG] = $5
    }
}
END {
    for (i=1; i<=nUG; i++) {
        print UG_OS[i]
    }
}
```

Associative Arrays

Array indexed by strings:[84] - Indexed with string keys - Different: `Array[5]` vs `Array["5"]` - Used for key-value pairs

Example - Associative array:[84]

```
BEGIN { FS = ":" }
{
    if ($4 == "UG") {
        n = split($6, ctaken, ",")
        for (i=1; i<=n; i++) {
            courses[ctaken[i]] = courses[ctaken[i]] " " $5
        }
    }
}
END {
    for (c in courses) {
```

```
        printf("%s: %s\n", c, courses[c])
    }
}
```

Output:

```
OS: 10MA20012 10CS20013
AI: 10CS20010
Algorithms: 10CS20010 10MA20012
ML: 10MA20012 10CS20013
Networks: 10CS20010 10MA20012 10CS20013
```

Looping over associative arrays:[84]

```
for (name in Array) {
    # Access entries as Array[name]
}
```

- Iterations not in sorted order of names

---

5. awk Functions

Built-in Functions

Numeric functions:[84] - `int(x)` - Integer part of x

String functions:[84] - `length(s)` - Length of string s - `index(s,t)` - Index of substring t in s (0 if not found) - `substr(s,b,l)` - Substring of s beginning at index b with length l - `toupper(s)` - Convert string s to uppercase - `tolower(s)` - Convert string s to lowercase

Array manipulation:[84] - `split(s,A,d)` - Split string s by delimiter d, store parts in array A - Returns number of parts - Array indexing is 1-based

Example - Built-in functions:[84]

```
{
    dept = substr($5, 3, 2)
    if (dept == "MA") {
        printf("%s %s has taken %s\n", $1, $3, $6)
        n = split($6, ctaken, ",")
        nst++
    }
}
END {
    printf("Number of Math students: %d\n", nst)
}
```

Output:

```
Abhik Das has taken Algorithms,OS,Networks,ML
Arvind Srinivasan has taken Algorithms,OS,ML
Gautam Kumar has taken Algorithms,AI
Anusha V Pillai has taken Networks,AI
Number of Math students: 4
```

User-Defined Functions

Function definition:[84]

```
function functionName(parameter1, parameter2, ...)
{
    statements
    return value
}
```

Example - Fibonacci function:[84]

```
function F(n) {
    if (n <= 1) { return n }
    return F(n-1) + F(n-2)
}
BEGIN {
    print "Fib(8) = " F(8)
}
```

Function Variable Scope

Variable scope rules:[84] - All variables are global - No local variable declaration provision - Function parameters act as local variables - Parameter passing by value only

Making local variables:[84] - Add local variables to parameter list - Don't pass values to all parameters - Unused parameters initialize to 0 or empty string

Example - Local variables:[84]

```
function oddsum(n, i, sum) {
    print "oddsum(" n ") called"
    sum = 0
    term = 1
    for (i=1; i<=n; i++) {
        sum += term
        term += 2
    }
    return sum
}
BEGIN {
    sum = 0
```

```
    for (i=1; i<=10; i++) {
        sum += oddsum(i)
    }
    print sum
}
```

Output:

```
oddsum(1) called
oddsum(2) called
...
oddsum(10) called
385
```

Runtime Input

Getting user input:[84]

```
BEGIN {
    printf("Enter a positive integer: ")
    getline n < "-"
    n = int(n)
    print "Fib(" n ") = " F(n)
}
```

Command execution:

```
$ gawk -f fib.awk
Enter a positive integer: 8
Fib(8) = 21
```

Setting Variables with -v Option

Pass values at command line:[84]

```
gawk -v n=6 -f script.awk
```

Example:[84]

```
$ gawk -v n=6 -f fib.awk
Entered argument: 6
Fib(6) = 8
```

---

6. File Operations in awk

Output Redirection

Writing to file:[84]

```
print "text" > "filename"    # Overwrite
print "text" >> "filename"   # Append
```

Example - Redirection:[84]

```
{
    dept = substr($5, 3, 2)
    if (dept == "MA") {
        printf("%s %s with roll no. %s has taken %s\n",
               $1, $3, $5, $6) >> "mathdetails.txt"
        nst++
    }
}
END {
    printf("Number of Math students: %d\n", nst)
}
```

File handling rules:[84] - No need to open or close file explicitly - `>` means overwrite (first print determines mode) - `>>` means append (after first print, no distinction) - Mode determined by first print statement - After first print, both `>` and `>>` behave same - Filename must be quoted

Verification:[84]

```
$ gawk -F: -f redirection.awk student.txt
Number of Math students: 4

$ cat mathdetails.txt
Abhik Das with roll no. 10MA20012 has taken Algorithms,OS,Networks,ML
Arvind Srinivasan with roll no. 10MA60012 has taken Algorithms,OS,ML
Gautam Kumar with roll no. 10MA60024 has taken Algorithms,AI
Anusha V Pillai with roll no. 09MA10001 has taken Networks,AI
```

C-like Features in awk

Similar syntax to C:[84] - Comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=` - New operators: `~` (match), `!~` (non-match), `**` (exponentiation) - Logical operators: `&&`, `||`, `!` - Arithmetic operators: `+`, `-`, `*`, `/`, `%`, `++`, `--` - Assignment operators: `=`, `+=`, `-=`, `*=`, `/=`, `%=` - Control flow: `if`, `if-else`, `while`, `for`, `break`, `continue` - Functions: `printf` and `sprintf` work exactly like C

---

7. Quick Reference Examples

Example 1: Print with Pattern

Command:

```
$ gawk -F: '{print $1 "  " $3}' student.txt
```

Output: Names split into fields

Example 2: Conditional Action

Command:

```
$ gawk -F: '{if ($4 == "UG") print $5}' student.txt
```

Output: Roll numbers of UG students only

Example 3: Counting Records

Command:

```
$ gawk -F: '{count++} END {print "Total records: " count}' student.txt
```

Output: Total number of records

Example 4: Field Arithmetic

Command:

```
$ gawk -F: 'BEGIN {sum=0} {sum += NF} END {print "Total fields: " sum}' student.txt
```

Output: Sum of fields in all records