Awk

What is awk?

- created by: Aho, Weinberger, and Kernighan
- scripting language used for ***manipulating data and generating reports***

- versions of awk
  - awk, nawk, mawk, pgawk, …
  - GNU awk: gawk

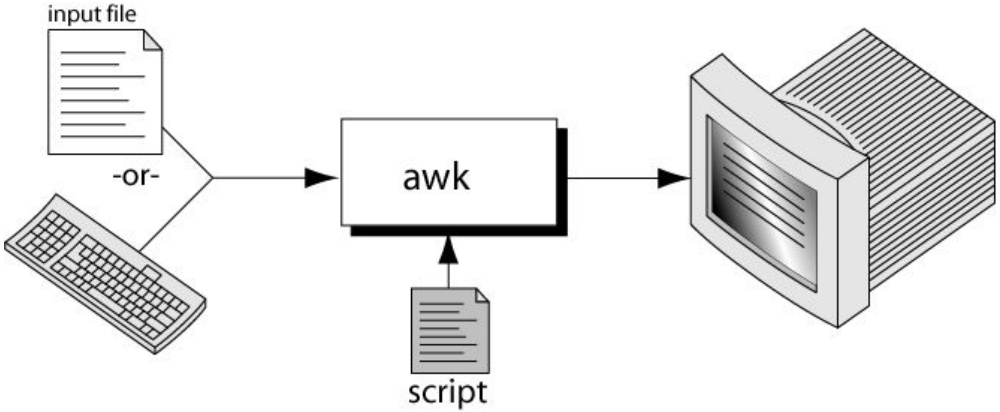
What can you do with awk?

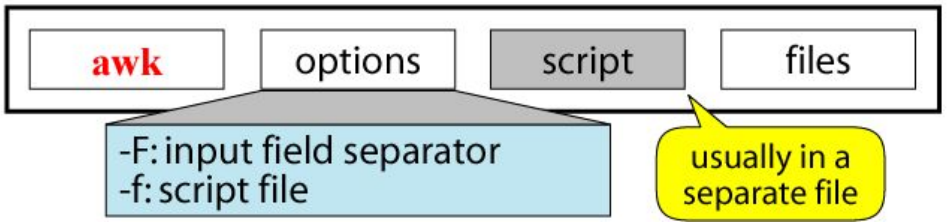
- awk operation:
  - scans a file line by line
  - splits each input line into fields
  - compares input line/fields to pattern
  - performs action(s) on matched lines

- Useful for:
  - transform data files
  - produce formatted reports

- Programming constructs:
  - format output lines
  - arithmetic and string operations
  - conditionals and loops

The Command: awk





Basic awk Syntax

- **`awk [options] 'script' file(s)`**

- **`awk [options] -f scriptfile file(s)`**

Options:

-F to change input field separator

-f to name script file

Basic awk Program

- consists of patterns & actions:

```
pattern {action}
```

  - if pattern is missing, action is applied to all lines
  - if action is missing, the matched line is printed
  - must have either pattern or action

Example:

```
awk '/for/' testfile
```

  - prints all lines containing string “for” in testfile

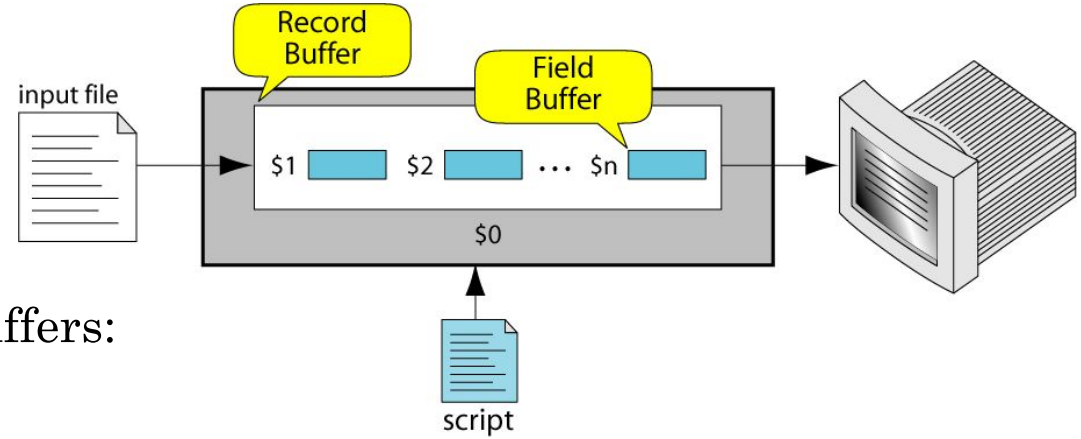
Basic Terminology: input file

- A field is a unit of data in a line
- Each field is separated from the other fields by the field separator
  - default field separator is whitespace
- A record is the collection of fields in a line
- A data file is made up of records

Example Input File - 4 records, 10 fields

```
/dev/disk1     112Gi  101Gi   11Gi    91%   26465116  2862242   90%   /
devfs          332Ki  332Ki    0Bi   100%   1150                0  100%   /dev
map -hosts       0Bi    0Bi     0Bi   100%      0                0  100%   /net
map auto_home    0Bi    0Bi     0Bi   100%      0                0  100%   /home
```

Buffers



- awk supports two types of buffers:

  record and field

- field buffer:
  - one for each fields in the current record.
  - names: $1, $2, …

- record buffer :
  - $0 holds the entire record

Some System Variables

FS	Field separator (default=whitespace)
RS	Record separator (default=\n)
NF	Number of fields in current record
NR	Number of the current record
OFS	Output field separator (default=space)
ORS	Output record separator (default=\n)
FILENAME	Current filename

Example: Records and Fields

```
% cat dfs
/dev/disk1     112Gi  101Gi   11Gi    91%  26465116  2862242   90%   /
devfs            332Ki  332Ki    0Bi    100%  1150                  0  100%   /dev
map  -hosts       0Bi    0Bi      0Bi     100%        0                  0  100%   /net
map auto_home   0Bi  0Bi    0Bi     100%        0                  0  100%   /home
```

```
% awk '{print NR, $0}' dfs
1   /dev/disk1     112Gi  101Gi   11Gi    91%  26465116  2862242   90%   /
2   devfs            332Ki  332Ki    0Bi    100%  1150                  0  100%   /dev
3   map -hosts       0Bi    0Bi      0Bi     100%        0                  0  100%   /net
4   map auto_home   0Bi  0Bi    0Bi     100%        0                  0  100%   /home
```

Example: Space as Field Separator

```
% cat dfs
/dev/disk1     112Gi  101Gi   11Gi    91%  26465116  2862242   90%   /
devfs            332Ki  332Ki    0Bi    100%  1150                    0  100%   /dev
map  -hosts       0Bi    0Bi      0Bi     100%        0                    0  100%   /net
map auto_home   0Bi  0Bi    0Bi     100%        0                   0  100%   /home
```

```
% awk '{print NR, $1, $2, $5}' dfs
1 /dev/disk1  112Gi 91%
2 devfs  332Ki 100%
3 map  -hosts 0Bi
4 map auto_home 0Bi
```

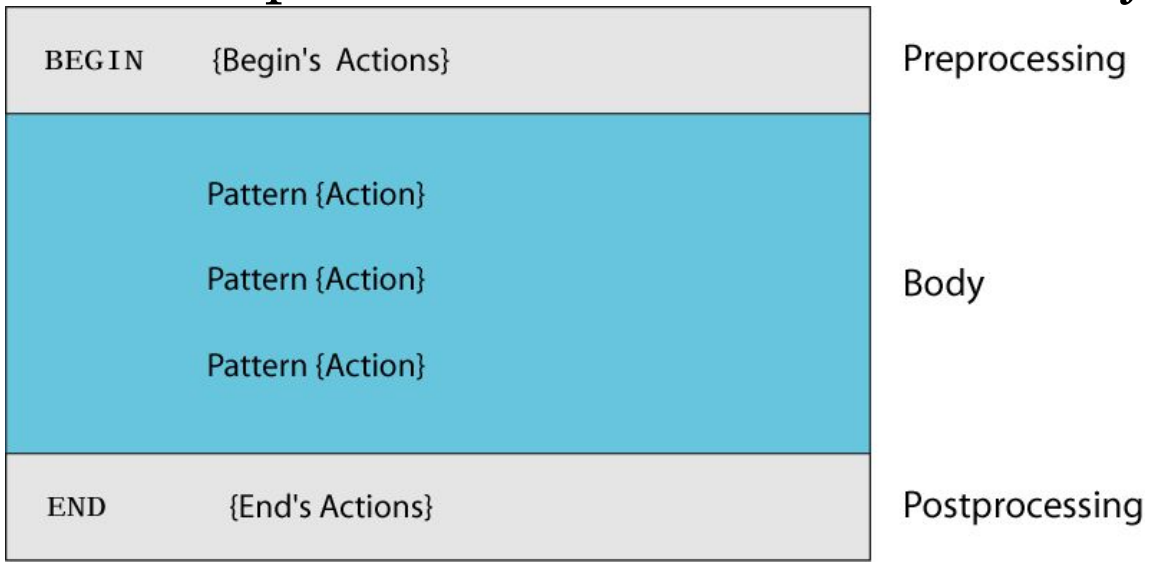
Example: Colon as Field Separator

```
% cat em2
Ram Nath:4424:5/12/66:543354
Govind Patel:5346:11/4/63:28765
Sara Bhooshan:1654:7/22/54:650000
Joe Rodrigues:1683:9/23/44:336500

% awk -F: '/Bhooshan/{print $1, $2}' em2
Sara Bhooshan 1654
```

AWK SCRIPTS

- awk scripts are divided into three major parts:



AWK SCRIPTS

- BEGIN: pre-processing
  - performs processing that must be completed before the file processing starts (i.e., before awk starts reading records from the input file)
  - useful for initialization tasks such as to initialize variables and to create report headings

AWK SCRIPTS

- BODY: Processing
  - contains main processing logic to be applied to input records
  - like a loop that processes input data one record at a time:
    - if a file contains 100 records, the body will be executed 100 times, one for each record

AWK SCRIPTS

- END: post-processing
  - contains logic to be executed after all input data have been processed
  - logic such as printing report grand total should be performed in this part of the script

Pattern / Action Syntax

```
pattern {statement}
```

(a) One Statement Action

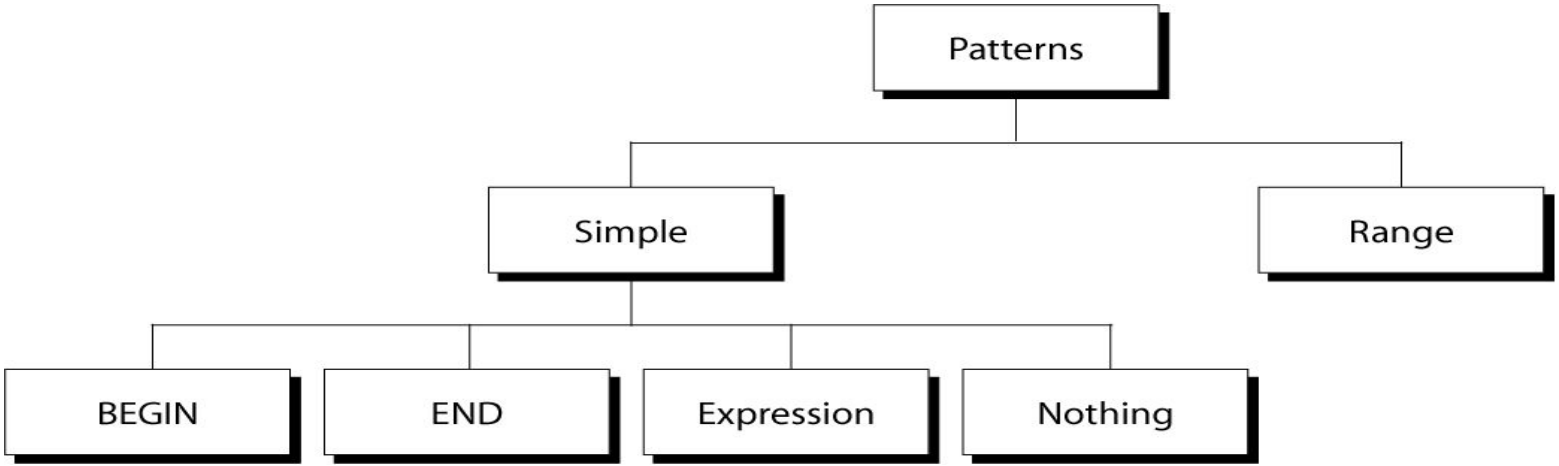
```
pattern {statement1; statement2; statement3}
```

(b) Multiple Statements Separated by Semicolons

```
pattern
{
    statement1
    statement2
    statement3
}
```

(c) Multiple Statements Separated by Newlines

Categories of Patterns



Expression Pattern types

- match
  - entire input record

  regular expression enclosed by ‘/’s
  - explicit pattern-matching expressions

  ~ (match), !~ (not match)

- expression operators
  - arithmetic
  - relational
  - logical

Example: match input record

```
% cat employees2
Ram Nath:4424:5/12/66:543354
Govind Patel:5346:11/4/63:28765
Sara Bhooshan:1654:7/22/54:650000
Joe Rodrigues:1683:9/23/44:336500

% awk -F: '/00$/' employees2
Sara Bhooshan:1654:7/22/54:650000
Joe Rodrigues:1683:9/23/44:336500
```

Example: explicit match

```
% cat datafile
northwest NW     3.0    .98    3   34
western   WE     5.3    .97    5   23
southwest SW     2.7    .8     2   18
southern  SO     5.1    .95    4   15
southeast SE     4.0    .7     4   17
eastern   EA     4.4    .84    5   20
northeast NE     5.1    .94    3   13
north     NO     4.5    .89    5    9
central   CT     5.7    .94    5   13

% awk '$3 ~ /\.[7-9]+/' datafile
southwest SW    2.7    .8     2   18
central   CT    5.7    .94    5   13
```

Examples: matching with REs

```
% awk '$2 !~ /E/{print $1, $2}' datafile
northwest NW
southwest SW
southern SO
north NO
central CT

% awk '/^[ns]/{print $1}' datafile
northwest
southwest
southern
southeast
northeast
north
```

Arithmetic Operators

Operator	Meaning	Example
+	Add	x + y
-	Subtract	x – y
*	Multiply	x * y
/	Divide	x / y
%	Modulus	x % y
^	Exponential	x ^ y

Example:

```
% awk '$3 * $4 > 500 {print $0}' file
```

Relational Operators

Operator	Meaning	Example
<	Less than	x < y
< =	Less than or equal	x < = y
==	Equal to	x == y
!=	Not equal to	x != y
>	Greater than	x > y
> =	Greater than or equal to	x > = y
~	Matched by reg exp	x ~ /y/
!~	Not matched by req exp	x !~ /y/

Logical Operators

Operator	Meaning	Example
&&	Logical AND	a && b
||	Logical OR	a || b
!	NOT	! a

Examples:

```
% awk '($2 > 5) && ($2 <= 15)
                        {print $0}' file
% awk '$3 == 100 || $4 > 50' file
```

Range Patterns

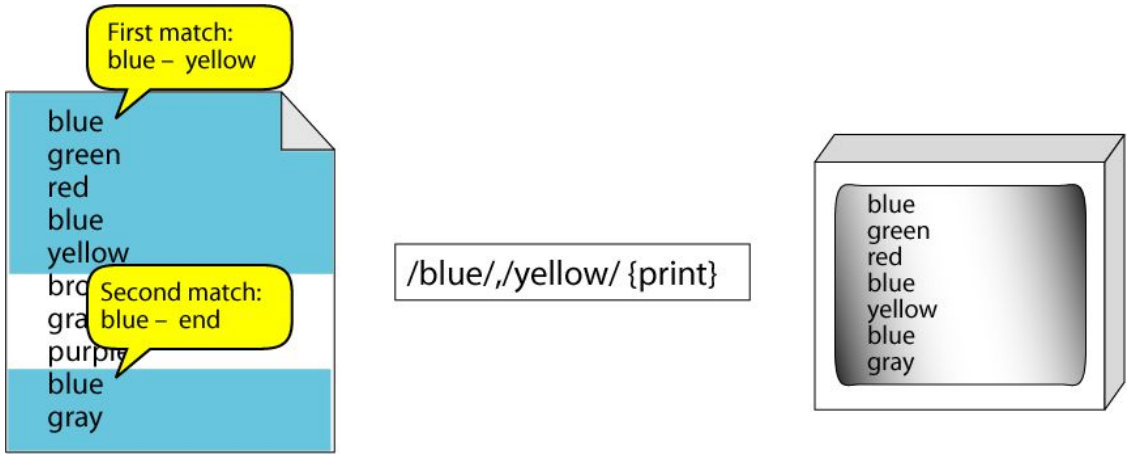
- Matches ranges of consecutive input lines

Syntax:

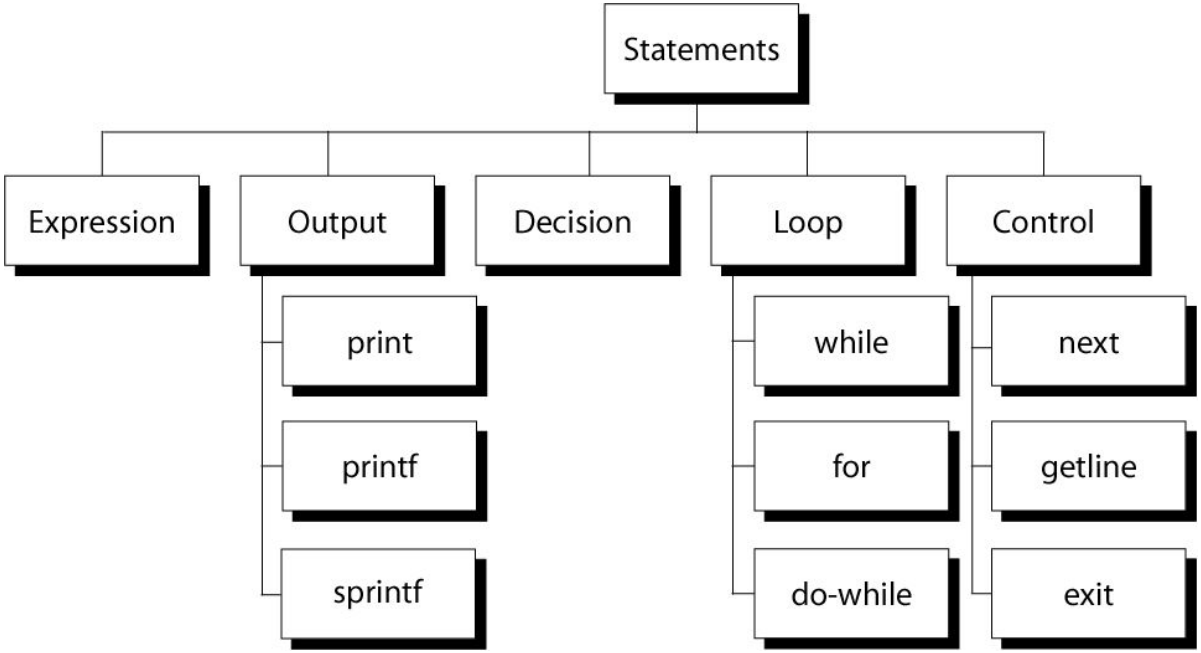
```
pattern1 , pattern2 {action}
```

- pattern can be any simple pattern
- **`pattern1`** turns action on
- **`pattern2`** turns action off

Range Pattern Example



AWK ACTIONS


Statements
Expression
Output
Decision
Loop
Control
print
printf
sprintf
while
for
do-while
next
getline
exit

AWK EXPRESSIONS

- Expression is evaluated and returns value
  - consists of any combination of numeric and string constants, variables, operators, functions, and regular expressions
- Can involve variables
  - As part of expression evaluation
  - As target of assignment

AWK VARIABLES

- A user can define any number of variables within an awk script
- The variables can be numbers, strings, or arrays
- Variable names start with a letter, followed by letters, digits, and underscore
- Variables come into existence the first time they are referenced; therefore, they do not need to be declared before use
- All variables are initially created as strings and initialized to a null string “”

AWK VARIABLES

Format:

```
variable = expression
```

Examples:

```
% awk '$1 ~ /Ram/
        {wage = $3 * $4; print wage}'
      filename
% awk '$4 == "MP"
        {$4 = "Madhya Pradesh"; print $0}'
      filename
```

AWK ASSIGNMENT OPERATORS

= assign result of right-hand-side expression to left-hand-side variable

++ Add 1 to variable

-- Subtract 1 from variable

+= Assign result of addition

-= Assign result of subtraction

*= Assign result of multiplication

/= Assign result of division

%=Assign result of modulo

^= Assign result of exponentiation

Awk example

- File: grades

```
ram 85 92 78 94 88
sita 89 90 75 90 86
laks 84 88 80 92 84
```

- awk script: average

```
# average five grades
{ total = $2 + $3 + $4 + $5 + $6
  avg = total / 5
  print $1, avg }
```

- Run as:

```
awk -f average grades
```

Output Statements

`print`

print easy and simple output

`printf`

print formatted (similar to C printf)

`sprintf`

format string (similar to C sprintf)

Function: print

- Writes to standard output
- Output is terminated by ORS
  - default ORS is newline
- If called with no parameter, it will print $0
- Printed parameters are separated by OFS,
  - default OFS is blank
- Print control characters are allowed:
  - \n \f \a \t \\ …

PRINT EXAMPLE

```
% awk '{print}' grades
ram 85 92 78 94 88
sita 89 90 75 90 86

% awk '{print $0}' grades
ram 85 92 78 94 88
sita 89 90 75 90 86

% awk '{print($0)}' grades
ram 85 92 78 94 88
sita 89 90 75 90 86
```

print Example

```
% awk '{print $1, $2}' grades
ram 85
sita 89

% awk '{print $1 "," $2}' grades
ram,85
sita,89
```

print Example

```
% awk '{OFS="-";print $1 , $2}' grades
ram-85
sita-89

% awk '{OFS="-";print $1 "," $2}' grades
ram,85
sita,89
```

Redirecting print output

- Print output goes to standard output

unless redirected via:

> “file”

>> “file”

| “command”

- will open file or command only once
- subsequent redirections append to already open stream

PRINT EXAMPLE

```
% awk '{print $1 , $2 > "file"}' grades

% cat file
ram 85
sita 89
laks 84
```

print Example

```
% awk '{print $1,$2 | "sort"}' grades
laks 84
ram 89
sita 85

% awk '{print $1,$2 | "sort -k 2"}' grades
laks 84
sita 85
ram 89
```

print Example

```
% date
Wed Nov 19 14:40:07 CST 2008


% date |
   awk '{print "Month: " $2 "\nYear: ",
  $6}'
Month: Nov
Year:  2008
```

printf: Formatting output

Syntax:

```
printf(format-string, var1, var2, …)
```

- works like C printf
- each format specifier in “format-string” requires argument of matching type

Format specifiers

%d, %i decimal integer

%c single character

%s string of characters

%f floating point number

%o octal number

%x hexadecimal number

%e scientific floating point notation

%% the letter “%”

Format specifier examples

Given: x = ‘A’, y = 15, z = 2.3, and $1 = Bob Smith	
Printf Format Specifier	**What it Does**
%c	***printf("The character is %c \n", x)*** **output: The character is A**
%d	***printf("The boy is %d years old \n", y)*** **output: The boy is 15 years old**
%s	***printf("My name is %s \n", $1)*** **output: My name is Bob Smith**
%f	***printf("z is %5.3f \n", z)*** **output: z is 2.300**

Format specifier modifiers

- between “%” and letter

  %10s

  %7d

  %10.4f

  %-20s

- meaning:
  - width of field, field is printed right justified
  - precision: number of digits after decimal point
  - “-” will left justify

sprintf: Formatting text

Syntax:

```
sprintf(format-string, var1, var2, …)
```

- Works like printf, but does not produce output
- Instead it returns formatted string

Example:

```
{
 text = sprintf("1: %d - 2: %d", $1, $2)
 print text
}
```

AWK BUILTIN FUNCTIONS

`tolower(string)`

- returns a copy of string, with each upper-case character converted to lower-case. Nonalphabetic characters are left unchanged.

Example: tolower("MiXeD cAsE 123")

returns "mixed case 123"

`toupper(string)`

- returns a copy of string, with each lower-case character converted to upper-case.

awk Example: list of products

```
103:sway bar:49.99
101:propeller:104.99
104:fishing line:0.99
113:premium fish bait:1.00
106:cup holder:2.49
107:cooler:14.89
112:boat cover:120.00
109:transom:199.00
110:pulley:9.88
105:mirror:4.99
108:wheel:49.99
111:lock:31.00
102:trailer hitch:97.95
```

awk Example: output

```
Marine Parts R Us
Main catalog
Part-id name                      price
=====================================
101     propeller               104.99
102     trailer hitch            97.95
103     sway bar                 49.99
104     fishing line              0.99
105     mirror                    4.99
106     cup holder                2.49
107     cooler                   14.89
108     wheel                    49.99
109     transom                 199.00
110     pulley                    9.88
111     lock                     31.00
112     boat cover              120.00
113     premium fish bait         1.00
=====================================
Catalog has 13 parts
```

AWK EXAMPLE: COMPLETE

```
BEGIN {
        FS= ":"
        print "Marine Parts R Us"
        print "Main catalog"
        print "Part-id\tname\t\t\t price"
        print "======================================="
}
{
        printf("%3d\t%-20s\t%6.2f\n", $1, $2, $3)
        count++
}
END {
        print "======================================="
        print "Catalog has " count " parts"
}
```

is output sorted ?

AWK ARRAY

- awk allows one-dimensional arrays
  to store strings or numbers
- index can be number or string

- array need not be declared
  - its size
  - its elements
- array elements are created when first used
  - initialized to 0 or “”

Arrays in awk

Syntax:

```
arrayName[index] = value
```

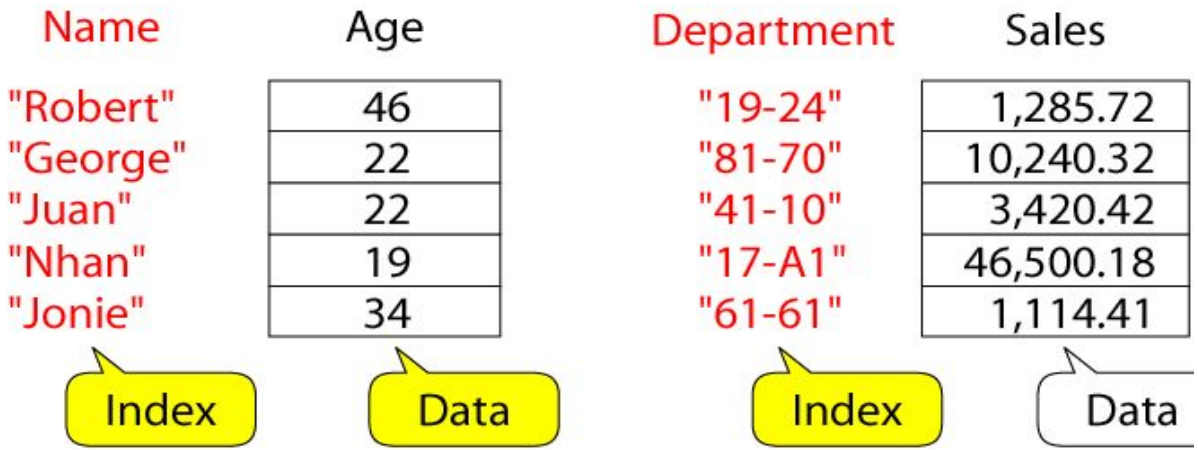
Examples:

```
list[1] = "one"
list[2] = "three"

list["other"] = "oh my !"
```

Illustration: Associative Arrays

- awk arrays can use string as index



AWK BUILTIN SPLIT FUNCTION

`split(string, array, fieldsep)`

- divides string into pieces separated by fieldsep, and stores the pieces in array
- if the fieldsep is omitted, the value of FS is used.

Example:

```
split("auto-da-fe", a, "-")
```

- sets the contents of the array a as follows:

```
a[1] = "auto"
a[2] = "da"
a[3] = "fe"
```

Example: process sales data

- input file:

Sales

1	clothing	3141
1	computers	9161
1	textbooks	21312
2	clothing	3252
2	computers	12321
2	supplies	2242
2	textbooks	15462

Illustration: process each input line

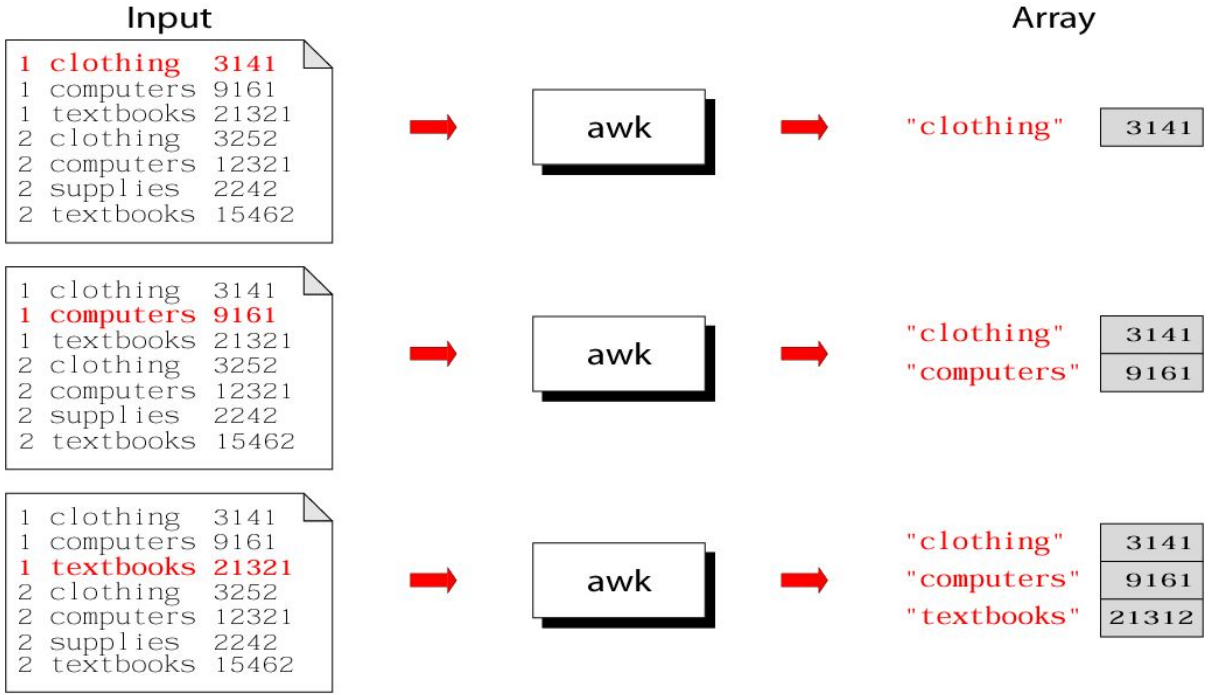
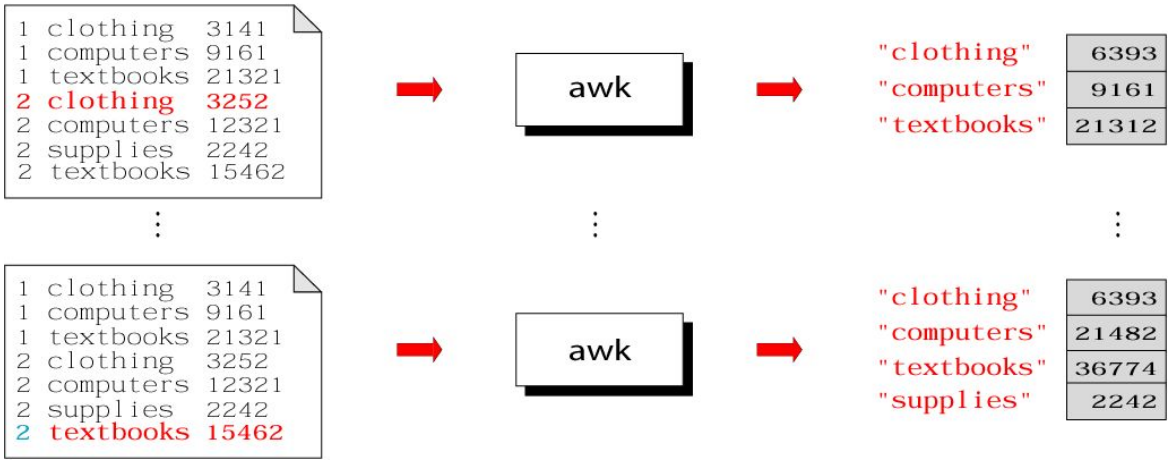


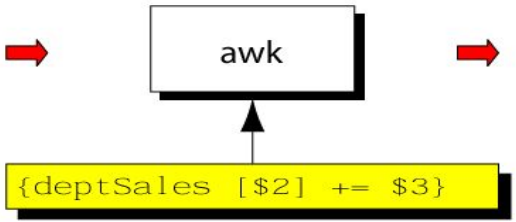
Illustration: process each input line



Summary: awk program

Sales

1	clothing	3141
1	computers	9161
1	textbooks	21312
2	clothing	3252
2	computers	12321
2	supplies	2242
2	textbooks	15462



"clothing"	6393
"computers"	21482
"textbooks"	36774
"supplies"	2242

deptSales

Example: complete program

```
% cat sales.awk
{
        deptSales[$2] += $3
}
END {
        for (x in deptSales)
                print x, deptSales[x]
}
% awk -f sales.awk sales
```

Delete Array Entry

- The delete function can be used to delete an element from an array.

Format:

```
delete array_name [index]
```

Example:

```
delete deptSales["supplies"]
```

Awk control structures

- Conditional
  - if-else
- Repetition
  - for
    - with counter
    - with array index
  - while
  - do-while

  - also: break, continue

IF STATEMENT

Syntax:

```
if (conditional expression)
 statement-1
else
 statement-2
```

Example:

```
if ( NR < 3 )
 print $2
else
 print $3
```

FOR LOOP

Syntax:

```
for (initialization; limit-test; update)
        statement
```

Example:

```
for (i = 1; i <= NR; i++)
{
        total += $i
        count++
}
```

for Loop for arrays

Syntax:

```
for (var in array)
        statement
```

Example:

```
for (x in deptSales)
{
      print x, deptSales[x]
}
```

WHILE LOOP

Syntax:

```
while (logical expression)
        statement
```

Example:

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

DO-WHILE LOOP

Syntax:

```
do
    statement
while (condition)
```

- statement is executed at least once, even if condition is false at the beginning

Example:

```
i = 1
do {
  print $0
  i++
} while (i <= 10)
```

LOOP CONTROL STATEMENTS

- **`break`**

  exits loop

- **`continue`**

  skips rest of current iteration, continues with next iteration

LOOP CONTROL EXAMPLE

```
for (x = 0; x < 20; x++) {
  if ( array[x] > 100) continue
  printf "%d ", x
  if ( array[x] < 0 ) break
}
```