AWK

interpreted programming language

tutorialspoint

SIMPLYEASYLEARNING

www.tutorialspoint.com


https://www.facebook.com/tutorialspointindia


https://twitter.com/tutorialspoint

About the Tutorial

This tutorial takes you through AWK, one of the most prominent text-processing utility on GNU/Linux. It is very powerful and uses simple programming language. It can solve complex text processing tasks with a few lines of code.

Starting with an overview of AWK, its environment, and workflow, the tutorial proceeds to explain the syntax, variables, operators, arrays, loops, and functions used in AWK. It also covers topics such as output redirection and pretty printing.

Audience

This tutorial will be useful for software developers, system administrators, or any enthusiastic reader inclined to learn how to do text processing and data extraction in Unix-like environment.

Prerequisites

You must have a basic understanding of GNU/Linux operating system and shell scripting.

Copyright & Disclaimer

© Copyright 2015 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com

tutorialspoint
SIMPLYEASYLEARNING

Table of Contents

tutorialspoint
SIMPLYEASYLEARNING

tutorialspoint
SIMPLYEASYLEARNING

tutorialspoint
SIMPLYEASYLEARNING

tutorialspoint
SIMPLYEASYLEARNING

1. OVERVIEW

AWK is an interpreted programming language. It is very powerful and specially designed for text processing. Its name is derived from the family names of its authors – **Alfred Aho, Peter Weinberger,** and **Brian Kernighan**.

The version of AWK that GNU/Linux distributes is written and maintained by the Free Software Foundation (FSF); it is often referred to as **GNU AWK**.

Types of AWK

Following are the variants of AWK:

AWK - Original AWK from AT & T Laboratory.

NAWK - Newer and improved version of AWK from AT & T Laboratory.

GAWK - It is GNU AWK. All GNU/Linux distributions ship GAWK. It is fully compatible with AWK and NAWK.

Typical Uses of AWK

Myriad of tasks can be done with AWK. Listed below are just a few of them:

- Text processing
- Producing formatted text reports
- Performing arithmetic operations
- Performing string operations, and many more

tutorialspoint
SIMPLYEASYLEARNING

2. AWK ENVIRONMENT

This chapter describes how to set up the AWK environment on your GNU/Linux system.

Installation Using Package Manager

Generally, AWK is available by default on most GNU/Linux distributions. You can use **which** command to check whether it is present on your system or not. In case you don't have AWK, then install it on Debian based GNU/Linux using Advance Package Tool **(APT)** package manager as follows:

```
[jeryy]$ sudo apt-get update
[jeryy]$ sudo apt-get install gawk
```

Similarly, to install AWK on RPM based GNU/Linux, use Yellowdog Updator Modifier **(YUM)** package manager as follows:

```
[root]# yum install gawk
```

After installation, ensure that AWK is accessible via command line.

```
[jerry]$ which awk
```

On executing the above code, you get the following result:

```
/usr/bin/awk
```

Installation from Source Code

As GNU AWK is a part of the GNU project, its source code is available for free download. We have already seen how to install AWK using package manager. Let us now understand how to install AWK from its source code.

The following installation is applicable to any GNU/Linux software, and for most other freely-available programs as well. Here are the installation steps:

Step 1 - Download the source code from an authentic place. The command-line utility **wget** serves this purpose.

```
[jerry]$ wget http://ftp.gnu.org/gnu/gawk/gawk-4.1.1.tar.xz
```


tutorialspoint
SIMPLYEASYLEARNING

Step 2 - Decompress and extract the downloaded source code.

```
[jerry]$ tar xvf gawk-4.1.1.tar.xz
```

Step 3 - Change into the directory and run configure.

```
[jerry]$ ./configure
```

Step 4 - Upon successful completion, the **configure** generates Makefile. To compile the source code, execute a **make** command.

```
[jerry]$ make
```

Step 5 - You can run the test suite to ensure the build is clean. This is an optional step.

```
[jerry]$ make check
```

Step 6 - Finally, install AWK. Make sure you have super-user privileges.

```
[jerry]$ sudo make install
```

That is it! You have successfully compiled and installed AWK. Verify it by executing the **awk** command as follows:

```
[jerry]$ which awk
```

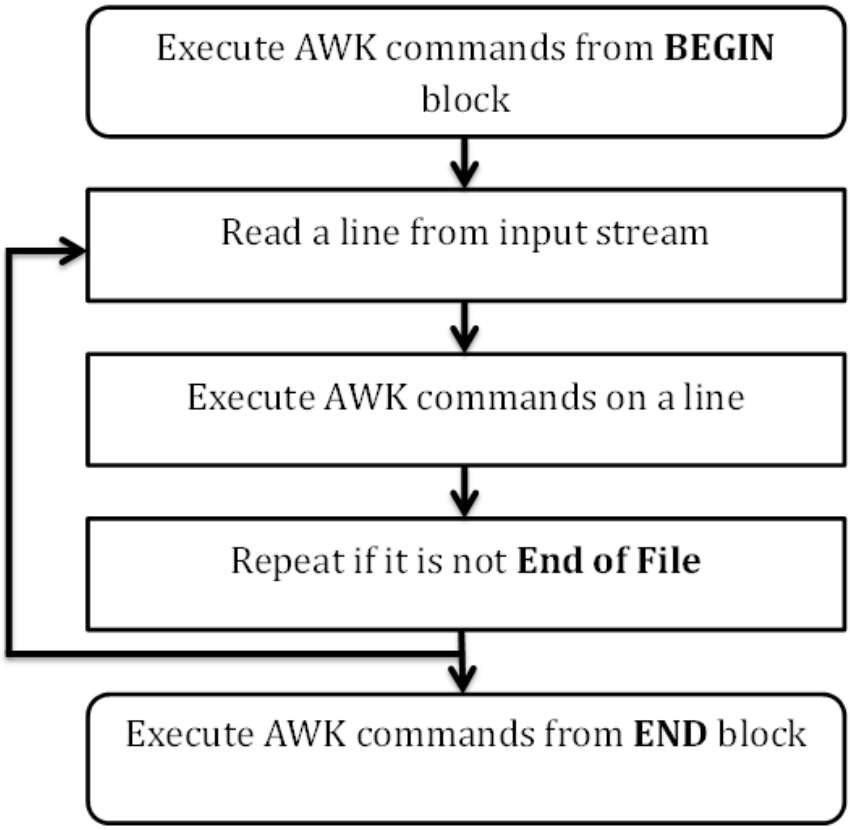
On executing this code, you get the following result:

```
/usr/bin/awk
```


tutorialspoint
SIMPLYEASYLEARNING

3. AWK WORKFLOW

To become an expert AWK programmer, you need to know its internals. AWK follows a simple workflow: Read, Execute, and Repeat. The following diagram depicts the workflow of AWK:



Read

AWK reads a line from the input stream (file, pipe, or stdin) and stores it in memory.

Execute

All AWK commands are applied sequentially on the input. By default, AWK executes commands on every line. We can restrict this by providing patterns.

Repeat

This process repeats until the file reaches its end.

tutorialspoint
SIMPLYEASYLEARNING

Program Structure

Let us now understand the program structure of AWK.

BEGIN Block

The syntax of the BEGIN block is as follows:

```
BEGIN {awk-commands}
```

The BEGIN block gets executed at program start-up. It executes only once. This is good place to initialize variables. BEGIN is an AWK keyword and hence it must be in upper-case. Please note that this block is optional.

Body Block

The syntax of the body block is as follows:

```
/pattern/ {awk-commands}
```

The body block applies AWK commands on every input line. By default, AWK executes commands on every line. We can restrict this by providing patterns. Note that there are no keywords for the Body block.

END Block

The syntax of the END block is as follows:

```
END {awk-commands}
```

The END block executes at the end of the program. END is an AWK keyword and hence it must be in upper-case. Please note that this block is optional.

Example

Let us create a file **marks.txt** which contains the serial number, name of the student, subject name, and number of marks obtained.

```
1)    Amit     Physics    80
2)    Rahul    Maths      90
3)    Shyam    Biology    87
4)    Kedar    English    85
5)    Hari     History    89
```

Let us now display the file contents with header by using AWK script.

tutorialspoint
SIMPLYEASYLEARNING

```
[jerry]$ awk 'BEGIN{printf "Sr No\tName\tSub\tMarks\n"} {print}' marks.txt
```

When this code is executed, it produces the following result:

```
Sr No Name     Sub         Marks
1)    Amit     Physics     80
2)    Rahul    Maths       90
3)    Shyam    Biology     87
4)    Kedar    English     85
5)    Hari     History     89
```

At the start, AWK prints the header from the BEGIN block. Then in the body block, it reads a line from a file and executes AWK's print command which just prints the contents on the standard output stream. This process repeats until file reaches the end.

tutorialspoint
SIMPLYEASYLEARNING

4. BASIC SYNTAX

AWK is simple to use. We can provide AWK commands either directly from the command line or in the form of a text file containing AWK commands.

AWK Command Line

We can specify an AWK command within single quotes at command line as shown:

```
awk [options] file ...
```

Example

Consider a text file **marks.txt** with following content:

```
1)    Amit     Physics    80
2)    Rahul    Maths      90
3)    Shyam    Biology    87
4)    Kedar    English    85
5)    Hari     History    89
```

Let us display the complete content of the file using AWK as follows:

```
[jerry]$ awk '{print}' marks.txt
```

On executing this code, you get the following result:

```
1)    Amit     Physics    80
2)    Rahul    Maths      90
3)    Shyam    Biology    87
4)    Kedar    English    85
5)    Hari     History    89
```

AWK Program File

We can provide AWK commands in a script file as shown:

```
awk [options] -f file ....
```


tutorialspoint
SIMPLYEASYLEARNING

First, create a text file **command.awk** containing the AWK command as shown below:

```
{print}
```

Now we can instruct the AWK to read commands from the text file and perform the action. Here, we achieve the same result as shown in the above example.

```
[jerry]$ awk -f command.awk marks.txt
```

On executing this code, you get the following result:

```
1)    Amit     Physics    80
2)    Rahul    Maths      90
3)    Shyam    Biology    87
4)    Kedar    English    85
5)    Hari     History    89
```

AWK Standard Options

AWK supports the following standard options which can be provided from the command line.

The -v Option

This option assigns a value to a variable. It allows assignment before the program execution. The following example describes the usage of the -v option.

```
[jerry]$ awk -v name=Jerry 'BEGIN{printf "Name = %s\n", name}'
```

On executing this code, you get the following result:

```
Name = Jerry
```

The –dump-variables[=file] Option

It prints a sorted list of global variables and their final values to file. The default file is **awkvars.out**.

```
[jerry]$ awk --dump-variables ''
[jerry]$ cat awkvars.out
```

On executing this code, you get the following result:

```
ARGC: 1
ARGIND: 0
```


tutorialspoint
SIMPLYEASYLEARNING

```
ARGV: array, 1 elements
BINMODE: 0
CONVFMT: "%.6g"
ERRNO: ""
FIELDWIDTHS: ""
FILENAME: ""
FNR: 0
FPAT: "[^[:space:]]+"
FS: " "
IGNORECASE: 0
LINT: 0
NF: 0
NR: 0
OFMT: "%.6g"
OFS: " "
ORS: "\n"
RLENGTH: 0
RS: "\n"
RSTART: 0
RT: ""
SUBSEP: "\034"
TEXTDOMAIN: "messages"
```

The –help Option

This option prints the help message on standard output.

```
[jerry]$ awk --help
```

On executing this code, you get the following result:

```
Usage: awk [POSIX or GNU style options] -f progfile [--] file ...
Usage: awk [POSIX or GNU style options] [--] 'program' file ...
POSIX options:          GNU long options: (standard)
     -f progfile        --file=progfile
     -F fs              --field-separator=fs
     -v var=val         --assign=var=val
```


tutorialspoint
SIMPLYEASYLEARNING

```
Short options:        GNU long options: (extensions)
     -b               --characters-as-bytes
     -c               --traditional
     -C               --copyright
     -d[file]         --dump-variables[=file]
     -e 'program-text' --source='program-text'
     -E file                --exec=file
     -g               --gen-pot
     -h               --help
     -L [fatal]       --lint[=fatal]
     -n               --non-decimal-data
     -N               --use-lc-numeric
     -O               --optimize
     -p[file]         --profile[=file]
     -P               --posix
     -r               --re-interval
     -S               --sandbox
     -t               --lint-old
     -V               --version
```

The –lint[=fatal] Option

This option enables checking of non-portable or dubious constructs. When an argument **fatal** is provided, it treats warning messages as errors. The following example demonstrates this:

```
[jerry]$ awk --lint '' /bin/ls
```

On executing this code, you get the following result:

```
awk: cmd. line:1: warning: empty program text on command line
awk: cmd. line:1: warning: source file does not end in newline
awk: warning: no program text at all!
```

The –posix Option

This option turns on strict POSIX compatibility, in which all common and gawk-specific extensions are disabled.

tutorialspoint
SIMPLYEASYLEARNING

The –profile[=file] Option

This option generates a pretty-printed version of the program in a file. The default file is **awkprof.out**. The following example demonstrates this:

```
[jerry]$ awk --profile 'BEGIN{printf"---|Header|--\n"} {print} END{printf"-
--|Footer|---\n"}' marks.txt > /dev/null
[jerry]$ cat awkprof.out
```

On executing this code, you get the following result:

```
# gawk profile, created Sun Oct 26 19:50:48 2014

    # BEGIN block(s)
    BEGIN {
          printf "---|Header|--\n"
    }

    # Rule(s)
    {
          print $0
    }

    # END block(s)
    END {
          printf "---|Footer|---\n"
    }
```

The –traditional Option

This option disables all gawk-specific extensions.

The –version Option

This option displays the version information of the AWK program.

```
[jerry]$ awk --version
```

When this code is executed, it produces the following result:

```
GNU Awk 4.0.1
```


tutorialspoint
SIMPLYEASYLEARNING

Copyright (C) 1989, 1991-2012 Free Software Foundation.

tutorialspoint
SIMPLYEASYLEARNING

5. BASIC EXAMPLES

This chapter describes several useful AWK commands and their appropriate examples. Consider a text file **marks.txt** to be processed with the following content:

```
1)    Amit     Physics    80
2)    Rahul    Maths      90
3)    Shyam    Biology    87
4)    Kedar    English    85
5)    Hari     History    89
```

Printing Column or Field

You can instruct AWK to print only certain columns from the input field. The following example demonstrates this:

```
[jerry]$ awk '{print $3 "\t" $4}' marks.txt
```

On executing this code, you get the following result:

```
Physics    80
Maths      90
Biology    87
English    85
History    89
```

In the file **marks.txt**, the third column contains the subject name and the fourth column contains the marks obtained in a particular subject. Let us print these two columns using AWK print command. In the above example, **$3** and **$4** represent the third and the fourth fields respectively from the input record.

Printing All Lines

By default, AWK prints all the lines that match pattern.

```
[jerry]$ awk '/a/ {print $0}' marks.txt
```

On executing this code, you get the following result:

```
2)    Rahul    Maths      90
3)    Shyam    Biology    87
```


tutorialspoint
SIMPLYEASYLEARNING

```
4)    Kedar    English    85
5)    Hari     History    89
```

In the above example, we are searching form pattern **a**. When a pattern match succeeds, it executes a command from the body block. In the absence of a body block - default action is taken which is print the record. Hence, the following command produces the same result:

```
[jerry]$ awk '/a/' marks.txt
```

Printing Columns by Pattern

When a pattern match succeeds, AWK prints the entire record by default. But you can instruct AWK to print only certain fields. For instance, the following example prints the third and fourth field when a pattern match succeeds.

```
[jerry]$ awk '/a/ {print $3 "\t" $4}' marks.txt
```

On executing this code, you get the following result:

```
Maths      90
Biology    87
English    85
History    89
```

Printing Column in Any Order

You can print columns in any order. For instance, the following example prints the fourth column followed by the third column.

```
[jerry]$ awk '/a/ {print $4 "\t" $3}' marks.txt
```

On executing the above code, you get the following result:

```
90    Maths
87    Biology
85    English
89    History
```

Counting and Printing Matched Pattern

Let us see an example where you can count and print the number of lines for which a pattern match succeeded.

tutorialspoint
SIMPLYEASYLEARNING

```
[jerry]$ awk '/a/{++cnt} END {print "Count = ", cnt}' marks.txt
```

On executing this code, you get the following result:

```
Count = 4
```

In this example, we increment the value of counter when a pattern match succeeds and we print this value in the END block. Note that unlike other programming languages, there is no need to declare a variable before using it.

Printing Lines with More than 18 Characters

Let us print only those lines that contain more than 18 characters.

```
[jerry]$ awk 'length($0) > 18' marks.txt
```

On executing this code, you get the following result:

```
3)    Shyam    Biology    87
4)    Kedar    English    85
```

AWK provides a built-in **length** function that returns the length of the string. **$0** variable stores the entire line and in the absence of a body block, default action is taken, i.e., the print action. Hence, if a line has more than 18 characters, then the comparison results true and the line gets printed.

tutorialspoint
SIMPLYEASYLEARNING

6. BUILT-IN VARIABLES

AWK provides several built-in variables. They play an important role while writing AWK scripts. This chapter demonstrates the usage of built-in variables.

Standard AWK Variables

The standard AWK variables are discussed below.

ARGC

It implies the number of arguments provided at the command line.

```
[jerry]$ awk 'BEGIN {print "Arguments =", ARGC}' One Two Three Four
```

On executing this code, you get the following result:

```
Arguments = 5
```

But why AWK shows 5 when you passed only 4 arguments? Just check the following example to clear your doubt.

ARGV

It is an array that stores the command-line arguments. The array's valid index ranges from 0 to ARGC-1.

```
[jerry]$ awk 'BEGIN { for (i = 0; i < ARGC - 1; ++i) { printf "ARGV[%d] =
%s\n", i, ARGV[i] } }' one two three four
```

On executing this code, you get the following result:

```
ARGV[0] = awk
ARGV[1] = one
ARGV[2] = two
ARGV[3] = three
```

CONVFMT

It represents the conversion format for numbers. Its default value is **%.6g**.

```
[jerry]$ awk 'BEGIN { print "Conversion Format =", CONVFMT }'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
Conversion Format = %.6g
```

ENVIRON

It is an associative array of environment variables.

```
[jerry]$ awk 'BEGIN { print ENVIRON["USER"] }'
```

On executing this code, you get the following result:

```
jerry
```

To find names of other environment variables, use **env** command.

FILENAME

It represents the current file name.

```
[jerry]$ awk 'END {print FILENAME}' marks.txt
```

On executing this code, you get the following result:

```
marks.txt
```

Please note that FILENAME is undefined in the BEGIN block.

FS

It represents the (input) field separator and its default value is space. You can also change this by using **-F** command line option.

```
[jerry]$ awk 'BEGIN {print "FS = " FS}' | cat -vte
```

On executing this code, you get the following result:

```
FS =  $
```

NF

It represents the number of fields in the current record. For instance, the following example prints only those lines that contain more than two fields.

```
[jerry]$ echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NF > 2'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
One Two Three
One Two Three Four
```

NR

It represents the number of the current record. For instance, the following example prints the record if the current record contains less than three fields.

```
[jerry]$ echo -e "One Two\nOne Two Three\nOne Two Three Four" | awk 'NR < 3'
```

On executing this code, you get the following result:

```
One Two
One Two Three
```

FNR

It is similar to NR, but relative to the current file. It is useful when AWK is operating on multiple files. Value of FNR resets with new file.

OFMT

It represents the output format number and its default value is **%.6g**.

```
[jerry]$ awk 'BEGIN {print "OFMT = " OFMT}'
```

On executing this code, you get the following result:

```
OFMT = %.6g
```

OFS

It represents the output field separator and its default value is space.

```
[jerry]$ awk 'BEGIN {print "OFS = " OFS}' | cat -vte
```

On executing this code, you get the following result:

```
OFS =  $
```

ORS

It represents the output record separator and its default value is newline.

tutorialspoint
SIMPLYEASYLEARNING

```
[jerry]$ awk 'BEGIN {print "ORS = " ORS}' | cat -vte
```

On executing the above code, you get the following result:

```
ORS = $
$
```

RLENGTH

It represents the length of the string matched by **match** function. AWK's match function searches for a given string in the input-string.

```
[jerry]$ awk 'BEGIN { if (match("One Two Three", "re")) { print RLENGTH } }'
```

On executing this code, you get the following result:

```
2
```

RS

It represents (input) record separator and its default value is newline.

```
[jerry]$ awk 'BEGIN {print "RS = " RS}' | cat -vte
```

On executing this code, you get the following result:

```
RS = $
$
```

RSTART

It represents the first position in the string matched by **match** function.

```
[jerry]$ awk 'BEGIN { if (match("One Two Three", "Thre")) { print RSTART } }'
```

On executing this code, you get the following result:

```
9
```

SUBSEP

It represents the separator character for array subscripts and its default value is **\034**.

```
[jerry]$ awk 'BEGIN { print "SUBSEP = " SUBSEP }' | cat -vte
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
SUBSEP = ^\$
```

$0

It represents the entire input record.

```
[jerry]$ awk '{print $0}' marks.txt
```

On executing this code, you get the following result:

```
1)    Amit     Physics    80
2)    Rahul    Maths      90
3)    Shyam    Biology    87
4)    Kedar    English    85
5)    Hari     History    89
```

$n

It represents the n^{th} field in the current record where the fields are separated by FS.

```
[jerry]$ awk '{print $3 "\t" $4}' marks.txt
```

On executing this code, you get the following result:

```
Physics    80
Maths      90
Biology    87
English    85
History    89
```

GNU AWK Specific Variables

GNU AWK specific variables are as follows:

ARGIND

It represents the index in ARGV of the current file being processed.

```
[jerry]$ awk '{ print "ARGIND   = ", ARGIND; print "Filename = ",
ARGV[ARGIND] }' junk1 junk2 junk3
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
ARGIND   =  1
Filename =  junk1
ARGIND   =  2
Filename =  junk2
ARGIND   =  3
Filename =  junk3
```

BINMODE

It is used to specify binary mode for all file I/O on non-POSIX systems. Numeric values of 1, 2, or 3 specify that input files, output files, or all files, respectively, should use binary I/O. String values of **r** or **w** specify that input files or output files, respectively, should use binary I/O. String values of **rw** or **wr** specify that all files should use binary I/O.

ERRNO

A string indicates an error when a redirection fails for **getline** or if **close** call fails.

```
[jerry]$ awk 'BEGIN { ret = getline < "junk.txt"; if (ret == -1) print
"Error:", ERRNO }'
```

On executing this code, you get the following result:

```
Error: No such file or directory
```

FIELDWIDTHS

A space separated list of field widths variable is set, GAWK parses the input into fields of fixed width, instead of using the value of the FS variable as the field separator.

IGNORECASE

When this variable is set, GAWK becomes case-insensitive. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN{IGNORECASE=1} /amit/' marks.txt
```

On executing this code, you get the following result:

```
1)    Amit     Physics    80
```


tutorialspoint
SIMPLYEASYLEARNING

LINT

It provides dynamic control of the **--lint** option from the GAWK program. When this variable is set, GAWK prints lint warnings. When assigned the string value **fatal**, lint warnings become fatal errors, exactly like **--lint=fatal**.

```
[jerry]$ awk 'BEGIN {LINT=1; a}'
```

On executing this code, you get the following result:

```
awk: cmd. line:1: warning: reference to uninitialized variable `a'
awk: cmd. line:1: warning: statement has no effect
```

PROCINFO

This is an associative array containing information about the process, such as real and effective UID numbers, process ID number, and so on.

```
[jerry]$ awk 'BEGIN { print PROCINFO["pid"] }'
```

On executing this code, you get the following result:

```
4316
```

TEXTDOMAIN

It represents the text domain of the AWK program. It is used to find the localized translations for the program's strings.

```
[jerry]$ awk 'BEGIN { print TEXTDOMAIN }'
```

On executing this code, you get the following result:

```
messages
```

The above output shows English text due to **en_IN** locale.

tutorialspoint
SIMPLYEASYLEARNING

7. OPERATORS

Like other programming languages, AWK also provides a large set of operators. This chapter explains AWK operators with suitable examples.

Arithmetic Operators

AWK supports the following arithmetic operators:

Addition

It is represented by **plus (+)** symbol which adds two or more numbers. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a + b) = ", (a + b) }'
```

On executing this code, you get the following result:

```
(a + b) =  70
```

Subtraction

It is represented by **minus (-)** symbol which subtracts two or more numbers. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a - b) = ", (a - b) }'
```

On executing this code, you get the following result:

```
(a - b) =  30
```

Multiplication

It is represented by **asterisk (*)** symbol which multiplies two or more numbers. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a * b) = ", (a * b) }'
```

On executing this code, you get the following result:

```
(a * b) =  1000
```


tutorialspoint
SIMPLYEASYLEARNING

Division

It is represented by **slash (/)** symbol which divides two or more numbers. The following example illustrates this:

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a / b) = ", (a / b) }'
```

On executing this code, you get the following result:

```
(a / b) =  2.5
```

Module

It is represented by **percent (%)** symbol which finds the module division of two or more numbers. The following example illustrates this:

```
[jerry]$ awk 'BEGIN { a = 50; b = 20; print "(a % b) = ", (a % b) }'
```

On executing this code, you get the following result:

```
(a % b) =  10
```

Increment and Decrement Operators

AWK supports the following increment and decrement operators:

Pre-Increment

It is represented by **++**. It increments the value of an operand by **1**. This operator first increments the value of the operand, then returns the incremented value. For instance, in the following example, this operator sets the value of both the operands, a and b, to 11.

```
awk 'BEGIN { a = 10; b = ++a; printf "a = %d, b = %d\n", a, b }'
```

On executing this code, you get the following result:

```
a = 11, b = 11
```

Pre-Decrement

It is represented by **--**. It decrements the value of an operand by **1**. This operator first decrements the value of the operand, then returns the decremented value. For instance, in the following example, this operator sets the value of both the operands, a and b, to 9.

tutorialspoint
SIMPLYEASYLEARNING

```
[jerry]$ awk 'BEGIN { a = 10; b = --a; printf "a = %d, b = %d\n", a, b }'
```

On executing the above code, you get the following result:

```
a = 9, b = 9
```

Post-Increment

It is represented by **++**. It increments the value of an operand by **1**. This operator first returns the value of the operand, then it increments its value. For instance, the following code sets the value of operand a to 11 and b to 10.

```
[jerry]$ awk 'BEGIN { a = 10; b = a++; printf "a = %d, b = %d\n", a, b }'
```

On executing this code, you get the following result:

```
a = 11, b = 10
```

Post-Decrement

It is represented by **--**. It decrements the value of an operand by **1**. This operator first returns the value of the operand, then it decrements its value. For instance, the following code sets the value of the operand a to 9 and b to 10.

```
[jerry]$ awk 'BEGIN { a = 10; b = a--; printf "a = %d, b = %d\n", a, b }'
```

On executing this code, you get the following result:

```
a = 9, b = 10
```

Assignment Operators

AWK supports the following assignment operators:

Simple Assignment

It is represented by **=**. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { name = "Jerry"; print "My name is", name }'
```

On executing this code, you get the following result:

```
My name is Jerry
```


tutorialspoint
SIMPLYEASYLEARNING

Shorthand Addition

It is represented by **+=**. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { cnt=10; cnt += 10; print "Counter =", cnt }'
```

On executing this code, you get the following result:

```
Counter = 20
```

In the above example, the first statement assigns value 10 to the variable **cnt**. In the next statement, the shorthand operator increments its value by 10.

Shorthand Subtraction

It is represented by **-=**. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { cnt=100; cnt -= 10; print "Counter =", cnt }'
```

On executing this code, you get the following result:

```
Counter = 90
```

In the above example, the first statement assigns value 100 to the variable **cnt**. In the next statement, the shorthand operator decrements its value by 10.

Shorthand Multiplication

It is represented by ***=**. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { cnt=10; cnt *= 10; print "Counter =", cnt }'
```

On executing this code, you get the following result:

```
Counter = 100
```

In the above example, the first statement assigns value 10 to the variable **cnt**. In the next statement, the shorthand operator multiplies its value by 10.

Shorthand Division

It is represented by **/=**. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { cnt=100; cnt /= 5; print "Counter =", cnt }'
```

On executing this code, you get the following result:

```
Counter = 20
```


tutorialspoint
SIMPLYEASYLEARNING

In the above example, the first statement assigns value 100 to the variable **cnt**. In the next statement, the shorthand operator divides it by 5.

Shorthand Modulo

It is represented by **%=**. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { cnt=100; cnt %= 8; print "Counter =", cnt }'
```

On executing this code, you get the following result:

```
Counter = 4
```

Shorthand Exponential

It is represented by **^=**. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { cnt=2; cnt ^= 4; print "Counter =", cnt }'
```

On executing this code, you get the following result:

```
Counter = 16
```

The above example raises the value of **cnt** by 4.

Shorthand Exponential

It is represented by ****=**. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { cnt=2; cnt **= 4; print "Counter =", cnt }'
```

On executing this code, you get the following result:

```
Counter = 16
```

This example also raises the value of **cnt** by 4.

Relational Operators

AWK supports the following relational operators:

Equal to

It is represented by **==**. It returns true if both operands are equal, otherwise it returns false. The following example demonstrates this:

```
awk 'BEGIN { a = 10; b = 10; if (a == b) print "a == b" }'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
a == b
```

Not Equal to

It is represented by **!=**. It returns true if both operands are unequal, otherwise it returns false.

```
[jerry]$ awk 'BEGIN { a = 10; b = 20; if (a != b) print "a != b" }'
```

On executing this code, you get the following result:

```
a != b
```

Less Than

It is represented by **<**. It returns true if the left-side operand is less than the right-side operand; otherwise it returns false.

```
[jerry]$ awk 'BEGIN { a = 10; b = 20; if (a < b) print "a < b" }'
```

On executing this code, you get the following result:

```
a < b
```

Less Than or Equal to

It is represented by **<=**. It returns true if the left-side operand is less than or equal to the right-side operand; otherwise it returns false.

```
[jerry]$ awk 'BEGIN { a = 10; b = 10; if (a <= b) print "a <= b" }'
```

On executing this code, you get the following result:

```
a <= b
```

Greater Than

It is represented by **>**. It returns true if the left-side operand is greater than the right-side operand, otherwise it returns false.

```
[jerry]$ awk 'BEGIN { a = 10; b = 20; if (b > a ) print "b > a" }'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing the above code, you get the following result:

```
b > a
```

Greater Than or Equal to

It is represented by **>=**. It returns true if the left-side operand is greater than or equal to the right-side operand; otherwise it returns false.

```
[jerry]$ awk 'BEGIN { a = 10; b = 10; if (a >= b) print "a >= b" }'
```

On executing this code, you get the following result:

```
b >= a
```

Logical Operators

AWK supports the following logical operators:

Logical AND

It is represented by **&&**. Its syntax is as follows:

```
expr1 && expr2
```

It evaluates to true if both expr1 and expr2 evaluate to true; otherwise it returns false. expr2 is evaluated if and only if expr1 evaluates to true. For instance, the following example checks whether the given single digit number is in octal format or not.

```
[jerry]$ awk 'BEGIN {num = 5; if (num >= 0 && num <= 7) printf "%d is in
octal format\n", num }'
```

On executing this code, you get the following result:

```
5 is in octal format
```

Logical OR

It is represented by **||**. The syntax of Logical OR is:

```
expr1 || expr2
```

It evaluates to true if either expr1 or expr2 evaluates to true; otherwise it returns false. expr2 is evaluated if and only if expr1 evaluates to false. The following example demonstrates this:

tutorialspoint
SIMPLYEASYLEARNING

```
[jerry]$ awk 'BEGIN {ch = "\n"; if (ch == " " || ch == "\t" || ch == "\n")
print "Current character is whitespace." }'
```

On executing this code, you get the following result:

```
Current character is whitespace.
```

Logical NOT

It is represented by **exclamation mark (!)**. The following example demonstrates this:

```
! expr1
```

It returns the logical compliment of expr1. If expr1 evaluates to true, it returns 0; otherwise it returns 1. For instance, the following example checks whether a string is empty or not.

```
[jerry]$ awk 'BEGIN { name = ""; if (! length(name)) print "name is empty
string." }'
```

On executing this code, you get the following result:

```
name is empty string.
```

Ternary Operator

We can easily implement a condition expression using ternary operator. The following example demonstrates this:

```
condition expression ? statement1 : statement2
```

When the condition expression returns true, statement1 gets executed; otherwise statement2 is executed. For instance, the following example finds the largest number from two given numbers.

```
[jerry]$ awk 'BEGIN { a = 10; b = 20; (a > b) ? max = a : max = b; print
"Max =", max}'
```

On executing this code, you get the following result:

```
Max = 20
```


tutorialspoint
SIMPLYEASYLEARNING

Unary Operators

AWK supports the following unary operators:

Unary Plus

It is represented by **+**. It multiplies a single operand by **+1**.

```
[jerry]$ awk 'BEGIN { a = -10; a = +a; print "a =", a }'
```

On executing this code, you get the following result:

```
a = -10
```

Unary Minus

It is represented by **-**. It multiplies a single operand by **-1**.

```
[jerry]$ awk 'BEGIN { a = -10; a = -a; print "a =", a }'
```

On executing this code, you get the following result:

```
a = 10
```

Exponential Operators

There are two formats of exponential operators:

Exponential Format 1

It is an exponential operator that raises the value of an operand. For instance, the following example raises the value of 10 by 2.

```
[jerry]$ awk 'BEGIN { a = 10; a = a ^ 2; print "a =", a }'
```

On executing this code, you get the following result:

```
a = 100
```

Exponential Format 2

It is an exponential operator that raises the value of an operand. For instance, the following example raises the value of 10 by 2.

```
[jerry]$ awk 'BEGIN { a = 10; a = a ** 2; print "a =", a }'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
a = 100
```

String Concatenation Operator

Space is a string concatenation operator that merges two strings. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN { str1="Hello, "; str2="World"; str3 = str1 str2; print
str3 }'
```

On executing this code, you get the following result:

```
Hello, World
```

Array Membership Operator

It is represented by **in**. It is used while accessing array elements. The following example prints array elements using this operator.

```
[jerry]$ awk 'BEGIN { arr[0] = 1; arr[1] = 2; arr[2] = 3; for (i in arr)
printf "arr[%d] = %d\n", i, arr[i] }'
```

On executing this code, you get the following result:

```
arr[0] = 1
arr[1] = 2
arr[2] = 3
```

Regular Expression Operators

This example explains the two forms of regular expressions operators.

Match

It is represented as **~**. It looks for a field that contains the match string. For instance, the following example prints the lines that contain the pattern **9**.

```
[jerry]$ awk '$0 ~ 9' marks.txt
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
2)   Rahul Maths 90
5)   Hari  History     89
```

Not Match

It is represented as **!~**. It looks for a field that does not contain the match string. For instance, the following example prints the lines that do not contain the pattern **9**.

```
[jerry]$ awk '$0 !~ 9' marks.txt
```

On executing this code, you get the following result:

```
1)   Amit  Physics     80
3)   Shyam Biology     87
4)   Kedar English     85
```


tutorialspoint
SIMPLYEASYLEARNING

8. REGULAR EXPRESSIONS

AWK is very powerful and efficient in handling regular expressions. A number of complex tasks can be solved with simple regular expressions. Any command-line expert knows the power of regular expressions.

This chapter covers standard regular expressions with suitable examples.

Dot

It matches any single character except the end of line character. For instance, the following example matches **fin, fun, fan,** etc.

```
[jerry]$ echo -e "cat\nbat\nfun\nfin\nfan" | awk '/f.n/'
```

On executing the above code, you get the following result:

```
fun
fin
fan
```

Start of Line

It matches the start of line. For instance, the following example prints all the lines that start with pattern **The**.

```
[jerry]$ echo -e "This\nThat\nThere\nTheir\nthese" | awk '/^The/'
```

On executing this code, you get the following result:

```
There
Their
```

End of Line

It matches the end of line. For instance, the following example prints the lines that end with the letter **n**.

```
[jerry]$ echo -e "knife\nknow\nfun\nfin\nfan\nnine" | awk '/n$/'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
fun
fin
fan
```

Match Character Set

It is used to match only one out of several characters. For instance, the following example matches pattern **Call** and **Tall** but not **Ball**.

```
[jerry]$ echo -e "Call\nTall\nBall" | awk '/[CT]all/'
```

On executing this code, you get the following result:

```
Call
Tall
```

Exclusive Set

In exclusive set, the carat negates the set of characters in the square brackets. For instance, the following example prints only **Ball**.

```
[jerry]$ echo -e "Call\nTall\nBall" | awk '/[^CT]all/'
```

On executing this code, you get the following result:

```
Ball
```

Alteration

A vertical bar allows regular expressions to be logically ORed. For instance, the following example prints **Ball** and **Call**.

```
[jerry]$ echo -e "Call\nTall\nBall\nSmall\nShall" | awk '/Call|Ball/'
```

On executing this code, you get the following result:

```
Call
Ball
```


tutorialspoint
SIMPLYEASYLEARNING

Zero or One Occurrence

It matches zero or one occurrence of the preceding character. For instance, the following example matches **Colour** as well as **Color**. We have made **u** as an optional character by using **?**.

```
[jerry]$ echo -e "Colour\nColor" | awk '/Colou?r/'
```

On executing this code, you get the following result:

```
Colour
Color
```

Zero or More Occurrence

It matches zero or more occurrences of the preceding character. For instance, the following example matches **ca, cat, catt,** and so on.

```
[jerry]$ echo -e "ca\ncat\ncatt" | awk '/cat*/'
```

On executing this code, you get the following result:

```
ca
cat
catt
```

One or More Occurrence

It matches one or more occurrences of the preceding character. For instance, the following example matches one or more occurrences of **2**.

```
[jerry]$ echo -e "111\n22\n123\n234\n456\n222"  | awk '/2+/'
```

On executing this code, you get the following result:

```
22
123
234
222
```


tutorialspoint
SIMPLYEASYLEARNING

Grouping

Parentheses () are used for grouping and the character **|** is used for alternatives. For instance, the following regular expression matches the lines containing either **Apple Juice** or **Apple Cake**.

```
[jerry]$ echo -e "Apple Juice\nApple Pie\nApple Tart\nApple Cake" | awk
'/Apple (Juice|Cake)/'
```

On executing this code, you get the following result:

```
Apple Juice
Apple Cake
```


tutorialspoint
SIMPLYEASYLEARNING

9. ARRAYS

AWK has associative arrays and one of the best thing about it is – the indexes need not to be continuous set of number; you can use either string or number as an array index. Also, there is no need to declare the size of an array in advance – arrays can expand/shrink at runtime.

Its syntax is as follows:

```
array_name[index]=value
```

Where **array_name** is the name of array, **index** is the array index, and **value** is any value assigning to the element of the array.

Creating Array

To gain more insight on array, let us create and access the elements of an array.

```
[jerry]$ awk 'BEGIN {
fruits["mango"]="yellow";
fruits["orange"]="orange"
print fruits["orange"] "\n" fruits["mango"]
}'
```

On executing this code, you get the following result:

```
orange
yellow
```

In the above example, we declare the array as **fruits** whose index is fruit name and the value is the color of the fruit. To access array elements, we use **array_name[index]** format.

Deleting Array Elements

For insertion, we used assignment operator. Similarly, we can use **delete** statement to remove an element from the array. The syntax of delete statement is as follows:

```
delete array_name[index]
```


tutorialspoint
SIMPLYEASYLEARNING

The following example deletes the element **orange**. Hence the command does not show any output.

```
[jerry]$ awk 'BEGIN {
fruits["mango"]="yellow";
fruits["orange"]="orange";
delete fruits["orange"];
print fruits["orange"]
}'
```

Multi-Dimensional Arrays

AWK only supports one-dimensional arrays. But you can easily simulate a multi-dimensional array using the one-dimensional array itself.

For instance, given below is a 3x3 three-dimensional array:

```
100 200 300
400 500 600
700 800 900
```

In the above example, array[0][0] stores 100, array[0][1] stores 200, and so on. To store 100 at array location [0][0], we can use the following syntax:

```
array["0,0"] = 100
```

Though we gave **0,0** as index, these are not two indexes. In reality, it is just one index with the string **0,0**.

The following example simulates a 2-D array:

```
[jerry]$ awk 'BEGIN {
array["0,0"] = 100;
array["0,1"] = 200;
array["0,2"] = 300;
array["1,0"] = 400;
array["1,1"] = 500;
array["1,2"] = 600;
# print array elements
print "array[0,0] = " array["0,0"];
print "array[0,1] = " array["0,1"];
```


tutorialspoint
SIMPLYEASYLEARNING

```
print "array[0,2] = " array["0,2"];
print "array[1,0] = " array["1,0"];
print "array[1,1] = " array["1,1"];
print "array[1,2] = " array["1,2"];
}'
```

On executing this code, you get the following result:

```
array[0,0] = 100
array[0,1] = 200
array[0,2] = 300
array[1,0] = 400
array[1,1] = 500
array[1,2] = 600
```

You can also perform a variety of operations on an array such as sorting its elements/indexes. For that purpose, you can use **assort** and **asorti** functions.

tutorialspoint
SIMPLYEASYLEARNING

10. CONTROL FLOW

Like other programming languages, AWK provides conditional statements to control the flow of a program. This chapter explains AWK's control statements with suitable examples.

If Statement

It simply tests the condition and performs certain actions depending upon the condition. Given below is the syntax of **if** statement:

```
if (condition)
    action
```

We can also use a pair of curly braces as given below to execute multiple actions:

```
if (condition)
{
    action-1
    action-1
    .
    .
    action-n
}
```

For instance, the following example checks whether a number is even or not:

```
[jerry]$ awk 'BEGIN {num = 10; if (num % 2 == 0) printf "%d is even
number.\n", num }'
```

On executing this code, you get the following result:

```
10 is even number.
```

If-Else Statement

In **if-else** syntax, we can provide a list of actions to be performed when a condition becomes false.

tutorialspoint
SIMPLYEASYLEARNING

The syntax of **if-else** statement is as follows:

```
if (condition)
    action-1
else
    action-2
```

In the above syntax, action-1 is performed when the condition evaluates to true and action-2 is performed when the condition evaluates to false. For instance, the following example checks whether a number is even or not:

```
[jerry]$ awk 'BEGIN {num = 11; if (num % 2 == 0) printf "%d is even
number.\n", num; else printf "%d is odd number.\n", num }'
```

On executing this code, you get the following result:

```
11 is odd number.
```

If-Else-If Ladder

We can easily create an **if-else-if** ladder by using multiple **if-else** statements. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN {
a=30;
if (a==10)
  print "a = 10";
else if (a == 20)
  print "a = 20";
else if (a == 30)
  print "a = 30";
}'
```

On executing this code, you get the following result:

```
a = 30
```


tutorialspoint
SIMPLYEASYLEARNING

11. LOOPS

This chapter explains AWK's loops with suitable example. Loops are used to execute a set of actions in a repeated manner. The loop execution continues as long as the loop condition is true.

For Loop

The syntax of **for** loop is:

```
for (initialisation; condition; increment/decrement)
    action
```

Initially, the **for** statement performs initialization action, then it checks the condition. If the condition is true, it executes actions, thereafter it performs increment or decrement operation. The loop execution continues as long as the condition is true. For instance, the following example prints 1 to 5 using **for** loop:

```
[jerry]$ awk 'BEGIN { for (i = 1; i <= 5; ++i) print i }'
```

On executing this code, you get the following result:

```
1
2
3
4
5
```

While Loop

The **while** loop keeps executing the action until a particular logical condition evaluates to true. Here is the syntax of **while** loop:

```
while (condition)
    action
```

AWK first checks the condition; if the condition is true, it executes the action. This process repeats as long as the loop condition evaluates to true. For instance, the following example prints 1 to 5 using **while** loop:

```
[jerry]$ awk 'BEGIN {i = 1; while (i < 6) { print i; ++i } }'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
1
2
3
4
5
```

Do-While Loop

The **do-while** loop is similar to the while loop, except that the test condition is evaluated at the end of the loop. Here is the syntax of **do-while** loop:

```
do
    action
while (condition)
```

In a **do-while** loop, the action statement gets executed at least once even when the condition statement evaluates to false. For instance, the following example prints 1 to 5 numbers using **do-while** loop:

```
[jerry]$ awk 'BEGIN {i = 1; do { print i; ++i } while (i < 6) }'
```

On executing this code, you get the following result:

```
1
2
3
4
5
```

Break Statement

As its name suggests, it is used to end the loop execution. Here is an example which ends the loop when the sum becomes greater than 50.

```
[jerry]$ awk 'BEGIN {sum = 0; for (i = 0; i < 20; ++i) { sum += i; if (sum
> 50) break; else print "Sum =", sum } }'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
Sum = 0
Sum = 1
Sum = 3
Sum = 6
Sum = 10
Sum = 15
Sum = 21
Sum = 28
Sum = 36
Sum = 45
```

Continue Statement

The **continue** statement is used inside a loop to skip to the next iteration of the loop. It is useful when you wish to skip the processing of some data inside the loop. For instance, the following example uses **continue** statement to print the even numbers between 1 to 20.

```
[jerry]$ awk 'BEGIN {for (i = 1; i <= 20; ++i) {if (i % 2 == 0) print i ;
else continue} }'
```

On executing this code, you get the following result:

```
2
4
6
8
10
12
14
16
18
20
```


tutorialspoint
SIMPLYEASYLEARNING

Exit Statement

It is used to stop the execution of the script. It accepts an integer as an argument which is the exit status code for AWK process. If no argument is supplied, **exit** returns status zero. Here is an example that stops the execution when the sum becomes greater than 50.

```
[jerry]$ awk 'BEGIN {sum = 0; for (i = 0; i < 20; ++i) { sum += i; if (sum
> 50) exit(10); else print "Sum =", sum } }'
```

On executing this code, you get the following result:

```
Sum = 0
Sum = 1
Sum = 3
Sum = 6
Sum = 10
Sum = 15
Sum = 21
Sum = 28
Sum = 36
Sum = 45
```

Let us check the return status of the script.

```
[jerry]$ echo $?
```

On executing this code, you get the following result:

```
10
```


tutorialspoint
SIMPLYEASYLEARNING

12. BUILT-IN FUNCTIONS

AWK has a number of functions built into it that are always available to the programmer. This chapter describes Arithmetic, String, Time, Bit manipulation, and other miscellaneous functions with suitable examples.

Arithmetic Functions

AWK has the following built-in arithmetic functions:

atan2(y, x)

It returns the arctangent of (y/x) in radians. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN {
  PI = 3.14159265
  x = -10
  y = 10
  result = atan2 (y,x) * 180 / PI;

  printf "The arc tangent for (x=%f, y=%f) is %f degrees\n", x, y, result
}'
```

On executing this code, you get the following result:

```
The arc tangent for (x=-10.000000, y=10.000000) is 135.000000 degrees
```

cos(expr)

This function returns the cosine of **expr**, which is expressed in radians. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN {
  PI = 3.14159265
  param = 60
  result = cos(param * PI / 180.0);

  printf "The cosine of %f degrees is %f.\n", param, result
```

```
}'
```

On executing this code, you get the following result:

```
The cosine of 60.000000 degrees is 0.500000.
```

exp(expr)

This function is used to find the exponential value of a variable.

```
[jerry]$ awk 'BEGIN {
  param = 5
  result = exp(param);

  printf "The exponential value of %f is %f.\n", param, result
}'
```

On executing this code, you get the following result:

```
The exponential value of 5.000000 is 148.413159.
```

int(expr)

This function truncates the **expr** to an integer value. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN {
  param = 5.12345
  result = int(param)

  print "Truncated value =", result
}'
```

On executing this code, you get the following result:

```
Truncated value = 5
```

log(expr)

This function calculates the natural logarithm of a variable.

```
[jerry]$ awk 'BEGIN {
```


tutorialspoint
SIMPLYEASYLEARNING

```
  param = 5.5
  result = log (param)

  printf "log(%f) = %f\n", param, result
}'
```

On executing this code, you get the following result:

```
log(5.500000) = 1.704748
```

rand

This function returns a random number N, between 0 and 1, such that 0 <= N < 1. For instance, the following example generates three random numbers:

```
[jerry]$ awk 'BEGIN {
  print "Random num1 =" , rand()
  print "Random num2 =" , rand()
  print "Random num3 =" , rand()
}'
```

On executing this code, you get the following result:

```
Random num1 = 0.237788
Random num2 = 0.291066
Random num3 = 0.845814
```

sin(expr)

This function returns the sine of **expr**, which is expressed in radians. The following example demonstrates this:

```
[jerry]$ awk 'BEGIN {
  PI = 3.14159265
  param = 30.0
  result = sin(param * PI /180)

  printf "The sine of %f degrees is %f.\n", param, result
}'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
The sine of 30.000000 degrees is 0.500000.
```

sqrt(expr)

This function returns the square root of **expr**.

```
[jerry]$ awk 'BEGIN {
  param = 1024.0
  result = sqrt(param)

  printf "sqrt(%f) = %f\n", param, result
}'
```

On executing this code, you get the following result:

```
sqrt(1024.000000) = 32.000000
```

srand([expr])

This function generates a random number using seed value. It uses **expr** as the new seed for the random number generator. In the absence of expr, it uses the time of day as the seed value.

```
[jerry]$ awk 'BEGIN {
  param = 10

  printf "srand() = %d\n", srand()
  printf "srand(%d) = %d\n", param, srand(param)
}'
On executing this code, you get the following result:
srand() = 1
srand(10) = 1417959587
```


tutorialspoint
SIMPLYEASYLEARNING

String Functions

AWK has the following built-in String functions:

asort(arr [, d [, how]])

This function sorts the contents of **arr** using GAWK's normal rules for comparing values, and replaces the indexes of the sorted values **arr** with sequential integers starting with 1.

```
[jerry]$ awk 'BEGIN {
     arr[0] = "Three"
     arr[1] = "One"
     arr[2] = "Two"

     print "Array elements before sorting:"
     for (i in arr) {
           print arr[i]
     }

     asort(arr)

     print "Array elements after sorting:"
     for (i in arr) {
           print arr[i]
     }
}'
```

On executing this code, you get the following result:

```
Array elements before sorting:
Three
One
Two
Array elements after sorting:
One
Three
Two
```


tutorialspoint
SIMPLYEASYLEARNING

asorti(arr [, d [, how]])

The behavior of this function is the same as that of **asort()**, except that the array indexes are used for sorting.

```
[jerry]$ awk 'BEGIN {
     arr["Two"] = 1
     arr["One"] = 2
     arr["Three"] = 3

     asorti(arr)

     print "Array indices after sorting:"
     for (i in arr) {
           print arr[i]
     }
}'
```

On executing this code, you get the following result:

```
Array indices after sorting:
One
Three
Two
```

gsub(regex, sub, string)

gsub stands for global substitution. It replaces every occurrence of **sub** with **regex**. The third parameter is optional. If it is omitted, then $0 is used.

```
[jerry]$ awk 'BEGIN {
     str = "Hello, World"

     print "String before replacement = " str

     gsub("World", "Jerry", str)

     print "String after replacement = " str
}'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
String before replacement = Hello, World
String after replacement = Hello, Jerry
```

index(str, sub)

It checks whether **sub** is a substring of **str** or not. On success, it returns the position where **sub** starts; otherwise it returns 0. The first character of **str** is at position 1.

```
[jerry]$ awk 'BEGIN {
     str = "One Two Three"
     subs = "Two"

     ret = index(str, subs)

     printf "Substring \"%s\" found at %d location.\n", subs, ret
}'
```

On executing this code, you get the following result:

```
Substring "Two" found at 5 location.
```

length(str)

It returns the length of a string.

```
[jerry]$ awk 'BEGIN {
     str = "Hello, World !!!"

     print "Length = ", length(str)
}'
```

On executing this code, you get the following result:

```
Length = 16
```

match(str, regex)

It returns the index of the first longest match of **regex** in string **str**. It returns 0 if no match found.

tutorialspoint
SIMPLYEASYLEARNING

```
[jerry]$ awk 'BEGIN {
     str = "One Two Three"
     subs = "Two"
     ret = match(str, subs)
     printf "Substring \"%s\" found at %d location.\n", subs, ret
}'
```

On executing this code, you get the following result:

```
Substring "Two" found at 5 location.
```

split(str, arr, regex)

This function splits the string **str** into fields by regular expression **regex** and the fields are loaded into the array **arr**. If **regex** is omitted, then FS is used.

```
[jerry]$ awk 'BEGIN {
     str = "One,Two,Three,Four"
     split(str, arr, ",")
     print "Array contains following values"
     for (i in arr) {
           print arr[i]
     }
}'
```

On executing this code, you get the following result:

```
Array contains following values
One
Two
Three
Four
```

sprintf(format, expr-list)

This function returns a string constructed from **expr-list** according to format.

```
[jerry]$ awk 'BEGIN {
     str = sprintf("%s", "Hello, World !!!")
```


tutorialspoint
SIMPLYEASYLEARNING

```
     print str
}'
```

On executing this code, you get the following result:

```
Hello, World !!!
```

strtonum(str)

This function examines **str** and return its numeric value. If str begins with a leading 0, it is treated as an octal number. If str begins with a leading 0x or 0X, it is taken as a hexadecimal number. Otherwise, assume it is a decimal number.

```
[jerry]$ awk 'BEGIN {
     print "Decimal num = " strtonum("123")
     print "Octal num = " strtonum("0123")
     print "Hexadecimal num = " strtonum("0x123")
}'
```

On executing this code, you get the following result:

```
Decimal num = 123
Octal num = 83
Hexadecimal num = 291
```

sub(regex, sub, string)

This function performs single substitution. It replaces the first occurrence of **sub** with **regex**. The third parameter is optional. If it is omitted, $0 is used.

```
[jerry]$ awk 'BEGIN {
     str = "Hello, World"
     print "String before replacement = " str
     sub("World", "Jerry", str)
     print "String after replacement = " str
}'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
String before replacement = Hello, World
String after replacement = Hello, Jerry
```

substr(str, start, l)

This function returns the substring of string **str**, starting at index **start** of length **l**. If length is omitted, the suffix of **str** starting at index **start** is returned.

```
[jerry]$ awk 'BEGIN {
     str = "Hello, World !!!"
     subs = substr(str, 1, 5)

     print "Substring = " subs
}'
```

On executing this code, you get the following result:

```
Substring = Hello
```

tolower(str)

This function returns a copy of string **str** with all upper-case characters converted to lower-case.

```
[jerry]$ awk 'BEGIN {
     str = "HELLO, WORLD !!!"

     print "Lowercase string = " tolower(str)
}'
```

On executing this code, you get the following result:

```
Lowercase string = hello, world !!!
```

toupper(str)

This function returns a copy of string **str** with all lower-case characters converted to upper case.

tutorialspoint
SIMPLYEASYLEARNING

```
[jerry]$ awk 'BEGIN {
     str = "hello, world !!!"
     print "Uppercase string = " toupper(str)
}'
```

On executing this code, you get the following result:

```
Uppercase string = HELLO, WORLD !!!
```

Time Functions

AWK has the following built-in time functions:

systime

This function returns the current time of the day as the number of seconds since the Epoch (1970-01-01 00:00:00 UTC on POSIX systems).

```
[jerry]$ awk 'BEGIN {
     print "Number of seconds since the Epoch = " systime()
}'
```

On executing this code, you get the following result:

```
Number of seconds since the Epoch = 1418574432
```

mktime(datespec)

This function converts **datespec** string into a timestamp of the same form as returned by systime(). The datespec is a string of the form **YYYY MM DD HH MM SS**.

```
[jerry]$ awk 'BEGIN {
     print "Number of seconds since the Epoch = " mktime("2014 12 14 30 20 10")
}'
```

On executing this code, you get the following result:

```
Number of seconds since the Epoch = 1418604610
```


tutorialspoint
SIMPLYEASYLEARNING

strftime([format [, timestamp[, utc-flag]]])

This function formats timestamps according to the specification in format.

```
[jerry]$ awk 'BEGIN {
     print strftime("Time = %m/%d/%Y %H:%M:%S", systime())
}'
```

On executing this code, you get the following result:

```
Time = 12/14/2014 22:08:42
```

The following time formats are supported by AWK:

Date format specification	Description
%a	The locale's abbreviated weekday name.
%A	The locale's full weekday name.
%b	The locale's abbreviated month name.
%B	The locale's full month name.
%c	The locale's appropriate date and time representation. (This is %A %B %d %T %Y in the C locale.)
%C	The century part of the current year. This is the year divided by 100 and truncated to the next lower integer.
%d	The day of the month as a decimal number (01–31).
%D	Equivalent to specifying %m/%d/%y.
%e	The day of the month, padded with a space if it is only one digit.
%F	Equivalent to specifying %Y-%m-%d. This is the ISO 8601 date format.
%g	The year modulo 100 of the ISO 8601 week number, as a decimal number (00–99). For example, January 1, 1993 is in week 53 of 1992. Thus, the year of its ISO 8601 week number is 1992, even though its year is 1993. Similarly, December 31, 1973 is in week 1 of 1974. Thus, the year of its ISO week number is 1974, even though its year is 1973.
%G	The full year of the ISO week number, as a decimal number.
%h	Equivalent to %b.
%H	The hour (24-hour clock) as a decimal number (00–23).

tutorialspoint
SIMPLYEASYLEARNING

%I	The hour (12-hour clock) as a decimal number (01–12).
%j	The day of the year as a decimal number (001–366).
%m	The month as a decimal number (01–12).
%M	The minute as a decimal number (00–59).
%n	A newline character (ASCII LF).
%p	The locale's equivalent of the AM/PM designations associated with a 12-hour clock.
%r	The locale's 12-hour clock time. (This is %I:%M:%S %p in the C locale.)
%R	Equivalent to specifying %H:%M.
%S	The second as a decimal number (00–60).
%t	A TAB character.
%T	Equivalent to specifying %H:%M:%S.
%u	The weekday as a decimal number (1–7). Monday is day one.
%U	The week number of the year (the first Sunday as the first day of week one) as a decimal number (00–53).
%V	The week number of the year (the first Monday as the first day of week one) as a decimal number (01–53).
%w	The weekday as a decimal number (0–6). Sunday is day zero.
%W	The week number of the year (the first Monday as the first day of week one) as a decimal number (00–53).
%x	The locale's appropriate date representation. (This is %A %B %d %Y in the C locale.)
%X	The locale's appropriate time representation. (This is %T in the C locale.)
%y	The year modulo 100 as a decimal number (00–99).
%Y	The full year as a decimal number (e.g. 2011).
%z	The time-zone offset in a +HHMM format (e.g., the format necessary to produce RFC 822/RFC 1036 date headers).
%Z	The time zone name or abbreviation; no characters if no time zone is determinable.

tutorialspoint
SIMPLYEASYLEARNING

Bit Manipulation Functions

AWK has the following built-in bit manipulation functions:

and

Performs bitwise AND operation.

```
[jerry]$ awk 'BEGIN {
     num1 = 10
     num2 = 6

     printf "(%d AND %d) = %d\n", num1, num2, and(num1, num2)
}'
```

On executing this code, you get the following result:

```
(10 AND 6) = 2
```

compl

It performs bitwise COMPLEMENT operation.

```
[jerry]$ awk 'BEGIN {
     num1 = 10

     printf "compl(%d) = %d\n", num1, compl(num1)
}'
```

On executing this code, you get the following result:

```
compl(10) = 9007199254740981
```

lshift

It performs bitwise LEFT SHIFT operation.

```
[jerry]$ awk 'BEGIN {
     num1 = 10
     printf "lshift(%d) by 1 = %d\n", num1, lshift(num1, 1)
}'
```

On executing this code, you get the following result:

tutorialspoint
SIMPLYEASYLEARNING

```
lshift(10) by 1 = 20
```

rshift

It performs bitwise RIGHT SHIFT operation.

```
[jerry]$ awk 'BEGIN {
     num1 = 10
     printf "rshift(%d) by 1 = %d\n", num1, rshift(num1, 1)
}'
```

On executing this code, you get the following result:

```
rshift(10) by 1 = 5
```

or

It performs bitwise OR operation.

```
[jerry]$ awk 'BEGIN {
     num1 = 10
     num2 = 6
     printf "(%d OR %d) = %d\n", num1, num2, or(num1, num2)
}'
```

On executing this code, you get the following result:

```
(10 OR 6) = 14
```

xor

It performs bitwise XOR operation.

```
[jerry]$ awk 'BEGIN {
     num1 = 10
     num2 = 6
     printf "(%d XOR %d) = %d\n", num1, num2, xor(num1, num2)
}'
```

On executing this code, you get the following result:

tutorialspoint
SIMPLYEASYLEARNING

```
(10 bitwise xor 6) = 12
```

Miscellaneous Functions

AWK has the following miscellaneous functions:

close(expr)

This function closes file of pipe.

```
[jerry]$ awk 'BEGIN {
     cmd = "tr [a-z] [A-Z]"
     print "hello, world !!!" |& cmd
     close(cmd, "to")
     cmd |& getline out
     print out;
     close(cmd);
}'
```

On executing this code, you get the following result:

```
HELLO, WORLD !!!
```

Does the script look cryptic? Let us demystify it.

- The first statement, **cmd = "tr [a-z] [A-Z]"** - is the command to which we establish the two way communication from AWK.
- The next statement, i.e., the **print** command, provides input to the **tr** command. Here **&|** indicates two-way communication.
- The third statement, i.e., **close(cmd, "to")**, closes the **to** process after competing its execution.
- The next statement **cmd |& getline out** stores the output into **out** variable with the aid of getline function.
- The next print statement prints the output and finally the **close** function closes the command.

delete

This function deletes an element from an array. The following example shows the usage of the **close** function:

tutorialspoint
SIMPLYEASYLEARNING

```
[jerry]$ awk 'BEGIN {
     arr[0] = "One"
     arr[1] = "Two"
     arr[2] = "Three"
     arr[3] = "Four"

     print "Array elements before delete operation:"
     for (i in arr) {
           print arr[i]
     }

     delete arr[0]
     delete arr[1]

     print "Array elements after delete operation:"
     for (i in arr) {
           print arr[i]
     }
}'
```

On executing this code, you get the following result:

```
Array elements before delete operation:
One
Two
Three
Four

Array elements after delete operation:
Three
Four
```

exit

This function stops the execution of a script. It also accepts an optional **expr** which becomes AWK's return value. The following example describes the usage of exit function.

tutorialspoint
SIMPLYEASYLEARNING

```
[jerry]$ awk 'BEGIN {
     print "Hello, World !!!"

     exit 10

     print "AWK never executes this statement."
}'
```

On executing this code, you get the following result:

```
Hello, World !!!
```

fflush

This function flushes any buffers associated with open output file or pipe. The following syntax demonstrates the function.

```
fflush([output-expr])
```

If no output-expr is supplied, it flushes the standard output. If output-expr is the null string (""), then it flushes all open files and pipes.

getline

This function instructs AWK to read the next line. The following example reads and displays the **marks.txt** file using getline function.

```
[jerry]$ awk '{getline; print $0}' marks.txt
```

On executing this code, you get the following result:

```
2)   Rahul Maths 90
4)   Kedar English     85
5)   Hari  History     89
```

The script works fine. But where is the first line? Let us find out.

At the start, AWK reads the first line from the file **marks.txt** and stores it into **$0** variable.

In the next statement, we instructed AWK to read the next line using getline. Hence AWK reads the second line and stores it into **$0** variable.

And finally, AWK's **print** statement prints the second line. This process continues until the end of the file.

tutorialspoint
SIMPLYEASYLEARNING

next

The **next** function changes the flow of the program. It causes the current processing of the pattern space to stop. The program reads the next line, and starts executing the commands again with the new line. For instance, the following program does not perform any processing when a pattern match succeeds.

```
[jerry]$ awk '{if ($0 ~/Shyam/) next; print $0}' marks.txt
```

On executing this code, you get the following result:

```
1)   Amit  Physics     80
2)   Rahul Maths 90
4)   Kedar English     85
5)   Hari  History     89
```

nextfile

The **nextfile** function changes the flow of the program. It stops processing the current input file and starts a new cycle through pattern/procedures statements, beginning with the first record of the next file. For instance, the following example stops processing the first file when a pattern match succeeds.

First create two files. Let us say **file1.txt** contains:

```
file1:str1
file1:str2
file1:str3
file1:str4
```

And **file2.txt** contains:

```
file2:str1
file2:str2
file2:str3
file2:str4
```

Now let us use the nextfile function.

```
[jerry]$ awk '{ if ($0 ~ /file1:str2/) nextfile; print $0 }' file1.txt
file2.txt
```

On executing this code, you get the following result:

tutorialspoint
SIMPLYEASYLEARNING

```
file1:str1
file2:str1
file2:str2
file2:str3
file2:str4
```

return

This function can be used within a user-defined function to return the value. Please note that the return value of a function is undefined if expr is not provided. The following example describes the usage of the return function.

First, create a **functions.awk** file containing AWK command as shown below:

```
function addition(num1, num2)
{
     result = num1 + num2

     return result
}

BEGIN {
     res = addition(10, 20)
     print "10 + 20 = " res
}
```

On executing this code, you get the following result:

```
10 + 20 = 30
```

system

This function executes the specified command and returns its exit status. A return status 0 indicates that a command execution has succeeded. A non-zero value indicates a failure of command execution. For instance, the following example displays the current date and also shows the return status of the command.

```
[jerry]$ awk 'BEGIN { ret = system("date"); print "Return value = " ret }'
```

On executing this code, you get the following result:

```
Sun Dec 21 23:16:07 IST 2014
```


tutorialspoint
SIMPLYEASYLEARNING

```
Return value = 0
```


tutorialspoint
SIMPLYEASYLEARNING

13. USER-DEFINED FUNCTIONS

Functions are basic building blocks of a program. AWK allows us to define our own functions. A large program can be divided into functions and each function can be written/tested independently. It provides re-usability of code.

Given below is the general format of a user-defined function:

```
function function_name(argument1, argument2, ...)
{
     function body
}
```

In this syntax, the **function_name** is the name of the user-defined function. Function name should begin with a letter and the rest of the characters can be any combination of numbers, alphabetic characters, or underscore. AWK's reserve words cannot be used as function names.

Functions can accept multiple arguments separated by comma. Arguments are not mandatory. You can also create a user-defined function without any argument.

function body consists of one or more AWK statements.

Let us write two functions that calculate the minimum and the maximum number and call these functions from another function called **main**. The **functions.awk** file contains:

```
# Returns minimum number
function find_min(num1, num2)
{
  if (num1 < num2)
    return num1
  return num2
}

# Returns maximum number
function find_max(num1, num2)
{
  if (num1 > num2)
    return num1
```


tutorialspoint
SIMPLYEASYLEARNING

```
  return num2
}

# Main function
function main(num1, num2)
{
  # Find minimum number
  result = find_min(10, 20)
  print "Minimum =", result

  # Find maximum number
  result = find_max(10, 20)
  print "Maximum =", result
}

# Script execution starts here
BEGIN {
  main(10, 20)
}
```

On executing this code, you get the following result:

```
Minimum = 10
Maximum = 20
```


tutorialspoint
SIMPLYEASYLEARNING

14. OUTPUT REDIRECTION

So far, we displayed data on standard output stream. We can also redirect data to a file. A redirection appears after the **print** or **printf** statement. Redirections in AWK are written just like redirection in shell commands, except that they are written inside the AWK program. This chapter explains redirection with suitable examples.

Redirection Operator

The syntax of the redirection operator is:

```
print DATA > output-file
```

It writes the data into the **output-file**. If the output-file does not exist, then it creates one. When this type of redirection is used, the output-file is erased before the first output is written to it. Subsequent write operations to the same output-file do not erase the output-file, but append to it. For instance, the following example writes **Hello, World !!!** to the file.

Let us create a file with some text data.

```
[jerry]$ echo "Old data" > /tmp/message.txt
[jerry]$ cat /tmp/message.txt
```

On executing this code, you get the following result:

```
Old data
```

Now let us redirect some contents into it using AWK's redirection operator.

```
[jerry]$ awk 'BEGIN { print "Hello, World !!!" > "/tmp/message.txt" }'
[jerry]$ cat /tmp/message.txt
```

On executing this code, you get the following result:

```
Hello, World !!!
```

Append Operator

The syntax of append operator is as follows:

```
print DATA >> output-file
```


tutorialspoint
SIMPLYEASYLEARNING

It appends the data into the **output-file**. If the output-file does not exist, then it creates one. When this type of redirection is used, new contents are appended at the end of file. For instance, the following example appends **Hello, World !!!** to the file.

Let us create a file with some text data.

```
[jerry]$ echo "Old data" > /tmp/message.txt
[jerry]$ cat /tmp/message.txt
```

On executing this code, you get the following result:

```
Old data
```

Now let us append some contents to it using AWK's append operator.

```
[jerry]$ awk 'BEGIN { print "Hello, World !!!" >> "/tmp/message.txt" }'
[jerry]$ cat /tmp/message.txt
```

On executing this code, you get the following result:

```
Old data
Hello, World !!!
```

Pipe

It is possible to send output to another program through a pipe instead of using a file. This redirection opens a pipe to command, and writes the values of items through this pipe to another process to execute the command. The redirection argument command is actually an AWK expression. Here is the syntax of pipe:

```
print items | command
```

Let us use **tr** command to convert lowercase letters to uppercase.

```
[jerry]$ awk 'BEGIN { print "hello, world !!!" | "tr [a-z] [A-Z]" }'
```

On executing this code, you get the following result:

```
HELLO, WORLD !!!
```

Two-Way Communication

AWK can communicate to an external process using **|&**, which is two-way communication. For instance, the following example uses **tr** command to convert lowercase letters to uppercase. Our **command.awk** file contains:

tutorialspoint
SIMPLYEASYLEARNING

```
BEGIN {
     cmd = "tr [a-z] [A-Z]"
     print "hello, world !!!" |& cmd
     close(cmd, "to")
     cmd |& getline out
     print out;
     close(cmd);
}
```

On executing this code, you get the following result:

```
HELLO, WORLD !!!
```

Does the script look cryptic? Let us demystify it.

- The first statement, **cmd = "tr [a-z] [A-Z]"**, is the command to which we establish the two-way communication from AWK.

- The next statement, i.e., the print command provides input to the **tr** command. Here **&|** indicates two-way communication.

- The third statement, i.e., **close(cmd, "to")**, closes the **to** process after competing its execution.

- The next statement **cmd |& getline out** stores the output into **out** variable with the aid of getline function.

- The next print statement prints the output and finally the **close** function closes the command.

tutorialspoint
SIMPLYEASYLEARNING

15. PRETTY PRINTING

So far, we have used AWK's **print** and **printf** functions to display data on standard output. But the printf function is much more efficient. This function has been borrowed from the C language and it is very helpful while producing formatted output. Here is the syntax of the printf statement:

```
printf fmt, expr-list
```

In the above syntax, **fmt** is a string of format specifications and constants. **expr-list** is a list of arguments corresponding to format specifiers.

Escape Sequences

Similar to any string, format can contain embedded escape sequences. Discussed below are the escape sequences supported by AWK:

New Line

The following example prints **Hello** and **World** in separate lines using newline character:

```
[jerry]$ awk 'BEGIN { printf "Hello\nWorld\n" }'
```

On executing this code, you get the following result:

```
Hello
World
```

Horizontal Tab

The following example uses horizontal tab to display different field:

```
[jerry]$ awk 'BEGIN { printf "Sr No\tName\tSub\tMarks\n" }'
```

On executing the above code, you get the following result:

```
Sr No    Name    Sub Marks
```


tutorialspoint
SIMPLYEASYLEARNING

Vertical Tab

The following example uses vertical tab after each filed:

```
[jerry]$ awk 'BEGIN { printf "Sr No\vName\vSub\vMarks\n" }'
```

On executing this code, you get the following result:

```
Sr No
    Name
        Sub
            Marks
```

Backspace

The following example prints a backspace after every field except the last one. It erases the last number from the first three fields. For instance, **Field 1** is displayed as **Field**, because the last character is erased with backspace. However, the last field **Field 4** is displayed as it is, as we did not have a **\b** after **Field 4**.

```
[jerry]$ awk 'BEGIN { printf "Field 1\bField 2\bField 3\bField 4\n" }'
```

On executing this code, you get the following result:

```
Field Field Field Field 4
```

Carriage Return

In the following example, after printing every field, we do a **Carriage Return** and print the next value on top of the current printed value. It means, in the final output, you can see only **Field 4**, as it was the last thing to be printed on top of all the previous fields.

```
[jerry]$ awk 'BEGIN { printf "Field 1\rField 2\rField 3\rField 4\n" }'
```

On executing this code, you get the following result:

```
Field 4
```

Form Feed

The following example uses form feed after printing each field.

```
[jerry]$ awk 'BEGIN { printf "Sr No\fName\fSub\fMarks\n" }'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
Sr No
    Name
        Sub
            Marks
```

Format Specifier

As in C-language, AWK also has format specifiers. The AWK version of the printf statement accepts the following conversion specification formats:

%c

It prints a single character. If the argument used for **%c** is numeric, it is treated as a character and printed. Otherwise, the argument is assumed to be a string, and the only first character of that string is printed.

```
[jerry]$ awk 'BEGIN { printf "ASCII value 65 = character %c\n", 65 }'
```

On executing this code, you get the following result:

```
ASCII value 65 = character A
```

%d and %i

It prints only the integer part of a decimal number.

```
[jerry]$ awk 'BEGIN { printf "Percentages = %d\n", 80.66 }'
```

On executing this code, you get the following result:

```
Percentages = 80
```

%e and %E

It prints a floating point number of the form [-]d.dddddde[+-]dd.

```
[jerry]$ awk 'BEGIN { printf "Percentages = %E\n", 80.66 }'
```

On executing this code, you get the following result:

```
Percentages = 8.066000e+01
```


tutorialspoint
SIMPLYEASYLEARNING

The **%E** format uses **E** instead of e.

```
[jerry]$ awk 'BEGIN { printf "Percentages = %e\n", 80.66 }'
```

On executing this code, you get the following result:

```
Percentages = 8.066000E+01
```

%f

It prints a floating point number of the form [-]ddd.dddddd.

```
[jerry]$ awk 'BEGIN { printf "Percentages = %f\n", 80.66 }'
```

On executing this code, you get the following result:

```
Percentages = 80.660000
```

%g and %G

Uses %e or %f conversion, whichever is shorter, with non-significant zeros suppressed.

```
[jerry]$ awk 'BEGIN { printf "Percentages = %g\n", 80.66 }'
```

On executing this code, you get the following result:

```
Percentages = 80.66
```

The **%G** format uses **%E** instead of **%e**.

```
[jerry]$ awk 'BEGIN { printf "Percentages = %G\n", 80.66 }'
```

On executing this code, you get the following result:

```
Percentages = 80.66
```

%o

It prints an unsigned octal number.

```
[jerry]$ awk 'BEGIN { printf "Octal representation of decimal number 10 =
%o\n", 10}'
```

On executing this code, you get the following result:

```
Octal representation of decimal number 10 = 12
```


tutorialspoint
SIMPLYEASYLEARNING

%u

It prints an unsigned decimal number.

```
[jerry]$ awk 'BEGIN { printf "Unsigned 10 = %u\n", 10 }'
```

On executing this code, you get the following result:

```
Unsigned 10 = 10
```

%s

It prints a character string.

```
[jerry]$ awk 'BEGIN { printf "Name = %s\n", "Sherlock Holmes" }'
```

On executing this code, you get the following result:

```
Name = Sherlock Holmes
```

%x and %X

It prints an unsigned hexadecimal number. The **%X** format uses uppercase letters instead of lowercase.

```
[jerry]$ awk 'BEGIN { printf "Hexadecimal representation of decimal number
15 = %x\n", 15}'
```

On executing this code, you get the following result:

```
Hexadecimal representation of decimal number 15 = f
```

Now let use **%X** and observe the result:

```
[jerry]$ awk 'BEGIN { printf "Hexadecimal representation of decimal number
15 = %X\n", 15}'
```

On executing this code, you get the following result:

```
Hexadecimal representation of decimal number 15 = F
```

%%

It prints a single **%** character and no argument is converted.

```
[jerry]$ awk 'BEGIN { printf "Percentages = %d%%\n", 80.66 }'
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
Percentages = 80%
```

Optional Parameters with %

With **%**, we can use the following optional parameters:

Width

The field is padded to the **width**. By default, the field is padded with spaces but when 0 flag is used, it is padded with zeroes.

```
[jerry]$ awk 'BEGIN { num1 = 10; num2 = 20; printf "Num1 = %10d\nNum2 =
%10d\n", num1, num2 }'
```

On executing this code, you get the following result:

```
Num1 =         10
Num2 =         20
```

Leading Zeros

A leading zero acts as a flag, which indicates that the output should be padded with zeroes instead of spaces. Please note that this flag only has an effect when the field is wider than the value to be printed. The following example describes this:

```
[jerry]$ awk 'BEGIN { num1 = -10; num2 = 20; printf "Num1 = %05d\nNum2 =
%05d\n", num1, num2 }'
```

On executing this code, you get the following result:

```
Num1 = -0010
Num2 = 00020
```

Left Justification

The expression should be left-justified within its field. When the input-string is less than the number of characters specified, and you want it to be left justified, i.e., by adding spaces to the right, use a minus symbol (–) immediately after the % and before the number.

In the following example, output of the AWK command is piped to the cat command to display the END OF LINE($) character.

```
[jerry]$ awk 'BEGIN { num = 10; printf "Num = %-5d\n", num }' | cat -vte
```


tutorialspoint
SIMPLYEASYLEARNING

On executing this code, you get the following result:

```
Num = 10   $
```

Prefix Sign

It always prefixes numeric values with a sign, even if the value is positive.

```
[jerry]$ awk 'BEGIN { num1 = -10; num2 = 20; printf "Num1 = %+d\nNum2 =
%+d\n", num1, num2 }'
```

On executing this code, you get the following result:

```
Num1 = -10

Num2 = +20
```

Hash

For %o, it supplies a leading zero. For %x and %X, it supplies a leading 0x or 0X respectively, only if the result is non-zero. For %e, %E, %f, and %F, the result always contains a decimal point. For %g and %G, trailing zeros are not removed from the result. The following example describes this:

```
[jerry]$ awk 'BEGIN { printf "Octal representation = %#o\nHexadecimal
representation = %#X\n", 10, 10}'
```

On executing this code, you get the following result:

```
Octal representation = 012

Hexadecimal representation = 0XA
```


tutorialspoint
SIMPLYEASYLEARNING