# AWK - 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.

# 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

**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, issue 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

# 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 execute commands on every line. We can restrict this by providing patterns.

### Repeat

This process repeats until the file reaches its end.

## Program Structure

Let us now understand the program structure of AWK.

### BEGIN block

The syntax of the BEGIN block is as follows −

**Syntax**

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 −

**Syntax**

/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 −

**Syntax**

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.

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.

**Example**

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

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

**Output**

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.

# AWK - 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 the 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 −

**Example**

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

On executing this code, you get the following result −

**Output**

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 ....

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.

**Example**

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

On executing this code, you get the following result −

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

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**.

**Example**

[jerry]$ awk --dump-variables ''

[jerry]$ cat awkvars.out

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

**Output**

ARGC: 1

ARGIND: 0

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.

**Example**

[jerry]$ awk --help

On executing this code, you get the following result −

**Output**

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

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 −

**Example**

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

On executing this code, you get the following result −

**Output**

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.

### The --profile[=file] option

This option generates a pretty-printed version of the program in file. Default file is **awkprof.out**. Below simple example illustrates this −

**Example**

[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 −

**Output**

# 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.

**Example**

[jerry]$ awk --version

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

**Output**

GNU Awk 4.0.1

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

# AWK - 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 −

### Example

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

On executing this code, you get the following result −

### Output

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.

### Example

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

On executing this code, you get the following result −

### Output

2) Rahul Maths 90

3) Shyam Biology 87

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 −

### Example

[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.

### Example

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

On executing this code, you get the following result −

### Output

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.

### Example

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

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

### Output

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.

### Example

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

On executing this code, you get the following result −

### Output

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.

### Example

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

On executing this code, you get the following result −

### Output

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.

# AWK - 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.

**Example**

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

On executing this code, you get the following result −

**Output**

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.

**Example**

[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 −

**Output**

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**.

**Example**

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

On executing this code, you get the following result −

**Output**

Conversion Format = %.6g

### ENVIRON

It is an associative array of environment variables.

**Example**

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

On executing this code, you get the following result −

**Output**

jerry

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

### FILENAME

It represents the current file name.

**Example**

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

On executing this code, you get the following result −

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

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**.

**Example**

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

On executing this code, you get the following result −

**Output**

OFMT = %.6g

### OFS

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

**Example**

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

On executing this code, you get the following result −

**Output**

OFS = $

### ORS

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

**Example**

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

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

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

2

### RS

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

**Example**

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

On executing this code, you get the following result −

**Output**

RS = $

$

### RSTART

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

**Example**

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

On executing this code, you get the following result −

**Output**

9

### SUBSEP

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

**Example**

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

On executing this code, you get the following result −

**Output**

SUBSEP = ^\$

### $0

It represents the entire input record.

**Example**

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

On executing this code, you get the following result −

**Output**

1) Amit Physics 80

2) Rahul Maths 90

3) Shyam Biology 87

4) Kedar English 85

5) Hari History 89

### $n

It represents the nth field in the current record where the fields are separated by FS.

**Example**

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

On executing this code, you get the following result −

**Output**

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.

**Example**

[jerry]$ awk '{

print "ARGIND = ", ARGIND; print "Filename = ", ARGV[ARGIND]

}' junk1 junk2 junk3

On executing this code, you get the following result −

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

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 −

**Example**

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

On executing this code, you get the following result −

**Output**

1) Amit Physics 80

### 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**.

**Example**

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

On executing this code, you get the following result −

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

4316

### TEXTDOMAIN

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

**Example**

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

On executing this code, you get the following result −

**Output**

messages

The above output shows English text due to **en\_IN** locale

# AWK - Operators

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

|  |  |
| --- | --- |
| **S.No.** | **Operators & Description** |
| 1 | [**Arithmetic Operators**](https://www.tutorialspoint.com/awk/awk_arithmetic_operators.htm)  AWK supports the following arithmetic operators. |
| 2 | [**Increment and Decrement Operators**](https://www.tutorialspoint.com/awk/awk_increment_decrement_operators.htm)  AWK supports the following increment and decrement operators. |
| 3 | [**Assignment Operators**](https://www.tutorialspoint.com/awk/awk_assignment_operators.htm)  AWK supports the following assignment operators. |
| 4 | [**Relational Operators**](https://www.tutorialspoint.com/awk/awk_relational_operators.htm)  AWK supports the following relational operators. |
| 5 | [**Logical Operators**](https://www.tutorialspoint.com/awk/awk_logical_operators.htm)  AWK supports the following logical operators. |
| 6 | [**Ternary Operator**](https://www.tutorialspoint.com/awk/awk_ternary_operators.htm)  We can easily implement a condition expression using ternary operator. |
| 7 | [**Unary Operators**](https://www.tutorialspoint.com/awk/awk_unary_operators.htm)  AWK supports the following unary operators. |
| 8 | [**Exponential Operators**](https://www.tutorialspoint.com/awk/awk_exponential_operators.htm)  There are two formats of exponential operators. |
| 9 | [**String Concatenation Operator**](https://www.tutorialspoint.com/awk/awk_string_concatenation_operator.htm)  Space is a string concatenation operator that merges two strings. |
| 10 | [**Array Membership Operator**](https://www.tutorialspoint.com/awk/awk_array_membership_operator.htm)  It is represented by **in**. It is used while accessing array elements. |
| 11 | [**Regular Expression Operators**](https://www.tutorialspoint.com/awk/awk_regular_expression_operators.htm)  This example explains the two forms of regular expressions operators. |

# AWK - 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.

### Example

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

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

### Output

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**.

### Example

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

On executing this code, you get the following result −

### Output

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**.

### Example

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

### Output

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**.

### Example

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

### Output

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**.

### Example

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

On executing this code, you get the following result −

### Output

Ball

## Alteration

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

### Example

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

On executing this code, you get the following result −

### Output

Call

Ball

## 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 **?**.

### Example

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

On executing this code, you get the following result −

### Output

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.

### Example

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

On executing this code, you get the following result −

### Output

ca

cat

catt

## One or More Occurrence

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

### Example

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

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

### Output

22

123

234

222

## 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**.

### Example

[jerry]$ echo -e "Apple Juice\nApple Pie\nApple Tart\nApple Cake" | awk

'/Apple (Juice|Cake)/'

On executing this code, you get the following result −

### Output

Apple Juice

Apple Cake

# AWK - 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 −

### Syntax

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.

### Example

[jerry]$ awk 'BEGIN {

fruits["mango"] = "yellow";

fruits["orange"] = "orange"

print fruits["orange"] "\n" fruits["mango"]

}'

On executing this code, you get the following result −

### Output

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 −

### Syntax

delete array\_name[index]

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

### Example

[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 −

### 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 −

### Example

[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"];

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 −

### Output

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

# AWK - 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 −

### Syntax

if (condition)

action

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

### Syntax

if (condition) {

action-1

action-1

.

.

action-n

}

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

### Example

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

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

### Output

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.

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

### Syntax

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 −

### Example

[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 −

### Output

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 −

### Example

[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 −

### Output

a = 30

# AWK - 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 −

### Syntax

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 −

### Example

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

On executing this code, you get the following result −

### Output

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 −

### Syntax

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 −

### Example

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

On executing this code, you get the following result −

### Output

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 −

### Syntax

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 −

### Example

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

On executing this code, you get the following result −

### Output

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.

### Example

[jerry]$ awk 'BEGIN {

sum = 0; for (i = 0; i < 20; ++i) {

sum += i; if (sum > 50) break; else print "Sum =", sum

}

}'

On executing this code, you get the following result −

### Output

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.

### Example

[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 −

### Output

2

4

6

8

10

12

14

16

18

20

## 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.

### Example

[jerry]$ awk 'BEGIN {

sum = 0; for (i = 0; i < 20; ++i) {

sum += i; if (sum > 50) exit(10); else print "Sum =", sum

}

}'

### Output

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.

### Example

[jerry]$ echo $?

On executing this code, you get the following result −

### Output

10

# AWK - 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.

|  |  |
| --- | --- |
| **S.No.** | **Built in functions & Description** |
| 1 | [**Arithmetic Functions**](https://www.tutorialspoint.com/awk/awk_arithmetic_functions.htm)  AWK has the following built-in arithmetic functions. |
| 2 | [**String Functions**](https://www.tutorialspoint.com/awk/awk_string_functions.htm)  AWK has the following built-in String functions. |
| 3 | [**Time Functions**](https://www.tutorialspoint.com/awk/awk_time_functions.htm)  AWK has the following built-in time functions. |
| 4 | [**Bit Manipulation Functions**](https://www.tutorialspoint.com/awk/awk_bit_manipulation_functions.htm)  AWK has the following built-in bit manipulation functions. |
| 5 | [**Miscellaneous Functions**](https://www.tutorialspoint.com/awk/awk_miscellaneous_functions.htm)  AWK has the following miscellaneous functions. |

# AWK - 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 −

### Syntax

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 −

### Example

# 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

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 −

### Output

Minimum = 10

Maximum = 20

# AWK - 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 −

### Syntax

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.

### Example

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

[jerry]$ cat /tmp/message.txt

On executing this code, you get the following result −

### Output

Old data

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

### Example

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

[jerry]$ cat /tmp/message.txt

On executing this code, you get the following result −

### Output

Hello, World !!!

## Append Operator

The syntax of append operator is as follows −

### Syntax

print DATA >> output-file

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.

### Example

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

[jerry]$ cat /tmp/message.txt

On executing this code, you get the following result −

### Output

Old data

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

### Example

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

[jerry]$ cat /tmp/message.txt

On executing this code, you get the following result −

### Output

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 −

### Syntax

print items | command

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

### Example

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

On executing this code, you get the following result −

### Output

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 −

### Example

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 −

### Output

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.

# AWK - Pretty Printing

So far we have used AWK's **print** and **printf** functions to display data on standard output. But printf is much more powerful than what we have seen before. This function is borrowed from the C language and is very helpful while producing formatted output. Below is the syntax of the printf statement −

### Syntax

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 −

**Example**

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

On executing this code, you get the following result −

**Output**

Hello

World

### Horizontal Tab

The following example uses horizontal tab to display different field −

**Example**

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

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

**Output**

Sr No Name Sub Marks

### Vertical Tab

The following example uses vertical tab after each filed −

**Example**

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

On executing this code, you get the following result −

**Output**

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**.

**Example**

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

On executing this code, you get the following result −

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

Field 4

### Form Feed

The following example uses form feed after printing each field.

**Example**

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

On executing this code, you get the following result −

**Output**

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.

**Example**

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

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

Percentags = 80

### %e and %E

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

**Example**

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

On executing this code, you get the following result −

**Output**

Percentags = 8.066000e+01

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

**Example**

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

On executing this code, you get the following result −

**Output**

Percentags = 8.066000E+01

### %f

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

**Example**

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

On executing this code, you get the following result −

**Output**

Percentags = 80.660000

### %g and %G

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

**Example**

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

**Output**

On executing this code, you get the following result −

Percentags = 80.66

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

**Example**

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

On executing this code, you get the following result −

**Output**

Percentags = 80.66

### %o

It prints an unsigned octal number.

**Example**

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

On executing this code, you get the following result −

**Output**

Octal representation of decimal number 10 = 12

### %u

It prints an unsigned decimal number.

**Example**

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

On executing this code, you get the following result −

**Output**

Unsigned 10 = 10

### %s

It prints a character string.

**Example**

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

On executing this code, you get the following result −

**Output**

Name = Sherlock Holmes

### %x and %X

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

**Example**

[jerry]$ awk 'BEGIN {

printf "Hexadecimal representation of decimal number 15 = %x\n", 15

}'

On executing this code, you get the following result −

**Output**

Hexadecimal representation of decimal number 15 = f

Now let use %X and observe the result −

**Example**

[jerry]$ awk 'BEGIN {

printf "Hexadecimal representation of decimal number 15 = %X\n", 15

}'

On executing this code, you get the following result −

**Output**

Hexadecimal representation of decimal number 15 = F

### %%

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

**Example**

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

On executing this code, you get the following result −

**Output**

Percentags = 80%

## Optional Parameters with %

With **%** we can use 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.

**Example**

[jerry]$ awk 'BEGIN {

num1 = 10; num2 = 20; printf "Num1 = %10d\nNum2 = %10d\n", num1, num2

}'

On executing this code, you get the following result −

**Output**

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 −

**Example**

[jerry]$ awk 'BEGIN {

num1 = -10; num2 = 20; printf "Num1 = %05d\nNum2 = %05d\n", num1, num2

}'

On executing this code, you get the following result −

**Output**

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.

**Example**

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

On executing this code, you get the following result −

**Output**

Num = 10 $

### Prefix Sign

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

**Example**

[jerry]$ awk 'BEGIN {

num1 = -10; num2 = 20; printf "Num1 = %+d\nNum2 = %+d\n", num1, num2

}'

On executing this code, you get the following result −

**Output**

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 −

**Example**

[jerry]$ awk 'BEGIN {

printf "Octal representation = %#o\nHexadecimal representaion = %#X\n", 10, 10

}'

On executing this code, you get the following result −

**Output**

Octal representation = 012

Hexadecimal representation = 0XA