Navigating Linux System Commands

A guide for beginners to the Shell and GNU coreutils

Sayan Ghosh

October 30, 2024

IIT Madras
BS Data Science and Applications

Disclaimer
This document is a companion activity book for the System Commands (BSSE2001) course taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program**. This book contains resources, references, questions and solutions to some common questions on Linux commands, shell scripting, grep, sed, awk, and other system commands.

This was prepared with the help and guidance of the course instructors:

Santhana Krishnan and **Sushil Pachpinde**

Colophon
This document was typeset with the help of KOMA-Script and LaTeX using the kaobook class.
The source code of this book is available at:
`https://github.com/sayan01/se2001-book`
(You are welcome to contribute!)

Edition
Compiled on October 30, 2024

UNIX is basically a simple operating system, but you have to be a genius to understand the simplicity.

– Dennis Ritchie

Preface

Through this work I have tried to make learning and understanding the basics of Linux fun and easy. I have tried to make the book as practical as possible, with many examples and exercises. The structure of the book follows the structure of the course *BSSE2001 - System Commands*, taught by **Prof. Gandham Phanikumar** at **IIT Madras BS Program.** .

The book takes inspiration from the previous works done for the course,

- Sanjay Kumar's Github Repository
- Cherian George's Github Repository
- Prabuddh Mathur's TA Sessions

as well as external resources like:

- Robert Elder's Blogs and Videos
- Aalto University, Finland's Scientific Computing - Linux Shell Crash Course

The book covers basic commands, their motivation, use cases, and examples. The book also covers some advanced topics like shell scripting, regular expressions, and text processing using `sed` and `awk`.

This is not a substitute for the course, but a companion to it. The book is a work in progress and any contribution is welcome at `https://github.com/sayan01/se2001-book`

Sayan Ghosh

Contents

List of Figures

List of Tables

AWK 1

1.1 What is AWK?

Awk is programming language meant for processing structured data in rows and columns.

The name **AWK** comes from the initials of its creators:

- Afred **A**ho
- Peter **W**einberger
- Brian **K**ernighan

Remark 1.1.1 Afred Aho was the author of *egrep* and thus **awk** works with Extended Regular Expressions directly, and not Basic Regular Expressions.

The most popular distribution of awk is **gawk** - GNU AWK, developed by the GNU team and distributed under the GPL.

Kernighan also released his version **nawk** - New AWK, that is used by BSD systems to avoid GPL.

AWK was the only scripting language in user space in early days along with the bourne shell. It inspired the creation of the **Perl** language which also has powerful text processing tools.

The awk language is a data-driven language, that is, it processes data line by line. It is a procedural language, but it is not object-oriented. It has a very simple syntax, and it is very powerful for text processing. For each line that is read, awk applies a set of rules to the line and executes the actions that are associated with the rules.

1.2 Basic Syntax

The basic syntax of an awk program is a series of pattern action pairs, written as:

```
pattern { action }
```

The entire input is split into records, and each record is split into fields. By default records are split using newlines, so each record is one line of input, but it can be changed.

Each record is tested against the pattern, and if the pattern matches, the action is executed.

Thus, an awk program has an implicit loop that reads each record, tests each record against the patterns, and executes the actions if they match.

Here we are checking for numbers in each line and if the number ends with a 5 then we are printing that it is a multiple of 5. Similarly if it ends with a 0, then we print that it is a multiple of 5 and 10. Observe how this is achieved using regex alternation and multiple pattern-action pairs.

```
$ cat script.awk
/5$|0$/ {
  print $0 " is a multiple of 5"
}

/0$/  {
  print $0 " is a multiple of 10"
}
$ seq 20 | awk -f script.awk
5 is a multiple of 5
10 is a multiple of 5
10 is a multiple of 10
15 is a multiple of 5
20 is a multiple of 5
20 is a multiple of 10
```

If the pattern is omitted, the action is executed for every record.

```
$ cat script.awk
{
  print "Line: " $0
}
$ seq 5 | awk -f script.awk
Line: 1
Line: 2
Line: 3
Line: 4
Line: 5
```

If the action is omitted, the default action is to print the record.

```
$ seq 50 | awk '/5$/'
5
15
25
35
45
```

1.2.1 Special Conditions

- **BEGIN** - This pattern is executed before the first record is read.
- **END** - This pattern is executed after the last record is read.

```
$ cat script.awk
BEGIN{
  print "Starting the script"
}
{
  print "Line: " $0
}
END{
  print "Ending the script"
}
$ seq 5 | awk -f script.awk
Starting the script
```

```
Line: 1
Line: 2
Line: 3
Line: 4
Line: 5
Ending the script
```

1.2.2 Ranges

The condition can also be a range of records, matched by their record number or by regular expression patterns. Both the ends are inclusive.

Here we are printing from the line starting with a capital T and till the line that ends with a full stop.

Example:

```
$ cat data.txt
Hello
This is the start
of a long sentence
spanning multiple
lines in the file.
Goodbye
$ awk '/^T/,/\.$/'
This is the start
of a long sentence
spanning multiple
lines in the file.
```

The record number can be matched using the `NR` variable. It is a variable built-in awk and it is automatically updated everytime a new record is read.

```
$ seq 10 | awk 'NR == 3, NR == 7'
3
4
5
6
7
```

1.2.3 Fields

Awk splits each record into fields. By default the fields are split by any kind of whitespace, including spaces and tabs. However, this can be changed to split by some other delimiter or even regular expression. This is useful for different structured files such as CSV and quoted CSV files.

The fields are numbered from 1 and can be accessed using the `$i` syntax.

```
$ echo "Hello World" | awk '{print $1}'
Hello
```

The variable `$0` variable contains the entire record.

We can also match a regular expression pattern against a specific field as well using the ~ syntax.

```
$ echo -e "Hello World\nhello universe\nthis
    is hello" | awk '$1 ~ /[hH]ello/'
Hello World
hello universe
```

1.2.4 Number of Fields and Number of Records

The number of fields in a record can be accessed using the `NF` variable. It is a built-in variable in awk.

```
$ echo "Hello World" | awk '{print NF}'
2
```

Here we are printing the number of fields in each line. The number of fields may change from record to record, the variable NR will also be updated accordingly automatically.

The number of records can be accessed using the `NR` variable. It is a built-in variable in awk.

```
$ seq 5 10 | awk 'END{print NR}'
6
```

Here we are printing the NR variable which is updated everytime a new record is read. Thus the NR variable stores the value i when it is reading the i^{th} record. However, we are not printing it for each record, rather we are printing it only after all the records are read, in the END block. The NR variable is not wiped after reading the last record, thus it still contains the record number of the last record, which is same as the total number of records as we count from 1.

1.2.5 Splitting Records by custom delimiter

By default awk splits records by whitespace. However, we can change this to split by some other delimiter. There are two variables that control this behavior:

- `FS` - Field Separator - This is the regular expression that is used to split the fields in a record.
- `FPAT` - Field Pattern - This is the regular expression that is used to match the fields in a record.

The `FS` variable can be set in the BEGIN block, or it can be set from the command line using the `-F` option. The `FPAT` variable can be set in the BEGIN block.

```
$ echo "Hello,World" | awk  '{print $1}'
Hello,World
$ echo "Hello,World" | awk -F, '{print $1}'
Hello
```

Here we can see that by default the record is split by whitespace, and thus the entire line is considered as a single field. However, when we set the `FS` variable to a comma, then the record is split by the comma and we get the first field as `Hello`.

The `FS` variable can also be explicitly set inside the script, if for example we do not have access to the command line flags.

```
$ echo "Hello,World" | awk 'BEGIN{FS=","}{
    print $1}'
Hello
```

The `FS` variable can also be set from the command line by using the `-v` flag.

```
$ echo "Hello,World" | awk -v FS="," '{print
    $1}'
Hello
```

Similarly, the `FPAT` variable can be set in the BEGIN block or sent from the commandline using the `-v` flag.

The `FPAT` variable is used to match the fields in a record. We can use regular expressions in this to match any particular pattern of fields if the field separator is not fixed.

```
$ cat script.awk
BEGIN{
  FPAT="[a-zA-Z]+"
}
{
  for(i=1; i<=NF;i++){
    printf $i " "
  }
  print ""
}
$ cat data.txt
Hello1This2is3a4file^with:lots'of'delimiters
however,the-fields=are+always/alphabets
so%we#can@simply!use*FPAT-variable?to"match
each]field.
$ awk -f script.awk data.txt
Hello This is a file with lots of delimiters
however the fields are always alphabets
so we can simply use FPAT variable to match
each field
```

In this example we have set the `FPAT` variable to match any alphabets. This is useful when the field separator is not fixed, but the fields themselves have a fixed pattern.

Then, for each line, we are printing all the fields in that line separated by a space.

1.3 Awk Scripts

Even though awk is great for one-lines, it is also great for writing scripts. The scripts can be written in a file and then executed using the `awk -f` flag. This is similar to how sed scripts also work, this chapter assumes familiarity with the previous chapter.

```
$ cat script.awk
BEGIN{
  print "Starting the script"
  FS=","
}
{
  print "First Field: " $1
}
END{
  print "Ending the script"
}
$ cat data.csv
Hello,World
Goodbye,Universe
$ awk -f script.awk data.csv
Starting the script
First Field: Hello
First Field: Goodbye
Ending the script
```

The entire contents of the script file is executed as an awk script. The BEGIN and END blocks are executed before and after the records are read respectively. All other pattern blocks [1] are executed for each record which match the pattern.

1: Here we are using a block without any pattern, so it matches all lines.

1.3.1 Command Line Arguments

Awk scripts can also take command line arguments. These arguments can be accessed using the `ARGV` array. The ARGC variable stores the number of arguments.

```
$ cat script.awk
BEGIN{
  print ARGC
  for(arg in ARGV){
    print ARGV[arg]
  }
}
$ awk -f script.awk hello how "are you"
4
awk
hello
how
are you
```

In awk, the first argument is the name of the program itself, and the rest of the arguments are the arguments passed to the program. This is consistent with C, which also uses variables argc and argv as parameter names in the main() function.

awk will assume that the first command line argument is the script and the rest of the command line arguments are the files to work on.

However, if we do not have any pattern-action pairs in the script and we do not have an END block as well, then awk not try to open the arguments as files. This is why we did not get any error in the last example. If we add a empty block after the begin, we will encounter the file not found error.

```
$ cat script.awk
BEGIN{
  print ARGC
  for(arg in ARGV){
    print ARGV[arg]
  }
}
{}
```

```
$ awk -f script.awk hello how "are you"
4
awk
hello
how
are you
awk: script.awk:5: fatal: cannot open file ‘
    hello’ for reading: No such file or
    directory
```

Note that the command block has no action in it, so it will not perform any action, but because it is present awk will still expect to open the file.

If we pass filenames as arguments instead to the same script, then the error subsides, even if nothing is printed from the files.

```
$ cat script.awk
BEGIN{
  print ARGC
  for(arg in ARGV){
    print ARGV[arg]
  }
}
{}
$ awk -f script.awk /etc/fstab ~/.bashrc
3
awk
/etc/fstab
/home/sayan/.bashrc
```

Remark 1.3.1 Note that having a action-less condition like `NR==5` will print the records by default which match the pattern, whereas having a condition-less action like `{ print $1 }` will execute that action for all lines. We can also have a pattern-less and action-less block as seen in the previous example `{}`. In this case, it will do nothing for all the lines. However awk will still expect one or more files to perform this NO-OP on, and providing a command line argument that is not a valid file path will throw an error as we saw. Further, even if the block is empty, awk will still read each line of the file, this can be verified by running the NO-OP on an infinite file like `/dev/zero` or `/dev/urandom` and observing that the command never stops.

```
$ awk '{}' /dev/zero
```

Press `Ctrl+C` to stop the command, as it may eat up your system resources since there are no newlines in `/dev/zero`, hence the record read in memory keeps on increasing.

This gives rise to two small NO-OP scripts that can be used to read the entire file.

```
$ awk '{}' /etc/fstab # does nothing
$ awk '1' /etc/fstab # prints the entire file
```

Thus `awk 1` can be used as a `cat` replacement.

Similarly `awk 0` or `awk {}` can be used as a `test -r` replacement to test if a file exists and is readable.

Exercise 1.3.1 Use the aforementioned tricks to write a script that first tests if the file path mentioned as the first argument of the script (`$1`) exists or not (using awk), and if it does exist, then print its contents (using awk).

Solution:

```
awk 0 "$1" && awk 1 "$1"
```

1.3.2 Shebang

Awk scripts can also be made executable by adding a shebang line at the top of the script. The current shell will then execute the script using the `awk` interpreter. Throughout this book we shall be using the `gawk` interpreter for running awk scripts. Similar to sed scripts, we have to mention the `-f` flag in the shebang.

```
$ cat script.awk
#!/usr/bin/gawk -f
BEGIN{
  print "hello"
}
$ chmod +x script.awk
$ ./script.awk
hello
```

To run the script the file has to be made executable using the `chmod +x` command.

1.4 Variables

Just like any other programming language, awk has support for variables to store data. There are three types of variables in awk, they are:

- Scalar - A variable that holds a single value of any valid data type.
- Indexed Array - An array that is indexed by numbers.
- Associative Array - An array that is indexed by strings.

Variables in awk are dynamically typed, that is, the type of the variable is determined by the value assigned to it. Further, the type of the variable can change during the execution of the program. The type is automatically determined by the value assigned to it or the operation performed on it. [2]

2: GNU Awk also has support for regex-typed variables, which are variables that can store a regular expression to match.

A variable is initially of the type `unassigned`, however as awk implicitly sets the default value of variables, that is equivalent to 0 and empty string `""`.

```
$ awk 'BEGIN { print (a == "" && a == 0 ? "a
    is untyped" : "a has a type!") ; print
    typeof(a) }'
a is untyped
unassigned
```

The type of a variable can be checked using the `typeof()` function.

```
$ awk 'BEGIN { a=5; b="hello"; c=5.5; print
    typeof(a); print typeof(b); print typeof(c
    ) }'
number
string
number
```

1.4.1 Loose Typing and Numeric Strings

In a dynamically typed language, it is easy to infer when a variable is string. But what about number inputs? Input, by design, are always strings in awk. However, if the string can be converted to a number, then it is treated as a number. These kinds of strings are called numeric strings.

When a numeric string is operated with a number, it is converted to a number in that expression. However, when it is operated with a string, it is converted to a string in that expression.

```
$ echo "9" | awk '{ print $1 < 10 }'
1
```

```
$ echo "9" | awk '{ print $1 < "10" }'
0
```

In the above example, the first expression is comparing a numeric string with a number, so the string is converted to a number, and we perform a numeric comparison, that is, is the number 9 less than the number 10? And thus the output is 1, which means true in awk.

In the second expression, we are comparing a numeric string with a string, so the numeric string is converted to a string, and we perform a string comparison, that is, does the string ”9” come lexicographically before the string ”10”? The answer is no since in string comparison we compare letter by letter, and ’9’ is not less than ’1’.

Only user input can be a numeric string, a string that looks like a number present in the script itself can never be treated like a number.

```
$ echo "hello 123" | awk '{ print typeof($1),
    typeof($2), typeof(1), typeof("1"), typeof
    ("one") }'
string strnum number string string
```

The above example demonstrates that the input fields can either be string or strnum (numeric string) if they look like a number, but never number. Similarly, the numbers in the script are always numbers, and never numeric strings, and the string in the script are always strings, never numeric strings.

You can read more about variable typing in awk here.

1.4.2 User-defined Variables

To define a variable, we simply assign a value to it. The variable is created when it is assigned a value. As awk is dynamically typed, the type of the variable is determined by the value assigned to it. A variable can be assigned a value using the = operator. This can be done in the following places:

- In the BEGIN block.
- In the pattern-action pairs.
- In the END block.
- In the command line using the `-v` flag.
- In the command line without the `-v` flag.

If the variable is assigned as a command line argument without the `-v` flag, the time of assignment of the variable depends on its order in the command line arguments.

If the assignment is done after the first file and before the second file, the variable will be unset for all the records of the first file and set to the value for the second file.

```
$ echo hello > hello
$ echo world > world
$ awk '{ print n $0 }' n=1 hello n=2 world
1hello
2world
```

The above example demonstrates that the variable `n` is set to 1 for the first file and 2 for the second file following the order of the command line arguments.

Similarly, if the variable is not defined at all, then it is treated as an empty string.

```
$ echo hello > hello
$ echo world > world
$ awk '{ print n $0 }' hello n=2 world
hello
2world
```

If we are using the -v flag to set the variable, it has to mandatorily be done before all files and before the awk script or script file also. However it can still be overwritten in subsequent arguments.

```
$ awk -v n=1 '{ print n $0 }' hello n=2 world
1hello
2world
```

The variables can also be declared in the `BEGIN`, `END`, or the pattern-action pairs.

A variable declared inside the blocks will be available for use in all subsequent blocks of that iteration and also of all blocks of the subsequent iterations. Unfamiliarity with this may lead to logical errors in the script.

```
$ cat script.awk
#!/bin/gawk -f

$0 % 2 == 0 {
  mult2 = 1
  print $0 " is a multiple of 2"
}

$0 % 5 == 0 {
  mult5 = 1
  print $0 " is a multiple of 5"
}

mult2 == 1 && mult5 == 1 {
  print $0 " is a multiple of 10"
}
$ seq 10 | awk -f script.awk
2 is a multiple of 2
4 is a multiple of 2
5 is a multiple of 5
```

```
5 is a multiple of 10
6 is a multiple of 2
6 is a multiple of 10
7 is a multiple of 10
8 is a multiple of 2
8 is a multiple of 10
9 is a multiple of 10
10 is a multiple of 2
10 is a multiple of 5
10 is a multiple of 10
```

As it is seen in the above example, the variables `mult2` and `mult5` are declared in the pattern-action pairs, and they are used in the subsequent pattern-action pair. However, even for next iterations, the variables are not reset, and they are still available for use in the next iteration. This causes logical error which prints that all numbers are multiples of 10. This is because the variables are available for use in all subsequent blocks of that iteration and also of all blocks of the subsequent iterations.

To fix this issue, we have to reset the variables at the end of the iteration.

```
$ cat script.awk
#!/bin/gawk -f

$0 % 2 == 0 {
  mult2 = 1
  print $0 " is a multiple of 2"
}

$0 % 5 == 0 {
  mult5 = 1
  print $0 " is a multiple of 5"
}

mult2 == 1 && mult5 == 1 {
  print $0 " is a multiple of 10"
}
```

```
{
  mult2 = 0
  mult5 = 0
}
$ seq 10 | awk -f script.awk
2 is a multiple of 2
4 is a multiple of 2
5 is a multiple of 5
6 is a multiple of 2
8 is a multiple of 2
10 is a multiple of 2
10 is a multiple of 5
10 is a multiple of 10
```

Due to this, the `BEGIN` block is the best place to declare variables that are used throughout the script. However, if your declaration simply sets a numeric variable to 0 or a string variable to an empty string, then it is not necessary to declare the variable at all, as awk implicitly sets undefined variables to 0 or empty string when they are used for the first time.

```
$ cat script.awk
#!/bin/gawk -f
BEGIN{
  prod = 1
}
{
  prod *= $0
  sum += $0
}
END{
  print "product:", prod
  print "sum:", sum
}
$ seq 5 | awk -f script.awk
product: 120
sum: 15
```

In the above example, we are calculating the product and sum of the numbers in the records. We need to initialize the variables `prod` and `sum` in the `BEGIN` block, as they are used throughout the script. However, as the `sum` variable is initialized to 0 we do not need to declare it explicitly. The `prod` variable is initialized to 1 as it is used in a multiplication operation, and multiplying by 0 will always result in 0, so it has to be explicitly declared in the `BEGIN` block.

1.4.3 Built-in Variables

There are a lot of built-in variables in awk, they either let us configure how awk works, or gives us access to the state of the program.

Variables that control awk

There are a lot of variables that control how awk works by configuring behaviours and edge-cases. We will not cover all of them, but some of the important ones.

A full list can be found in the manual.

- `FS` - Field Separator - This is the regular expression that is used to split the fields in a record. It cannot match a null string.
- `FPAT` - Field Pattern - This is the regular expression that is used to match the fields in a record. It can match a null string.
- `FIELDWIDTHS` - Field Width - This is the width of the fields in a record.
- `CONVFMT` - Conversion Format - This is the format used to convert numbers to strings.
- `OFMT` - Output Format - This is the format used to print numbers.

- ORS - Output Record Separator - This is the string that is printed after each record.
- OFS - Output Field Separator - This is the string that is printed between each field.
- RS - Record Separator - This is the string that is used to split input records.

FIELDWIDTHS and FPAT are gawk extensions and are not available in all versions of awk.

Let us play around with these variables to understand them better.

FS:

```
$ cat script.awk
#!/bin/gawk -f

BEGIN{
  FS=","
}

NR != 1{
  print $1
  marks+=$2
  students++
}
END{
  print "Average Marks:", marks/students
}
$ cat data.csv
Name,Marks
Adam,62
Bob,76
Carla,67
Delphi,89
Eve,51
$ awk -f script.awk data.csv
Adam
Bob
Carla
Delphi
```

```
Eve
Average Marks: 69
```

In this example we are splitting the records by a comma, and then printing the name of the student and calculating the average marks of the students.

As the file is a normal CSV file, we can use the `FS` to split the records by the comma delimiter.

If we do not set the `FS` variable, then the records are split by whitespace, and the entire line is considered as a single field. Try it out with names with multiple words and see how the fields are split.

We also use two user-defined variables `marks` and `students` to calculate the average marks of the students.

We can also use the `NR` variable to get the number of records instead of creating and maintaining a separate `students` variable. However note that we also have a header row, so we have to exclude that from the count.

```
$ cat script.awk
#!/bin/gawk -f

BEGIN{
  FS=","
}

NR != 1{
  print $1
  marks+=$2
}
END{
  print "Average Marks:", marks/(NR-1)
}
$ cat data.csv
Name,Marks
Adam,62
```

```
Bob,76
Carla,67
Delphi,89
Eve,51
$ awk -f script.awk data.csv
Adam
Bob
Carla
Delphi
Eve
Average Marks: 69
```

We will cover NR and other built-in variables in the next subsection.

FPAT:

The FPAT variable is used to match the fields in a record. It can match a null string. If the field separator is not fixed, then we can use the FPAT variable to match the fields in a record. If we set the FPAT variable, then the FS variable is ignored, and vice-versa.

If we want to replicate the previous example using the FPAT variable, we can do it as follows:

```
$ cat script.awk
#!/bin/gawk -f

BEGIN{
  FPAT="[^,]*"
}

NR != 1{
  print $1
  marks+=$2
}
END{
  print "Average Marks:", marks/(NR-1)
}
$ cat data.csv
```

```
Name,Marks
Adam,62
Bob,76
Carla,67
Delphi,89
Eve,51
$ awk -f script.awk data.csv
Adam
Bob
Carla
Delphi
Eve
Average Marks: 69
```

As we can see the output remains the same, but the `FPAT` variable is used to match the fields in the record.

The regex `[^,]*` matches any character except a comma, and it matches zero or more of these characters. We will discuss more about regex subsequent sections.

FIELDWIDTHS:

Sometimes the fields in a record are of fixed width, and they are not separated by any delimiter. In that case we can use the `FIELDWIDTHS` variable to specify the width of each field.

Let us create a file that contains all the student roll numbers.

```
$ for term in {21..24}f{1..3}; do for i in
    {1..10000}; do printf "%s%06d\n" $term "$i
    " ; done ; done > student-data
```

It is left as an exercise to the reader to explain the above command and how it is creating all the roll numbers.

Then we can use the FIELDWIDTHS variable to split the records by the field width and extract the unique years and terms. [3]

3: In this case we already know the answer as we ourself craeted the data using those parameters, but assume that the data was not created by us and we want to extract these details from the data.

```
$ cat script.awk
#!/bin/gawk -f

BEGIN{
  FIELDWIDTHS="2 1 1 6"
}

{
  years[$1]++
  terms[$3]++
}
END{
  print "Years:"
  for(year in years){
    print year
  }
  print "Terms:"
  for(term in terms){
    print term
  }
}
$ awk -f script.awk student-data
Years:
21
22
23
24
Terms:
1
2
3
```

As it is visible, the awk script uses the fixed width fields to split the record and then extract them. We then use an associative array [4] to store the count of each. In the END we use a for loop to iterate over all the keys and print them.

4: Like a dictionary in python

We have not covered either associative arrays or loops till now, we will cover them in the later sections.

OFMT:

The `OFMT` variable is used to set the output format of numbers.

This is a C-style format string, and it is used to convert numbers to strings during printing.

The default value is `"\%.6g"`.

```
$ cat script.awk
#!/bin/gawk -f

{
  sum+=$1
}
END{
  print "Average:", sum/NR
  OFMT="%d"
  print "Average:", sum/NR
}
$ seq 10 | awk -f script.awk
Average: 5.5
Average: 5
```

In this example, we are calculating the average of the numbers in the records. The input were the numbers from 1 to 10. The actual average is 5.5 as seen in the first line of output where the `OFMT` is not modified yet. After that, the `OFMT` is set to `"\%d"`, which is the format string for integers, and then the average is printed again, but this time as an integer, so the fractional part gets truncated.

ORS:

The `ORS` variable is used to set the output record separator. By default, the `ORS` variable is set to the newline character `"\n"`.

Consider the following example where we set the FS and ORS variable to extract a comma separated list of user names from the /etc/passwd file.

```
$ cat script.awk
#!/bin/gawk -f

BEGIN{
  FS=":"
  ORS=","
}
{
  print $1
}
$ awk -f script.awk /etc/passwd
root,bin,daemon,mail,ftp,http,nobody,dbus,
     systemd-coredump,systemd-network,systemd-
     oom,systemd-journal-remote,systemd-resolve
     ,systemd-timesync,tss,uuidd,_talkd,avahi,
     named,cups,dnsmasq,git,nm-openconnect,nm-
     openvpn,ntp,openvpn,polkitd,rpc,rpcuser,
     rtkit,saned,sddm,usbmux,sayan,brltty,
     gluster,qemu,colord,dhcpcd,fwupd,geoclue,
     libvirt-qemu,mysql,monero,mpd,nbd,passim,
     postgres,redis,tor,unbound,metabase,test,
     flatpak,
```

Here first we are setting the FS to be the colon symbol, which is required as the /etc/passwd file is colon separated. Then we print only the username field, which is the first field. But we set the ORS to be a comma, so that the output is a comma separated list of user names.

This is a useful use of awk, and usually if you are want to accomplish this task you will usually do this using a single line from the command line directly, without making an elaborate script file.

```
awk -F: -vORS=, '{print $1}' /etc/passwd
```

One of the most common use of changing ORS and

`RS` is to convert from **CRLF** to **LF** or vice-versa.

Exercise 1.4.1 Write an awk script that will convert a file from **CRLF** to **LF**.

Variables that convey information

There are many variables that awk sets on its own to convey information about the state of the program. They are also updated by awk whenever applicable.

- `NF` - Number of Fields - This is the number of fields in the current record. It starts from 1.
- `NR` - Number of Records - This is the number of records that have been read so far. Also refered to as the Record Number. It starts from 1.
- `FNR` - File Number of Records - This is the number of records that have been read so far in the current file. It starts from 1 and resets to 1 when a new file is read.
- `FILENAME` - File Name - This is the name of the current file being read.
- `RSTART` - Record Start - This is the index of the start of the matched string in the last call to the `match()` function.
- `RLENGTH` - Record Length - This is the length of the matched string in the last call to the `match()` function.
- `ENVIRON` - Environment Variables - This is an associative array that contains the environment variables that awk recieves.
- `ARGC` - Count of Command Line Arguments - It is the number of command line arguments passed from the shell. It also counts the name of the executable as the first argument.

- ARGV - The indexed array that contains all the command line arguments passed from the shell when running the awk program. The first argument is the name of the program itself (most times awk or gawk) and the rest arguments follow. This does not include the awk script itself, or the flags passed to the command.

FNR and NR:

We have already seen that NR denotes the record number in that iteration and is automatically updated by awk every next record.

The FNR variable is similar to the NR variable; it also stores the record number. However, the FNR variable resets from 1 for each new file, whereas the NR variable keeps on increasing for all the records read from all the files.

If we are iterating over only a single file, then the FNR and NR variables will have the same value. However, if we are iterating over multiple files, then the FNR variable will reset to 1 for each new file, whereas the NR variable will keep on increasing for all the records read from all the files. So the value of FNR will always be less than or equal to the value of NR.

In this example we are using the <(command) syntax to pass the output of the command as a file to awk. We are doing this so we can pass multiple files to awk. Here we can observe that the FNR variable resets to 1 for each new file, whereas the NR variable keeps on increasing for all the records read from all the files.

```
$ awk '{ print FNR, NR }' <(seq 5) <(seq 5)
1 1
2 2
3 3
4 4
5 5
1 6
2 7
3 8
4 9
5 10
```

The variables are equal for the first file, but they differ for the subsequent files. This can be exploited as a condition to check if we are in the first file or not.

For example, if we want to print the lines common to both the files, we can do it as follows:

```
$ awk ' FNR==NR{ lines[$0]=1 } FNR!=NR &&
    lines[$0]' <(seq 0 3 100) <(seq 0 5 100)
0
15
30
45
60
75
90
```

Here we are using `FNR==NR` to check if we are in the first file or not. For the first file we are simply storing all the lines in a dictionary. For the second file, which we check using `FNR!=NR`, we are checking if the line is already present in the dictionary. This means that the line has occured in file one, as well as in the second file, then we print the file. Here the files are multiples of 3 and multiples of 5, so the lines common are the multiples of 15.

NF:

The `NF` variable stores the number of fields in the current record. This can not only be used to see the number of fields in the record, but also to access the last field of the record.

```
$ cat data.txt
hello,world
hello,world,how,are,you
$ awk -F, '{ print NF, $NF }' data.txt
2 world
5 you
```

Here we are using the `NF` variable to print the number of fields in the record and the `$NF` to print the last field in each record.

FILENAME:

The `FILENAME` variable stores the name of the current file being read. This can be used in multitude of ways, like to print the file name in the output, or to check if we are in a specific file or not.

Consider the following examples:

```
$ cat data1.txt
This is the first file
There are two lines in this file
$ cat data2.txt
This is the
second file
and it has
four lines.
$ awk ' { print FILENAME ":" FNR ":" $0 }'
    data*
data1.txt:1:This is the first file
data1.txt:2:There are two lines in this file
data2.txt:1:This is the
data2.txt:2:second file
data2.txt:3:and it has
data2.txt:4:four lines.
```

This example demonstrates how we can use the `FILENAME` variable to print the file name in the output. This is useful if there are a lot of files and we want to prefix their lines with the filename and line number to find which file has a particular line.

Or, if you want to print the filename at the start of the file, we can use `FNR` to test if it is the first line, then print the filename. We also have to explicitly print all lines.

```
$ cat data1.txt
This is the first file
There are two lines in this file
$ cat data2.txt
This is the
second file
and it has
four lines.
$ awk ' FNR==1 { print "==" FILENAME "==" } 1'
     data*
==data1.txt==
This is the first file
```

```
There are two lines in this file
==data2.txt==
This is the
second file
and it has
four lines.
```

Or, if we want to print how many lines each files have:

```
$ cat data1.txt
This is the first file
There are two lines in this file
$ cat data2.txt
This is the
second file
and it has
four lines.
$ awk -f script.awk data*
data2.txt:4
data1.txt:2
```

Here we use a dictionary where the key is the FILENAME and the value keeps increasing for each line that exists in the file. This creates a dictionary with the number of lines in each file as we are iterating over all lines of all files.

We can then print the dictionary at the end to get the number of lines in each file. Note that order of the files is not guaranteed, as awk does not guarantee the order of the keys in the dictionary like python before version 3.7.

Exercise 1.4.2 Run `wc -l data*` and observe the output, can you mimic this output using awk? We have already seen how to store and print the number of lines for each file, can you also print the total sum?

Hint: Use NR.

ENVIRON:

The ENVIRON variable is an associative array that contains the environment variables that awk receives.

Thus we can also use the shell's environment variables in awk.

```
$ export a=5
$ awk 'BEGIN{ print ENVIRON["a"] }'
5
```

Note that the name of the variable has to be quoted since we are accessing a dictionary and the name of the variable is merely a key in the ENVIRON dictionary. If we do not quote the variable's name, awk will think we are refering to a variable in **awk** named a and resolve it to an empty string (or its value if defined) and try to access that key in the ENVIRON dictionary, which might not exist or be a different output than expected.

ARGC and ARGV

The ARGC and ARGV variables, like in C, store the number of commandline arguments passed to the awk program and the string array of each argument. Any flags passed to awk is not part of the ARGV array. The awk script or the awk script file path is also not a part of the array or the count of ARGC.

```
$ cat script.awk
BEGIN {
  for(arg in ARGV){
    print(ARGV[arg])
  }
}
$ awk -f script.awk a b c
awk
```

```
a
b
c
```

Not unlike C, the first argument is the name of the executable, and the rest are the arguments passed to the executable.

We can directly iterate over the array using a for-each loop, and print the arguments passed to the awk program.

We can also use a C-style for loop to iterate over the array using the `ARGC` count.

```
$ cat script.awk
BEGIN {
  for(i=0; i<ARGC; i++){
    print(ARGV[i])
  }
}
$ awk -f script.awk a b c
awk
a
b
c
```

We will discuss the `RSTART` and `RLENGTH` variables when we discuss the `match()` function.

1.5 Functions

If we have a set of code that performs a specific operation and might be re-used several times, we can move it into its own function and call the function.

Awk has support for functions, and we can define our own functions in awk.

When a function is called, expressions that create the function's actual parameters are evaluated completely before the call is performed.

1.5.1 User-defined Functions

Definitions of functions can appear anywhere between the rules of an awk program. There is no need to put the definition of a function before all uses of the function. This is because awk reads the entire program before starting to execute any of it.

Syntax of a function definition is as follows:

```
function name(parameter-list) {
  statements
}
```

The name of a function has to follow the same rules as that of a variable and cannot be a name that is already used by a variable, array, or built-in functions.

`parameter-list` is an optional list of the function's arguments and local variable names, separated by commas. When the function is called, the argument names are used to hold the argument values given in the call.

A function cannot have two parameters with the same name, nor may it have a parameter with the same name as the function itself.

The parameter list does not enforce the number of arguments passed to the function, and the function can be called with fewer arguments as well. Any argument that is not passed is treated as an empty string in string contexts and zero in numeric contexts.

```
$ cat script.awk
function test(a,b,c,d,e,f){
  print a,b,c,d,e,f
}

BEGIN{
  test(1,2,3)
  test(1,2,3,4,5)
  test(1,2,3,4,5,6)
}
$ awk -f script.awk
1 2 3
1 2 3 4 5
1 2 3 4 5 6
```

However providing more arguments than the function expects throws an error.

All the built-in variables are also available inside the function.

```
$ cat script.awk
function foo(){
  print NR ": " $0
}

/5/{
  foo()
}
$ seq 50 | awk -f script.awk
5: 5
15: 15
25: 25
35: 35
45: 45
50: 50
```

During execution of the function body, the arguments and local variable values hide, or shadow, any variables of the same names used in the rest of the program. The shadowed variables are not accessible in the function definition, because there is no way to name them while their names have been taken away for the arguments and local variables. All other variables used in the awk program can be referenced or set normally in the function's body.

The arguments and local variables last only as long as the function body is executing. Once the body finishes, you can once again access the variables that were shadowed while the function was running.

All the built-in functions return a value to their caller. User-defined functions can do so also, using the return statement.

```
$ cat script.awk
function sqr(x){
  return x*x
}

{
  print sqr($0)
}
$ seq 5 | awk -f script.awk
1
4
9
16
25
```

Here the function takes a parameter x and returns its square x^2. This is then used by the print function to print it to the screen.

A function can also be called recursively.

```
$ cat script.awk
function factorial(x){
  if(x==1) return 1
  return x * factorial(x-1)
}

{
  print factorial($0)
}
$ seq 5 | awk -f script.awk
1
2
6
24
120
```

1.5.2 Pass by Value and Pass by Reference

In awk, all arguments that are not array are passed by value. This means that the variable itself is not passed to the function when calling a function, rather its value is copied and passed to the function. If the function changes the value inside the function it has no effect on the variable used by the caller to pass the value to the function and that variable's value remains unchanged.

```
$ cat script.awk
function alter(x){
  print "X inside " x
  x=x+1
  print "X inside " x
}

BEGIN{
  x=5
  print "X outside " x
  alter(x)
  print "X outside " x
}
$ awk -f script.awk
X outside 5
X inside 5
X inside 6
X outside 5
```

Howecer, if we pass an array to a function, then the array is passed by reference. This means that the function can change the array and the changes will be reflected in the array used by the caller to pass the array to the function.

```
$ cat script.awk
function alter(l){
  l[0] = 0
}

BEGIN{
  l[0]=1
  print l[0]
  alter(l)
  print l[0]
}
$ awk -f script.awk
1
0
```

1.5.3 Built-in Functions

There are multiple built-in functions in awk which make the programming of scripts easier.

Each built-in function accepts a certain number of arguments. In some cases, arguments can be omitted. The defaults for omitted arguments vary from function to function and are described under the individual functions. In some awk implementations, extra arguments given to built-in functions are ignored. However, in gawk, it is a fatal error to give extra arguments to a built-in function.

Numeric Functions

There are multiple in-built functions that help perform numeric manipulation to ease the development process. They are:

- `atan2(y,x)` - Returns the arc tangent of y/x in radians.
- `cos(x)` - Return the cosine of x in radians.
- `exp(x)` - Returns the exponential of $x = e^x$.
- `int(x)` - Return the nearest integer to x, located between x and zero and truncated toward zero. For example, `int(3)` is 3, `int(3.9)` is 3, `int(-3.9)` is −3, and `int(-3)` is −3 as well.
- `log(x)` - Return the natural logarithm of x, if x is positive; otherwise, return NaN ("not a number") on IEEE 754 systems. Additionally, gawk prints a warning message when x is negative.
- `rand()` - Return a random number. The values of `rand()` are uniformly distributed between zero and one. The value could be zero but is never one.
- `sin(x)` - Return the sine of x, with x in radians.
- `sqrt(x)` - Return the positive square root of x. gawk prints a warning message if x is negative. Thus, `sqrt(4)` is 2.
- `srand(x)` - Set the starting point, or seed, for generating random numbers to the value x. Each seed value leads to a particular sequence of random numbers. Thus, if the seed is set to the same value a second time, the same sequence of random numbers is produced again. If no argument is provided then the current date and time is used to set the seed value. This generates unpredictable pseudo-random numbers. It returns the previous seed value.

String Functions

`gawk` understands locales and does all string processing in terms of characters, not bytes. This distinction is particularly important to understand for locales where one character may be represented by multiple bytes. Thus, for example, `length()` returns the number of characters in a string, and not the number of bytes used to represent those characters. Similarly, `index()` works with character indices, and not byte indices.

String functions are used to manipulate strings in awk. They are:

- `asort(source[,dest[,how]])` - gawk sorts the values of source and replaces the indices of the sorted values of source with sequential integers starting with one. If the optional array dest is specified, then source is duplicated into dest. dest is then sorted, leaving the indices of source unchanged. This means that `asort` can also work on associative arrays, however, after sorting the keys are no longer preserved, rather it is converted into an indexed array.
- `asorti(source[,dest,[how]])` - Similar to `asort` , however, it sorts the indices of an associative array instead of the values and produces a indexed array.
- `gensub(regexp, replacement, how[, target])` - This is a **gawk** extension that allows for general substitution. Search the target string target for matches of the regular expression `regexp`. If how is a string beginning with 'g' or 'G' (short for "global"), then replace all matches of `regexp` with `replacement`. Otherwise, treat `how` as a number indicating which match of `regexp` to replace. Treat numeric

values less than one as if they were one. If no target is supplied, use `$0`. It **returns** the modified string as the result of the function. The original target string is **not changed**. The returned value is always a string, even if the original target was a number or a regexp value. `gensub()` is a general substitution function. Its purpose is to provide more features than the standard `sub()` and `gsub()` functions.

- `gsub(regex,replacement[, target])` - This function searches the target string `target` for all occurrences of the regular expression `regex`, and replaces them with `replacement`. It returns the number of substitutions made. The target string `target` **is changed**. If `target` is omitted then the `$0` is searched and altered.
- `index(in,find)` - This function searches the string `in` for the first occurrence of the string `find`, and returns the position in `in` where that occurrence begins. If `find` is not found in `in`, then `index()` returns zero. The indices are 1-indexed.
- `length([string])` - This function returns the length of the string `string`. If it is omitted then the entire `$0` is used.
- `match(string,regexp[, array])` - This function searches the string `string` for the longest, leftmost substring matched by the regular expression `regexp`. It returns the position in `string` where the matched substring begins, or zero if no match is found. It also sets the `RSTART` and `RLENGTH` variables. `RSTART` is set to the index of the start of the match, and `RLENGTH` is set to the length of the match.
- `patsplit(string, array[ ,fieldpat[, seps])` - This function splits the string `string` into fields in the array `array` using the field pattern `fieldpat`. It uses `fieldpat` to match fields. It

is a regular expression. If `fieldpat` is omitted then the value of `FPAT` is used instead. It returns the number of fields created. The original string `string` is not changed. The separator strings are stored in the `seps` array. The `seps` array is indexed by the field number. The value of each element is the separator string that was found before the corresponding field. The `seps` array has one more element than the `array` array.

- `split(string, array[, fieldsep[, seps]])` - This function splits the string `string` into fields in the array `array`, using the field separator `fieldsep`. It returns the number of fields created. The original string `s` is not changed. If `fieldsep` is omitted, then `FS` is used.
- `sprintf(fmt,expr-list)` - This function returns the string resulting from formatting the arguments according to the format string `fmt`. The format string is the same as that used by the `printf()` function. The result is not printed.
- `strtonum(str)` - This function converts the string `str` to a number. It returns the numeric value represented by `str`. If `str` starts with 0 then the number is treated as an octal number. If the string starts with `0x` or `0X` then the number is treated as a hexadecimal.
- `sub(regexp, replacement[, target])` - This function searches the target string `target` for the leftmost longest occurrence of the regular expression `regexp`, and replaces it with `replacement` . It returns the number of substitutions made (zero or one). The target string `t` is changed. If `target` is omitted then the `$0` is used.
- `substr(string, start[, length])` - This function returns the substring of `string` starting at position `start` and of length `length`. If `length`

is omitted then the substring till the end of the string is taken.

- `tolower(string)` - This function returns a copy of the string `string` with all the upper-case characters converted to lower-case.
- `toupper(string)` - This function returns a copy of the string `string` with all the lower-case characters converted to upper-case.

Let us explore some of these functions with examples.

asort and asorti

```
$ cat script.awk
BEGIN{
  a["one"] = "banana"
  a["three"] = "cherry"
  a["two"] = "apple"
  asort(a)
  for(i in a){
    print i, a[i]
  }
}
$ awk -f script.awk
1 apple
2 banana
3 cherry
```

However, if we use `asorti` instead of `asort`, then the keys are sorted instead of the values.

```
$ cat script.awk
BEGIN{
  a["one"] = "banana"
  a["three"] = "cherry"
  a["two"] = "apple"
  asorti(a)
  for(i in a){
    print i, a[i]
  }
}
```

```
$ awk -f script.awk
1 one
2 three
3 two
```

gensub

```
$ cat script.awk
BEGIN{
  s = "Hello world"
  print gensub(/l+/, "_", "g", s)
}
$ awk -f script.awk
He_o wor_d
```

Similarly you can try out with `gsub` and `sub`.

index and length

```
$ cat script.awk
BEGIN{
  s = "This is a string"
  print index(s, "is")
  print length(s)
}
$ awk -f script.awk
3
16
```

match

```
$ cat script.awk
BEGIN{
  s = "Hello World"
  print match(s,/l+/,a)
  print RSTART
  print RLENGTH
  for(i in a){
    print i ": " a[i]
  }
}
$ awk -f script.awk
3
```

```
3
2
0start: 3
0length: 2
0: ll
```

Using the a array we can get the matched string, the start of the match and the length of the match.

split and patsplit

```
$ cat script.awk
BEGIN{
  s = "this,is,a,csv,file"
  split(s,a,",",seps)
  for(i in a){
    print i, a[i]
  }
  for(i in seps){
    print i, seps[i]
  }
}
$ awk -f script.awk
1 this
2 is
3 a
4 csv
5 file
1 ,
2 ,
3 ,
4 ,
```

However, the field separator may not always be a fixed string. In that case, we can use the `patsplit` function.

```
$ cat script.awk
BEGIN{
  s = "This!is.a,line^with*different&
     delimiters"
  patsplit(s,a,/[A-Za-z]+/,seps)
  for(i in a){
    print i, a[i]
  }
  for(i in seps){
    print i, seps[i]
  }
}
$ awk -f script.awk
1 This
2 is
3 a
4 line
5 with
6 different
7 delimiters
0
1 !
2 .
3 ,
4 ^
5 *
6 &
7
```

In this case, the `seps` array is useful as it stores the separators that were found before each field. Note that the first and last element of the `seps` array are empty strings, as there is no separator before the first field and after the last field. Also note that the `seps` array starts from 0 and not 1.

substr

```
$ cat script.awk
BEGIN{
  s = "Hello World"
  print substr(s,1,4)
  print substr(s,7)
}
$ awk -f script.awk
Hell
World
```

If we do not specify the length, then the substring is taken till the end of the string.

tolower and toupper

```
$ cat script.awk
BEGIN{
  s = "A string"
  print tolower(s)
  print toupper(s)
}
$ awk -f script.awk
a string
A STRING
```

There are other types of in-built functions as well, such as Input Output Functions, Time Functions, and Bitwise Functions which we will not explore in depth here, but students are encouraged to go through the manual for those as well.

The `system()` function is of particular interest as it allows us to run shell commands from within awk. We will cover this in a later section.

1.6 Arrays

Arrays are a collection of elements, where each element is identified by an index or a key. In awk, arrays are associative, which means that the index can be a string or a number, or indexed, where the key is numeric.

Arrays in awk are created on the fly, and there is no need to declare them before using them. The first time an array element is referenced, the array is created.

```
$ cat script.awk
BEGIN{
  arr[0] = "hello"
  arr[1] = "world"
  print arr[0] arr[1]
  delete arr[0]
  print arr[0] arr[1]
  delete arr
  print arr[0] arr[1]
}
$ awk -f script.awk
helloworld
world
```

Elements of the arrays can be accessed using the index or key. The individual elements can be deleted using the `delete` function. We can also delete the entire array by passing the array name to the `delete` keyword.

1.6.1 Associative Arrays

Associative arrays are arrays where the index is a string or a number. The index can be any string or number, and the elements are stored in the array in no particular order. In awk, unlike modern python, the order of the keys in the array is not guaranteed. The keys, even if they are numbers, are stored as strings and need not be contiguous.

```
$ cat script.awk
BEGIN{
  arr["name"] = "John"
  arr["age"] = 24
  for(i in arr){
    print i, arr[i]
  }
  print "age" in arr
  print "gender" in arr
}
$ awk -f script.awk
```

```
age 24
name John
1
0
```

Observe a few points here:

- The keys are stored as strings, even if they are numbers.
- The keys are not stored in any particular order.
- We can check if a key exists in the array using the `in` keyword.
- We can iterate over the keys of the array using the `for` loop.

1.6.2 Multidimensional Arrays

POSIX awk supports multidimensional arrays by converting the indices to strings and concatenating them with a separator. The separator is the value of the `SUBSEP` variable. The default value of `SUBSEP` is the string `"\034"`, which is the ASCII value of the record separator. After the value is stored there is no way to retrieve the original indices, so it is not possible to iterate over the keys of the array.

We can check if an index exists in the array using the `in` keyword.

```
$ cat script.awk
BEGIN{
  arr[0,0] = 5
  print arr[0,0]
  print (0,0) in arr
  print (1,1) in arr
}
$ awk -f script.awk
5
1
0
```

However, this is not really a multidimensional array. It stores all the values in a single associative single dimensional array.

However, gawk supports true multidimensional arrays. Elements of a subarray are referred to by their own indices enclosed in square brackets, just like the elements of the main array.

```
$ cat script.awk
BEGIN {
  arr[0][0] = 5
  print arr[0][0]
  print 0 in arr
  print 0 in arr[0]
  print 1 in arr
}
$ awk -f script.awk
5
1
1
0
```

Similarly we can use nested for-loops to iterate over the elements of the array.

```
$ cat script.awk
BEGIN{
  arr[0][0] = 0
  arr[0][1] = 1
  arr[1][0] = 2
  arr[1][1] = 3

  for(i in arr){
    for(j in arr[i]){
      printf arr[i][j] " "
    }
    printf "\n"
  }
}
$ awk -f script.awk
0 1
```

```
2 3
```

1.7 Regular Expressions

As we have seen before, regular expressions are patterns that describe sets of strings. They are used to search for patterns in text.

Awk by default uses extended regular expressions.

A regular expression can be used as a pattern by enclosing it in slashes. Then the regular expression is tested against the entire text of each record. (Normally, it only needs to match some part of the text in order to succeed.)

For example, the below script tests if the record contains the string `bash` and prints the first field if it does. Here we are iterating over the records of the file `/etc/passwd` and printing those users who can login using the bash shell.

```
$ awk '/bash/ { print $1 }' /etc/passwd
root:x:0:0::/root:/bin/bash
sayan:x:1000:1001:Sayan:/home/sayan:/bin/bash
postgres:x:946:946:PostgreSQL
```

We can also match a regex on any arbitrary field or string using the ~ operator. Similarly, we can negate the match using the `!~` operator.

```
$ cat script.awk
$5 ~ /[A-Z]/{
  print $1, $5
}
$ awk -F: -f script.awk /etc/passwd
nobody Kernel Overflow User
dbus System Message Bus
systemd-coredump systemd Core Dumper
systemd-network systemd Network Management
```

```
systemd-oom systemd Userspace OOM Killer
systemd-journal-remote systemd Journal Remote
systemd-resolve systemd Resolver
systemd-timesync systemd Time Synchronization
_talkd User for legacy talkd server
avahi Avahi mDNS/DNS-SD daemon
named BIND DNS Server
nm-openconnect NetworkManager OpenConnect
nm-openvpn NetworkManager OpenVPN
ntp Network Time Protocol
openvpn OpenVPN
polkitd PolicyKit daemon
rpc Rpcbind Daemon
rpcuser RPC Service User
rtkit RealtimeKit
saned SANE daemon user
sddm SDDM Greeter Account
sayan Sayan
brltty Braille Device Daemon
gluster GlusterFS daemons
qemu QEMU user
colord Color management daemon
fwupd Firmware update daemon
geoclue Geoinformation service
libvirt-qemu Libvirt QEMU user
mysql MariaDB
nbd Network Block Device
passim Local Caching Server
postgres PostgreSQL user
redis Redis in-memory data structure store
metabase Metabase user
flatpak Flatpak system helper
```

Here we are matching the regex `[A-Z]` on the fifth field of the `/etc/passwd` file and printing the first and fifth fields if the regex matches. This prints all those records which have a capitalized name for the user.

1.7.1 Matching Empty Regex

An empty regex matches the invisible empty string at the start and end of the string and between each character. It can be visualized by using the `gsub()` function and replacing the empty strings with a character.

```
$ cat script.awk
BEGIN{
  a = "hello"
  gsub(//, "X", a)
  print a
}
$ awk -f script.awk
XhXeXlXlXoX
```

1.7.2 Character Classes

Awk supports character classes in regular expressions. A character class is a set of characters enclosed in square brackets. It matches any one of the characters in the set. The character classes discussed in the earlier chapters are consistent with the character classes supported by awk.

Remark 1.7.1 Awk also supports collating symbols and equivalence classes, however gawk does not support these.

Awk always matches the longest leftmost string that satisfies the regular expression. This is called the maximal match rule. For simple match/no-match tests, this is not so important. But when doing text matching and substitutions with the `match()`, `sub()`, `gsub()`, and `gensub()` functions, it is very important.

Understanding this principle is also important for regexp-based record and field splitting.

1.7.3 Regex Constant vs String Constant

The righthand side of a ~ or `!~` operator need not be a regexp constant (i.e., a string of characters between slashes). It may be any expression. The expression is evaluated and converted to a string if necessary; the contents of the string are then used as the regexp. A regexp computed in this way is called a dynamic regexp or a computed regexp.

```
$ cat script.awk
BEGIN{
  r = "[A-Z].*apple"
}
$0 ~ r
$ awk -f script.awk /usr/share/dict/words
Balmawhapple
John-apple
```

Here we are matching the regex stored in the variable `r` against the records of the file `/usr/share/dict/words`.

Remark 1.7.2 When using the ~ and `!~` operators, be aware that there is a difference between a regexp constant enclosed in slashes and a string constant enclosed in double quotes. If you are going to use a string constant, you have to understand that the string is, in essence, scanned twice: the first time when awk reads your program, and the second time when it goes to match the string on the lefthand side of the operator with the pattern on the right. This is true of any string-valued expression (such as `r`, shown in the previous example), not just string constants.

This matters when we are using escape sequences in the string constant. If using string constants then we have to escape the character twice, once for the string and once for the regex.

For example, `/\*/` is a regexp constant for a literal `*`. Only one backslash is needed. To do the same thing with a string, you have to type `"\\*"`. The first backslash escapes the second one so that the string actually contains the two characters `\` and `*`.

However, we should always use regex constants over string constants because:

- String constants are more complicated to write and more difficult to read. Using regexp constants makes your programs less error-prone. Not understanding the difference between the two kinds of constants is a common source of errors.
- It is more efficient to use regexp constants. awk can note that you have supplied a regexp and store it internally in a form that makes pattern matching more efficient. When using a string constant, awk must first convert the string into this internal form and then perform the pattern matching.
- Using regexp constants is better form; it shows clearly that you intend a regexp match.

1.7.4 Ignoring Case

Usually regex is case sensitive, and there is no way to make a regex case insensitive from the regex itself. `grep` tackles this by providing the `-i` flag.

In awk, we can perform case insensitive matching in three ways:

- By matching both the uppercase and lowercase characters. `[wW]` matches both `w` and `W`.
- By using the `tolower()` or `toupper()` functions. We can convert the data to lowercase and have the regex in lowercase as well.
- Using the `IGNORECASE` variable. If set to any non-zero value, awk will ignore case when matching regex.

```
$ cat script.awk
BEGIN {
  x = "Hello"
  if (x ~ /hello/) {
    print "first check"
  }
  IGNORECASE = 1
  if (x ~ /hello/) {
    print "second check"
  }
}
$ awk -f script.awk
second check
```

As visible, the first check fails as the regex is case sensitive, however the second check passes as we have set the `IGNORECASE` variable to a non-zero value.

1.8 Control Structures

Just like any programming language, awk supports control structures like if-else, for, while, and do-while loops, which can be used to control the flow of the program and create programs for any use case.

1.8.1 if-else

Conditional branching is done using the `if-else` construct in awk. The syntax is similar to that of other programming languages.

if

```
$ cat script.awk
BEGIN{
  x = 5
  if(x > 0){
    print "Positive"
  }
}
$ awk -f script.awk
Positive
```

The `if` statement checks if the value of `x` is greater than zero and prints `Positive` if it is. Any boolean expression can be used in the `if` statement.

Remark 1.8.1 A non-zero value is considered true and a zero value is considered false in awk, similar to other programming languages and unlike shell scripting.

We can also use the regex matching operator in the `if` statement.

```
$ cat script.awk
BEGIN{
  x = "Hello"
  if(x ~ /l+/){
    print "Matched"
  }
}
$ awk -f script.awk
Matched
```

Here we are checking if the string `x` contains one or more `l` characters and printing `Matched` if it does.

if-else

```
$ cat script.awk
{
  if($1 > 0){
    print "Positive"
  } else {
    print "Negative"
  }
}
$ awk -f script.awk <<< "1"
Positive
$ awk -f script.awk <<< "-1"
Negative
```

Now we can conditionally perform one block of code or another based on the value of the expression. Since boolean expressions can have only two values, we can use the `else` keyword to execute a block of code if the expression evaluates to false. This ensures that one of the two blocks of code is always executed. [*]

However, note that not all real world scenarios are boolean in nature. For example, all numbers cannot be classified as either positive or negative. Since zero is neither positive nor negative. We can use the `else if` construct to handle such cases.

if-else if

```
$ cat script.awk
{
  if($1 > 0){
```

[*] Unless there is a runtime error or the program is terminated.

```
    print "Positive"
  } else if($1 < 0){
    print "Negative"
  } else {
    print "Zero"
  }
}
$ awk -f script.awk <<< "1"
Positive
$ awk -f script.awk <<< "-1"
Negative
$ awk -f script.awk <<< "0"
Zero
```

Now the program can handle all cases, positive, negative, and zero.

We can add as many `else if` blocks as required to handle all the cases.

The program will try to find the first block whose condition is true and execute that block. If no block is found then the `else` block is executed.

If a block is found then the program will not check the conditions of the other blocks.

```
$ cat script.awk
BEGIN {
  x = 15
  if(x % 5 == 0){
    print "Divisible by 5"
  } else if(x % 3 == 0){
    print "Divisible by 3"
  }
  else if(x % 2 == 0){
    print "Divisible by 2"
  }
}
$ awk -f script.awk
Divisible by 5
```

Even though the number is divisible by 3 and 5, only the first block is executed as the program stops checking the conditions once a block is matched.

1.8.2 Ternary Operator

The ternary operator is a shorthand for the `if-else` construct. It is used to assign a value to a variable based on a condition.

```
$ cat script.awk
{
  print ($1 > 0) ? "Positive" : "Negative"
}
$ awk -f script.awk <<< "1"
Positive
$ awk -f script.awk <<< "-1"
Negative
```

1.8.3 for loop

The `for` loop is used to iterate over a sequence of values.

AWK includes both the C-style `for` loop and the `for-in` loop.

C-style for loop

```
$ cat script.awk
BEGIN{
  for(i = 1; i <= 5; i++){
    print i
  }
}
$ awk -f script.awk
1
2
```

```
3
4
5
```

The `for` loop has three parts separated by semi-colons. The first part is the initialization, the second part is the condition, and the third part is the increment or decrement.

When the loop is first encountered, the initialization part is executed. Then the condition is checked. If the condition is true, then the block of code is executed. After the block of code is executed, the increment part is executed. Then the condition is checked again. This process is repeated until the condition is false.

This means that the variable `i` is incremented by one in each iteration and the loop runs until `i` is less than or equal to 5. After the loop is done, the value of `i` is 6. [5]

5: If this is not clear why the value of `i` is 6, try dry-running the code on a piece of paper to get hang of how C-style for loops work.

However, the value of `i` can be changed inside the loop as well.

```
$ cat script.awk
BEGIN{
  for(i = 1; i <= 5; i++){
    print i
    if(i == 3){
      i = 6
    }
  }
}
$ awk -f script.awk
1
2
3
```

The upadte rule can have any kind of operation, not just increments.

```
$ cat script.awk
BEGIN{
  for(i = 1; i <= 1024; i *= 2){
    print i
  }
}
$ awk -f script.awk
1
2
4
8
16
32
64
128
256
512
1024
```

Here we are doubling the value of `i` in each iteration to print the first ten powers of 2.

for-in loop

The `for-in` loop is used to iterate over the elements of an array. However, awk does not have indexed arrays, so the `for-in` loop is used to iterate over the keys of an associative array.

Remark 1.8.2 The order of the keys in an associative array is not guaranteed, so the order of the keys in the `for-in` loop is not guaranteed.

```
$ cat script.awk
BEGIN {
    assoc["key1"] = "val1"
    assoc["key2"] = "val2"
    assoc["key3"] = "val3"
    for (key in assoc)
```

```
        print key, assoc[key];
}
$ awk -f script.awk
key3 val3
key1 val1
key2 val2
```

Here we are iterating over the keys of the associative array `assoc` and printing the key and value of each element. The order of the keys is not guaranteed to be same as the order of insertion.

1.8.4 Iterating over fields

The biggest use-case of for-loops in awk is to iterate over the fields of a record.

```
$ cat script.awk
{
  for(i = 1; i <= NF; i++){
    print i, $i
  }
}
$ grep '^[^#]' /etc/passwd | head -n1
nobody:*:-2:-2:Unprivileged User:/var/empty:/
    usr/bin/false
$ grep '^[^#]' /etc/passwd | head -n1 | awk -F
    : -f script.awk
1 nobody
2 *
3 -2
4 -2
5 Unprivileged User
6 /var/empty
7 /usr/bin/false
```

1.8.5 while loop

The while loop in awk is similar to the while loop in other programming languages.

```
$ cat script.awk
BEGIN{
  i = 1
  while(i <= 5){
    print i
    i++
  }
}
$ awk -f script.awk
1
2
3
4
5
```

The `while` loop is used to execute a block of code as long as the condition is true. This is useful if you do not know how many times the loop is supposed to run. We can also use the `break` and `continue` statements inside the loop to break out of the loop or skip the current iteration.

```
$ cat script.awk
BEGIN{
  i = 1
  while(i <= 10){
    if(i == 3){
      i++
      continue
    }
    print i
    i++
    if(i == 5){
      break
    }
  }
}
$ awk -f script.awk
1
2
4
```

This script skips the iteration when `i` is 3 and breaks out of the loop when `i` is 5.

We can also match regex inside the condition.

```
$ cat script.awk
{
  i = 1
  while($i ~ /^[A-Z]/){
    print i, $i
    i++
  }
}
$ cat data.txt
Hello World
This is a test
$ awk -f script.awk data.txt
1 Hello
2 World
1 This
```

The script prints the fields of each record as long as the field starts with an uppercase letter.

1.8.6 do-while loop

The **do loop** is a variation of the while looping statement. The do loop executes the body once and then repeats the body as long as the condition is true.

```
$ cat script.awk
BEGIN{
  i = 1
  do {
    print i
    i++
  } while(i <= 5)
}
$ awk -f script.awk
1
2
3
4
5
```

This means that even if the condition is false, the body is executed at least once. There is no difference with the while loop after the first iteration.

```
$ cat script.awk
BEGIN{
  i = 6
  do {
    print i
    i++
  } while(i <= 5)
}
$ awk -f script.awk
6
```

1.8.7 switch-case

Switch Case is a control structure that allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Remark 1.8.3 Switch case is a **gawk** extension and is not available in POSIX awk.

The switch statement allows the evaluation of an expression and the execution of statements based on a case match. Case statements are checked for a match in the order they are defined. If no suitable case is found, the default section is executed, if supplied.

Each case contains a single constant, be it numeric, string, or regexp. The switch expression is evaluated, and then each case's constant is compared against the result in turn.

Syntax:

```
switch (expression) {
case value or regular expression:
    case-body
default:
    default-body
}
```

Control flow in the switch statement works as it does in C. Once a match to a given case is made, the case statement bodies execute until a `break`, `continue`, `next`, `nextfile`, or `exit` is encountered, or the end of the switch statement itself.

We will discuss `next` and `nextfile` in a later subsection.

```
$ cat script.awk
{
  switch($0){
    case 1:
      print "One"
      break
    case 2:
      print "Two"
      break
    case 3:
      print "Three"
      break
    case 4:
      print "Four"
      break
    case 5:
      print "Five"
      break
    default:
      print "Default"
  }
}
$ gawk -f script.awk <<< 5
Five
$ gawk -f script.awk <<< 2
Two
$ gawk -f script.awk <<< 6
Default
```

Here we are matching the value of the record against the cases and printing the corresponding value. Observe that we need to specify the keyword `break` after each case. If we however do not use it, then the program will continue to execute the next case as well.

```
$ cat script.awk
{
  switch($0){
    case 1:
      print "One"
    case 2:
      print "Two"
    case 3:
      print "Three"
    case 4:
      print "Four"
    case 5:
      print "Five"
    default:
      print "Default"
  }
}
$ gawk -f script.awk <<< 5
Five
Default
```

```
$ gawk -f script.awk <<< 2
Two
Three
Four
Five
Default
```

This behaviour, although might seem counter-intuitive, is useful in some cases. For example, if we want to execute the same block of code for multiple cases, then we can do so without repeating the code.

```
$ cat script.awk
{
  switch($0 % 10){
    case 0:
        print "Multiple of 10"
    case 5:
        print "Multiple of 5"
        break
    default:
        print "Not a multiple of 5 or 10"
  }
}
$ gawk -f script.awk <<< 15
Multiple of 5
$ gawk -f script.awk <<< 20
Multiple of 10
Multiple of 5
$ gawk -f script.awk <<< 7
Not a multiple of 5 or 10
```

However, unlike the **break** keyword, the **continue** keyword does not directly affect the switch statement, but rather the loop that encloses it.

```
$ cat script.awk
BEGIN {
    for (i = 1; i <= 3; i++) {
        switch (i) {
            case 1:
                print "Case 1"
                continue
            case 2:
                print "Case 2"
                break
            case 3:
                print "Case 3"
        }
        print "End of loop iteration", i
    }
}
$ gawk -f script.awk
Case 1
Case 2
End of loop iteration 2
Case 3
End of loop iteration 3
```

Matching Regex

We can also match regex in the case statement. This is unlike most programming languages where only constants are allowed.

```
$ cat script.awk
{
  switch($0){
    case /^[A-Z]+$/:
      print "Strictly Uppercase"
      break
    case /^[a-z]+$/:
      print "Strictly Lowercase"
      break
    default:
      print "Neither"
  }
}
$ gawk -f script.awk <<< "HELLO"
Strictly Uppercase
$ gawk -f script.awk <<< "hello"
Strictly Lowercase
```

```
$ gawk -f script.awk <<< "Hello"
Neither
```

1.8.8 break

The break statement jumps out of the innermost for, while, or do loop that encloses it.

```
$ cat script.awk
{
    num = $1
    for (divisor = 2; divisor * divisor <= num
     ; divisor++) {
        if (num % divisor == 0)
            break
    }
    if (num % divisor == 0)
        printf "Smallest divisor of %d is %d\n
     ", num, divisor
    else
        printf "%d is prime\n", num
}
$ awk -f script.awk <<< 15
Smallest divisor of 15 is 3
$ awk -f script.awk <<< 17
17 is prime
```

Here we are finding the smallest divisor of a number. If the number is prime, then the smallest divisor is the number itself.

1.8.9 continue

The continue statement skips the rest of the loop body and starts the next iteration of the loop.

```
$ cat script.awk
{
    for (i = 1; i <= 10; i++) {
        if (i % 2 == 0)
            continue
        print i
    }
}
$ awk -f script.awk
1
3
5
7
9
```

Here we are printing all the odd numbers from 1 to 10.

1.8.10 next and nextfile

. . . BEGINFILE and ENDFILE

1.9 Printing

1.9.1 print

1.9.2 printf

1.9.3 OFS and ORS

1.10 Files and Command Line Arguments

1.10.1 FILENAME, ARGV and ARGC

1.11 Input using getline

1.11.1 getline without arguments

1.11.2 getline into variable

1.11.3 getline with input file

1.11.4 getline into variable from file

1.11.5 getline with pipe

1.11.6 getline into a variable from a pipe

1.11.7 getline with coprocess

1.12 Redirection and Piping

1.12.1 Output Redirection

1.12.2 Input Redirection

1.12.3 Piping to other commands

and try to understand how they work. Further it is beneficial to try to create scripts of their own to solve problems they face using **bash**, **sed**, and **awk**.