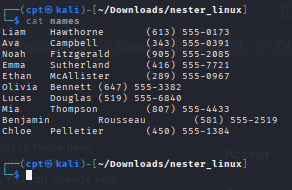
Same as sed, awk considered non-interactive text editor but provide greater/wider range of capabilities. This tool even has its own scripting languages, syntactically similar to C. Like sed, awk executes a set of instructions for each line of input. You can specify instructions on the command line or create a script file.

**awk ’***instructions***’** *files*

*Records and Fields*

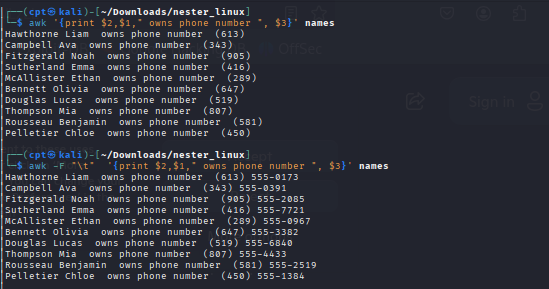
Awk makes the assumption that its input is structured and not just an endless string of characters. In the simplest case, it takes each input line as a record and each word, separated by spaces or tabs, as a field. (The characters separating the fields are often referred to as *delimiters*.) The following records in the file *names* has three fields, separated by either a space or a tab.



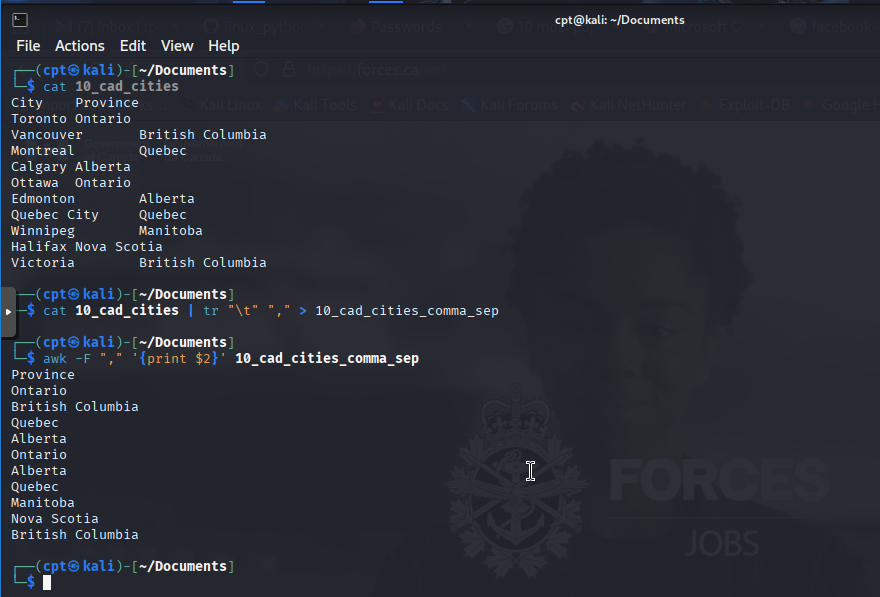
Two or more consecutive spaces and/or tabs count as a single delimiter.

*Referencing and Separating Fields*

Awk allows you to refer to fields in actions using the field operator $. This operator is followed by a number or a variable that identifies the position of a field by number. “$1” refers to the first field, “$2” to the second field, and so on. “$0” refers to the entire input record. The following example displays the last name first and the first name second, followed by the phone number. Notice that if explicit delimiter is not specified, both space and tab are considered a valid delimiter.

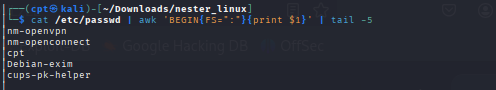


Use the *-F* option to change the field separator to a comma.



It is usually a better practice, and more convenient, to specify the field separator in the script itself. The system variable FS can be defined to change the field separator. Because this must be done before the first input line is read, we must assign this variable in an action controlled by the BEGIN rule.

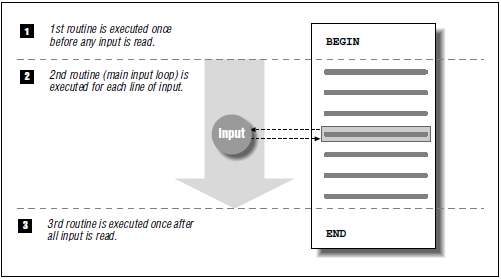
**BEGIN { FS = ":" }**



It’s important to understand the basic model that awk offers the programmer. Part of the reason why awk is easier to learn than many programming languages is that it offers such a well-defined and useful model to the programmer. An awk program consists of what we will call a main input loop. A loop is a routine that is executed over and over again until some condition exists that terminates it. You don’t write this loop, it is given—it exists as the framework within which the code that you do write will be executed. The main input loop in awk is a routine that reads one line of input from a file and makes it available for processing. The main input loop is executed as many times as there are lines of input. As you saw in the “Hello, world” examples, this loop does not execute until there is a line of input. It terminates when there is no more input to be read. Awk allows you to write two special routines that can be executed before any input is read and after all input is read. These are the procedures associated with the BEGIN and END rules, respectively. In other words, you can do some preprocessing before the main input loop is ever executed and you can do some post processing after the main input loop has terminated. The BEGIN and END procedures are optional.

You can think of an awk script as having potentially three major parts: what happens before, what happens during, and what happens after processing the input. Figure 7-1 shows the relationship of these parts in the flow of control of an awk

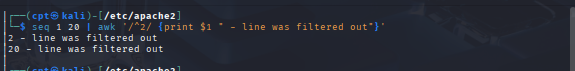
script.



When awk reads an input line, it attempts to match each pattern-matching rule in a script. Only the lines matching the particular pattern are the object of an action. If no action is specified, the line that matches the pattern is printed (executing the print statement is the default action). Consider the following script:

**/ˆ$/ { print "This is a blank line." }**

In the example below, we filtered out all the line that start with digit 2 and additional sentence was appended to it.



*Expressions*

The use of expressions in which you can store, manipulate, and retrieve data is quite different from anything you can do in sed, yet it is a common feature of most programming languages. A variable is an identifier that references a value. To define a variable, you only have to name it and assign it a value. The name can only contain letters, digits, and underscores, and may not start with a digit. Case distinctions in variable names are important: Salary and salary are two different variables. Variables are

not declared; you do not have to tell awk what type of value will be stored in a variable. Each variable has a string value and a numeric value, and awk uses the appropriate value based on the context of the expression. (Strings that do not consist of numbers have a numeric value of 0.) Variables do not have to be initialized; awk automatically initializes them to the empty string, which acts like 0 if used as

a number. The following expression assigns a value to x:

x = 1

x is the name of the variable, = is an assignment operator, and 1 is a numeric constant.

The following expression assigns the string “Hello” to the variable z:

z = "Hello"

A space is the string concatenation operator. The expression:

z = "Hello" "World"

concatenates the two strings and assigns “HelloWorld” to the variable z.

The dollar sign ($) operator is used to reference fields. The following expression

assigns the value of the first field of the current input record to the variable w:

w = $1

A variety of operators can be used in expressions. Arithmetic operators are listed below

+ Addition

- Subtraction

\* Multiplication

/ Division

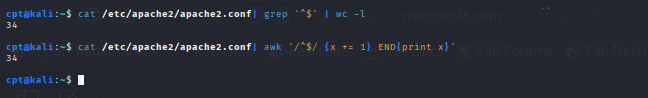
% Modulo

ˆ Exponentiation

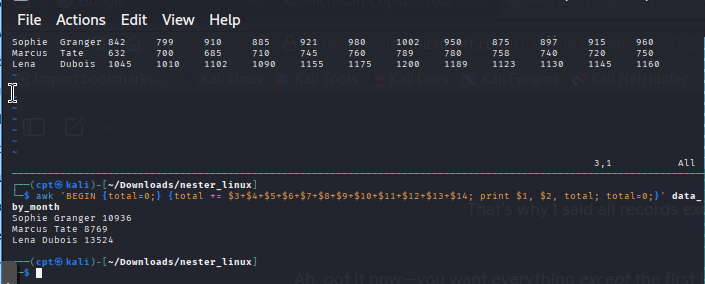
\*\* Exponentiation

Look at the following example, which counts each blank line in a file.

/ˆ$/ {x += 1} END{print x}



One more a bit more complex example of summing 12 month data for each person.

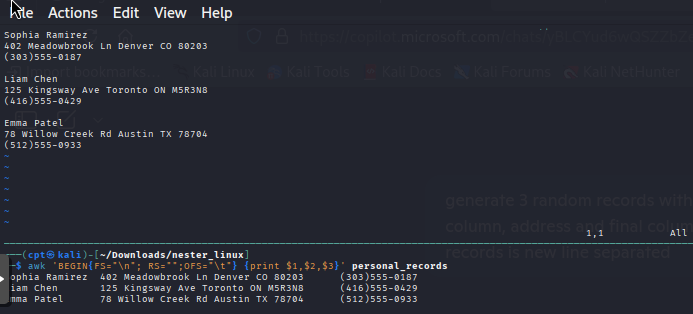


*System Variables*

There are a number of system or built-in variables defined by awk. Awk has two types of system variables. The first type defines values whose default can be changed, such as the default field and record separators. The second type defines values that can be used in reports or processing, such as the number of fields found in the current record, the count of the current record, and others. These are automatically updated by awk; for example, the current record number and input file name. There are a set of default values that affect the recognition of records and fields on input and their display on output. The system variable FS defines the field separator. By default, its value is a single space, which tells awk that any number of spaces and/or tabs separate fields. FS can also be set to any single character, or to a regular expression. Earlier, we changed the field separator to a comma in order

to read a list of names and addresses. The output equivalent of FS is OFS, which is a space by default. Awk defines the variable NF to be the number of fields for the current input record. Awk also defines RS, the record separator, as a newline. RS is a bit unusual; it’s the only variable where awk only pays attention to the first character of the value. The output equivalent to RS is ORS, which is also a newline by default. In the next section, “Working with Multiline Records,” we’ll show how to change the default record separator. Awk sets the variable NR to the number of the current input record. It can be used to number records in a list. The variable FILENAME contains the name of the current input file. The variable FNR is useful when multiple input files are used as it provides the number of the current record relative to the current input file.

Let’s suppose we have a file of records but instead of having one record corresponding to one line, data if saved in blocks. By changing the perception of what is column separator, we can parse this data with ease.



*Relational and Boolean Operators*

Relational and Boolean operators allow you to make comparisons between two

expressions. The relational operators are found in Table 7-4.

Table 7−4: Relational Operators

Operator Description

< Less than

> Greater than

<= Less than or equal to

>= Greater than or equal to

== Equal to

!= Not equal to

˜ Matches

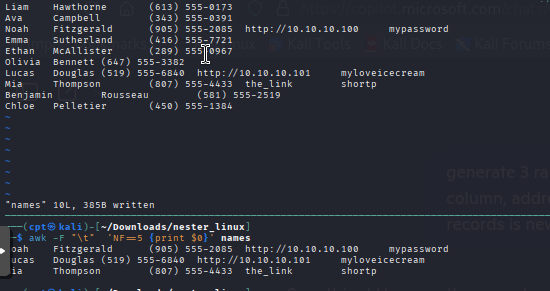
!˜ Does not match

A relational expression can be used in place of a pattern to control a particular

action. For instance, if we wanted to limit the records selected for processing to

those that have five fields, we could use the following expression:

NF == 5



Boolean operators allow you to combine a series of comparisons.

|| Logical OR

&& Logical AND

! Logical NOT

The parentheses in the next example show which expression would be evaluated first based on the rules of precedence.

(NR > 1 && NF >= 2) || $1 ˜ /\t/

In other words, both of the expressions in parentheses must be true or the right hand side must be true. You can use parentheses to override the rules of precedence, as in the following example which specifies that two conditions must be true.

NR > 1 && (NF >= 2 || $1 ˜ /\t/)

The first condition must be true and either of two other conditions must be true.

Given an expression that is either true or false, the ! operator inverts the sense of the expression.

! (NR > 1 && NF > 3)

This expression is true if the parenthesized expression is false.

*String Functions*

The built-in string functions are much more significant and interesting than the numeric functions. Because awk is essentially designed as a string-processing language, a lot of its power derives from these functions.

Awk Function Description

**gsub**(r,s,t) Globally substitutes s for each match of the regular expression r in the string t. Returns the number of substitutions. If t is not supplied, defaults to $0.

**index**(s,t) Returns position of substring t in string s or zero if not present.

**length**(s) Returns length of string s or length of $0 if no string is supplied.

**match**(s,r) Returns either the position in s where the regular expression r

**begins**, or 0 if no occurrences are found. Sets the values of RSTART and RLENGTH.

**split**(s,a,sep) Parses string s into elements of array a using field separator sep; returns number of elements. If sep is not supplied, FS is used. Array splitting works the same way as field splitting.

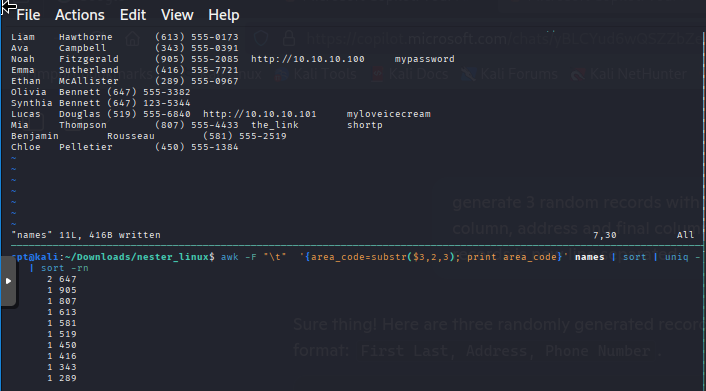
**sprintf**(“fmt ”,expr) Uses pr intf for mat specification for expr.

**sub**(r,s,t) Substitutes s for first match of the regular expression r in the string t. Returns 1 if successful; 0 otherwise. If t is not supplied, defaults to $0.

**substr**(s,p,n) Returns substring of string s at beginning position p up to a maximum length of n. If n is not supplied, the rest of the string from p is used.

**tolower**(s) Translates all uppercase characters in string s to lowercase and returns the new string.

**toupper**(s) Translates all lowercase characters in string s to uppercase and returns the new string.



*Conditional Statements*

A conditional statement allows you to make a test before performing an action. In the previous chapter, we saw examples of pattern matching rules that were essentially conditional expressions affecting the main input loop. In this section, we look at conditional statements used primarily within actions.

A conditional statement is introduced by if and evaluates an expression placed in parentheses.

The syntax is:

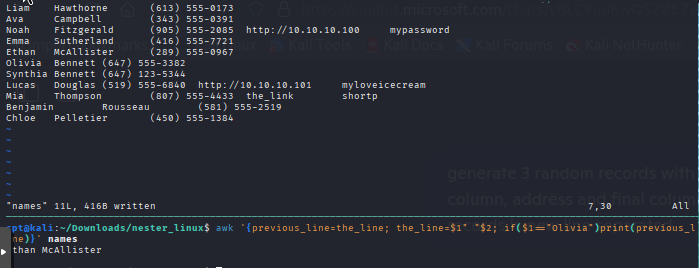
if ( expression )

action1

[else

action2]

Simple Example. Let’s say we need to display first and last name of the record prior to all records with first name “Olivia”:



*The getline Function*

The getline function is used to read another line of input. Not only can getline read from the regular input data stream, it can also handle input from files and pipes. The getline function is similar to awk’s next statement. While both cause the next input line to be read, the next statement passes control back to the top of the script. The getline function gets the next line without changing control in the

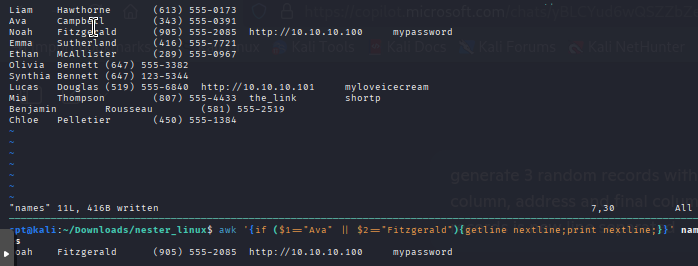
script. Possible return values are:

1 If it was able to read a line.

0 If it encounters the end-of-file.

-1 If it encounters an error.

Let’s solve a similar task to the previous one but instead of printing record located prior to the current one that match the condition, we must print next one following the current one that match the condition.



*The system( ) Function*

The system( ) function executes a command supplied as an expression.\* It does not, however, make the output of the command available within the program for processing. It returns the exit status of the command that was executed. The script waits for the command to finish before continuing execution.

