awk tutorial

Why awk?

The Awk text-processing programming language and is a useful tool for manipulating text.

- Awk recognizes the concepts of "file", "record", and "field".
- A file consists of records, which by default are the lines of the file. One line becomes one record.
- Awk operates on one record at a time.
- A record consists of fields, which by default are separated by any number of spaces or tabs.
- Field number 1 is accessed with $1, field 2 with $2, and so forth. $0 refers to the whole record.

Now, for an explanation of the { print } code block. In awk, curly braces are used to group blocks of code together, similar to C. Inside our block of code, we have a single print command. In awk, when a print command appears by itself, the full contents of the current line are printed.

```
$ awk '{ print $0 }' /etc/passwd
```

In awk, the $0 variable represents the entire current line, so print and print $0 do exactly the same thing.

```
$ awk '{ print "" }' /etc/passwd
```

```
$ awk '{ print "hello" }' /etc/passwd
```

Running this script will fill your screen with hello's.

AWK Variables

awk variables are initialized to either zero or the empty string the first time they are used.

Variables

- Variable declaration is not required
- May contain any type of data, their data type may change over the life of the program
- Must begin with a letter and continuing with letters, digits and underscores
- Are case senstive
- Some of the commonly used built-in variables are:

    - NR -- The current line's sequential number
    - NF -- The number of fields in the current line
    - FS -- The input field separator; defaults to whitespace and is reset by the -F command line parameter

```
$ echo “3 56” > calc
$ echo “567 89” >> calc
$ cat calc
3 56
567 89
$ awk '{d=($2-($1-4));s=($2+$1);print d/sqrt(s),d*d/s }'calc
7.42077 55.0678
-18.5066 342.494
```

in above example we have a file calc with two rows and two columns. Note that the final statement, a "print" in this case, does not need a semicolon. It doesn't hurt to put it in, though.

Integer variables can be used to refer to fields. If one field contains information about which other field is important, this script will print only the important field:

```
$ awk '{imp=$1; print $imp }' calc
```

The special variable NF tells you how many fields are in this record. This script prints the first and last field from each record, regardless of how many fields there are:
if now calc file is

```
$ echo “3 56 abd” > calc
$ echo “567 89 xyz” >> calc
$ cat calc
3 56 abd
567 89 xyz

$ awk '{print $1,$NF }' calc
3 abd
567 xyz
```

Begin and End

Any action associated with the BEGIN pattern will happen before any line-by-line processing is done. Actions with the END pattern will happen after all lines are processed.
1.One is to just mash them together, like so: >

```
$ awk 'BEGIN{print"fee"} $1=="foo"{print"fi"}
       END{print"fo fum"}' filename
```

AWK Arrays

awk has arrays, but they are only indexed by strings. This can be very useful, but it can also be annoying. For example, we can count the frequency of words in a document (ignoring the icky part about printing them out):

```
awk '{for(i=1;i <=NF;i++) freq[$i]++ }' filename
```

The array will hold an integer value for each word that occurred in the file. Unfortunately, this treats "foo", "Foo", and "foo," as different words. Oh well. How do we print out these frequencies? awk has a special "for" construct that loops over the values in an array. This script is longer than most command lines, so it will be expressed as an executable script:

```
#!/usr/bin/awk -f
{for(i=1;i <=NF;i++) freq[$i]++ }
END{for(word in freq) print word, freq[word]}
```

AWK Regular expressions and blocks

```
awk '/pattern_to_match/ {actions}' input_file
awk '/foo/ { print }' abc.txt
cat abc.txt | awk '/[0-9]+.[0-9]*/ { print }'
```

Expressions and blocks

fredprint

$1 == "fred" { print $3 }
root
$5 ~ /root/ { print $3 }

AWK Conditional statements

```
awk '{
    if ( $1 ~ /root/ )
   {
    print $1
   }
 }' /etc/passwd
```

Both scripts function identically. In the first example, the boolean expression is placed outside the block, while in the second example, the block is executed for every input line, and we selectively perform the print command by using an if statement. Both methods are available, and you can choose the one that best meshes with the other parts of your script.

```
if
{
        if ( $1 == "foo" ) {
                if ( $2 == "foo" ) {
                        print "uno"
                } else {
                        print "one"
                }
        } else if ($1 == "bar" ) {
                print "two"
        } else {
                print "three"
        }
 }
```

```
if
! /matchme/ { print $1 $3 $4 }
{
        if ( $0 !~ /matchme/ ) {
                print $1 $3 $4
        }
}
```

Both scripts will output only those lines that don't contain a matchme character sequence. Again, you can choose the method that works best for your code. They both do the same thing.

($1 == "foo") && ($2 == "bar") { print }