Lecture 11B: awk: Like Cut, But Better

Introduction

An excellent resource can be found here: why you should learn just a little awk.

awk commands look like this:

```
awk '{ }'
```

Inside the single quotes is a small program with its own syntax. Let's learn a little bit of it.

```
echo 'this is a test' | awk '{print $0}'
```

```
this is a test
```

As you can see, `$0` is returning the whole string. `awk` is very useful for dividing by a delimiter, similar to what `cut` does. But `awk` is much more powerful.

```
echo "this is a test" | awk '{print $1}'
```

```
this
```

```
echo "this is a test" | awk '{print $2}'
```

```
is
```

```
echo "this is a test" | awk '{print $3}'
```

```
a
```

```
echo "this is a test" | awk '{print $4}'
```

```
test
```

Special Note about Quotes

Notice that we've been using strong quotes when passing input to `awk`. It should be clear why: notice that `$1` in the context of *Bash* has one meaning (ie. the first argument passed to the script) and a completely *different* meaning in the context of *awk* (ie. the first field). We don't *want* any substitution to occur to our input before it gets to `awk`. Another way to think of it is: we don't want Bash to be interpreting our input. We want our instructions to be taken literally by `awk`.

If that makes sense, great. If it isn't clear, just remember: put instructions to `awk` in strong quotes. If you get to the point where you are calling `awk` from inside a script and you want to pass variables into it, rest assured there are ways to do it but they are outside of the scope for this course.

Number of Fields

By default, `awk` considers one or more spaces to be a delimiter. This is why our simple example has worked without any special setup.

NF contains the number of fields. So you can substitute $4 with $NF and get the same result.

```
 echo "this is a test" | awk '{print $NF}'
```

```
test
```

```
 echo "this is a test" | awk '{print NF}'
```

```
4
```

You can also subtract from NF:

```
 echo "this is a test" | awk '{print (NF-1)}'
```

```
3
```

```
 echo "this is a test" | awk '{print $(NF-1)}'
```

```
a
```

```
 echo "this is a test" | awk '{print $1, $(NF-1)}'
```

```
this a
```

Cars Example

Instead of piping into `awk`, let's give it our file `cars`:

```
 awk '{print $1, $2}' cars
```

```
plym fury
chevy nova
ford mustang
volvo gl
ford ltd
Chevy nova
fiat 600
honda accord
ford thundbd
toyota tercel
chevy impala
ford bronco
```

Let's print out prices for our car lot. Buyers don't need to know how old these and used these cars are. :)

`awk '{print $1,$2":₩t $"$NF}' cars

```
 plym fury:  $ 2500
chevy nova: $ 3000
ford mustang:   $ 17000
volvo gl:   $ 9850
ford ltd:   $ 10500
Chevy nova: $ 3500
fiat 600:   $ 450
honda accord:   $ 6000
ford thundbd:   $ 17000
toyota tercel:  $ 750
chevy impala:   $ 1550
ford bronco:    $ 9525
```

Notice that I entered some formatting in my print statement, to get my output looking the way I want it to.

Using NR

Another useful variable to remember is `NR` . This will return the number of the line being parsed. So one useful thing you can do is to number your lines:

```
awk '{print NR". " $0}' cars
```

```
 1. plym    fury    77      73      2500
2. chevy   nova    79      60      3000
3. ford    mustang 65      45      17000
4. volvo   gl      78      102     9850
5. ford    ltd     83      15      10500
6. Chevy   nova    80      50      3500
7. fiat    600     65      115     450
8. honda   accord  81      30      6000
9. ford    thundbd 84      10      17000
10. toyota  tercel  82      180     750
11. chevy   impala  65      85      1550
12. ford    bronco  83      25      9525
```

Here's another way to use NR: `awk 'NR == 1, NR ==  6' cars`

```
 plym    fury    77      73      2500
chevy   nova    79      60      3000
ford    mustang 65      45      17000
volvo   gl      78      102     9850
ford    ltd     83      15      10500
Chevy   nova    80      50      3500
```

...with this you can specify a line (or several!) to be parsing.

Number Comparisons

If the third field equals 65:

```
awk '$3 == 65' cars
```

```
 ford    mustang 65      45      17000
fiat    600     65      115     450
```

```
chevy   impala  65      85      1550
```

Notice we have no `{print}` statements, so the default behaviour is just to print the entire line.

If the fifth field is less than or equal to 3000:

```
awk '$5 <= 3000' cars
```

```
 plym    fury    77      73      2500
chevy   nova    79      60      3000
fiat    600     65      115     450
toyota  tercel  82      180     750
chevy   impala  65      85      1550
```

Regular Expressions

`awk` will accept regular expressions as well as extended regular expressions. By default, we look at the entire line. The regular expression here (^t) matches every line that starts with **t**. We put the regular expression between slashes. (/ /).

```
awk '/^t/ {print $1,$2,$4}' cars
```

```
toyota tercel 180
```

We can also specify a field to be matched. Let's look at the fifth field (price), and match all with four digits. Notice the tilde (~). Notice also that we are considering field number 5 to be a *complete line* by itself. Thus our regex should use `^...$` to avoid 'spillage'.

```
awk '$5 ~ /^[0-9]{4}$/' cars
```

```
 plym    fury    77      73      2500
chevy   nova    79      60      3000
volvo   gl      78      102     9850
Chevy   nova    80      50      3500
honda   accord  81      30      6000
chevy   impala  65      85      1550
ford    bronco  83      25      9525
```

Specifying a Delimiter

This can be done with the option `-F` followed by the delimiter you choose. For example:

```
awk -F',' '{print $4}' wireshark.csv
```

Summary

- `awk '<match> {<what to print>}' <file>` : Basic syntax. **Use strong quotes**.
- `{print $0}` : Print whole line.

- `$1, $2 ... $NF` : Fields 1 to *n*. awk's default delimiter is one or more spaces.
- `NR` : Number of rows.
- `/<regex>/` : Put regular expressions between slashes.
- `$1 ~ /<regex>/` : Use regex on specific field (for example, the first column).
- `-F'<delimiter>'` : Specify a delimiter.