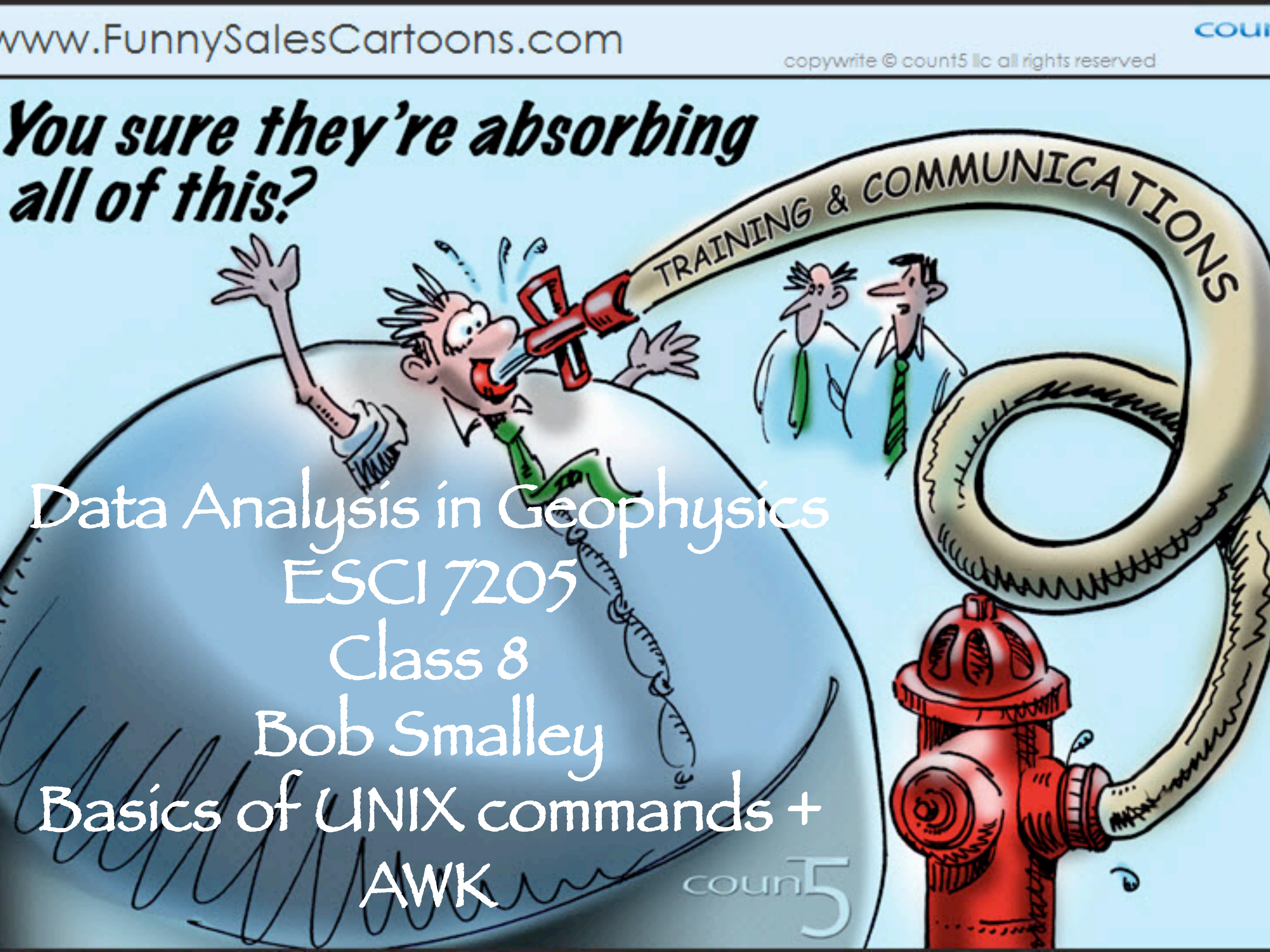
www.FunnySalesCartoons.com
copywrite © count5 llc all rights reserved
You sure they're absorbing all of this?
TRAINING & COMMUNICATIONS
count5

Data Analysis in Geophysics
ESCI 7205
Class 8
Bob Smalley
Basics of UNIX commands + AWK

Request on homeworks

Make sure your name is on every page you hand in (especially if not stapled together)

Please include your username (you could even use that as your name!).

Please include where I can find the files on the computer (this was not a problem when we only had the Sun's, but now I have to check 2 systems).

Shell note – how to drive yourself crazy

Do this!

```
echo don't ever do this
```

The shell will take what you think is an apostrophe as a single quote and will start escaping everything from there on to the next single quote (or the end of the file if there are no more single quotes) causing major problems as the quotes will not match (and the error report will come from a usually good line of code where the shell finds the next quote!).
It will take you forever to find this one.!

```
559 $ echo don't ever do this<CR>
> as the shell will not see the newline and stop input to echo <CR>
> until you enter another '
dont ever do this
as the shell will not see the newline and stop input to echo
until you enter another
560 $
```

Notice there are no quotes in the output (even though there are two in the input).

```
$ echo I\'d recommend you do this
I'd recommend you do this
```

Unless this does not work either (we will see more about this later).

Output formatting

Printing

`printf`: shell command for formatted printing

So far we have just been copying stuff from standard-in, files, pipes, etc to the screen or another file.

Say I have a file with names and phone numbers. I would like to print it out with vertically aligned columns.

(so my printout is not in the same format as the input file)

File contents:

```
Bob 4929
Chuck 4882
```

Desired display:

```
Bob       4929
Chuck     4882
```

printf

is a command from the C programming language to control printing.

```
% cat printex.sh
#!/bin/bash
printf "Hello world.  \n" represents a newline
a=`echo "Hello world." | wc | awk '{print $2}' `
b=`echo "Hello world." | wc -w`
printf "This phrase contains %d words from printf\n" $a
echo This phrase contains $b words from echo
% printex.sh
Hello world.
This phrase contains 2 words from printf
This phrase contains 2 words from echo
%
```

```
#!/bin/bash
printf "Hello world.  \n" # represents a newline
a=`echo "Hello world." | wc | awk '{print $2}' `
b=`echo "Hello world." | wc -w`

printf "This phrase contains %d words from printf\n" $a

echo This phrase contains $b words from echo
```

We need the double quotes " . . . " to define an argument (the stuff inside the quotes) for printf.

No parenthesis, string to print out in quotes, followed by list of variables separated by spaces.

```
#!/bin/bash
printf "Hello world.\n"
a=`echo Hello world. | wc | awk '{print $2}' `
printf "This phrase contains %d words\n" $a
```

The argument in double quotes has

- Regular text ("Hello world", "This phrase contains ")

- Some funny new thing - <u>`%d`</u> – <u>a format specifier</u>.

- The already known specification for a new line - `\n`

```
#!/bin/bash
printf "Hello world.\n"
a=`echo Hello world. | wc | awk '{print $2}' `
printf "This phrase contains %d words\n" $a
```

We also have another argument, the `$a`, which is a shell variable, at the end.

Note that the items are delimited with <u>spaces</u>, not commas.

```
#!/bin/bash
printf "Hello world.\n"
a=`echo Hello world. | wc | awk '{print $2}' `
printf "This phrase contains %d words\n" $a
```

We also have an example of awk (which can be understood from the quick introduction given previously to allow awk use in the homework).

Ignoring the details, this line assigns the value of the shell variable a to be the number of words in the string "Hello world." The variable a therefore contains an integer value (as a character string).

```
536 $ echo $a
hello world
537 $ printf "number words in a, %d\n" `echo $a | wc -w`
number words in a, 2
```

Simple errors – if have more data to print out (wc without any switches prints out 3 items) than format specifiers (the % things), it repeats the "…" until it runs out of stuff to print (or the format specifier cannot print the data type at which point it will complain).

```
538 $ printf "number words in a, %d\n" `echo $a | wc`
number words in a, 1
number words in a, 2
number words in a, 12
539 $ printf "number words in a, %d %d\n" `echo $a | wc`
number words in a, 1 2
number words in a, 12 0
```

`printf`: format specifiers

How to specify the format for printing various types of things

```
printf "This phrase contains %d words\n" $a
```

We are going to print out what is in the double quotes filling the `%things` with the values found in the variables listed at the end.

`printf`: format specifiers

```
printf "This phrase contains %d words\n" $a
```

No problem for everything but the `%d`.

And what is that shell variable `$a` at the end?

`printf`: format specifiers

```
printf "This phrase contains %d words\n" $a
```

The shell variable a contains the number of words in "Hello world". We want this (number) information where the %d is located in the format specification.

The %d and the $a are "paired".

`printf`: format specifiers

```
printf "This phrase contains %d words\n" $a
```

The `%d` format specifier is used to control how contents of the shell variable, `a`, are printed.

`printf`: format specifiers

Specify how to print various types of things

`%d` signed decimal integer

(The word decimal means base 10, as opposed to octal – base 8, or hexadecimal – base 16, it does not mean a number with a decimal point.

The word integer means a whole number, no decimal point and fraction).

So – a base 10 integer.

`printf`: format specifiers

Modifying how decimal integers are printed.

```
%<N>.<DIGITS>d
```

says use a field `N` characters wide, with `DIGITS` digits (uses leading zeros to make it the specified number digits, `DIGITS` can be > `N` (in which case `N` gets overridden), or `DIGITS` can be left out).

`printf`: format specifiers

Specify how to print various types of things

```
printf "This phrase contains %d words\n" $a
This phrase contains 2 words

printf "This phrase contains %3d words\n" $a
This phrase contains   2 words

printf "This phrase contains %3.0d words\n" $a
This phrase contains   2 words

printf "This phrase contains %3.3d words\n" $a
This phrase contains 002 words

printf "This phrase contains %3.4d words\n" $a
This phrase contains 0002 words
```

`printf`: format specifiers

Specify how to print various types of things

```
%d Print the associated argument as signed decimal number
%i Same as %d
%o Print the associated argument as unsigned octal number
%u Print the associated argument as unsigned decimal number
%x Print the associated argument as unsigned hexadecimal number
with lower-case hex-digits (a-f)
%X Same as %x, but with upper-case hex-digits (A-F)
%f Interpret & print associated argument as floating point number
%e Interpret associated argument as double, print in <N>±e<N> fmt
%E Same as %e, but with an upper-case E in the printed format
```

`printf`: format specifiers

Specify how to print various types of things

```
%g Interprets associated argument as double, prints like %f or %e
%G Same as %g, but print it like %E
%c Interprets the associated argument as character: only the
      first character of a given argument is printed
%s Interprets the associated argument literally as string (bunch
      of characters)
%b Interprets the associated argument as a string and interprets
      escape sequences in it
%q Prints the associated argument in a format, that it can be re-
      used as shell-input (escaped spaces etc..)
```

`printf`: format specifiers

Modifiers are specified between the `%` and the character that specifies/identifies the format:

`<N>` Any number: specifies a minimum field width, if the text to print is smaller, it's padded with spaces

`*` The asterisk: the width is given as an argument before the variable or string to be printed. Usage - the "`*`" corresponds to the "`20`" in the ex below

```
$ printf "%*s\n" 20 "test string"
         test string
$ a="test string"
$ printf "%*s\n" $(( 11 + `echo $a | wc -c` )) "$a"
            test string
```

`printf`: format specifiers

Modifiers are specified between the `%` and the character that specifies/identifies the format:

`#` "Alternative format" for numbers: see table below

`-` Left-bound text printing into the field (standard is right-bound)

`0` Pads numbers with zeros, not spaces

`<space>` Pad a positive number with a space, where a minus (-) is for negative numbers

`+` Prints all numbers signed (+ for positive, - for negative)

`printf`: format specifiers

Precision for single precision or double precision floating point numbers can be specified by using `<DIGITST>`<u>`.<DIGITSF>`</u>, where `<DIGITST>` is the total number of character positions (sign, decimal point, and digits) and <u>`<.DIGITSF>`</u> is the number of digits for precision in the fractional part <u>after</u> the decimal point.

```
$ printf "%.10f\n" 14.3
14.3000000000
```

Combine with <N> (total # characters, "-", decimal point, e)

```
$ printf "%15.10f\n" 14.3
__14.3000000000
```

`printf`: format specifiers

If `<DIGITST>` or `<DIGITSF>` is an asterisk (*), the precision is read from the argument that precedes the number to print.

```
$ a=14.3
$ printf "%.*f\n" 10 $a
14.3000000000
$ printf "%*.*f\n" 15 10 $a
__14.3000000000
```

In the last example it moves right to take a total of 15 spaces.

printf: format specifiers

Scientific notation

(seems to ignore the <DIGITT> field if it is too small for the number of characters needed to print out the number. Fortran of C would complain)

```
$ printf "%*.*e\n" 6 4 143200000000
1.4320e+11
$ printf "%*.*e\n" 6 4 -143.200000000
-1.4320e+02
$ printf "%*.*e\n" 6 3 -143.200000000
-1.432e+02
$ printf "%.*e\n" 3 -143.200000000
-1.432e+02
$ printf "%*.*e\n" 15 3 -143.200000000
     -1.432e+02
$
```

Later on we will talk about how computers represent numbers

Integer format
(integers, counting numbers)

Floating point format
(numbers with decimal point)

Floating point numbers can be
Real or Complex

`printf`: format specifiers

For strings, the precision specifies the maximum number of characters to print (i.e. the maximum field width).

(We already saw) For integers, it specifies the number of characters/digits to print (with leading zero- or blank-padding).

Escape codes

`\\`	Prints the character \ (backslash)
`\a`	Prints the alert character (ASCII code 7 decimal)
`\b`	Prints a backspace
`\f`	Prints a form-feed (goes to end of page)
`\n`	Prints a newline (is <CR><LF> on SUN and Mac OS)
`\r`	Prints a carriage-return (may or may not go to next line)
`\t`	Prints a horizontal tabulator
`\v`	Prints a vertical tabulator
`\<NNN>`	Interprets <NNN> as octal number and prints the corresponding character from the character set
`\0<NNN>`	same as \<NNN> *
`\x<NNN>`	Interprets <NNN> as hexadecimal number and prints the corresponding character from the character set (3 digits)*
`\u<NNNN>`	same as \x<NNN>, but 4 digits *
`\U<NNNNNNNN>`	same as \x<NNN>, but 8 digits *

(* - indicates nonstandard, may not work)

A few of the most useful format specifiers

%s	String. Print all the characters of the corresponding argument
%c	ASCII character. Print the <u>first character</u> only of the corresponding argument
%d	Decimal integer
%f	Floating-point format
%E	Scientific notation floating-point format

Review

you can control how many spaces are reserved for the formatted print (%) by adding numbers between the % and specifier (s, d, f, etc.)

`%10s` - 10 character string print

`%5d` - 5 spaces for signed decimal integer

`%10.2f` - 10 spaces for float and prints 2 digits after decimal point.

Default format is right justified. _ _ 1.02

To make formatted text left justified, add a minus sign, `-`, after the `%`

`%-10.2f` becomes 1.02 _ _

Shell note – how to drive yourself crazy

Some more funny business.

In GPS we use the DOY very heavily for the "date".

When using DOY, it is advantageous to use 3 digits, padding with zeros for days < DOY=100.

Say I'm working on data from day 032

```
551 $ x=032
552 $ echo $x
032
```

So far so good

But if I use **x** in an arithmetic expression, the **((…))** construct, and use **$** to return the value,

```
551 $ x=032
552 $ echo $x
032
553 $ echo $(( x ))
26
```

(oops!) What's this?

Turns out that bash interprets numbers defined with leading a zero(s) as octal (and a leading **0x** or **0X** as hexadecimal)

So 32 base 8 = 3*8+2=26 base 10
Not what I wanted!

To use **032** as a base 10 number you have to override the assumption that it is octal (or hex) using the **<BASE>#** construct, where **base** specifies the value of the base (here 10)

```
551 $ x=032
552 $ echo $x
032
553 $ echo $(( x ))
26
554 $ echo $(( 10#$x ))
32
555 $
```

Note that

```
echo $(( 10#x ))
```

Is incorrect as **x** is not a valid digit (you need the **$x** to get the value of **x**).

Another solution to the DOY problem is to work in "regular" numbers, `x=32`, and turn them into 3 digit text strings using `printf` when you need them in that format.

Introduction

AWK programming language

awk Programming Language

standard UNIX language that is geared for text processing and creating formatted reports

But is very valuable to seismologists because it uses floating point math, and is designed to work with columnar data

syntax similar to C and bash

one of the most useful UNIX tools at your command

(Softpanorama says "AWK is a simple and elegant pattern scanning and processing language.")

`awk` considers text files as having records (lines), which have fields (columns)

Performs floating & integer arithmetic and string operations

Has loops and conditionals

Can define your own functions (subroutines)

Can execute UNIX commands within the scripts and process the results

Basic structure of **awk** use

The essential organization of an **awk** program follows the form:

```
pattern { action }
```

The pattern specifies when the action is performed.

Like most UNIX utilities, **awk** is line oriented.

It may include an explicit test pattern that is performed with each line read as input.

If there is no explicit test pattern, or the condition in the test pattern is true, then the action specified is taken.

The default test pattern (no explicit test) is something that matches every line, i.e. is always true.

(This is the blank or null pattern.)

Versions/Implementations

`awk`: original awk

`nawk`: new awk, dates to 1987

`gawk`: GNU awk has more powerful string functionality

- NOTE -

We are going to use **awk** as the generic program name (like Kleenex for facial tissue)

Wherever you see **awk**, you can use **nawk** (or **gawk** if you are using that on a LINUX box).

Every CERI UNIX system has at least **awk**. The SUNs also have nawk.

You want to use **nawk**. I suggest adding this line to your .cshrc or .bashrc file on the SUNs

```
alias awk 'nawk' (csh) or alias awk=nawk (bash)
```

Rumor has it that in OS X **awk** is actually **nawk**, although I've not been able to establish this. If this is the case no changes to **nawk** codes are necessary (except the name). If you have lots of scripts with **nawk** and it is not found/installed, put the following in your .cshrc or .bashrc file

```
alias nawk='awk' (csh) or alias nawk=awk (bash)
```

Command line functionality

you can call **awk** from the command line two ways, we have seen/used the first – put the **awk** commands on the command line in the construct ' {…} ' or read **awk** commands from a script file.

```
awk [options] 'pattern { commands }' variables infile(s)
awk –f scriptfile variables infile(s)
```

or you can create an executable **awk** script

```
%cat << EOF > test.awk
#!/usr/bin/awk
some set of commands
EOF

%chmod 755 test.awk
%./test.awk
```

How **awk** treats text

awk commands are applied to every record or line of a file that passes the test.

it is designed to separate the data in each line into a number of fields and can processes what is in each of these fields.

essentially, each field becomes a member of an array with the first field identified as `$1`, second field `$2` and so on.

`$0` refers to the entire record (all fields).

Field Separator

How does **awk** break the line into fields.
It needs a field separator.
The default <u>field separator</u> is one or more white spaces

```
$1  $2     $3  $4    $5  $6    $7       $8        $9        $10   $11
1   1918    9   22    9   54   49.29   -1.698    98.298    15.1  ehb
```

So `$1 = 1, $2=1918, …, $10=15.1, $11=ehb`

Notice that the fields may be integer, floating point (have a decimal point) or strings.

awk is generally smart enough to figure out how to use them.

`print`

`print` is one of the most common **awk** commands (e.x. for an input line)

```
1 1918    9    22    9    54   49.29   -1.698    98.298    15.1   ehb
```

The following **awk** command '{…}' will produce

```
%awk '{ print $2 $8}'   /somewhere/inputfile
1918-1.698
```

The two output fields (1918 and -1.698) are run together - this is probably not what you want.

This is because **awk** is insensitive to white space in the inside the command ' {...} '

print

```
%awk '{ print $2 $8}'  /somewhere/inputfile
1918-1.698
```

The two output fields (1918 and -1.698) are run together – two solutions if this is not what you want.

```
%awk '{ print $1 "   " $8}'  /somewhere/inputfile
1918 -1.698
%awk '{ print $1, $8}'  /somewhere/inputfile
1918 -1.698
```

The awk print command different from the UNIX printf commad (more similar to echo).

any string (almost – we will see the caveat in a minute) or numeric text can be explicitly output using double quotes "..."

Assume our input file looks like this

```
1 1 1918 9 22 9 54 49.29  -1.698 98.298 15.0 0.0  0.0 ehb
```

Specify the character strings you want to put out with the "…".

```
1 1 1918 9 22 9 54 49.29  -1.698  98.298 15.0 0.0  0.0 ehb FEQ  x
1 1 1918 9 22 9 54 49.29   9.599 -92.802 30.0 0.0  0.0 ehb FEQ  x
1 1 1918 9 22 9 54 49.29   4.003  94.545 20.0 0.0  0.0 ehb FEQ  x
```

```
%awk '{print "latitude:",$9,"longitude:",$10,"depth:",$11}' SUMA.Loc
latitude: -1.698 longitude: 98.298 depth: 15.0
latitude: 9.599 longitude: -92.802 depth: 30.0
latitude: 4.003 longitude: 94.545 depth: 20.0
```

Does it for each line.

Notice that the output does not come out in nice columnar output (similar to the input).

If you wanted to put each piece of information on a different line, you can specify a newline several ways

```
%awk '{print "latitude:",$9; print "longitude:",$10}' SUMA.Loc
%awk '{print "latitude:",$9}{print "longitude:",$10}' SUMA.Loc
%awk '{print "latitude:",$9"\n""longitude:",$10}'  SUMA. Loc
latitude: -1.698
longitude: 98.298
```

Stop printing with ";" or } (the } marks the end of statement/block) and then do another print statement (you need to start a new block with another { if you closed the previous block),

or put out a new line character (\n) (it is a character so it has to be in double quotes "...").

awk variables

You can create **awk** variables inside your **awk** blocks in a manner similar to sh/bash. These variables can be character strings or numeric - integer, or floating point.

awk treats a number as a character string or various types of number based on context.

awk variables
In Shell we can do this

```
796 $ a=text
797 $ b='test $TEXT'
798 $ c="check $0"
799 $ d=10.7
800 $ echo $a $b, $c, $d
text test $TEXT, check —bash, 10.7
```

No comma between $a and $b, so no comma in output (spaces count and are output as spaces, comma produces comma in output)

In awk we would do the same thing like this (spaces don't count here, comma produces spaces in output)

```
809 $ echo text | nawk '{b="test $TEXT";a=b;c="'$0'";d=10.7;print $1, a, b, c, d}'
text test $TEXT test $TEXT -bash 10.7
810 $ echo text | nawk '{b="test $TEXT";a=b;c="'$0'";d=10.7;print $1, a b, c, d}'
text test $TEXTtest $TEXT -bash 10.7
```

Aside - Several ways to enter **awk** command(s) (some more readable than others

separate commands on the command line by ";"

```
809 $ echo text | nawk '{b="test $TEXT";a=b;c="'$0'";d=10.7;print $1, a, b, c, d}'
text test $TEXT test $TEXT -bash 10.7
```

Or you can put each command on its own line (newlines replace the ";"s)

```
506 $ echo text | nawk '{
b="test $TEXT"
a=b
c="'$0'"
d=10.7
print $1, a, b, c, d
}'
text test $TEXT test $TEXT -bash 10.7
507 $
```

Aside - Several ways to enter **awk** command(s) (some more readable than others)

Or you can make an executable shell file – `tst.awk` – the file has what you would type on the terminal (plus a #!/bin/sh at the beginning).

```
$ vi tst.awk
i#!/bin/sh
nawk '{
b="test $TEXT"
a=b
c="'$0'"
d=10.7
print $1, a, b, c, d
}'esc
:wq
$ x tst.awk
$ echo text | tst.awk
text test $TEXT test $TEXT ./tst.awk 10.7
```

Aside - Several ways to enter awk command(s)

Make a file, `tst.awk.in`, containing an awk "program" (the commands you would type in). Notice that here we do not need the single quotes (stuff not going through shell) or the `{}` around the block of commands (outside most set of braces optional as file is used to define the block). Use `–f` to id file with commands.

```
$ vi tst.awk.in
i#!/bin/sh
b="test $TEXT"
a=b
c="'$0'"
d=10.7
print $1, a, b, c, d
esc
:wq
$ echo text | tst.awk –f tst.awk.in
text test $TEXT test $TEXT ./tst.awk 10.7
```

```
$ cat tst.awk.in
b="test $TEXT"
a=b
c="'$0'"
d=10.7
print $1, a, b, c, d
```

Back to awk variables

```
#!/bin/sh
column=$1
nawk '{print $'$column'}'

822 $ ls -l *tst
-rwx------@ 1 robertsmalley  staff  2250 Aug 16  2004 az_map_tst
-rwx------@ 1 robertsmalley  staff   348 Aug 16  2004 tst
823 $ ls -l *tst | Column2.sh 9
az_map_tst
tst
824 $
```

Here **column** is a shell variable that is set to the first command line argument, **$1**.
'**$column**' is then expanded to **9** , the value of the first command line argument above, creating the **awk** variable **$9**

```
nawk '{print $9}'
```

And another tangent - this example also demonstrates a very powerful (***and very dangerous***) idea and capability.

The field to be printed is determined in real-time while the program is being executed.

It is not "hard coded".

So the program is effectively writing itself.
This is a very, very simple form of self-modifying-code.
(self-modifying-code is very hard to debug because you don't know what is actually being executed! You better hope is is the guilty party.)

You will find that it is very convenient to write scripts to write scripts!

You can write shell scripts (or C or Fortran for that matter) to write new shell scripts.

You can write shell scripts (or C or Fortran for that matter) to write SAC macros, etc. (Vocabulary/jargon - SAC macros are like shell scripts, but are in the SAC command “language”.)

Built-In Variables

`FS`: Field Separator

The default field separator is the space, what if we want to use some other character.

The password file looks like this

```
root:x:0:1:Super-User:/:/sbin/sh
```

The field separator seems to be (is) “:”
We can reset the field separator using the –F command line switch (the lower case switch, -f, is for specifying scriptfiles as we saw before).

```
% awk –F":"  '{print $1, $3}'  /etc/passwd
root 0
```

Built-In Variables

`FS`: Field Separator

There is another way to reset the FS variable that is more general (in that it does not depend on having a command line switch to do it – so it works with other built-in variables).

```
root:x:0:1:Super-User:/:/sbin/sh

% awk 'BEGIN {FS=":"} {print $1, $3}'  /etc/passwd
root 0
```

More **awk** program syntax

`BEGIN {…}` : the begin block contains everything you want done before **awk** procedures are implemented (before it starts processing the file)

`{…}[{…}…]` (list of procedures to be carried out on all lines)

`END {…}` : the end block contains everything you want done after the whole file has been processed.

BEGIN and **END** specify actions to be taken before any lines are read, and after the last line is read.

The **awk** program:

```
BEGIN { print "START" }
    { print }
END { print "STOP" }
```

adds one line with "**START**" before printing out the file and one line "**STOP**" at the end.

Field Separator

Can use multiple characters for Field Separators simultaneously

```
FS = "[:, -]+"
```

Built-In Variables

`NR`: record number is another useful built-in **awk** variable
it takes on the current line number, starting from 1

```
root:x:0:1:Super-User:/:/sbin/sh

% awk –F":"  '{print NR, $1, $3}'  /etc/passwd
1 root 0
```

`RS` : record separator specifies when the current record ends and the next begins
default is “`\n`” (newline)
useful option is `""` (blank line)

`OFS` : output field separator
default is `"   "` (whitespace)

`ORS` : output record separator
default is a `"\n"` (newline)

`NF` : number of fields in the current record

think of this as **awk** looking ahead to the next `RS` to count the number of fields in advance

```
$ echo 1 1 1918 9 22 9 54 49.29  -1.698 98.298 15.0 0.0  0.0 ehb
FEQ  x | nawk '{print NF}'
16
```

`FILENAME` : stores the current filename

`OFMT` : output format for numbers
example `OFMT="%.6f"` would make all numbers output as floating points

Accessing shell variables in **awk**

3 methods to access shell variables inside a **awk** script ...

Method 1 - Say I have a script with command arguments and I want to include them in the output of **awk** processing of some input data file:

Now we have a little problem

The Shell takes `$0, $1, etc.` as (the value of) variables for the command that is running, its first argument, etc.

While **awk** takes `$0, $1, etc.` as (the value of) variables that refer to the whole line, first field, second field, etc. of the input file.

So what does `'{print $1}'` refer to?

So what does `'{print $1}'` refer to?

It refers to the **awk** definition (the single quotes protect it from the shell) – the first field on the line coming from the input file.

The problem is getting stuff into **awk** other than what is coming from the input stream (a file, a pipe, stndin).

In addition to the shell command line parameter/ field position variable name problem, say I have some variable in my shell script that I want to include in the **awk** processing (using the shell variables `$0, $1,` is really the same as this problem with the addition of the variable name confusion).

What happens if I try

```
$ a=1
$ awk '{print $1, $2, $a}'
```

awk will be very disappointed in you!

Unlike the shell **awk** does not evaluate variables within strings.

If I try putting the shell variables into quotes to make them part of a character string to be output

```
'{print "$0\t$a" }'
```

awk would print out

```
$0      $a
```

Inside quotes in **awk**, the $ is not a metacharacter (unlike inside double quotes in the shell where variables are expanded). Outside quotes in **awk**, the $ corresponds to a field (so far), not evaluate and return the value of a variable (you could think of the field as a variable, but you are limited to variables with integer names).

The `\t` is a tab

```
{print "$0\t$a" }
```

Another difference between **awk** and a shell processing the characters within double quotes.

AWK understands special characters follow the "\" character like "`t`".

The Bourne and C UNIX shells do not.

To make a long story short, what we have to do is stop protecting the $ from the shell by judicious use of the single quotes.

For number valued variables, just use single quotes, for text valued variables you need to tell **awk** it is a character string with double quotes.

```
$ a=A;b=1;c=C
$ echo $a $c | nawk '{print $1 '$b' $2, "'$0'"}'
A1C –bash
```

If you don't use double quotes for character variables, it may not work

```
$ echo $a $b | nawk '{print $1 '$b' $2, '$0'}'
A1C -0
```

The single quotes really group like this.

```
$ echo $a $c | awk '{print $1 '$b' $2, "'$0'"}'
A1C –bash
```

The first single quote turns off shell interpretation of everything after it until the next single quote. So the single quote before the `$b` turns off the quote before the `{`. The `$b` gets passed to the shell for interpretation. It is a number so `awk` can handle it without further ado. The single quote after the `$b` turns off shell interpretation again, until the single quote before the `$0`.

The single quotes really group like this.

```
$ echo $a $c | awk '{print $1 '$b' $2, "'$0'"}'
A1C –bash
```

The `$0` returns the name of the program you are running, in this case the shell **–bash**. This is a character string so it needs to be in double quotes, thus the `"'$0'"`. The single quote after the `$0` turns "quoting" back on and it continues to the end of the awk block of code , dignified by the `}`.

The quotes are <u>switches</u> that turn shell interpretation off (first one) and back on (second one).

The single quotes really group like this.

```
$ echo $a $c | awk '{print $1 '$b' $2, "'$0'"}'
A1C -bash
```

Practically, since you always have the first and last, you can think about the ones about the `'$b'` and `'$0'` as pairs – but they really match up operationally as discussed.

Same for variables you define - if it is a text string you have to say it is a text string with the double quotes.

```
$ b=B
$ echo $a $b | nawk '{print $1 "'$b'" $2}'
ABC
```

If the variable was a number, you can still print it out as a text string (**awk** treats numbers as text strings or numbers as necessary in context, while text strings are stuck as text strings.)

```
$ b = 1
$ echo $a $b | nawk '{print $1 "'$b'" '$b' $2}'
A11C
```

Aside

How to print out quotes
(this is a very long line, no \ for continuation – wraps on its own).

```
620 $ echo $a $c | nawk '{print $1, '$b', $2, "'$0'", "\"", "\"ab
\"", "*", "'"''''"'", "a'"'"'b", "/" }'
A 1 C -bash " "ab" * ''' a'b /
```

Aside
How to print out quotes

```
581:> nawk 'BEGIN { print "Dont Panic!" }'
Dont Panic!
582:> nawk 'BEGIN { print "Don'\''t Panic!" }'
Don't Panic!
583:> nawk 'BEGIN { print "Don'"'"'t Panic!" }'
Don't Panic!
586:> echo Don”’”t Panic! | nawk "{print}"
Don't Panic!
584:> echo Don\'t Panic! | nawk '{print}'
Don't Panic!
585:> echo Don\'t Panic! | nawk "{print}"
Don't Panic!
```

Look carefully at the 2 lines above – you can (sometimes) use either quote (‘ or “) to protect the nawk program (depends on what you are trying to also protect from the shell).

Aside
How to print out quotes

```
alpaca.587:> nawk 'BEGIN { print "\"Don’\’’t Panic!\"" }'
"Don’t Panic!"
```

Method 2. Assign the shell variables to **awk** variables after the body of the script, but before you specify the input file

```
VAR1=3
VAR2=“Hi”

awk '{print v1, v2}' v1=$VAR1 v2=$VAR2 input_file
```

```
3 Hi
```

Also note: **awk** variables are referred to using just their name (no $ in front)

There are a couple of constraints with this method

Shell variables assigned using this method are not available in the **BEGIN** section (will see this, and **END** section, soon).

If variables are assigned after a filename, they will not be available when processing that filename

```
awk '{print v1, v2}' v1=$VAR1 file1 v2=$VAR2 file2
```

In this case, **v2** is not available to **awk** when processing file1.

Method 3. Use the `-v` switch to assign the shell variables to **awk** variables.

This works with **awk**, but not all flavors.

```
awk -v v1=$VAR1 -v v2=$VAR2 '{print v1, v2}' input_file
```

Aside - why use variables?

Say I'm doing some calculation that uses the number π.

I can put `3.1416` in whenever I need to use it.

But say later I decide that I need more precision and want to change the value of π to `3.1415926`.

It is a pain to have to change this and as we have seen global edits sometimes have unexpected (mostly because we were not really paying attention) side effects.

Aside - Why use variables?

Using variables (the first step to avoid hard-coding) – if you use variables you don't have to modify the code in a thousand places where you used `3.1416` for π.

If you had set a variable

```
pi=3.1416
```

And use `$pi`, it becomes trivial to change its value everywhere in the script by just editing the single line

```
pi=3.1415926
```

you don't have to look for it & change it everywhere

Examples:

Say we want to print out the owner of every file

Record/Field/column separator (**RS=" "**)

The output of **ls -l** is

```
-rwxrwxrwx   1 rsmalley user     7237 Jun 12  2006 setup_exp1.sh
```

So we need fields 3 and 9.

Do using an executable shell script

Create the file **owner.nawk** and make it executable.

```
$ vi owner.nawk
i#!/bin/awk -f
BEGIN { print "File\tOwner" }
{ print $9, "\t", $3}
END { print " - DONE -" }esc
:wq
$ x owner.nawk
```

Now we have to get the input into the program.

Pipe the long directory listing into our shell script.

```
507:> ls -l | owner.nawk
File    Owner
*CHARGE-2002-107*        rsmalley
022285A.cmt      rsmalley
190-00384-07.pdf         rsmalley
. . .
zreal2.f         rsmalley
zreal2.o         rsmalley
 - DONE –
508:>
```

So far we have just been selecting and rearranging fields. The output is a simple copy of the input field.

What if you wanted to change the format of the output with respect to the input

Considering that **awk** was written by some of UINX's developers, it might seem reasonable to guess that they "reused" some useful UNIX tools.

If you guessed that you would be correct.

So if you wanted to change the format of the output with respect to the input – you just use the UNIX `printf` command.

We already saw this command, so we don't need to discuss it any further (another UNIX philosophy based attitude).

```
$ echo text | awk '{
b="test $TEXT"
a=b
c="'$0'"
d=10.7
printf("%s, %s, %s, %s, %6.3f\n", $1, a, b, c, d)
}'
text, test $TEXT, test $TEXT, -bash, 10.700
```

Notice a few differences with the UNIX `printf` command

You need parens (…) around the arguments to the `printf` command.

You need commas between the items in the variable list.

```
$ echo text | awk '{
b="test $TEXT"
a=b
c="'$0'"
d=10.7
printf("%s, %s, %s, %s, %6.3f\n", $1, a, b, c, d)
}'
text, test $TEXT, test $TEXT, -bash, 10.700
```

The output of `printf` goes to stndout.

<u>`sprintf`</u>

Same as `printf` but sends formatted print output to a string variable rather to stndout

```
n=sprintf ("%d plus %d is %d", a, b, a+b);
```