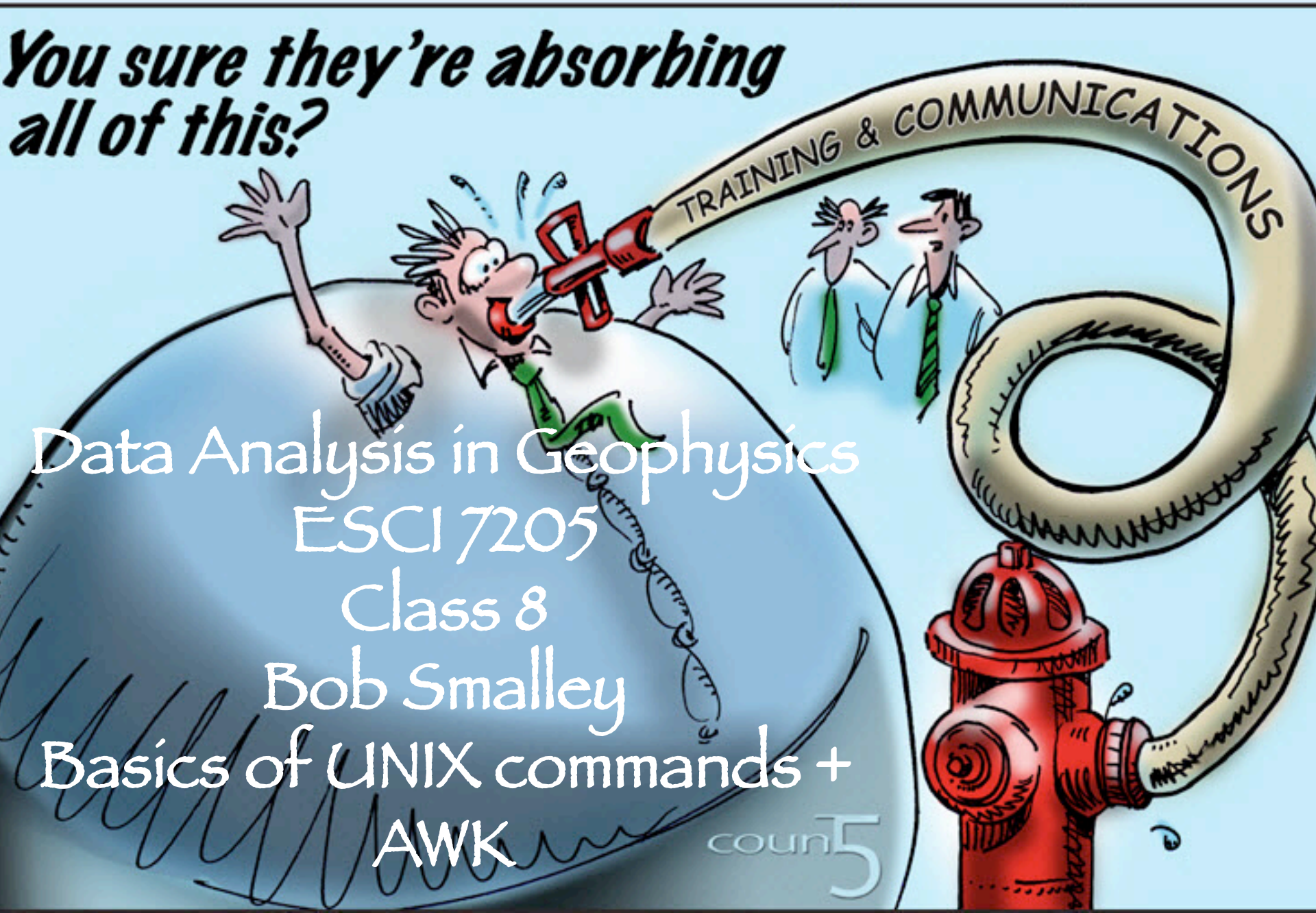


Data Analysis in Geophysics
ESCI 7205
Class 8
Bob Smalley
Basics of UNIX commands + AWK

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 a LINUX box).

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.")

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.

Most of your awk programs will look something like

awk '{print $3, $5}' input.file

prints out third and fifth fields of each line of file

or

awk '/help/ {print $3, $5}' input.file

prints out third and fifth fields of each line of file that contains the string - help

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
```

`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" in nawk and gawk)

Can execute UNIX commands within the scripts and process the results

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.)

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 field separator 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

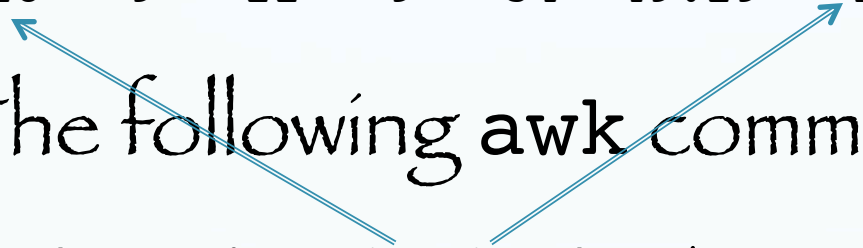
Notice that the fields may be integer, floating point (have a decimal point) or strings.

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 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 (or C or Fortran code for that matter).

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
```

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 , signified by the }.

The quotes are switches 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.

`sprintf`

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);
```

"Simple" awk example:

Say I have some sac files with the horrid IRIS DMC format file names

`1999.289.10.05.26.0000.IU.KMBO.00.LHZ.SAC`

and it would rename it to something more "user friendly" like **`KMBO.LHZ`** to save on typing while doing one of Chuck's homeworks.

```
alpaca.540:> more rename.sh
#!/bin/sh

#to rename horrid iris dmc file names

#call with rename.sh A x y
#where A is the char string to match, x and y are the field
#numbers in the original file name you want to use in the
#final name, and using the period/dot for the field seperator

#eg if the file names look like
#1999.289.10.05.26.0000.IU.KMBO.00.LHZ.SAC
#and you would ;ike to rename it KMBO.LHZ
#the 8th field is the station name, KMBO
#and the 10th field is the component name, LHZ
#so you would call rename.sh SAC 8 10
#(it will do it for all file names in your directory
#containing the string "SAC")

for file in `ls -1 *$1*`
do
mv $file `echo $file | nawk -F. '{print $'$2'"."$'$3'}'`
done
alpaca.541:>
```

Loop is in Shell, not awk.

string functions

`index(months,mymonth)`

Built-in string function `index`, returns the starting position of the occurrence of a substring (the second parameter) in another string (the first parameter), or it will return `0` if the string isn't found.

```
months="Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec"

         000000000111111111122222222223333333334444444444
         123456789012345678901234567890123456789012345678
                                                  ↑
```

```
print index(months,"Aug")
29
```

To get the number associated with the month (based on the string with the 12 months) add 3 to the index (29+3=32) and divide by 4 (32/4=8, Aug is 8th month).

The string months was designed so the calculation gave the month number.

Good place for tangent –

Functions (aka Subroutines)

We have used the word functions quite a bit, but what are they (definition with respect to programming?)

Blocks of code that are semi-independent from the rest of the program and can be used multiple times and from multiple places in a program (sometimes including themselves – recursive).

They can also be used for program organization.

```
<Placemark>
 <name>PELD</name>
 <styleUrl>#CAPStyleMap</styleUrl>
                <description><![CDATA[
<table width="580" cellpadding="0" cellspacing="0">
  <tr>
    <td align="left" valign="top">
      <p>
        <font color="00000000">
PELD -33.14318 -70.67493 US|CAP [5] 1993 1997 1998 1999 2003 CHILE OKRT Peldehue
</font>
      </p>
 <td width="10" align="left" valign="top"> </td>
    <td align="right" valign="top">
      <table border="0" cellspacing="0" cellpadding="0" bgcolor="white">
        <tr>
        </tr>
      </table>
    </td>
  </tr>
</table>]]></description>
 <Point>
<coordinates>  -70.67493000, -33.14318000,    0.0000</coordinates>
 </Point>
 </Placemark>
```

```
 <Placemark>
 <name>COGO</name>
 <styleUrl>#CAPStyleMap</styleUrl>
                <description><![CDATA[
<table width="580" cellpadding="0" cellspacing="0">
  <tr>
    <td align="left" valign="top">
      <p>
        <font color="00000000">
COGO -31.15343 -70.97526 US|CAP [4] 1993 1996 2003 2008 CHILE OKRT Cogoti
</font>
      </p>
 <td width="10" align="left" valign="top"> </td>
    <td align="right" valign="top">
      <table border="0" cellspacing="0" cellpadding="0" bgcolor="white">
        <tr>
        </tr>
      </table>
    </td>
  </tr>
</table>]]></description>
 <Point>
<coordinates>  -70.97526000, -31.15343000,    0.0000</coordinates>
 </Point>
 </Placemark>
```

This is a piece of kml code (the language of Google Earth). Notice that the only difference between what is in the two boxes is the stuff in red.

```
<Placemark>
 <name>PELD</name>
 <styleUrl>#CAPStyleMap</styleUrl>
                <description><![CDATA[
<table width="580" cellpadding="0" cellspacing="0">
  <tr>
    <td align="left" valign="top">
      <p>
        <font color="00000000">
PELD -33.14318 -70.67493 US|CAP [5] 1993 1997 1998 1999 2003 CHILE OKRT Peldehue
</font>
      </p>
 <td width="10" align="left" valign="top"> </td>
    <td align="right" valign="top">
      <table border="0" cellspacing="0" cellpadding="0" bgcolor="white">
        <tr>
        </tr>
      </table>
    </td>
  </tr>
</table>]]></description>
 <Point>
<coordinates>  -70.67493000, -33.14318000,    0.0000</coordinates>
 </Point>
 </Placemark>
```

```
 <Placemark>
 <name>COGO</name>
 <styleUrl>#CAPStyleMap</styleUrl>
                <description><![CDATA[
<table width="580" cellpadding="0" cellspacing="0">
  <tr>
    <td align="left" valign="top">
      <p>
        <font color="00000000">
COGO -31.15343 -70.97526 US|CAP [4] 1993 1996 2003 2008 CHILE OKRT Cogoti
</font>
      </p>
 <td width="10" align="left" valign="top"> </td>
    <td align="right" valign="top">
      <table border="0" cellspacing="0" cellpadding="0" bgcolor="white">
        <tr>
        </tr>
      </table>
    </td>
  </tr>
</table>]]></description>
 <Point>
<coordinates>  -70.97526000, -31.15343000,    0.0000</coordinates>
 </Point>
 </Placemark>
```

This is a prime example of when one would want to use a subroutine (unfortunately kml does not have subroutines– but we will pretend it does).

(so in kml, if you have 500 points this code is repeated 500 times with minor variations)
(see why kml desperately needs subroutines?)

```
Define it (define inputs)

<Placemark>
 <name>name</name>
 <styleUrl>#CAPStyleMap</styleUrl>
                <description><![CDATA[
<table width="580" cellpadding="0" cellspacing="0">
  <tr>
    <td align="left" valign="top">
      <p>
        <font color="00000000">
description
</font>
      </p>
 <td width="10" align="left" valign="top"> </td>
    <td align="right" valign="top">
      <table border="0" cellspacing="0" cellpadding="0" bgcolor="white">
        <tr>
        </tr>
      </table>
    </td>
  </tr>
</table>]]></description>
 <Point>
<coordinates>  location</coordinates>
 </Point>
 </Placemark>

Go back to calling routine
```

The idea of functions and subroutines is to write the code once with some sort of placeholder in the red parts.

We will also need to put some wrapping around it (a name, ability to get and return data from calling routine, etc.) and have a way to "call" it.

```
Function KML_Point (name, description ,location )

<Placemark>
 <name>name</name>
 <styleUrl>#CAPStyleMap</styleUrl>
                <description><![CDATA[
<table width="580" cellpadding="0" cellspacing="0">
  <tr>
    <td align="left" valign="top">
      <p>
        <font color="00000000">
description
</font>
      </p>
 <td width="10" align="left" valign="top"> </td>
    <td align="right" valign="top">
      <table border="0" cellspacing="0" cellpadding="0" bgcolor="white">
        <tr>
        </tr>
      </table>
    </td>
  </tr>
</table>]]></description>
 <Point>
<coordinates>  location</coordinates>
 </Point>
 </Placemark>

Go back to calling routine
```

Let's say the subroutine name is KML_Point and it takes 3 arguments, a character string for the name, a character string with the description and a character string with the location (lat, long, elevation).

```
Function KML_Point (name, description ,location )

<Placemark>
 <name>name</name>
 <styleUrl>#CAPStyleMap</styleUrl>
               <description><![CDATA[
<table width="580" cellpadding="0" cellspacing="0">
  <tr>
    <td align="left" valign="top">
      <p>
        <font color="00000000">
description
</font>
      </p>
 <td width="10" align="left" valign="top"> </td>
    <td align="right" valign="top">
      <table border="0" cellspacing="0" cellpadding="0" bgcolor="white">
        <tr>
        </tr>
      </table>
    </td>
  </tr>
</table>]]></description>
 <Point>
<coordinates>  location</coordinates>
 </Point>
 </Placemark>

Go back to calling routine



. . . . .

Somewhere in my program

Call KML_Point("PELD","PELD -33.14318 -70.67493 US|CAP [5] 1993 1997 1998 1999 2003 CHILE OKRT Peldehue","-70.67493000, -33.14318000,    0.0000")

COGO_Name="COGO"
COGO_Desc="COGO -31.15343 -70.97526 US|CAP [4] 1993 1996 2003 2008 CHILE OKRT Cogoti"
COGO_Loc="-70.97526000, -31.15343000,    0.0000"
Call KML_Point($COGO_NAME,$COGO_Desc,$COGO_Loc)
```

Now in my program I can call this "subroutine" and don't have to repeat all the common information. An even better way to do below is to have the data in an array (soon) and do a loop over the elements in the array.

Can also use subroutines to organize your program, rather than just for things you have to do lots of times.

This also allows you to easily change the calculation in the subroutine by just replacing it (works for single use or multiple use subroutines – e.g. raytracer in inversion program.)

Functions (aka Subroutines)
(nawk and gawk, not awk)

Format -- "`function`", then the name, and then the parameters separated by commas, inside parentheses.

Followed by "{}", the code block that contains the actions that you'd like this function to execute.

```
function monthdigit(mymonth) {
return (index(months,mymonth)+3)/4
}
```

awk provides a "return" statement that allows the function to return a value.

```
function monthdigit(mymonth) {
return (index(months,mymonth)+3)/4
}
```

This function converts a month name in a 3-letter string format into its numeric equivalent. For example, this:

```
print monthdigit("Mar")
```

....will print this:

3

Example

```
607 $ cat fntst.sh
#!/opt/local/bin/nawk -f
#return integer value of month, return 0 for "illegal" input
#legal input is 3 letter abbrev, first letter capitalized
{
    if (NF = 1) {
    print monthdigit($1)
    } else {
        print;
    }
}

function monthdigit(mymonth) {
months="Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec";;
if ( index(months,mymonth) == 0 ) {
return 0
} else {
return (index(months,mymonth)+3)/4
}
}
```

Example

```
607 $ cat fntst.dat
Mar
Jun
JUN
608 $ fntst.sh fntst.dat
3
6
0
609 $ cat callfntst.sh
#!/bin/sh
echo $1 is month number `echo $1 | fntst.sh`
610 $ callfntst.sh May
May is month number 5
611 $
```

`substr(string,StartCharacter,NumberOfCharacters)`

cut specific subset of characters from string

`string`: a string variable or a literal string from which a substring will be extracted.

`StartCharacter`: starting character position.

`NumberOfCharacters`: maximum number characters or length to extract.

(if `length(string)` is shorter than `StartCharacter+NumberOfCharacters`, your result will be truncated.)

`substr()` won't modify the original string, but returns the substring instead.

Back to strings

Sub-strings

```
substr(string,StartCharacter,NumberOfCharacters)

oldstring="How are you?"

newstr=substr(oldstring,9,3)
```

What is newstr in this example?

`match()` searches for a regular expression.

match returns the starting position of the match, or zero if no match is found, and sets two variables called `RSTART` and `RLENGTH`.

`RSTART` contains the return value (the location of the first match), and

`RLENGTH` specifies its span in characters (or -1 if no match was found).

string substitution

`sub()` and `gsub()`.

Modify the original string.

```
sub(regexp,replstring,mystring)
```

`sub()` finds the first sequence of characters in `mystring` matching `regexp`, and replaces that sequence with `replstring`.

`gsub()` performs a global replace, swapping out all matches in the string.

string substitution sub() and gsub().

```
oldstring="How are you doing today?"
sub(/o/,"O",oldstring)
print oldstring
HOw are you doing today?

oldstring="How are you doing today?"
gsub(/o/,"O”,oldstring)
print oldstring
HOw are yOu dOing tOday?
```

Other string functions

`length` : returns the number of characters in a string

```
oldstring="How are you?"
length(oldstring) # returns  12
```

`tolower/toupper` : converts string to all lower or to all upper case

(could use this to fix out previous example to take May or MAY.)

Continuing with the features mentioned in the introduction.

`awk` does arithmetic (integer, floating point, and some functions – sin, cos, sqrt, etc.) and logical operations.

Some of this looks like math in the shell, but ...

awk does floating point math!!!!!

awk stores all variables as strings, but when math operators are applied, it converts the strings to floating point numbers if the string consists of numeric characters (can be interpreted as a number)

awk's numbers are sometimes called stringy variables

Arithmetic Operators

All basic arithmetic is left to right associative

+ : addition

- : subtraction

* : multiplication

/ : division

% : remainder or modulus

^ : exponent

other standard C programming operators (++, --, =+,...)

Arithmetic Operators

```
…| awk '{print $6, $4/10., $3/10., "0.0"}' |…
```

Above is easy, fields 4 and 3 divided by 10

How about this

```
`awk '{print $3, $2, (2009-$(NF-2))/'$CIRCSC' }'
 $0.tmp`
```

`(NF-2)` is the number of fields on the line minus 2,

`$(NF-2)` is the value of the field in position `(NF-2)`,

which is subtracted from 2009, then everything is divided by `$CIRCSC` passed in from script.

Arithmetic Operators

Math functions

```
…| awk '{print $1*cos($2*0.01745)}'
```

Arguments to trig functions have to be specified in RADIANS, so if have degrees, divide by π/180.

```
MAXDISP=`awk '{print sqrt($3^2+$4^2)}' $SAMDATA/
ARIA_coseismic_offsets.v0.3.table | sort -n -r |
head -1`
```

a trick

If a field is composed of both strings and numbers, you can multiply the field by 1 to remove the string.

```
% head test.tmp
 1.5  2008/09/09 03:32:10 36.440N 89.560W  9.4
 1.8  2008/09/08 23:11:39 36.420N 89.510W  7.1
 1.7  2008/09/08 19:44:29 36.360N 89.520W  8.2

% awk '{print $4,$4*1}' test.tmp
36.440N 36.44
36.420N 36.42
36.360N 36.36
```

Selective execution

So far we have been processing every line (using the default test pattern which always tests true).

`awk` recognizes regular expressions and conditionals at test patterns, which can be used to selectively execute `awk` procedures on the selected records

Selective execution

Simple test for character string /test pattern/, if found, does stuff in {...}, from command line

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

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

or within a script

```
$ cat esciawk1.sh
#!/bin/sh
awk -F":" '/root/ {print $1, $3}'
$ cat /etc/passwd | esciawk1.sh
root 0
```

Use/reuse other UNIX features/tools to make much more powerful selections.

Selective execution

Or using a scriptfile and input file

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

$ cat esciawk1.nawk
/root/ {print $1, $3}
$ esciawk1.sh -f esciawk1.nawk < /etc/passwd
root 0
```

Relational Operators

Relational operators return 1 if true and 0 if false

!!! opposite of bash/shell test command

All relational operators left to right associative

< : test for less than

<= : test for less than or equal to

> : test for greater than

>= : test for greater than or equal to

== : test for equal to

!= : test for not equal

Unlike bash, the comparison and relational operators in `awk` don't have different syntax for strings and numbers.

ie: `==` only in `awk`

rather than == or `-eq` using test.

Boolean (Logical) Operators

Boolean operators return 1 for true & 0 for false

!!! opposite of bash/shell test command

&& : logical AND; tests that both expressions are true

left to right associative

|| : logical OR ; tests that one or both of the expressions are true

left to right associative

! : logical negation; tests that expression is true

Selective execution
Boolean Expressions in test pattern.

```
awk '((/\|US/||/US\|/)&&!/continuous/)&&(/BOLIVIA/||/BODEGA/||/
^SP/)'$ARGONLY' {print $3,$2, " 12 0 4 1 ", $1$5}' $GPSDATA
```

`\|` - have to escape the pipe symbol

`(/\|US/||/US\|/)` - parens group terms

`/continuous/` - simple pattern match

Plus more self-modification

`'$ARGONLY'` - is equal to one of the two choices `ARGONLY=<CR>` or `ARGONLY='&&/ARGENTINA/'` make it up as you go along (first one is nothing, second adds a test and logical to combine with everything in parentheses).

Selective execution

Relational, Boolean expressions in test pattern

```
… | awk '('$LATMIN'<=$2&&$2<='$LATMAX') {print
$0}' | …

awk '('$LONMIN'<=$1)&&($1<='$LONMAX')&&
('$LATMIN'<=$2)&&($2<='$LATMAX')&&
($10>='$MINMTEXP')&&$3>50 {print $1, $2, $3, $4,
$5, $6, $7, $8, $9, $10, '$MECAPRINT' }'
$HCMTDATA/$FMFILE
```

Also passing shell variables into awk

Selective execution

Regular Expressions in test pattern.

```
awk '((/\|US/||/US\|/)&&!/continuous/)&&(/BOLIVIA/||/BODEGA/||/
^SP/| /^AT[0-9]/ |/^RL[0-9]/)'$ARGONLY' {print $3,$2, " 12 0 4 1
", $1$5}' $GPSDATA
```

`/^AT[0-9]/` - regular expressions (beginning of line - `^`, range of characters - `[0-9]`)

Selective execution

shell variable in test pattern.

```
awktst_shal=\(\$3\<60\&\&\$4\>10\)
awk ''$nawktst_shal' {print $0}'
```

Notice the escapes in the definition of the variable **awktst**.

These \ escapes the (and $ and & and get stripped out by the shell inside the ' ' before going to nawk.

Also notice the quotes **''$nawktst_shal' …'**
(more self modifying code)

Selective execution

New structure

conditional-assignment expression - "?:"

Test?true:false

```
…| awk '{print ($7>180?$7-360:$7), $6,
$4/10., $3/10., "0.0 0.0 0.0"}' |…
```

Does the test `$7>180`, then prints out `$7-360` if true, (else) or `$7` if false.
Is inside the "print".

```
nawk '{print ($1>=0?$1:360+$1)}'
```

Syntax: (test?stmt1:stmt2)

This will do a test

(in this case: $1>=0)

If true it will output stmt1 ($1)

(does this: `nawk '{print $1}'`

If false it will output stmt2 (360+$1)

(does this: `nawk '{print 360+$1}'`

(in this case we are changing longitudes from the range/format -180<=lon<=180 to the range/format 0<=lon<=360)

Selective execution

```
$ cat tmp
isn't that special!

$ cat tmp | nawk '$2=="that" {print $0}'
isn't that special!

$ cat tmp | nawk '{ if ($2=="that") print $0}'
isn't that special!

$ cat tmp | nawk '{ if ($2=="I") print $0}'
$
```

Effect of \ and quotes.

```
513 $ awktst_shal=\(\$3\<60\&\&\$4\>10\)
514 $ echo $awktst_shal
($3<60&&$4>10)
```

All the backslashes "go away" when there are no quotes.

The backslashes get "consumed" by the shell protecting the following metacharacter so it "comes out".

Effect of \ and quotes.

```
515 $ awktst_shal="\(\$3\<60\&\&\$4\>10\)"
516 $ echo $awktst_shal
\($3\<60\&\&$4\>10\)
```

The "..." protect most metacharacters from the shell.

This keeps most the backslashes, but "..." evaluates $ and `...`, so the backslashes in front of the $ go away, they get "consumed" by the shell, as they protect the $ from the shell.

Effect of \ and quotes.

```
517 $ awktst_shal='\(\$3\<60\&\&\$4\>10\)'
518 $ echo $awktst_shal
\(\$3\<60\&\&\$4\>10\)
519 $
```

The '...' protects all metacharacters from the shell.

This keeps all the backslashes.

Write a file with nawk commands and execute it.

```
#!/bin/sh
#general set up
ROOT=$HOME
SAMDATA=$ROOT/geolfigs
ROOTNAME=$0_ex
VELFILEROOT=`echo $latestrtvel`
VELFILEEXT=report
VELFILE=${SAMDATA}/${VELFILEROOT}.${VELFILEEXT}
#set up for making gmt input file
ERRORSCALE=1.0
SEVENFLOAT="%f %f %f %f %f %f %f "
FORMATSS=${SEVENFLOAT}"%s %f %f %f %f\\\\n"
GMTTIMEERRSCFMT="\$2, \$3, \$4, \$5, ${ERRORSCALE}*\$6, ${ERRORSCALE}*\
$7, \$8”
#make the station list
STNLIST=`$SAMDATA/selplot $SAMDATA/gpsplot.dat pcc`
#now make nawk file
echo $STNLIST {printf \"$FORMATSS\", $GMTTIMEERRSCFMT, \$1, \$9,
$ERRORSCALE, \$6, \$7 } > ${ROOTNAME}.nawk

#get data and process it
nawk -f $SAMDATA/rtvel.nawk $VELFILE | nawk -f ${ROOTNAME}.nawk
```

Notice all the "escaping" ("\" character) in the shell variable definitions (`FORMATSS` and `GMTTIMEERRSCFMT`) and the echo.

Look at the nawk file – it looses most of the escapes.

The next slide shows the nawk file at the top and the output of applying the nawk file to an input data file at the bottom.

```
/ALGO/||/ANT2/||/ANTC/||/ARE5/||/AREQ/||/ASC1/||/AUTF/||/
BASM/||/BLSK/||/BOGT/||/BOR4/||/BORC/||/BRAZ/||/CAS1/||/
CFAG/||/COCR/||/CONZ/||/COPO/||/CORD/||/COYQ/||/DAV1/||/
DRAO/||/EISL/||/FORT/||/FREI/||/GALA/||/GAS0/||/GAS1/||/
GAS2/||/GAS3/||/GLPS/||/GOUG/||/HARB/||/HARK/||/HART/||/
HARX/||/HUET/||/IGM0/||/IGM1/||/IQQE/||/IQTS/||/KERG/||/
KOUR/||/LAJA/||/LHCL/||/LKTH/||/LPGS/||/MAC1/||/MARG/||/
MAW1/||/MCM1/||/MCM4/||/OHI2/||/OHIG/||/PALM/||/PARA/||/
PARC/||/PMON/||/PTMO/||/PWMS/||/RIOG/||/RIOP/||/SALT/||/
SANT/||/SYOG/||/TOW2/||/TPYO/||/TRTL/||/TUCU/||/UDEC/||/
UEPP/||/UNSA/||/VALP/||/VESL/||/VICO/||/HOB2/||/HRA0/||/DAVR/
{printf "%f %f %f %f %f %f %f %s %f %f %f %f\n", $2, $3, $4,
$5, 1.0*$6, 1.0*$7, $8, $1, $9, 1.0, $6, $7 }

-78.071370 45.955800 -6.800000 -8.600000 0.040000 0.040000
0.063400 ALGO 12.296000 1.000000 0.040000 0.040000⏎
-70.418680 -23.696350 26.500000 8.800000 1.010000 1.010000
-0.308300 ANT2 0.583000 1.000000 1.010000 1.010000⏎
-71.532050 -37.338700 15.000000 -0.400000 0.020000 0.040000
-0.339900 ANTC 8.832000 1.000000 0.020000 0.040000⏎
-71.492800 -16.465520 -9.800000 -13.000000 0.190000 0.120000
-0.061900 ARE5 3.348000 1.000000 0.190000 0.120000⏎
-71.492790 -16.465510 14.100000 3.800000 0.030000 0.020000
-0.243900 AREQ 7.161000 1.000000 0.030000 0.020000⏎ . . .
```

Looping Constructs in awk

awk loop syntax are very similar to C and perl

`while`: continues to execute the block of code as long as condition is true.

If not true on first test, which is done before going through the block, it will never go through block.

Do stuff in "block" between { ... }

```
while ( x==y ) {
    . . .
    block of commands
    . . .
}
```

do/while

do the block of commands between { ... } and while, while the test is true

```
do {
   . . .
   block of commands
   . . .
} while ( x==y )
```

The difference between the while loop on the last slide and the do/while here (notice the while at the end) is when the condition is tested. It is tested prior to running the block of commands in a while loop, but tested after running the block of commands in a do/while loop (so at least one trip through the block of commands will occur)

for loops

The for loop, allows iteration/counting as one executes the block of code in {...}.

It is one of the most common loop structures.

```
for ( x=1; x<=NF; x++) {
   . . .
   block of commands
   . . .
}
```

This is an extremely useful/important construct as it allows applying the block of commands to the elements of an array

(at least numerical arrays with all the elements "filled-in").

New structure

```
for ( q in myarray ) …
```

In programs that use arrays, you often need a loop that executes once for each element of an array.

awk has a special kind of for statement for scanning an array:

```
for (var in array) body
```

This loop executes body once for each index in array that your program has previously used, with the variable var set to that index.

New structure

```
for ( q in myarray ) …
```

The q here is a dummy variable. It is made up and initialized on-the-fly.

Its value changes on each trip (loop) through the following block of code.

Its value may or may not retain the last value after the loop finishes (on Mac it seems to).

break and continue

break: breaks out of a loop

continue: restarts at the beginning of the loop

```
x=1
while (1) {
   if ( x == 4 ) {
      x++
      continue
   }
   print "iteration",x
   if ( x > 20 ) {
      break
   }
   x++
}
```

If / else if / else blocks

similar to bash but syntax is different (no `then` or `fi`, uses braces { . . . } to define block instead)

```
if ( conditional1 ) {
    . . .
    block of commands
    . . .
} else if ( conditional2 ) {
    . . .
    block of commands
    . . .
} else {
    . . .
    block of commands
    . . .
}
```

`else if` and `else` are optional

you can have an `if` loop w/o an `else if` or `else`, but you can't have an `else if` or `else` w/o an `if`

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.