Data Analysis in Geophysics
ESCI 7205

Class 9

Bob Smalley

AWK

Computers make it easier to do a lot of things, but most of the things they make it easier to do don't need to be done.

Andy Rooney

"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"

        00000000011111111112222222222333333333344444444
        12345678901234567890123456789012345678901234567 8
```

```
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)

```
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_Pooint (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_Pooint (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
```

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)

```
. . . . .

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)
```

and do a loop over the elements in the array.

Recursion (just for fun for you out of the box thinkers, or those of you who will do it accidently.)

definition of recursion.

Recursion: See "Recursion".

Recursion.

A routine that calls itself.

Classic example – Factorial.

N!=N*(N-1)*(N-2)*…*2
For N≥2
N!=1 for N=1
N!=1 for N=0

N! undefined for N<1.

How to calculate.

Say I have a routine NFact that calculates the factorial of a number.

Recursion.

One possible way to implement the Factorial function.

My main program will call the subroutine `NFact` with the number `N` whose factorial I want.

My subroutine `NFact` will then do this.

Look at the number.

If it is `0` or `1`, return `1`.

If `N` is ≥`2`, calculate `N*NFact(N-1)`

Recursion.

So this is what would get done for `N=4`

```
NFact(4)

4*NFact(3)

4*3*Nfact(2)

4*3*2*Nfact(1)

4*3*2*1
```

And now, finally, I can evaluate it.

definition of recursion.

Recursion: If you still don't get it, see "Recursion". .

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 fields minus 2, then `$(NF-2)` is the value of the field in position `(NF-2)`, which is subtracted from `2009`, then everything is divided by `$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\|/)` – group terms

`/continuous/` - simple pattern match

Plus more self-modification

`'$ARGONLY'` - One of `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 \ escape 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)

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.

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".

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
#cat ${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⏎ . . .
```

```
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}'
$
```

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

<u>do</u> the block of commands between { ... } and <u>while</u>, while the test is true

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

The difference between <u>while</u> (last slide) and <u>do/while</u> (notice the <u>while</u> at the end) is when the condition is tested. It is tested <u>prior</u> to running the block of commands for a <u>while</u> loop, but tested <u>after</u> running the block of commands in a <u>do/while</u> 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”).

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`

Example

Checkbook balancing program in awk

- Simple tab-delimited text file into which recent deposits and withdrawals are entered.

- The idea is to hand this data file to an awk script that would automatically add up all the amounts and report the balance.

Input file format:

Fields are separated by one or more tabs.

After the date (field 1, $1), there are two fields: "exp field" and "inc field".

When entering an expense, a four-letter nickname is entered in the exp field, and a "-" (blank entry) in the inc field.
When entering a deposit, a four-letter nickname is entered in the inc field, and a "-" (blank entry) in the exp field.

Here's what an expense (debit) looks like:

```
23 Aug 2000   food   -   -   Y   Jimmy's Buffet      30.25
```

Here's what a deposit looks like:

```
23 Aug 2000   -      inco    -   Y      Boss Man      2001.00
```

Fields

```
1111111111➔2  ➔  3333 ➔ 4 ➔ 5 ➔ 66666666 ➔     7777777
```

Note, there are tabs (not spaces) between the fields, which you can't see in the display.

Now for the code

set up global variables

```
#!/usr/bin/awk -f
BEGIN {
    FS="\t+"
    months="Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec"
}
```

"`#!...`" allows execution directly from shell.

`BEGIN` block gets executed before nawk starts processing our checkbook file.

Set FS to "\t+" (one or more tabs).

In addition, we define a string called months.

More functions/subroutines

three basic kinds of transactions, credit (doincome), debit (doexpense) and transfer (dotransfer).

```
function doincome(mybalance) {
    mybalance[curmonth,$3] += amount
    mybalance[0,$3] += amount
}
function doexpense(mybalance) {
    mybalance[curmonth,$2] -= amount
    mybalance[0,$2] -= amount
}
function dotransfer(mybalance) {
    mybalance[0,$2] -= amount
    mybalance[curmonth,$2] -= amount
    mybalance[0,$3] += amount
    mybalance[curmonth,$3] += amount
}
```

The main code block will process each line of the checkbook file sequentially, calling one of these functions so that the appropriate transactions are recorded in an awk array.

All three functions accept one argument, called mybalance.

mybalance is a placeholder for a two-dimensional array, which we'll pass in as an argument.

We will be storing the data in a 2-dimensional "array".

What is an "array"?

An array is a table of values, called elements.

The elements of an array are distinguished by their indices.

Indices in **awk** may be either <u>numbers</u> or <u>strings</u>.
(arrays are "associative", not numerical)

(as **awk** maintains a single set of names for naming variables, arrays and functions, you cannot have a variable and an array with the same name in the same **awk** program.)

`awk` arrays

numerical array indices `in awk` start at 1 (in most computer programming languages, except fortran and matlab, arrays start at 0)

arrays are commonly indexed by numbers, but in `awk`, they can be indexed by strings

to explicitly set an array element, use brackets to specify which index of the array you are setting

```
myarray[1]="jim"     #note, strings appear in quotes
myarray[2]=456
```

or

```
myarray["name"]="jim"  #index strings appear in quotes too
```

or

```
for ( x in myarray ) {
          print myarray[x]
}
```

x gets set to an index variable by use of the <u>in</u> function, but the access order of the index variables is random

Arrays in awk superficially resemble arrays in other programming languages; but there are fundamental differences.

The most fundamental or significant difference is that any number or string may be used as an array index in awk, not just consecutive integers.

(in the end in awk, array indicies, even numerical ones, are strings)

In awk, *you* also don't need to specify the size of an array before *you* start to use it.

Arrays in **awk** are associative.
This means that each array is a collection of pairs: an index, and its corresponding array element value:

```
Element 4      Value 30
Element 2      Value "foo"
Element 1      Value 8
Element 3      Value ""
```

The pairs are shown in jumbled order because the array index order is irrelevant and has nothing to do with storage in memory.

One advantage of associative arrays is that new pairs can be added at any time.

Adding a 10th element whose value is "number ten" to our example array.

```
Element 10    Value "number ten"
Element 4     Value 30
Element 2     Value "foo"
Element 1     Value 8
Element 3     Value ""
```

Now the array is sparse, which just means some indices are missing: it has elements 1 through 4 and 10, but doesn't have elements 5 through 9.

Indices of associative arrays don't have to be positive integers.
Any number, or even a string, can be an index.
Here is an array which translates words from English into French:

```
Element "dog" Value "chien"
Element "cat" Value "chat"
Element "one" Value "un"
Element 1     Value "un”
```

We use the number one in each language spelled-out and in numeric form--a single array can have both numbers and strings as indices.

(array subscripts in **awk** are actually always strings)

The principal way of using an array is to refer to one of its elements.

An array reference is an expression which looks like this:

```
array[index]
```

Here, array is the name of an array.

The expression index is the index of the element of the array that you want.

Array elements are assigned values just like `awk` variables:

```
array[subscript] = value
```

`array` is the name of your array.

subscript is the index of the element of the array that you want to assign a value.

value is the value you are assigning to that element of the array.

mis-indexing of arrays (when they are indexed by integers) is one of the most common bugs in programming.

If you mis-index an array in awk, it just makes a new element with that index and a null value. (Wastes space and does not return value you were trying to obtain.)

To explicitly set an array element, use brackets to specify which index of the array you are setting.

strings – when used as indices or values – have to be in quotes

```
BEGIN {
animals["dog"] = "perro"
animals["cat"] = "gato"
stuff[1]=1
stuff[4]=4
stuff[-1]=-1
stuff[0]=0
print animals["dog"]
print stuff[1]
print stuff[2]
print stuff[3]
print stuff[4]
print stuff[-1]
print stuff[0]
 }
```

Reference to elements that don't exist

Execute the nawk script

```
smalley$ nawk -f arrays.nawk
perro
1


4
-1
0
smalley$
```

Null output for the ones that don't exist to delete an array element, use the delete command

```
delete myarray[1]
```