Data Analysis in Geophysics
ESCI 7205

Class 10

Bob Smalley

Basics of UNIX commands

Computers may save time but they sure waste a lot of paper. About 98 percent of everything printed out by a computer is garbage that no one ever reads.

Andy Rooney

Review

awk Working Methodology

- awk reads the input files one line at a time.

- For each line, it matches with given pattern in the given order, if matches performs the corresponding action.

- If no pattern matches, no action will be performed.

- In the above syntax, either search pattern or action are optional, But not both.

- If the search pattern is not given, then awk performs the given actions for each line of the input.

http://www.thegeekstuff.com/2010/01/awk-introduction-tutorial-7-awk-print-examples/

Review

awk Working Methodology Continued

- If the action is not given, print all that lines that matches with the given patterns which is the default action.

- Empty braces with out any action does nothing. It won't perform default printing operation.

- Each statement in Actions should be delimited by semicolon.

http://www.thegeekstuff.com/2010/01/awk-introduction-tutorial-7-awk-print-examples/

Say we have this file and we want to put it into numerical order in an awk array.

```
$ more data.txt
4
1
3
2
a
7
B
$
```

Try this.

(grey box – look at raw and sorted file, blue box – fill array with sorted elements and numerical index, yellow box print out array indices and values.)

```
$ more awkex1.nawk
#!/bin/bash
cat data.txt
echo ------
sort -n data.txt
echo ------
sort -n data.txt | \
awk 'BEGIN {c=0} {
if ( $0 > 0 ) {
print c, $0
  myarray[c]=$0; c++;
  }
}
END {
    for ( c in myarray ) printf ":: %s %s ",c,myarray[c]; printf
"\n";
    }
'
$
```

Input data file (look at it)
Sort first (not part of awk – have a tool to do this – reuse as per UNIX philosophy), pipe to awk

Try this.

(grey box – look at raw and sorted file, blue box – fill array with sorted elements and numerical index, yellow box print out array indices and values.)

```
$ more awkex1.nawk
#!/bin/bash
cat data.txt
echo ------
sort -n data.txt
echo ------
sort -n data.txt | \
awk 'BEGIN {c=0}
{if ( $0 > 0 ) {
print c, $0
  myarray[c]=$0; c++;
  }
}
END {
    for ( q in myarray ) printf ":: %s %s ",q,myarray[q]; printf
"\n";
    }
'
$
```

Initialize count

Put data (from non empty lines) in array

Print array

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.

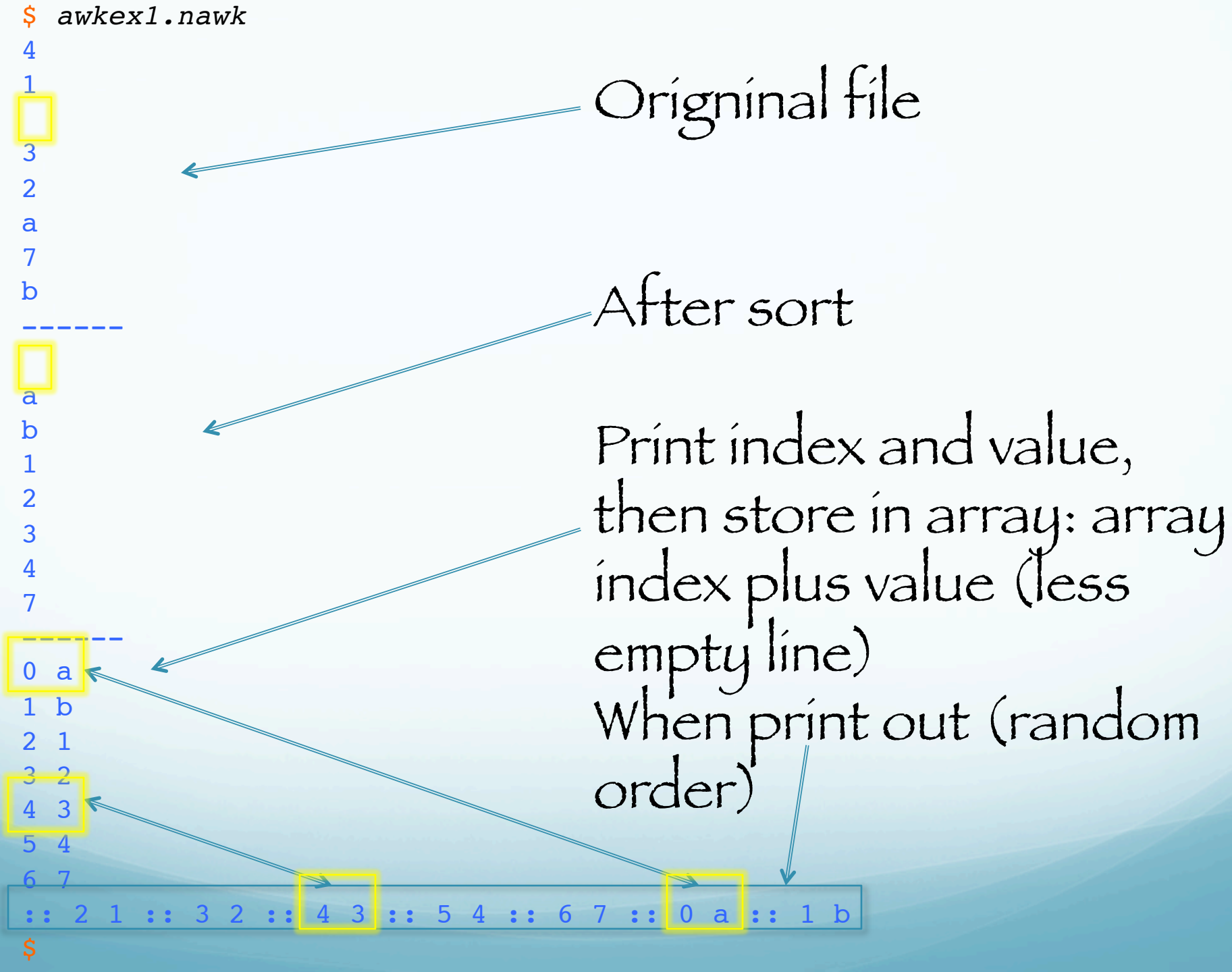
New structure

```
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).


$ awkex1.nawk
4
1
3
2
a
7
b
------
a
b
1
2
3
4
7
------
0 a
1 b
2 1
3 2
4 3
5 4
6 7
:: 2 1 :: 3 2 :: 4 3 :: 5 4 :: 6 7 :: 0 a :: 1 b
$
Orignal file
After sort
Print index and value, then store in array: array index plus value (less empty line)
When print out (random order)

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167
651 $ cat awk_arrays1.awk
nawk '
{Ip[$3]++;}
END
{for (var in Ip)
print var, "access", Ip[var],"
times"}
' Iplogs.txt
652 $ awk_arrays1.awk
202.178.23.4 access 3  times
125.25.45.221 access 1  times
202.188.3.2 access 2  times
123.12.23.122 access 1  times
164.78.22.64 access 1  times
```

Want to count how many times each unique IP address accessed.

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

Do with awk array.

Define array elements on first reference, increment on each reference (from zero or empty to 1, the ++, on first reference, then keeps counting).

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167
651 $ cat awk_arrays1.awk
nawk '
{Ip[$3]++;}
END
{for (var in Ip)
print var, "access", Ip[var],"
times"}
' Iplogs.txt
652 $ awk_arrays1.awk
202.178.23.4 access 3  times
125.25.45.221 access 1  times
202.188.3.2 access 2  times
123.12.23.122 access 1  times
164.78.22.64 access 1  times
```

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

Third field ($3) is an ip address. This is used as an index of an array called Ip.

For each line, it increments the value of the corresponding ip index.

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167
651 $ cat awk_arrays1.awk
nawk '
{Ip[$3]++;}
END
{for (var in Ip)
print var, "access", Ip[var],"
times"}
' Iplogs.txt
652 $ awk_arrays1.awk
202.178.23.4 access 3  times
125.25.45.221 access 1  times
202.188.3.2 access 2  times
123.12.23.122 access 1  times
164.78.22.64 access 1  times
```

Finally in the END section, the indices (not the position in the array as in Fortran, C, C++, etc. arrays) are the list of unique IP address and the corresponding values are the occurrence counts.

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

In awk arrays the index is both the element identifier and can be data/information.

The value is also data/information associated with the index.

The "regular" index concept does not apply, there is no method to identify (index) or access the values in a counting (address offset) manner and attempts to do so produce what looks like random ordering.

"Regular Index"	awk index	value
1	202.178.23.4	3
2	125.25.45.221	1
3	202.188.3.2	2
4	123.12.23.122	1
5	164.78.22.64	1

The indices and values of each element of an array don't even have to be of the same type (character or numeric – but numeric data is really numeric in your mind, it is a character string to awk, although awk can try to use it as a number if you refer to it in a mathematical context) or length.

"Regular Index"	awk index	value
1	202.178.23.4	3
2	125.25.45.221	1
3	202.188.3.2	2
4	123.12.23.122	1
5	164.78.22.64	1
6	character index	character value

Setting array indices and values.

```
END{
Ip["A"]="letterA"
B="B"
LB="letterB"
Ip[B]
Ip[B]=LB
Ip["C"]=1.2
Ip[D]="letterD"
for (var in Ip)
Print var,\
 "is index, \
element value", Ip[var]
}
```

```
$ awk_arrays1.awk
 is index, element value letterD
A is index, element value letterA
202.178.23.4 is index, element value 3
B is index, element value letterB
C is index, element value 1.2
$
```

Everything is a character string to awk, although it will take numbers without the quotes. If you try to use an un-defined variable (D) as an index it sets the index to null (notice missing character, is not putting out blank).

Setting array indices and values.

```
END{
Ip["A"]="letterA"
B="B"
LB="letterB"
Ip[B]
Ip[B]=LB
Ip["C"]=1.2
Ip[D]="letterD"
Ip[E]="letterE"
for (var in Ip)
Print var,\
 "is index, \
element value", Ip[var]
}
```

Now you have reset the value associated with index null to something else.

```
$ awk_arrays1.awk
 is index, element value letterE
A is index, element value letterA
202.178.23.4 is index, element value 3
B is index, element value letterB
C is index, element value 1.2
$
```

```
END{
Ip["A"]="letterA"
B="B"
LB="letterB"
Ip[B]
Ip[B]=LB
Ip["C"]=1.2
Ip[D]="letterD"
Ip[E]=1.2a
for (var in Ip)
Print var,\
 "is index, \
element value", Ip[var]
}
```

```
$ awk_arrays1.awk
 is index, element value letterD
A is index, element value letterA
202.178.23.4 is index, element value 3
B is index, element value letterB
C is index, element value 1.2
$
```

If you try to assign something that is not a character string (no quotes) or an existing variable to an undefined index **awk** seems to ignore it completely - output is one line shorter than what you think it should be – is two null elements (we really don't know what is in memory).

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167
```

```
{ Ip[$3]++;
count[$3]+=$NF; }
```

Don't really need the semicolons

Count how many times each unique IP address accessed (from before), and calculate how many sites each accessed.

Two arrays, the index used in both arrays is same — which is the IP address (third field).

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167
```

Identify day with maximum number accesses.

array named "date" has date as its index and occurrence count as the value of the array. This one line does all the "work" of calculating accesses.

```
{ date[$1]++; }
END
{
 for (count in date)
 { if ( max < date[count]
   { max = date[count];
     maxdate = count;
   }
 } print "Maximum access is on" maxdate;
}
```

Don't really need all the semicolons

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64    12
200607 012312 202.188.3.2     13
210607 084849 202.178.23.4    34
210607 121435 202.178.23.4    32
210607 132423 202.188.3.2    167
```

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

max is a variable which has the count value and is used to find array element in date with max count (evidently starts out undefined, 0, or minimum).

maxdate is variable with date index for which the count is maximum.

```
{ date[$1]++; }
END
{
 for (count in date)
 { if ( max < date[count]
   { max = date[count];
     maxdate = count;
   }
 } print "Maximum access ", date[count], " is on" maxdate;
}
```

```
651 $ awk -f ex3.awk Iplogs.txt
Maximum access 3 is on 210607
```

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167
```

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

Original example output did not put out maximum number of accesses.

```
{ date[$1]++; }
END
{
 for (count in date)
 { if ( max < date[count]
   { max = date[count];
     maxdate = count;
   }
 } print "Maximum access is on" maxdate;
}
```

```
651 $ awk -f ex3.awk Iplogs.txt
Maximum access is on 210607
```

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167
```

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

And whenever you put something on the web and allow comments, somebody comes along with an "improvement" (to the code, not the English).

I think solve example 3 more effective is

```
awk ‘max < $1 { max = $1 } END { print "Maximum access is on"
max }' Iplogs.txt

Maximum access is on 210607
```

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167
```

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

```
awk '
{ a[i++] = $0 }
END
{ for (j=i-1; j>=0;)
  print a[j--]
}' Iplogs.txt
```

Reverse the order of lines in a file

Starts by recording all lines in the array 'a'. Index i also serves to count number lines read in (evidently starts out at 1).

```
651 $ awk -f ex3.awk Iplogs.txt
Maximum access 3 is on 210607
```

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167

awk '!($1 in array)
{ array[$1]; print $1
' Iplogs.txt

651 $ ex5.awk
180607
190607
200607
210607
```

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

Remove duplicate and nonconsecutive dates (first field `$1`, for lines use whole line `$0`, but whole line, not just date, has to be duplicate) using awk.

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

```
650 $ cat Iplogs.txt
180607 093423 123.12.23.122 133
180607 121234 125.25.45.221 153
190607 084849 202.178.23.4   44
190607 084859 164.78.22.64   12
200607 012312 202.188.3.2    13
210607 084849 202.178.23.4   34
210607 121435 202.178.23.4   32
210607 132423 202.188.3.2   167
```

Data format: [date] [time] [ip-address] [number-of-websites-accessed]

```
awk '!($1 in array)
{ array[$1]; print $1
' Iplogs.txt
```

```
651 $ ex5.awk
180607
190607
200607
210607
```

Reads every line from file `Iplogs.txt`, uses "`in`" operator to check if current test pattern (CTP=`$1`) exists in the array "`a`".

If the CTP does not exist in "`a`" (the `!`), it stores the CTP as that array index(the date) and prints the current line.

http://www.thegeekstuff.com/2010/03/awk-arrays-explained-with-5-practical-examples/

you can also set arrays using the `split` command

```
split(“string”,destination array,separator)
```

`split` also returns the number of indices

```
numelements=split("Jan,Feb,Mar,Apr,May",mymonths,",”)
```

Splits the string into array elements using the “,” to break the string into elements, and returns `numelements=5` and `mymonths[1]=“Jan”`

A multi-dimensional awk array is an array in which an element is identified by a sequence of indices, instead of a single index.

For example, a two-dimensional array requires two indices.

The usual way to refer to an element of a two-dimensional array named grid is with `grid[x,y]`.

http://people.cs.uu.nl/piet/docs/nawk/nawk_87.html

Multi-dimensional arrays are supported in awk through concatenation of indices into one string.

What happens is that awk converts the indices into strings and concatenates them together, with a separator between them.

This creates a single string that describes the values of the separate indices.

http://people.cs.uu.nl/piet/docs/nawk/nawk_87.html

The combined string is used as a single index into an ordinary, one-dimensional array.

The separator used is the value of the built-in variable `SUBSEP`.

Once the element's value is stored, awk has no record of whether it was stored with a single index or a sequence of indices.

The two expressions `foo[5,12]` and `foo[5 SUBSEP 12]` always have the same value.

http://people.cs.uu.nl/piet/docs/nawk/nawk_87.html

The default value of `SUBSEP` is the string `"\034"`, which contains a nonprinting character that is unlikely to appear in an awk program or in the input data.

http://people.cs.uu.nl/piet/docs/nawk/nawk_87.html

Need to choose an unlikely character due to the fact that index values containing a string matching `SUBSEP` lead to combined strings that are ambiguous.

Suppose `SUBSEP` were `"@"`;
then `foo["a@b", "c"]`
and `foo["a", "b@c"]`
would be indistinguishable because both would actually be stored as
`foo["a@b@c"]`.

http://people.cs.uu.nl/piet/docs/nawk/nawk_87.html

Because **SUBSEP** is "\034", such confusion can arise only when an index contains the character with ASCII code 034, which is a rare event.

The following example treats its input as a two-dimensional array of fields; it rotates this array 90 degrees clockwise and prints the result. It assumes that all lines have the same number of elements.

```
awk '{
    if (max_nf < NF)
        max_nf = NF
     max_nr = NR
     for (x = 1; x <= NF; x++)
        vector[x, NR] = $x
}
END {
    for (x = 1; x <= max_nf; x++) {
        for (y = max_nr; y >= 1; --y)
            printf("%s ", vector[x, y])
            printf("\n")
    }
}'
```

http://people.cs.uu.nl/piet/docs/nawk/nawk_87.html

When given the input:

```
1  2  3  4  5  6
2  3  4  5  6  1
3  4  5  6  1  2
4  5  6  1  2  3
```

it produces:

```
4  3  2  1
5  4  3  2
6  5  4  3
1  6  5  4
2  1  6  5
3  2  1  6
```

http://people.cs.uu.nl/piet/docs/nawk/nawk_87.html

Summary

awk emulates multidimensional arrays with single-dimensional arrays by combining two or more indices into a single string.

From the point of view of awk, it looks like a single index, but to it is composed of two or more discrete parts.

Back to our checkbook

Record information into "`mybalance`" as follows.

The first dimension of the array ranges from `0` to `12`, and specifies the entire year (`0`) or month (number of month).

Our second dimension is a four-letter category, like "`food`" or "`inco`"; this is the actual category we're dealing with.

(remember that the dimensions are not fixed – we can add categories at will)

So, to find the entire year's balance for the food category, you'd look in

```
mybalance[0,"food"].
```

To find June's income, you'd look in

```
mybalance[6,"inco"].
```

Arrays are passed by reference.

We also refer to several global variables:

`curmonth, (numeric value of month of current record),`

`$2` (expense category),

`$3` (income category).

```
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
}
```

Passing of information between calling routine and subroutine.

Two basic ways.

By reference

Tell subroutine where the information is in the memory and the subroutine uses it. Changes made by the subroutine are global.

By value

Give the subroutine a copy of the information. Any changes made by the subroutine are local to its copy of the data.

The main code block contains the code that parses each line of input data.

Remember, because we have set `FS` correctly, we can refer to the first field as `$1`, the second field as `$2`, etc.

When the functions are called, they can access the current values of `curmonth`, `$2`, `$3` and `amount` from inside the function.

```
#main program
{
    curmonth=monthdigit(substr($1,4,3))
    amount=$7

    #record all the categories encountered
    if ( $2 != "-" )
        globcat[$2]="yes"
    if ( $3 != "-" )
        globcat[$3]="yes"

    #tally up the transaction properly
    if ( $2 == "-" ) {
        if ( $3 == "-" ) {
            print "Error: inc and exp fields are both blank!"
            exit 1
        } else {
            #this is income
            doincome(balance)
            if ( $5 == "Y" )
                doincome(balance2)
        }
```

```
    } else if ( $3 == "-" ) {
        #this is an expense
        doexpense(balance)
        if ( $5 == "Y" )
        doexpense(balance2)
    } else {
        #this is a transfer
        dotransfer(balance)
        if ( $5 == "Y" )
            dotransfer(balance2)
    }
}
#end of main program
END {
    bal=0
    bal2=0
    for (x in globcat) {
        bal=bal+balance[0,x]
        bal2=bal2+balance2[0,x]
        }
        printf("Your available funds: %10.2f\n", bal)
        printf("Your account balance: %10.2f\n", bal2)
}
```

Input file:

```
23 Aug 2000   food   -   -   Y   Jimmy's Buffet      30.25
23 Aug 2000   -      inco  -   Y   Boss Man           2001.00
```

Output to the screen:

```
Your available funds:    1174.22
Your account balance:    2399.33
```

Shell arrays (now that we know what they are – does the Shell have them?)

The shell also has arrays.

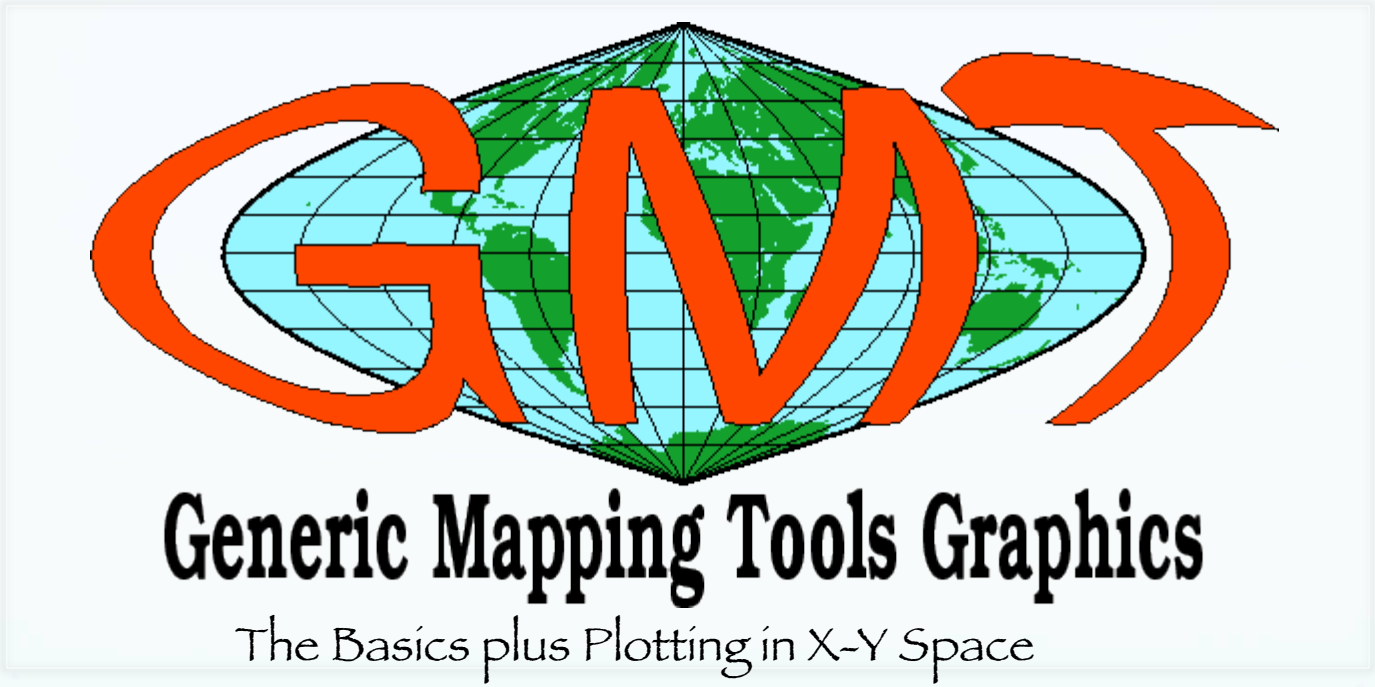
In the Shell, the index has to be a number (but the numbers don't have to be consecutive and it does not eat up memory for empty indices).

Shell arrays

```
#!/bin/sh
#call with array_gamit.sh [yr][v]
…
YRS='2011 2008 2004'
…
DAYS[2004]='037049 055059 084112 114115 235238 243244 333349’
DAYS[2008]='005031’
DAYS[2011]='072116’
…
for YR in $YRS
do
…
for day in ${DAYS[${YR}]}
do
…
STRTDOY=`echo $day | nawk '{print substr($1,1,3)}'`
STOPDOY=`echo $day | nawk '{print substr($1,4,3)}'`
…
done
…
done
```

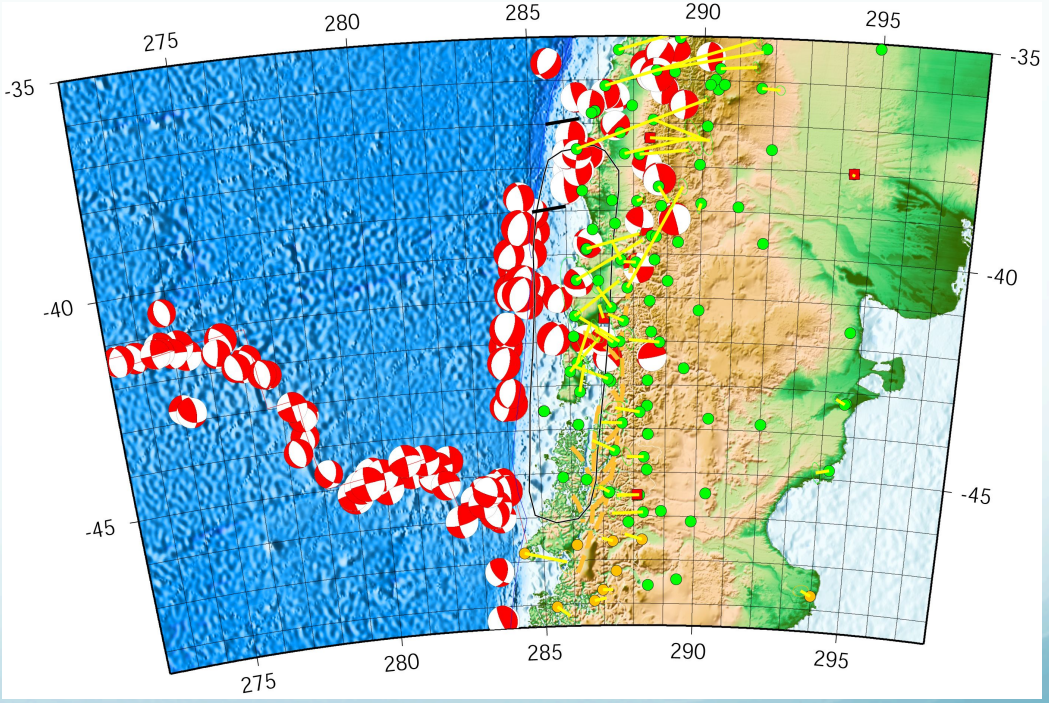
to test if an element exists, can use

```
for ( 1 in myarray ) {
print “It’s there”
} else {
print “It’s missing”
}
```



Generic mapping tools (GMT)

Goal – make scientific illustrations ("generic" of GMT is generic to geo sciences)



Goal – make scientific illustrations

Maps

- Color/bw/shaded topography and bathymetry,
- Point data (earthquakes, seismic or gps stations, etc.),
- Line data (faults, eq rupture zones, roads),
- Vector fields w/ error ellipses,
- Focal mechanisms
- 3D surface
- Cross sections
- Profiles
- Other stuff

What is GMT

GMT is an open source collection of ~60 tools (and and additional 35 support tools) for manipulating geographic and Cartesian data sets

(including filtering, trend fitting, gridding, projecting, etc.)

What is GMT

Produces PostScript File (PS).

Make illustrations ranging from simple x-y plots to contour maps to artificially illuminated surfaces and 3-D perspective views

GMT supports ~30 map projections and transformations and comes with support data such as GSHHS coastlines, rivers, and political boundaries.

If it does not have a map projection *you* want: it is open source and UNIX.
(i.e. *you* can do it *yourself*)

Design Philosophy

Follows the design philosophy of UNIX – filters (linear, single data stream):

data → processing → final illustration.

Processing flow is broken down to a series of elementary steps.

Each step is accomplished by a separate GMT or UNIX tool (machine shop philosophy).

Design Philosophy

Benefits (UNIX only has benefits):

(1) only a few programs are needed

(in the world where 60+35 is a "few", maybe they are referring to the log of the number of programs.)

(2) each program is small and easy to update and maintain

(maybe – alternate is each task is subroutine that is small and easy to maintain)

Design Philosophy

Benefits (UNIX only has benefits):

(3) each step is independent of the previous step and the data type and can therefore be used in a variety of applications

(4) the programs can be chained together in shell scripts or with pipes, thereby creating a process tailored to do a user-specific task

Design Philosophy

GMT was deliberately written for command line usage, not a windows (or interactive) environment, in order to maximize power and flexibility (i.e. it is hard to use).

Written by Paul Wessel and Walter Smith while graduate students at Lamont Doherty/Columbia University in the mid 80's when the SUN workstations came out (and UNIX took over the world).

.

(Now at the University of Hawaii and NOAA respectively

The GMT homepage is: gmt.soest.hawaii.edu

GMT documentation

Tutorial

Technical Reference and Cookbook

(aka Manual)

both available on web

http://gmt.soest.hawaii.edu/

in HTML, PDF, and PostScript format.

As is standard with UNIX

GMT is well documented with (UNIX style) "man" pages (also on web).

Entering GMT program/filter name all by itself, or errors in the command specification (switches, not data) that cause GMT to fall over – dumps the man page to standard error.

What does/can GMT do?

-Filtering 1-D and 2-D data

(simple processing, GMT is NOT a general Number Cruncher)

output is reprocessed data

Plotting 1-D and 2-D data

- points, lines (symbols, fill, geologic symbols on faults, etc.)

- vector fields

2-D images – grayscale and color, illumination

3-D perspective of 2-D images

histograms, rose diagrams

text

focal mechanism beachballs

Data preparation

gridding, resampling, conversion

Contouring

data base: extraction, merge

cross sections

projection/map transformation (map sphere to plane)

output is reprocessed data

Bookkeeping and bunch of other stuff

GMT Processing Output

1-D ASCII Tables — For example, a (x, y) series may be filtered and the filtered values output.

ASCII output is written to the standard output stream.

GMT Processing Output

2-D binary (netCDF or user-defined) grid files

Programs that grid ASCII (x, y, z) data or operate on existing grid files produce this type of output.

GMT Output

Reports – Several GMT programs read input files and report statistics and other information.

Nearly all programs have an optional "verbose" operation, which reports on the progress of computation.

Such text is written to the standard error stream

GMT Output

The bulk of GMT output goes to

PostScript

The plotting programs all use the *PostScript* page description language to describe the output.

These commands are stored as ASCII text (they are a program in the POSTSCRIPT language).

output is "PostScript" program – generally ascii text, but not too readable.

(GMT files can get amazingly BIG)

```
% Map boundaries
%
S 1050 1050 1050 0 360 arc S
S 1050 1050 1074 0 360 arc S
S 24 W
S 1050 1050 1062 -135 -90 arc S
S 1050 1050 1062 135 180 arc S
S 1050 1050 1062 45 90 arc S
S 1050 1050 1062 -45 0 arc S
S 1050 1050 1062 -90 -90 arcn S
S 2 W
S [] 0 B
%
% End of basemap
%
S [] 0 B
%%Trailer
%%BoundingBox: 0 0 647 647
% Reset translations and scale and call showpage
S -295 -295 T 4.16667 4.16667 scale 0 A
showpage
```

GMT Output

If you are really ambitious, you can directly edit this file using vi…but in general, don't.

GMT Output

Postscript is translated by postscript capable (usually laser) printers.

(it is an extra feature one has to buy).

GMT Output

Postscript is also the native language of

- Adobe Illustrator/Photoshop

- ghostscript,

- ghostview.

GMT Output

I frequently use Illustrator to edit GMT produces Postscript prior to using the figures in papers, presentations, or posters

Apart from the built-in support for coastlines, GMT completely decouples data retrieval/ management from the main GMT programs.

(puts the onus on user! UNIX philosophy)

GMT uses architecture-independent file formats

(flat files – least common denominator).

Effective use of GMT is really effective application of the UNIX philosophy.

Installation/Maintenance - done for us

(by Mitch/Deshone – THANKS.

Somewhat complicated, not for average user.)

Setup – basic setup done for us

(don't have to define GMTHOME, path, etc. if use standard CERI .login and .cshrc files)

Installation/Maintenance.

Some common data sets

(GTOPO-30, ETOPO-5, Predicted bathy, etc.)

are installed

"**.gmtdefaults**" (generic, is .gmtdefaults4 for version 4) file in your home or working directory.

(if you've copied something from the tutorial or gotten a script from someone else and it comes out "funny", the "default" settings may be the culprit).