The awk Pattern Scanning and Processing Language

- scans text files line-by-line and searches for patterns.
- works in a way similar to sed but it is more versatile.
- Sample runs:

```
>>> awk 'length>52 {print $0}' filein
>>>                     % length is the # of char in a line
>>>
>>> awk 'NF%2==0 {print $1}' filein
>>>                     % NF = number of fields
>>>
>>> awk '$1=log($1); print' filein
>>>                     % replaces the 1st argu with..
>>>
```

awk Pattern Morphing and Processing

```
>>> awk '{print $3 $2}' filein
>>> awk '$1 != prev {print $0; prev=$1}' filein
>>>                     % print all lines for which the
>>>                     % argu is diff from the 1st argu
>>>                     % of the previous line
>>>
>>> awk '$2~/A|B|C/ {print $0}' filein
>>>                     % prints all lines with A or B
>>>                     % or C in the 2nd argu
>>>
```

- General invocation options:
  1. `awk -f filewithawkcommands inputfile`
  2. `awk '{awk-commands}' inputfile`

awk basic file-instruction layout

BEGIN	{declarations; action(s);}
$pattern_1$	{ action(s); }
$pattern_2$	{ action(s); }
$pattern_3$	{ action(s); }
.....	.....
$pattern_n$	{ action(s); }
END	{ action(s); }

- Either pattern or action may be left out.
- If *no* action exists, simply the input matching line is placed on the output.

Records and Fields

- Input is divided into "records" – ended by a terminator character whose default value is `\n`.
- `FILENAME`: the name of the current input file.
- Each record is divided into "fields" separated by white-space blanks *OR* tabs.
- Fields are referred to as `$1`, `$2`, `$3`, ...
- The entire string (record) is denoted as `$0`
- `NR`: is the number of current record.
- `NF`: number of fields in the line
- `FS`: field separator (default " ")
- `RS`: record separator (default `\n`)

Printing in awk

1. `{print}`
   ⇒ print the entire input file to output.
2. `{print $2, $1}`
   ⇒ print $field_2$ and $field_1$ from input file.
3. `{ print NR, NF, $0 }`
   ⇒ print the number of the *current* record, the *number of its fields*, and the entire record.
4. `{ print $1 > "foo"; print $2 > "bar" }`
   ⇒ print fields into multiple output files; >> can be also used.
5. `{ print $1 > $2 }`
   ⇒ the name of $field_2$ is used as a file (for output).
6. `{ printf("%8.2f %-20s \n",$1, $2); }`
   ⇒ pretty-printing with C-like notation.

Patterns in awk

- patterns in front of actions act as *selectors*.
- awk file: special keywords BEGIN and END provide the means to gain control before and after the processing of awk:

```
BEGIN   { FS=":" }
        { print $2 }
END     { print NR }
```

- Output:

```
gympie:~/Samples$ cat awkfile1
alex:delis
mike:hatzopoulos
dimitris:achlioptas
elias:koutsoupias
alex:eleftheriadis
gympie:~/Samples$ awk -f awk1 awkfile1
delis
hatzopoulos
achlioptas
koutsoupias
eleftheriadis
5
gympie:~/Samples$
```

Regular Expressions (some initial material)

- `/smith/`
  ⇒ find all lines that contains the string "smith"
- `/[Aa]ho|[Ww]einberger|[Kk]ernigham/`
  ⇒ find all lines containing the strings "*Aho*" or "*Weinberger*" or "*Kernighham*" (starting either with lower or upper case).
  - `|` : alternative
  - `+` : one or more
  - `?` zero or one
  - `[a-zA-Z0-9]` : matches any of the letters or digits
- `/\/.*\//` : ⇒ matches any set of characters enclosed *between* two slashes.
- `$1~/[jJ]ohny/` **or** `$1!~/[jJ]ohny/`
  ⇒ matches (or not!) all records whose first field in *Johny* or *johny*.

Relational Expressions: <, <=, ==, !=, >=, >

- `'$2 > $1 + 100'`
  ⇒ selects lines whose records comply with the condition.
- `'NF%2 == 0'`
  ⇒ project lines with even number of records.
- `'$1 >= "kitsos"'`
  ⇒ display all lines whose first parameter is alphanumerically greater or equal to `"kitsos"`.
- `'$1 > $2'`
  ⇒ similarly as above but arithmetic comparison.

Combinations of Patterns:

- || (OR), && (AND) and ! (not).
- Expressions evaluated left-to-right
- Example: `($1 >= "s") && ($1 < "t")`
  `&& ( $1 !="smith" )`

Pattern Ranges:

- `'/start/,/stop/'` : prints all lines that contain string `start` or `stop`.

Built-in Functions

- `{print (length($0)),$0 }` **OR** `{print length,$0}`
- `sqrt, log (base e), exp, int, cos(x), sin(x), srand(x), atan2(y,x)`
- `substr(s,m,n)`: produces the string `s` that starts at position `m` and is at most `n` characters.
- `index(s1,s2)`: return the position in which `s2` starts in the string `s1`.
- `x=sprintf("%8.3f %10d \n", $1, $2);`
  $\Rightarrow$ sets string `x` to values produced by `$1` and `$2`.

Variables, Expressions and Assignments

- `awk` uses int/char variables based on context.
  - `x=1`
  - `x='smith'`
  - `x="3"+"4"` (`x` is set to 7)
  - variable are set in the `BEGIN` section of the code but by default, are initialized *anywhere* to `NULL` (or implicitly to zero)

    ```
    { s1 += $1 ; s2 += $2 }
    END { print s1, s2 }
    ```

    if `$1` and `$2` are floats, `s1, s2`, also function as floats.

Regular Expressions and Metacharacters

- Regular-expression Metacharacters are:
  `\, ^, $, [, ], |, (, ), *, +, ?`
- A basic regular expression (**BRE**) is:
  - a non-metacharacter matches itself such as `A`.
  - an escape character that matches a *special symbol*: `\t` (tab), `\b` (backspace), `\n` (newline) etc.
  - a quoted metacharacter (matching itself): `\*` matches the *star* symbol.
  - `^` matches the *beginning* of a string.
  - `$` matches the *end* of a string.
  - `.` matches any *single* character.
  - a character class `[ABC]` matches a *single* `A`, `B`, or `C`.
  - character classes abbreviations `[A-Za-z]` matches *any single* character.
  - a complementary class of characters `[^0-9]` matches any character *except* a digit
    (what would the pattern `/^[^0-9]/` match?)

More Complex Regular Expressions using BREs

◇ Operators that can combine BREs (see below `A`, `B`, `r`) into larger regular expressions:

`A|B` matches `A` or `B` (alternation)

`AB` `A` followed by `B` (concatenation)

`A*` zero or more `A`s (closure)

`A+` at least one `A` or more (positive closure)

`A?` matches the null string or `A` (zero or one)

`(r)` matches the same string as `r` (parentheses)

Examples:

- `/^[0-9]+$/`
  matches any input lines that consists of only digits.
- `/^[+-]?[0-9]+[.]?[0-9]*$/`
  matches a decimal number with an optional sign and optional fraction.
- `/^[A-Za-z]|^[A-Za-z][0-9]$/`
  a letter or a letter followed by a digit.
- `/^[A-Za-z][0-9]?$/`
  a letter or a letter followed by a digit.
- `/\/.*\//`
  matches any set of characters enclosed between two slashes
- `$1~/[jJ]ohny/`
  matches all records whose first field is *Johny* or *johny*
- `$1!~/[jJ]ohny/`
  matches all records whose first field is not *Johny* or *johny*.

Dealing with Field Values

```
gympie:~/Samples$ cat awk2
{   if ($2> 1000)
          $2 = "too big";
    print;
}
gympie:~/Samples$
```

```
gympie:~/Samples$ awk -f awk2 test5
ddd   100
eee too big
rrr   99
fff   899
f11 too big
f2    992
gympie:~/Samples$
```

Splitting a string into its Elements using an array

- The function `split()` helps separate a string into a number of token (each token being part of the resulting array).

```
BEGIN{ sep= ";" }
{ n = split ($0, myarray, sep); }
END {
        print "the string is:"$0;
        print "the number of tokens is="n;
        print "The tokens are:"
        for (i=1;i<=n;i++)
                print myarray[i];
    }
```

```
gympie:~/Samples$ cat data3
alexis;delis;apostolos;nikolaos
gympie:~/Samples$ awk -f awk3 data3
the string is:alexis;delis;apostolos;nikolaos
the number of tokens is=4
The tokens are:
alexis
delis
apostolos
nikolaos
gympie:~/Samples$
```

Arrays

- Feature: Arrays are not declared - they are simply used!
- 'X[NR]=$0' assigns current line to the NR element of array X
- Arrays can be used to collect statistics:

```
gympie:~/Samples$ more awk4
/apple/      {X["apple"]++}
/orange/     {X["orange"]++}
/grape/      {X["grape"]++}
END {
    print "Apple Occurrences = " X["apple"];
    print "Orange Occurrences = " X["orange"];
    print "Grape Occurrences = " X["grape"];
    }
gympie:~/Samples$
```

```
gympie:~/Samples$ awk -f awk4 text5
Apple Occurrences = 8
Orange Occurrences = 5
Grape Occurrences = 4
gympie:~/Samples$
```

Control Flow Statements

- `{ statements }`
- `if (expression) statement`
- `if (expression) statement1 else statement2`
- `while (expression) statement`
- `for (expression1; expression2; expression3) statement`
- `for (var in array) statement`
- `do statement while (expression)`
- `break` // immediately leave innermost enclosing while, for or do
- `continue` //start next iteration of innermost enclosing while, for or do
- `next` //start next iteration of main input loop
- `exit`
- `exit expression` //return expression value as program status

Example with while

```
gympie:~/Samples$ cat awk5
{    i=1
     while (i <= NF ) {
          print $i;
          i++;
          }
}
gympie:~/Samples$
```

```
gympie:~/Samples$ cat data4
mitsos kitsos mpellos
alexis mitsos apostolos nikolaos
aggeliki ourania eleftheria mitsos
gympie:~/Samples$ awk -f awk5 data4
mitsos
kitsos
mpellos
alexis
mitsos
apostolos
nikolaos
aggeliki
ourania
eleftheria
mitsos
gympie:~/Samples$
```

Similar effect with for-loop

```
gympie:~/Samples$ cat awk6
{ for (i=1; i<=NF; i++)
    print $i;
}
gympie:~/Samples$
```

```
gympie:~/Samples$ awk -f awk6 data4
mitsos
kitsos
mpellos
alexis
mitsos
apostolos
nikolaos
aggeliki
ourania
eleftheria
mitsos
gympie:~/Samples$
```

Population Table

```
Asia          Indonesia    230      376
Asia          Japan        160      154
Asia          India        1024     1267
Asia          PRChina      1532     3705
Asia          Russia       175      6567
Europe        Germany      81       178
Europe        UKingdom     65       120
N.America     Mexico       130      743
N.America     Canada       41       3852
S.America     Brazil       150      3286
S.America     Chile        8        112
```

```
gympie:~/Samples$ more awkgeo
BEGIN{
    printf("%10s %12s %8s %10s\n","COUNTRY","AREA","POP","CONTINENT");
    printf("---------------------------------------------------------\n")
;
    }
    {
    printf("%10s %12s %8d   %-12s\n",$2, $4, $3, $1);
    area = area + $4;
    pop = pop + $3;
    }
END {
    printf("---------------------------------------------------------\n")
;
    printf("%10s in %12d km^2 %8d mil people live   \n\n", "TOTAL:",area, po
p);
    }
gympie:~/Samples$
```

Outcome

```
gympie:~/Samples$ awk -f awkgeo continents
   COUNTRY          AREA       POP  CONTINENT
-------------------------------------------------------------
 Indonesia           376       230   Asia
     Japan           154       160   Asia
     India          1267      1024   Asia
   PRChina          3705      1532   Asia
    Russia          6567       175   Asia
   Germany           178        81   Europe
  UKingdom           120        65   Europe
    Mexico           743       130   N.America
    Canada          3852        41   N.America
    Brazil          3286       150   S.America
     Chile           112         8   S.America
-------------------------------------------------------------
    TOTAL: in        20360 km^2     3596 mil people live

gympie:~/Samples$
```

Computing and Graphing Deciles - User-defined Functions

```
# input: numbers from 0 to 100 - one at a line
# output: decile population graphed

    { x[int($1/10)]++ ; }

END {
    for (i=0; i<10; i++)
        printf("%2d - %2d: %3d %s\n",
                    10*i, 10*i+9, x[i], rep(x[i],"*") );
    printf("100:      %3d %s\n",x[10], rep(x[10],"*") );
    }

#returns string of n s's
function rep(n,s) {
    t= "";
    while (n-- > 0)
        t = t s
    return t
    }
```

Outcome (deciles)

```
gympie:~/src-set003$ awk -f awk.deciles data6
 0 -  9:    3 ***
10 - 19:    3 ***
20 - 29:    5 *****
30 - 39:    6 ******
40 - 49:   12 ************
50 - 59:   14 **************
60 - 69:   14 **************
70 - 79:   12 ************
80 - 89:    6 ******
90 - 99:    5 *****
100:        2 **
gympie:~/src-set003$
```

User-defined Functions

- Function definitions may occur anywhere a pattern-action statement can.
- Functions often are listed at the end of an awk script and are separated by either newlines or semicolons.
- They contain a `return expression` statement that returns control along with the value of the `expression`.
- Example:

```
function mymax( a, b) {
  return a > b ? a : b
  }
```

- Recursive invocation:

```
  { print mymax($1, mymax($2,$3) ) }
```

Built-in String Functions

Function Name	*Description*
`gsub(r,s)`	substitute `s` for `r` globally in $0; return number of substitutions made
`gsub(r,s,t)`	substitute `s` for `r` globally in string `t`; return number of substitutions made
`index(s,t)`	return first position of `t` in `s`; otherwise zero
length(s)	return number of characters in `s`
`match(s,r)`	test whether `s` contains a substring matched by `r`; return index or 0.
`split(s,a)`	split `s` into array a on `FS`; return number of fields
`split(s,a,fs)`	as above – `fs` is the defined field seperator
`sprintf(ftm,exprlst)`	format an expression list
`sub(r,s)`	substitute `s` for the leftmost longest substring of $0 matched by `r`; return number of subs made.
`sub(r,s,t)`	substitute `s` for the leftmost longest substring of `t` matched by `r`; return number of subs made.
`substr(s,p)`	return suffix of `s` starting at position `p`