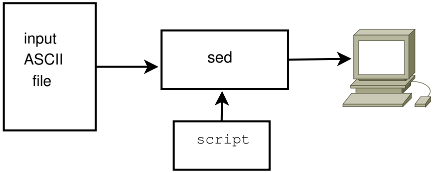
sed and awk Programming

March 2017

sed

- Character Stream Processor for ASCII files
  - not really an editor!
- Operational model: `sed` scans the input ASCII file on a line-by-line fashion and applies a set of rules to all lines.
- `sed` has three options:

  `-e` : script is on the command line (default case)
  `-f` : finds all rules that are applied in a specific (script) file.
  `-n` : suppresses the output



Invoking sed

- bash > sed -e 'address command' inputfile
- bash > sed -f script.sed inputfile
- each instructions given to sed consists of an address and command.
- Sample sed-script file:

```
#This line is a comment
2,14 s/A/B/
30d
40d
```

1. From lines 2 to 14 substitute the character A with B
2. Line 30 - delete it!
3. Line 40 - delete it!

sed 's/[0-9]//g'

```
gympie:~/Samples$ cat lista
john         32       london
eduardo      19       brazilia
winnie       97       cordoba
jean         21       athens
marco         7       buenosaires
filip        23       telaviv
dennis       15       brisbane
louis        31       heraclion
dimi         34       heraclion
ji           27       washington
hyseyin      33       izmir
gympie:~/Samples$
```

```
gympie:~/Samples$ cat lista | sed 's/[0-9]//g'
```

```
john                  london
eduardo               brazilia
winnie                cordoba
jean                  athens
marco                  buenosaires
filip                 telaviv
dennis                brisbane
louis                 heraclion
dimi                  heraclion
ji                    washington
hyseyin               izmir
gympie:~/Samples$
```

Substitution at the front and at the end of a line

```
gympie:~/Samples$ cat lista | sed 's/$/>>>/'
```

```
john          32        london>>>
eduardo       19        brazilia>>>
winnie        97        cordoba>>>
jean          21        athens>>>
marco          7        buenosaires>>>
filip         23        telaviv>>>
dennis        15        brisbane>>>
louis         31        heraclion>>>
dimi          34        heraclion>>>
ji            27        washington>>>
hyseyin       33        izmir>>>
```

```
gympie:~/Samples$ cat lista | sed 's/$/>>>/g' | \
                  sed 's/^/<<</g'
```

```
<<<john          32        london>>>
<<<eduardo       19        brazilia>>>
<<<winnie        97        cordoba>>>
<<<jean          21        athens>>>
<<<marco          7        buenosaires>>>
<<<filip         23        telaviv>>>
<<<dennis        15        brisbane>>>
<<<louis         31        heraclion>>>
<<<dimi          34        heraclion>>>
<<<ji            27        washington>>>
<<<hyseyin       33        izmir>>>
gympie:~/Samples$
```

Entire-Pattern and Numbered-Buffer Substitutions

- & : designates the entire pattern (just matched).
- \(and \): designate a numbered pattern later on identified by its respective number-id such as: \1, \2, \3, etc.

```
     &
s/-----/---&----/
```

```
      \1      \2         \3
s/\(---\)\(-------\)\(--------\)/---\1----\2-----\3---/
```

Examples with Entire/Numbered-Buffers Substitutions

```
gympie:~/Samples$ cat tilefona
Alex Delis                 6973304567
Mike Hatzopoulos           6934400567
Thomas Sfikopulos          6945345098
Stavros Kolliopulos        6911345123
Aggelos Kiagias            6978098765
gympie:~/Samples$
```

```
gympie:~/Samples$ cat tilefona | sed \
 's/\([0-9]\{4\}\)\([0-9]\{2\}\)\([0-9]\{4\}\)/\1-\2-\3/'
```

```
Alex Delis                 6973-30-4567
Mike Hatzopoulos           6934-40-0567
Thomas Sfikopulos          6945-34-5098
Stavros Kolliopulos        6911-34-5123
Aggelos Kiagias            6978-09-8765
gympie:~/Samples$
```

Another Example

```
gympie:~/Samples$ cat pricelist
```

```
**This is the price list**
   of good today
Breakfast        10.03
Lunch            11.45
Dinner            7.56
```

```
gympie:~/Samples$ sed 's/[0-9]/$&/' pricelist
```

```
**This is the price list**
   of good today
Breakfast        $10.03
Lunch            $11.45
Dinner            $7.56
```

```
gympie:~/Samples$ sed 's/[0-9]/$&/3' pricelist
```

```
**This is the price list**
   of good today
Breakfast        10.$03
Lunch            11.$45
Dinner            7.5$6
gympie:~/Samples$
```

Local and *global* substitutions

```
gympie:~/Samples$ cat text2
I had a black dog, a white dog, a yellow dog and
a fine white cat and a pink cat as well as a croc.
These are my animals: dogs, cats and a croc.
```

```
gympie:~/Samples$ cat text2 | sed '1 s/dog/DOG/g'
```

```
I had a black DOG, a white DOG, a yellow DOG and
a fine white cat and a pink cat as well as a croc.
These are my animals: dogs, cats and a croc.
```

```
gympie:~/Samples$ cat text2 | sed '1 s/dog/DOG/'
```

```
I had a black DOG, a white dog, a yellow dog and
a fine white cat and a pink cat as well as a croc.
These are my animals: dogs, cats and a croc.
```

```
gympie:~/Samples$ cat text2 | sed 's/dog/DOG/g'
```

```
I had a black DOG, a white DOG, a yellow DOG and
a fine white cat and a pink cat as well as a croc.
These are my animals: DOGs, cats and a croc.
```

```
gympie:~/Samples$ cat text2 | sed '1,2 s/cat/CAT/2'
```

```
I had a black dog, a white dog, a yellow dog and
a fine white cat and a pink CAT as well as a croc.
These are my animals: dogs, cats and a croc.
gympie:~/Samples$
```

Suppressing the outpur (*-n*) - creating new (*p*/*w*)

```
gympie:~/Samples$ ls -l
total 48
-rw-r--r-- 1 ad ad  328 2010-03-05 11:54 lista
drwxr-xr-x 2 ad ad 4096 2010-03-05 14:21 MyDir1
drwxr-xr-x 2 ad ad 4096 2010-03-05 14:21 MyDir2
-rw-r--r-- 1 ad ad    0 2010-03-04 23:45 out1
-rw-r--r-- 1 ad ad  112 2010-03-05 10:08 pricelist
-rwxr-xr-x 1 ad ad   51 2010-03-03 18:23 script1
-rw-r--r-- 1 ad ad 1603 2010-03-04 23:42 text1
-rw-r--r-- 1 ad ad  146 2010-03-05 13:56 text2
-rw-r--r-- 1 ad ad  165 2010-03-05 09:56 tilefona
```

```
gympie:~/Samples$ ls -l | sed  -n "/^-/s/\([-rwx]*\).*:..\(.*\)/\1\2/p"
```

```
-rw-r--r-- lista
-rw-r--r-- out1
-rw-r--r-- pricelist
-rwxr-xr-x script1
-rw-r--r-- text1
-rw-r--r-- text2
-rw-r--r-- tilefona
gympie:~/Samples$
```

```
gympie:~/Samples$ ls -l | \
                 sed  -n "/^-/s/\(..........\).*:..\(.*\)/\1\2/w 2alex1"
```

Transforming Characters (option *y*)

```
gympie:~/Samples$ more text2
I had a black dog, a white dog, a yellow dog and
a fine white cat and a pink cat as well as a croc.
These are my animals: dogs, cats and a croc.
```

```
gympie:~/Samples$ cat text2 | sed 'y/abcdt/ADCBQ/'
```

```
I hAB A DlACk Bog, A whiQe Bog, A yellow Bog AnB
A fine whiQe CAQ AnB A pink CAQ As well As A CroC.
These Are my AnimAls: Bogs, CAQs AnB A CroC.
gympie:~/Samples$
```

Additional `sed` Input and Output Commands

- Next (`n`): forces `sed` to read the next text line from input file.
- Append Next (`N`): adds the next input line to the current content of the pattern space.
- Print (`p`): copies the current content of the pattern space to the standard output.
- Print First Line (`P`): prints the cotent of the pattern space upto and including a newline character.
- List (`l`): displays “hidden” characters found in the lines of the file.
- Read (`r`): reads from a file
- Write (`w`): writes to a file

The Next Command (n)

```
gympie:~/Samples$ cat sedn
/^[a-z]/{
    n
    /^$/d
    }
```

```
gympie:~/Samples$ cat -n text2
     1  I had a black dog, a white dog, a yellow dog and
     2
     3  a fine white cat and a pink cat as well as a croc.
     4
     5
     6
     7  These are my animals: dogs, cats and a croc.
gympie:~/Samples$ sed -f sedn  text2
I had a black dog, a white dog, a yellow dog and

a fine white cat and a pink cat as well as a croc.


These are my animals: dogs, cats and a croc.
gympie:~/Samples$
```

→`n` forces sed to read the next line from input. Before reading the next line, sed copies the current content of the pattern space to the output, deletes the current text in the pattern space, and then refills it with the next input line. After reading, it applies the script.

Append Next (N) command

```
gympie:~/Samples$ cat text3
11111111
22222222
bbbbbbbb
cccccccv
jhdskjhj
ldjlkjds
lkdjsj44
gympie:~/Samples$
```

```
gympie:~/Samples$ more sedN
    {
    N
    s/\n/  /
    }
gympie:~/Samples$
```

```
gympie:~/Samples$ !sed
sed -f sedN text3
11111111  22222222
bbbbbbbb  cccccccv
jhdskjhj  ldjlkjds
lkdjsj44
```

→ While `n` clears the pattern space before inputting the next line, append (`N`) does not; it adds the next input line to the current content of the pattern space.

A more interesting example with command N

```
gympie:~/Samples$ cat text2
I had a black dog, a white dog, a yellow dog and

a fine white cat and a pink cat as well as a croc.


These are my animals: dogs, cats and a croc.


This is a test
gympie:~/Samples$
```

```
gympie:~/Samples$ cat sednotN
/^$/    {
    $!N
    /^\n$/D
    }
gympie:~/Samples$
```

```
gympie:~/Samples$ sed -f sednotN text2
I had a black dog, a white dog, a yellow dog and

a fine white cat and a pink cat as well as a croc.

These are my animals: dogs, cats and a croc.

This is a test
gympie:~/Samples$
```

Understading the script

- What happens, should you replace `D` with `d`?
  - `$!N` means “if line is not the last line”
  - `$N` means “if line is the last line in the text”
  - `D` command: delete up to the first embedded newline in the pattern space. Start next cycle, but skip reading from the input if there is still data in the pattern space.
  - `d` command: delete pattern space. Start next cycle.

The *p* command

```
gympie:~/Samples$ sed -n '2,3p' text3
```

```
22222222
bbbbbbbb
```

```
gympie:~/Samples$ sed 'p' text3
```

```
11111111
11111111
22222222
22222222
bbbbbbbb
bbbbbbbb
cccccccv
cccccccv
jhdskjhj
jhdskjhj
ldjlkjds
ldjlkjds
lkdjsj44
lkdjsj44
gympie:~/Samples$
```

P command: prints content of the pattern-space upto including a newline char

```
gympie:~/Samples$ cat text4
I had a black dog, a white dog,
a yellow dog and a pink lion
    a fine white cat and
    a pink cat as well as a croc.
These are my animals:
dogs, cats and a croc.
This is a test
gympie:~/Samples$
```

```
gympie:~/Samples$ cat setprintkt
$!N
/\n /P
D
```

```
gympie:~/Samples$ sed -f setprintkt text4
a yellow dog and a pink lion
    a fine white cat and
gympie:~/Samples$
```

A good way to see "invisible" characters

```
gympie:~/Samples$ sed  -n 'l' text4
```

```
I had a black dog, a white dog, $
a yellow dog and a pink lion$
\ta fine white cat and $
\ta pink cat as well as a croc.$
These are my animals: $
dogs, cats and a croc.$
This is a test$
gympie:~/Samples$
```

Reading files in a text with `r`

```
gympie:~/Samples$ cat maintext

This is blah blah blah...
and more blah blah blah blah..
and even more....
blah blah blah...
gympie:~/Samples$ cat mainheader
  THIS IS THE TEXT
gympie:~/Samples$ cat maindate

Sat Mar  6 18:17:14 EET 2010
gympie:~/Samples$
```

```
gympie:~/Samples$ cat sedread
1 r mainheader
$ r maindate
gympie:~/Samples$
```

```
gympie:~/Samples$ sed -f sedread maintext

  THIS IS THE TEXT
This is blah blah blah...
and more blah blah blah blah..
and even more....
blah blah blah...

Sat Mar  6 18:17:14 EET 2010
gympie:~/Samples$
```

Separating lines to different files with `w` command

```
Mon 7:00 Get up!
Tue 7:00 Get up!
Wed 7:00 Get up!
Thu 7:00 Get up!
Fri 7:00 Get up!
Mon 7:30 Get Washed
Tue 7:30 Get Washed
...... etc etc
```

```
gympie:~/Samples$ cat sedwrite
/Mon/w Mon.log
/Tue/w Tue.log
/Wed/w Wed.log
/Thu/w Thu.log
/Fri/w Fri.log
gympie:~/Samples$ sed -nf sedwrite log-events
```

```
gympie:~/Samples$ cat sedwrite
/Mon/w Mon.log
/Tue/w Tue.log
/Wed/w Wed.log
/Thu/w Thu.log
/Fri/w Fri.log
gympie:~/Samples$ ls *log
Fri.log  Mon.log  Thu.log  Tue.log  Wed.log
gympie:~/Samples$
```

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)

- `/simth/`
  ⇒ 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);`
  ⇒ 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`