Awk Essential Training: Lynda.com
Notes by Rosalind Xu 18' Summer 2016

Awk: named after inventors: simple data filtering and manipulation.
Good for: text files divided into lines, with fields on each line; joining, merging, etc...
Versions: awk, nawk, gawk—> the one we get if we type gawk on command line. (gawk was installed by "brew install". Now your awk is automatically gawk, but it wasn't by default)

1. awk '{print $2, $1}' names.txt
    1. all awk programs start with "awk"
    2. the part within single quote is the actual
    3. names.txt is the input file
    4. { } means apply this command to every **line** in input
    5. $2 is the second field in line, $1 is the first field in line
    6. comma: in print: use a field separator
    7. awk 'NR==3' <input>: prints the third record (no need to use the command 'print')
    8. command substitution can be understood in awk. However, all awk arguments have to be enclosed in { }. The arguments inside the first { } will not be interpreted as awk arguments, so please don't do that.
2. awk looks at each line of text, each field separated by a space
    1. $0 refers to the whole line
    2. print $2 "," $1 —> concatenation
    3. print NF —> print Number of Fields
        1. **Caution:** while NR refers to record number, NF refers to number of fields; it is a numeric variable. To call a particular field, use $<field_number>.
    4. awk '/up/{print NF, $0}' name.txt
        1. /up/ : a pattern. The action within {} is only applied to lines with this pattern
        2. other patterns: NF==6, etc.
        3. awk 'NF==6' name.txt: print lines matching this patterN
    5. awk '/up/{print "UP:", NF, $0} /down/{print "DOWN:", NF, $0}' name.txt: multiple actions in awk
3. awk flags:
    1. -f: the following program is to be applied to a file
        1. exp: awk -f swap.sh name.txt
        2. inside swap: only {print $2, $1}: no single quotes; no need to be protected from shell

2. -F: use what as field delimitor
    1. exp: awk -F , '{...}' name.txt
    2. -F t : t == tab
3. -v: specify the value of a variable
    1. exp awk -v hi=Hello '{print $1, hi}' then type in "hello awk", what to expect is "hello, Hello" (use ctrl+x) to end the program
    2. **Caution:** use -v to pass external variables to awk! use -v once for each variable.
4. files: can specify multiple file; awk will process one by one
5. The input and output can be redirected awk '..... ' <input> > <output>

6. Defining record and field:
    1. By default: field: any combinations of spaces and tabs
    2. -F exp: awk -F , '{print $2}' <input>
    3. other field separators: ABC; "a b" (multi-character separators are not supported in MacOSX...) ; "" (empty string) —> **Caution:** blank line as RS, no separator (every character is separated) as FS; " "(space) —> space; '[,!]' —> , or ! (single quotes: protect from shell); or just a blank line as RS: "\n\n+".
    4. specify within an awk program: with special variable FS
        1. exp: awk '{FS="," ; print $2}' <input> semicolon: specify the end of one command
        2. caution: awk separate input into fields and records before calling the commands.
        3. use BEGIN pattern to let the commands execute first: awk 'BEGIN{FS=","} {print $2}' <input>
    5. specify within an awk program record separator: RS
        1. exp: 'BEGIN{RS="!";FS=","} {print $2}' <input>
    6. output field/record separators: OFS ORS
        1. exp awk 'BEGIN{RS="\n"; FS=" "; OFS=","; ORS="!"} {print $2, $1}' names.txt
    7. to specify a range of records: NR==3, NR==8{print}. No quotes. NR==3(or 8) can be any regular expression; they specify the beginning and end.
    8. To loop through the records: specify the range of records at the beginning of { }; all commands in { } will be looped through them automatically. If no specification, all records will be looped through. BEGIN{ } happens before looping starts. The END{ } do not loop through any record; commands are just executed.
        1. If for loops inside a { } appears to be only applicable to the last record, you have used the wrong syntax...the for syntax is: for (initialization; condition; increment) (new line) body, no { }!!

7. More built-in variables:

1. NR: record number exp: use NR==6 as a pattern to extract the sixth records
2. FILENAME: filename of file awk is processing
3. FNR: record number within that file
4. $0: the whole line
5. $#: the #the field in that record
6. $NF: the last field
7. $(NF-1) : next to last field
8. as long as $(a_number), it specifies a field
9. field can also be changed (only in memory; not in input) exp: $2="hi"
10. any field can be specified: $100="hi", even if original NF=5

8. Creating user-defined variables
    1. **Caution:** in awk, the variables within the same awk program are valid across { }, even if it is an index in for-statement, etc. Avoid using repetitive indices!
    2. In awk, it is unnecessary (and wrong) to do put $ in front of every variable. $ is only used to let bash knows that something is a variable; however, when the variable is inside awk, it will be interpreted correctly as a variable, because bash knows awk.
    3. exp. awk '{hello=$1; goodbye=$2; print hello, goodbye}'
    4. all awk inputs are case-sensitive
    5. variable type: inter-converted based on context
        1. exp: awk '{a=1; b=3; print a/b}' —> 0.333333
    6. awk multiplies first then concatenates
    7. parentheses used to specify what happens first

9. all awk variables are global variables awk 'BEGIN{a=1} {print a}'

10. Math operators: +,-,*,/,%,^ (not exactly the same as bash..)
11. increment/decrement operators:
    1. a=3; b=++a;print a,b —> 4 4 (the value of a is also changed..)
12. assignment operators: =, +=, -=, *=, /=, %=, ^=
13. comparison operators: ==, !=, <, <=, >, >= (1 for true, 0 for false...different from bash)
14. string operators: concatenation: space. ~, !~ —> whether string value matches regular expression. for null, however, the most convenient way is to use !="" or ==".
15. Arrays: [ ] (different from bash...)
    1. awk '{a[1]=$1; a[2]=$2; a[3]=$3; print a[1], a[2], a[3]}'
    2. awk only allows 1-dimensional arrays

16. Regular expressions

1. format: / / exp: /abc/; matches “abc”, “mabc”, etc.
    1. watch spaces! no space within / / unless otherwise wanted
2. comparison: ~ and !~ : whether something matches a regular expression
    1. exp awk ‘$4~/up/{print}’ <input>
3. special patterns for regular expressions:
    1. . : matches any single character exp: /a.c/
    2. \: escapes . ; \ ; / exp: /a\.c/ matches “a.c”
    3. ^: matches the beginning of; $: matches the end of
        1. exp: /^abc/ matches “abcd”
        2. exp /abc$/ matches “dabc”
    4. [ ] matches any character in set
        1. exp: /a[xyz]c/ matches “axc”
        2. exp: ranges: /a[a-zA-Z]c/ matches “amc”
        3. exp: not: /a[^a-z]c/ does not match “abc” but matches “a1c”
    5. * : matches 0 or more occurrences of the previous item
        1. exp: /ab*c/ matches “abbbbbc”
    6. + : matches 1 or more occurrences of the previous item
        1. exp: /ab+c/ does not match “ac” but “abc”
    7. ? : matches 0 or 1 occurrence of the previous item; i.e. the previous item is optional
    8. { }: certain number of repeats
        1. exp: /ab{3}c/ matches only “abbbc”
        2. exp: /ab{3,}c/: matches 3 or more repeats of b
        3. exp: /ab{3,4}c/ matches “abbbc” and “abbbbc”
    9. ( ): specify a group to repeat
        1. exp: /(ab)+c/ matches “ababc”
4. **greediness**: regular expression will match as many items as it can
    1. exp: /<.+>/ will match all of “<hi>…..<hi>”
    2. to only get the first "<hi>”: /<[^>]+>/

17. Control Structures: similar to C. Awk is a full-fledged programming language.
18. if statement:
    1. if (condition) { (new line) if-statements (new line) } else{(new line) else-statements (new line)}
    2. condition: == : numeric equality; ~: regular expressions matching
    3. numeric values are considered false if its value is 0; true if its value is something else
    4. string values are considered false if it’s empty string; true otherwise
    5. use semicolon at the end of **each statement** in if-statements, not in conditions
    6. the spaces and line breaks are optional; for awk program, any number of breaks and spaces can be used around any element (except assignment)

7. exp: awk ‘{if (NF < 8) {print “hi”} else {print “good”}}’
8. in an awk file:
    1. {if (NF < 8) {print “hi”} else {print “good”}}
    2. execute the file awk -f 1.awk <input>
9. in a shell file:
    1. exp: awk ‘{if (NF < 8) {print “hi”} else {print “good”}}’
    2. execute: sh 1.sh
    3. or: chmod +x 1.sh then: ./1.sh

19. for statement:
    1. {for (initialization (=, not ==!); condition; increment) {(new line) body}. The { } is optional for single-line statements but always a good idea to use. (Since unlike bash, awk does not have do…done pair to enclose a statement.)
        1. **Caution:** all awk statements need to be enclosed in { }, even for, while, if, etc.!
        2. If you want to put a for loop inside a while loop, be careful! The for loop will finish as long as the initial condition for the while loop is true!
20. BEGIN: at the beginning of the program, do this this and that; END: at the end of the program, do this this and that.
    1. exp: BEGIN{ beginning of file) } { (main body) } END{ (end of file)}
    2. END is **very** important! if something is written in END, it will not be looped through every line. if it is in { } with no patterns ahead, it will be looped through every line of input!
21. printf(): same syntax as C
    1. printf (format, value…)
    2. exp: awk -F ‘{printf("%s\t%s\t%d\n”, $1,$2,$3}
    3. %#s: the field is # length %-#s: left justified
    4. %f: six digits after decimal point by default. %#1.#2f: #1: width (including decimal; #2: #digits after decimal
    5. %d: integer

22. String manipulation:
    1. the first letter is letter 1, not 0
    2. string functions: none of these is going to alter the input file
        1. length([string])
        2. index(string,target): return the **first** find of target in string; indexed by the first letter of target
        3. match(string,regexp==regular expression): looks for the first find like index; returns index; also sets RSTART to index, RLENGTH to length of the match
        4. substr(string, start, [,length]): looks for the substring starting at start with length length. If length is not specified, it’s the substring starting

at start till the end.
5. sub(regexp, newval[, string]): replaces the **first** match of regexp with newval
6. gsub(regexp, newval[, string]):replaces **all** matches of regexp with newval
7. split(string, array[, regexp]): break fields into **subfields** and store in array; default: regexp = FS.

23. Arrays
    1. associative arrays: a[“hi”]=1, a[“good”]=2...etc. To iterate through, use for (index in array) {body}
    2. sort can be used to sort arrays. exp: <output> | sort -rn -k 2
        1. -rn: reverse, numerical
        2. -k: key is field 2 of output
        3. in awk program: format: asort(a, a, "@val_num_desc”). See gnu. The second a is destination.
    3. simulate a multi-dimensional arrays
        1. exp. array[i “,” j] = 2, etc.
    4. delete (see gnu)
    5. **Caution:** watch the indices!!! tempDistance[,2] is not a valid index!

24. Math functions:
    1. int(x): integer part of x
    2. rand( ) 0-1; never 1
    3. srand([x]): seed random
    4. sqrt(x)
    5. sin(x), cos(x)
    6. atan2(y,x): atan2(0,-1) = Pi
    7. log(x)
    8. exp(x)

25. Pipes: pipe the output of a command as the input of another command
    1. can be used to preprocess the input, etc.

26. CSV file: refer to the course

27. Awk scripts: use #!/bin/bash

then type awk -Ft ‘BEGIN {...} NR > 1{#Skip header…….} END { …..} > <output.txt>
use chmod +x <file> to make it executable