

LING 408/508: Computational Techniques for Linguists

Lecture 12

Adminstrivia

- Homework 6 on awk out today

awk

```
sandiway@sandiway-VirtualBox: ~
File Edit View Search Terminal Help
sandiway@sandiway-VirtualBox:~$ which awk
/usr/bin/awk
sandiway@sandiway-VirtualBox:~$ which mawk
/usr/bin/mawk
sandiway@sandiway-VirtualBox:~$
```

NAME

awk - pattern-directed scanning and processing language

SYNOPSIS

```
awk [ -Ffs ] [ -v var=value ] [ -mrn ] [ -mfn ] [ -f prog [ prog ] [ file ... ]
```

DESCRIPTION

`Awk` scans each input `file` for lines that match any of a set of patterns specified literally in `prog` or in one or more files specified as `-f file`. With each pattern there can be an associated action that will be performed when a line of a `file` matches the pattern. Each line is matched against the pattern portion of every pattern-action statement; the associated action is performed for each matched pattern. The file name means the standard input. Any `file` of the form `var=value` is treated as an assignment, not a file name, and is executed at the time it would have been opened if it were a file name. The option `-v` followed by `var=value` is an assignment to be done before `prog` is executed; any number of `-v` options may be present. `-F fs` option defines the input field separator to be the regular expression `fs`.

awk

- Why awk?
 - use the command line for extracting textual data
 - a pattern-matching language
 - can be very fast ...
 - <https://brenocon.com/blog/2009/09/dont-mawk-awk-the-fastest-and-most-elegant-big-data-munging-language/>

When one of these newfangled “[Big Data](#)” sets comes your way, the very first thing you have to do is data munging: shuffling around file formats, renaming fields and the like. Once you’re dealing with hundreds of megabytes of data, even simple operations can take plenty of time.

For one recent ad-hoc task I had — reformatting 1GB of textual feature data into a form Matlab and R can read — I tried writing implementations in several languages, with help from my classmate [Elijah](#). The results really surprised us:

Language	Time (min:sec)	Speed (vs. gawk)	Lines of code	Notes	Type
mawk	1:06	7.8x	3	Mike Brennan's Awk , system default on Ubuntu/Debian Linux.	VM
java	1:20	6.4x	32	version 1.6 (-server didn't matter)	VM+ JIT
c-ish c++	1:35	5.4x	42	g++ 4.0.1 with -O3, using stdio.h	Native
python	2:15	3.8x	20	version 2.5, system default on OSX 10.5	VM
perl	3:00	2.9x	17	version 5.8.8, system default on OSX 10.5	VM
nawk	6:10	1.4x	3	Brian Kernighan's "One True Awk" , system default on OSX, *BSD	?
c++	6:50	1.3x	48	g++ 4.0.1 with -O3, using fstream, stringstream	Native
ruby	7:30	1.1x	22	version 1.8.4, system default on OSX 10.5; also tried 1.9, but was slower	Interpreted
gawk	8:35	1x	3	GNU Awk , system default on RedHat/Fedora Linux	Interpreted Linux

awk

- Powerful pattern-matching is at the heart of awk:
- <https://www.gnu.org/software/gawk/manual/gawk.html>
- https://www.gnu.org/software/gawk/manual/html_node/Regexp.html

A pattern-action statement has the form

```
pattern { action }
```

A missing { action } means print the line; a missing pattern always matches. Pattern-action statements are separated by newlines or semicolons.

awk

- Manpage:

<http://manpages.ubuntu.com/manpages/bionic/en/man1/awk.1plan9.html>

```
An input line is normally made up of fields separated by white space, or by regular
expression FS. The fields are denoted $1, $2, ..., while $0 refers to the entire line.
If FS is null, the input line is split into one field per character.
```

Regular expressions (regex):

<http://manpages.ubuntu.com/manpages/bionic/en/man7/regex.7.html>

awk

- Manpage:

<http://manpages.ubuntu.com/manpages/bionic/en/man1/awk.1plan9.html>

```
if( expression ) statement [ else statement ]
while( expression ) statement
for( expression ; expression ; expression ) statement
for( var in array ) statement
do statement while( expression )
break
continue
{ [ statement ... ] }
expression          # commonly var = expression
print [ expression-list ] [ > expression ]
printf format [ , expression-list ] [ > expression ]
return [ expression ]
next                 # skip remaining patterns on this input line
nextfile            # skip rest of this file, open next, start at top
delete array[ expression ]# delete an array element
delete array       # delete all elements of array
exit [ expression ] # exit immediately; status is expression
```

- a bit like the Bash shell programming language...
- a bit like Perl, actually Perl is a bit like awk

awk

- `awk` is a very useful command
 - it allows you process files line by line and extract matching information
 - Words on a line:
 - `$1` is word #1 in a line
 - `$2` is word #2 in a line (separated from #1 by space(s)) *etc.*
 - Some simple Awk code:
 - `print $3` means print word #3 in a line
 - `vname=0` set variable `vname` to 0 (note: no `$`)
 - (arithmetic expressions ok on the right side of the `=`, e.g. `vname=vname+2`)
 - `if (...) { ... } else { ... }` conditional: e.g. `$1 >= 3`
 - `;` separates statements
 - **Syntax:**
 - `awk 'BEGIN { ..1.. } { ..2.. } END { ..3.. }' data.txt`
 - means execute awk code block `{ ..1.. }` at the beginning
 - then process each line of `data.txt` using awk code block `{ ..2.. }`
 - then at the end execute awk code block `{ ..3.. }`
 - `BEGIN { ..1.. }` is optional
 - `END { ..3.. }` is also optional

man awk for examples

awk

- Manpage:

<http://manpages.ubuntu.com/manpages/bionic/en/man1/awk.1plan9.html>

EXAMPLES

```
length($0) > 72
    Print lines longer than 72 characters.

{ print $2, $1 }
    Print first two fields in opposite order.

BEGIN { FS = ",[ \t]*|[ \t]+" }
    { print $2, $1 }
    Same, with input fields separated by comma and/or blanks and tabs.

    { s += $1 }
END { print "sum is", s, " average is", s/NR }
    Add up first column, print sum and average.

/start/, /stop/
    Print all lines between start/stop pairs.

BEGIN { # Simulate echo(1)
    for (i = 1; i < ARGV; i++) printf "%s ", ARGV[i]
    printf "\n"
    exit }
```

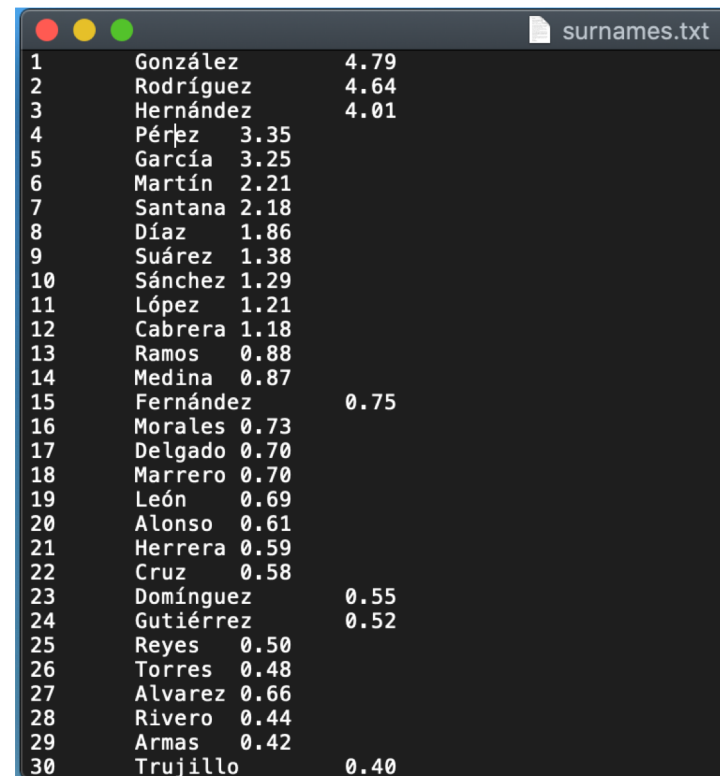
awk

- awk is locale sensitive: Ubuntu 18.04 LTS (and later) supports UTF-8 by default

```
sandiway@sandiway-VirtualBox: ~  
File Edit View Search Terminal Help  
sandiway@sandiway-VirtualBox:~$ which awk  
/usr/bin/awk  
sandiway@sandiway-VirtualBox:~$ which mawk  
/usr/bin/mawk  
sandiway@sandiway-VirtualBox:~$ echo $LANG  
en_US.UTF-8  
sandiway@sandiway-VirtualBox:~$ more /etc/default/locale  
# File generated by update-locale  
LANG="en_US.UTF-8"  
sandiway@sandiway-VirtualBox:~$ █
```

awk

- Example:
 - Top 30 surnames and percentages in the **Canary Islands** according to Wikipedia
 - https://en.wikipedia.org/wiki/List_of_the_most_common_surnames_in_Europe
 - Filename: surnames.txt
 - 3 fields: rank, name, and percentage of population
 - fields separated by [TAB] (ASCII 9)

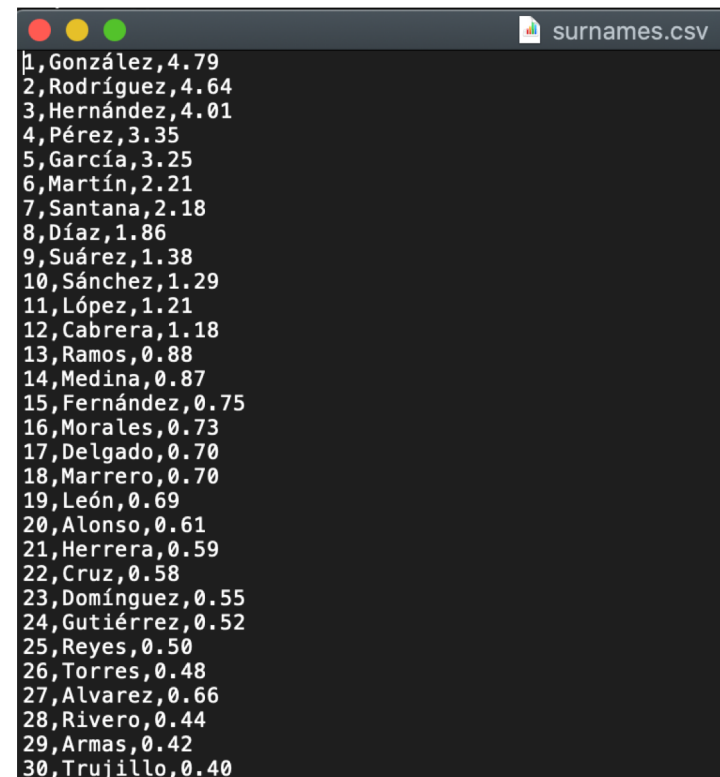


```
surnames.txt
1 González 4.79
2 Rodríguez 4.64
3 Hernández 4.01
4 Pérez 3.35
5 García 3.25
6 Martín 2.21
7 Santana 2.18
8 Díaz 1.86
9 Suárez 1.38
10 Sánchez 1.29
11 López 1.21
12 Cabrera 1.18
13 Ramos 0.88
14 Medina 0.87
15 Fernández 0.75
16 Morales 0.73
17 Delgado 0.70
18 Marrero 0.70
19 León 0.69
20 Alonso 0.61
21 Herrera 0.59
22 Cruz 0.58
23 Domínguez 0.55
24 Gutiérrez 0.52
25 Reyes 0.50
26 Torres 0.48
27 Alvarez 0.66
28 Rivero 0.44
29 Armas 0.42
30 Trujillo 0.40
```

Note accent marks: UTF-8

awk

- Example:
 - Top 30 surnames and percentages in the **Canary Islands** according to Wikipedia
 - https://en.wikipedia.org/wiki/List_of_the_most_common_surnames_in_Europe
 - Filename: surnames.csv
 - 3 fields: rank, name, and percentage of population
 - fields separated by a single comma



```
surnames.csv
1, González, 4.79
2, Rodríguez, 4.64
3, Hernández, 4.01
4, Pérez, 3.35
5, García, 3.25
6, Martín, 2.21
7, Santana, 2.18
8, Díaz, 1.86
9, Suárez, 1.38
10, Sánchez, 1.29
11, López, 1.21
12, Cabrera, 1.18
13, Ramos, 0.88
14, Medina, 0.87
15, Fernández, 0.75
16, Morales, 0.73
17, Delgado, 0.70
18, Marrero, 0.70
19, León, 0.69
20, Alonso, 0.61
21, Herrera, 0.59
22, Cruz, 0.58
23, Domínguez, 0.55
24, Gutiérrez, 0.52
25, Reyes, 0.50
26, Torres, 0.48
27, Alvarez, 0.66
28, Rivero, 0.44
29, Armas, 0.42
30, Trujillo, 0.40
```

awk: FS variable

4.5.1 Whitespace Normally Separates Fields

Fields are normally separated by whitespace sequences (spaces, TABs, and newlines), not by single spaces. Two spaces in a row do not delimit an empty field. The default value of the field separator FS is a string containing a single space, " ". If awk interpreted this value in the usual way, each space character would separate fields, so two spaces in a row would make an empty field between them. The reason this does not happen is that a single space as the value of FS is a special case—it is taken to specify the default manner of delimiting fields.

```
awk 'BEGIN { OFS=""; print "*",FS,"*" }'
```

```
* *
```

- Notes:

- OFS (Output Field Separator):
 - "In the output, the [print] items are normally separated by single spaces."
- awk strings:
 - "A *string constant* consists of a sequence of characters enclosed in double quotation marks."

awk

- Run the following awk one-liner to figure out what the code does:

```
1. awk '{print $2}' surnames.txt
```

awk

- Run the following awk one-liner to figure out what the code does:

```
2. awk '{print $2}' surnames.csv
```

awk

- Run the following awk one-liner to figure out what the code does:

3. `awk 'BEGIN {FS=","} {print $2}' surnames.csv`

4. `awk -F, '{print $2}' surnames.csv`

awk

- Run the following awk one-liner to figure out what the code does:

5. `awk '{if ($3>=1) {print $2}}' surnames.txt`

6. `awk '{if ($3>=1.5) {print $2, $3}}' surnames.txt`

- Note:

- "A *numeric constant* stands for a number. This number can be an integer, a decimal fraction, or a number in scientific (exponential) notation."

Homework 6

- Write awk code to:
 1. print a table of and **calculate the total percentage of population** for the top 10, 20 and 30 surnames
 2. read and print out the table with table headings **aligned** with the field values (use printf)

Rank	Name	Approximate percentage
1	González	4.79
2	Rodríguez	4.64
3	Hernández	4.01
4	Pérez	3.35
5	García	3.25

Homework 6

- for `printf` documentation, read:
https://www.gnu.org/software/gawk/manual/html_node/Printf.html#Printf

5.5 Using `printf` Statements for Fancier Printing

For more precise control over the output format than what is provided by `print`, use `printf`. With `printf` you can specify the width to use for each item, as well as various formatting choices for numbers (such as what output base to use, whether to print an exponent, whether to print a sign, and how many digits to print after the decimal point).

- **Basic Printf:** Syntax of the `printf` statement.
- **Control Letters:** Format-control letters.
- **Format Modifiers:** Format-specification modifiers.
- **Printf Examples:** Several examples.

Homework 6

- Due Monday by midnight
- Usual rules:
 - email to me
 - one PDF file
 - subject: Homework 6 408/508 *Your name*