Advanced UNIX

Scripting, data exploration, text parsing

What you should know

- Basic shell navigation - cd, ls, pwd
- Making file and directories - touch, mkdir
- Moving files - mv, cp
- Basic file handling - head, tail, cat, less
- Some experience in a script editor

Advanced Unix Materials

Log onto the server using isu username and password. Fetch materials from github:

```
$ ssh isuusername@bcbio.gdcb.iastate.edu
$ git clone https://github.com/j23414/adv-unix-workshop.git
```

Slides: **https://github.com/j23414/adv-unix-workshop**

Workshop Outline

1. Creating shell scripts
2. Exploring data with **grep**, **sort**, **uniq** and **wc**
3. **sed** - search/replace and line deletion
4. **awk** - advanced data processing language

Extra: All Unix Commands

```
$ echo $PATH            # prints path to installed Unix commands
$ cd <path there>       # go to folder containing Unix commands
$ ls *                  # list available commands
$ man <command>         # show manual page for command

$ which <command>       # tells which executable is used

$ who                   # who is also logged in?
$ top                   # what commands are currently running?
```

Part One

Creating shell scripts

What are shellscripts

- Anything you type into your terminal, can be pasted into a file and executed

- The code in the shellscript is read line-by-line by the bash interpreter, just like the lines you type into your terminal

hello.sh

Another ‘Hello World’ example…

```
#! /bin/bash

echo ‘Hello world’

touch file.txt                # creates a file
ls -ltr                       # lists files
```

Hashbang (#!)

You need to tell the system what program should interpret your script

Syntax: #! <path to program>

Examples (first line of file):

```
#! /bin/bash
#! /usr/bin/python
```

Comments

Anything following a ‘#’ is ignored by bash

```
# list doc files in the current directory
ls *.doc
```

Comments are notes to human readers

Your turn…

Create hello.sh

```
$ cd                    # go to home directory
$ touch hello.sh        # create a script file
$ vi hello.sh           # open editor
```

```
#! /bin/bash
echo ‘Your message here….’
```

Run the script

```
$ bash hello.sh                    # Method 1


$ ls -l hello.sh
$ chmod 755 hello.sh               # Change to executable
$ ls -l hello.sh
$ ./hello.sh                       # Method 2
```

‘./’ is used to run an executable file

Part Two

grep, sort, uniq, wc

Review:
Piping and Redirection

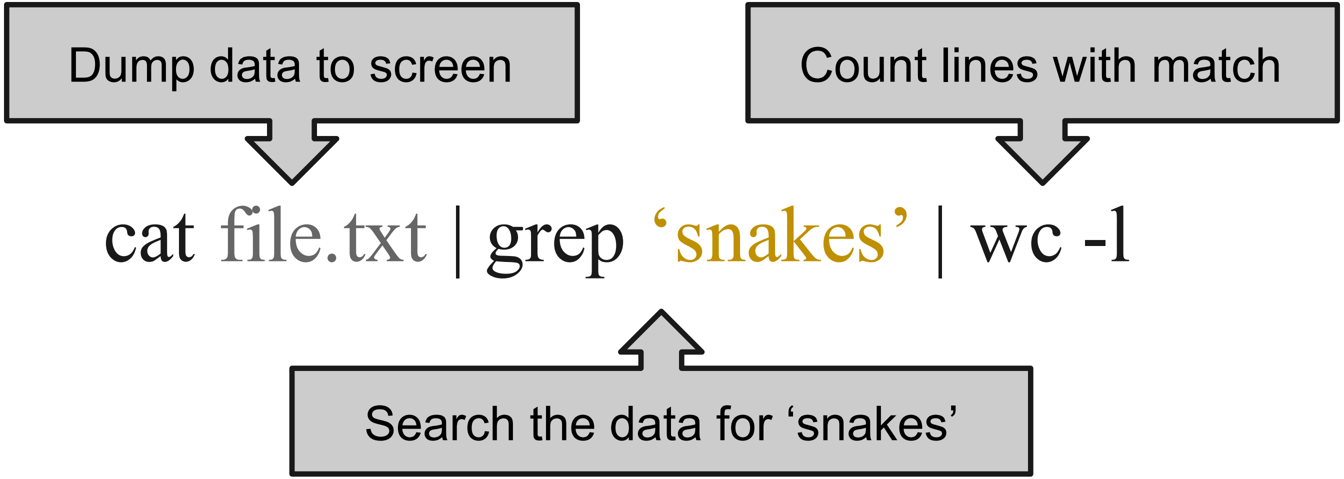
Redirection

Given A and B are programs and f is a file

A | B	Pipe output of A to input of B
A > f	Overwrite f with A’s output
A >> f	Append A’s output to end of f

Pipelines

Many UNIX tools can be linked into piplines



Dump data to screen

Count lines with match

```
cat file.txt | grep ‘snakes’ | wc -l
```

Search the data for ‘snakes’

Warnings

```
# a.txt will be empty after both commands
$ head a.txt > a.txt
$ head a.txt | A | B > a.txt
```

Never open a file for both reading and writing in one pipeline

Four powerful tools

1. wc - count lines, words, or characters
2. grep - search tool
3. sort - flexible sort tool
4. uniq - find unique lines

Sample data and exercises

Move to section-2/ which contains the files:

1. h[12345].txt (5 files)
2. script.sh
3. solutions.sh
4. unsorted.tab

wc

word **c**ount - count lines, words and characters

Options:

-l,	--lines	line count
-w,	--words	word count
-m,	--chars	character count
-L,	--max-line-length	

wc examples

```
# prints count of lines, words, and bytes
$ wc h*.txt
# Word count, like in MS Word
$ man bash | wc -w
# Count files in the working directory
$ ls | wc -l
```

grep

- prints lines matching the search pattern
- for multiple files, tells which files matched
- has lots of very powerful options

Syntax:

```
$ grep [options] <pattern> <files>
$ <in> | grep [options] <pattern>
```

Examples 2.1

```
$ grep ‘primrose’ h*.txt
$ grep ‘not to be’ h*.txt
```

If your shell is not coloring the matches, run the following command:

```
$ alias grep=’grep --color=auto’
```

Exercise 2.1

Practice **wc** and basic **grep**

- Navigate to section-2/
- Make script.sh executable (chmod 755)
- Open file script.sh
- Follow the instructions for Exercise 2.1

some grep options

--help list of options and brief explanations

-c, --count

-v, --invert-match

-i, --ignore-case

-w,--word-regexp

-l, --files-with-match

-o, --only-matching

-E, --extended-regexp

-A, --after-context

-B, --before-context

-C, --context

-h, --no-filename

-L, --files-without-match

Examples 2.3

```
$ grep ‘Fred’ m.tab                # Lines containing ‘Fred’
$ grep -c ‘Fred’ m.tab             # Count ‘Fred’ matches
$ grep -v ‘Fred’ m.tab             # Lines except ‘Fred’
$ grep -i ‘g’ m.tab                # Case insensitive match of ‘g’

$ grep -C1 ‘rose’ h*.tab           # Shows the context, 1 line
$ grep -wC1 ‘rose’ h*.tab          # Only match rose word
```

Two more options

-E, --extended-regexp

These commands require regular expressions to be really useful

Regular Expressions (1)

.	matches any character except a newline
*	matches 0 or more of previous character
+	matches 1 or more of previous character
[xyz]	matches characters x, y and z
[^xyz]	matches characters OTHER than x, y and z
^	anchors match at the BEGINNING of the line
$	anchors match at the END of the line
\	escapes the following special character

Example 2.4

```
$ grep -E ‘[aeiou]’ m.tab         # Match vowels
$ grep -E ‘[^aeiou]’ m.tab        # Match consonants
$ grep -E ‘^[1-5]’ m.tab          # Begins with numbers 1-5
$ grep -E ‘[0-9]*G$’ m.tab        # Ends in number and ‘G’

$ grep -E ‘[a-z]+able’ h*.txt     # Match words that have ‘able’
$ grep -oE ‘[a-z]+able’ h*.txt    # Only print out match
$ grep -E ‘^\[.*\]$’ h*.txt       # Prints stage directions
```

sort – reorder a file

--help list of options and brief explanations

-g,	--general-numeric-sort	
-n,	--numeric-sort	
-r,	--reverse	
-u,	--unique	
-k,	--key=POS	Sort by column

uniq

--help		list of options and brief explanations
-c,	--count	count occurences of each line
-d,	--repeated	print only duplicated lines
-u,	--unique	print only uniq lines

Input must be sorted!

Only compares two adjacent lines!

Example 2.7

```
# The following two are identical
$ sort m.tab | uniq
$ sort -u m.tab

# Try these
$ sort m.tab | uniq -c
$ sort m.tab | uniq -d
$ sort m.tab | uniq -u
$ sort -k2 m.tab
```

Pipeline strategies

```
grep | sort | uniq
grep | sort | uniq | wc
<input> | sort | uniq -c | sort -n
```

Strategy: Build the pipelines up incrementally, checking output at each step

Exercise 2.2

Practice building pipelines

Navigate to section-2/

Follow the instructions for Exercise 2.2

Part Three

Substitution with sed

Sample Data

Move into **section-3/**, find the following:

- m.tab - similar to unsorted.tab in Part 2
- ids.txt - a file of info on imaginary people
- s.fa - a protein sequence file

The power of sed

```
$ sed ‘s/This/That/g’ yourfile.txt  # text replacement
```

- search and replace (with style)
- extract specific patterns from files
- delete specific lines or ranges of lines

sed will not hurt your data

sed reads your data and writes to output.

The output will pour into your terminal unless redirected to a pipe or file.

Your original file is perfectly safe

sed won’t, but YOU can

NEVER REDIRECT TO ORIGIN

--- Pipelines should not be circular ---

The following will destroy z.txt:

```
prog1 z.txt | prog2 > z.txt    # BAD!!!
```

Test drive sed...

```
$ sed ‘ ’ m.tab                     # prints everything
$ sed -n ‘ ’ m.tab                  # prints nothing

$ sed -n ‘/Fred/p’ m.tab            # same as “grep ‘Fred’ m.txt”
$ sed ‘/Fred/p’ m.tab               # duplicate ‘Fred’ lines

$ sed ‘s/Fred/George/’ m.tab        # 1st time text replacement
$ sed ‘s/Fred/George/g’ m.tab       # global text replacement
```

sed syntax

sed **[OPTIONS]** **<command>**

sed **[OPTIONS]** ‘**[LINE_ADDRESS]** **PROCEDURE**’

sed workflow

```
for each line of input
    remove trailing newline character
    if line matches the address
        perform user’s procedure
    if -n option is NOT set
        append newline and print
```

Addresses - by number

1	Matches line number 1
12	Matches line number 12
2,5	Matches lines 2 to 5
5,$	Matches lines 5 and on

```
$ sed -n ‘1p’ m.tab                    # prints 1st line
$ sed -n ’5,$p’ m.tab                  # prints lines 5 and on
```

Addresses - by expression

/ham/	Matches lines with pattern ‘ham’
/a/,/b/	Matches from lines matching a to b
1,/ham/	Matches lines 1 to matching ‘ham’
/ham/,$	Matches from ‘ham’ to the end

```
$ sed -n ‘/ham/p’  m.tab              # prints lines matching ham
$ sed -n ’/start/,/stop/p’ m.tab   # print between two patterns
```

Procedure: deletion (d)

When the line matches the address, sed does not print, rather it moves onto the next line

Examples 3.1: deletion (d)

The address can be a number or a regular expression:

```
$ sed ‘d’ m.tab              # delete everything
$ sed ‘1d’ m.tab             # delete 1st line
$ sed ‘/Fred/d’ m.tab        # delete lines containing ‘Fred’
$ sed ‘5,10d’ m.tab          # delete lines 5 to 10
$ sed ‘10,$d’ m.tab          # delete lines from 10 on
$ sed ‘/R/,/T/d’ m.tab       # delete lines between R and T
$ sed ‘/Fred/,/Duffy/d’ m.tab
```

! operator, invert selection

Addresses can be negated with !

1!	Matches lines NOT equal to 1
2,5!	Matches lines NOT between 2 and 5

```
$ sed -n ‘1!p’ m.tab                  # prints all except 1st line
$ sed -n ’2,5!p’ m.tab                # prints all except lines 5 and on
$ sed -n ’/Bob/!p’ m.tab# prints all except Bob lines
```

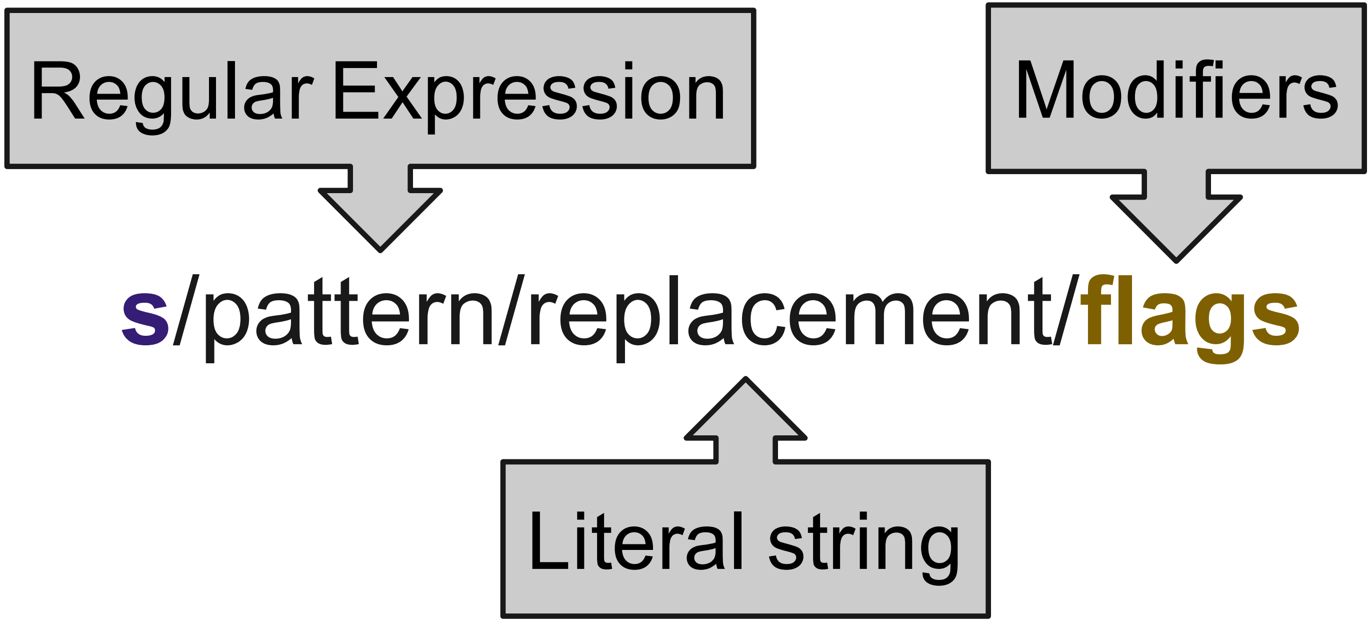
Regular Expressions (1)

.	matches any character except a newline
*	matches 0 or more of the previous char
[...]	matches any of the enclosed
[^...]	matches everything EXCEPT the enclosed
^	anchors match at the BEGINNING of the line
$	anchors match at the END of the line
\	escapes the following special character

Examples 3.2: regex

```
$ sed ‘/[TA]$/d’ m.tab           # Remove if ends in T or A
$ sed ‘/[^TA]$/d’ m.tab          # Remove if not ends in T or A
$ sed ‘/^Scene/d’ h1.txt         # Remove if starts with ‘Scene’
$ sed ‘/^\[/d’ h1.txt            # Remove if starts with ‘[‘
```

substitution (s)



Regular Expression

Modifiers

s/pattern/replacement/**flags**

Literal string

Examples

```
# replace each line’s 1st ‘this’ with ‘that’
cat “this this this” | sed ‘s/this/that/’

# replace EVERY ‘this’ with ‘that’ (global flag)
cat “this this this” | sed ‘s/this/that/g’
```

Examples 3.3

```
$ sed ‘s/Fred/George/’ m.tab
$ sed ‘/Feb/,/Sep/ s/Fred/George/’ m.tab
$ sed ‘s/[1-5]/*/g’ m.tab
$ sed ‘s/\[.*\]//’ h*.txt

$ sed ‘s/ /\n/’ s.fa             # Instead use: tr ‘ ‘ ‘\n’ s.fa
$ sed ‘s/\|/\t/’ s.fa            # Instead use: tr ‘|‘ ‘\t’ s.fa
```

Exercise 1.1 (ids.txt)

1. cat ids.txt, check format (anything weird?)
2. delete ONLY those who are absent
3. delete ONLY those who are present
4. delete all entries after Mark
5. delete entries with an ‘*’ after the name

Optional Challenge

Pull out gi, ref number, description and species from a protein sequence.

More advanced sed using a combination of pattern matching and substitution.

Extended Expressions (2)

(...) captures the enclosed sequence

\n recalls *n*th captured sequence

+ matches 1 or more of the previous characters

| OR

All of these require the **-r** argument (**-E** on mac)

Exercise 1.2

s.fa is formatted as so:

```
>gi|<gi>|ref|<ref>| <description> [<species>]
<sequence line 1>
...
<sequence line N>
```

1. Extract the 4 header regions individually
2. Write the gi and ref to a comma-delimited file

Print only if substituted

Problem:

```
$ sed -r ‘s/^>gi\|([0-9]+).*/\1/’ s.fa
```

You want a list of integers, but all the lines in the input still print

Solution:

```
$ sed -rn ‘s/^>gi\|([0-9]+).*/\1/p’ s.fa
```

Extraction strategy

To one of more words from a line:

- Start with the term to be extracted

```
sed -rn ‘s/.*([0-9]+).*/\1/p’
```

- Make pattern unambiguous by adding context

```
sed -r ‘s/.*id=([0-9]+).*/\1/’
```

- If no context is necessary, just use `grep -o`

```
grep -oE ‘[0-9]+’
```

Part Four

AWK: columnar data

What is AWK?

- AWK is a full programing language
  - variables and arrays (like Perl hashes)
  - loops and conditional statements
  - math, string processing, and user defined functions
- Sed-like addressing and regular expressions
- Automatically splits lines into words

Terms

- **record** - usually a line of input
- **field** - records are split into fields
- **command** - a **condition/procedure** pair
- **condition** - a logical test
- **procedure** - code block that is run
  if the condition is TRUE

AWK Pseudocode

```
BEGIN { do initial stuff }
for each record in input
    split record into fields
    for each command
        if condition is TRUE
            do procedure
END { do final stuff }
```

Outline

- Condition statements
    - condition only calls
    - fields and conditional logic
    - field separator
- Procedure statements
    - print
    - mathematical operators

1. Condition Statements

Condition only calls

AWK Rule 1: If the *command* consists only of a *condition*, the *procedure* defaults to print *record*.

Sample Data

Navigate to **section-4**

- diamonds.tab - Borrowed from Hadley
- d.tab - 25 lines of diamonds.tab

Examples 4.1

```
awk ‘/Ideal/’  d.tab            # sed -n ‘/Ideal/p’ d.tab
awk ‘/Fair/,/Ideal/’  a.tab
awk ‘1,5’ a.tab                 # fyi doesn’t work
awk ‘/[GH]/’ a.tab
```

AWK Fields

AWK breaks lines into fields
By default, fields are separated by whitespaces, e.g

```
0.7␣Fair␣G␣VS1␣56.2
$1  $2   $3 $4  $5
```

A field can be accessed by prefixing ‘$’ to the field number, e.g. $2 is ‘Fair’, $3 is ‘G’

```
awk ‘$3 == “G”’ d.tab          # print if 3rd field is G
```

Comparison Operators (1)

~	Regular expression match
!~	Regular expression non-match
==	Equals (don’t use ‘=’)
!=	Not equals
<	Less than
>	Greater than
>=	Greater than or equal to
<=	Less than or equal to
/a/,/b/	TRUE between matches (like in sed)

Examples 4.2

Now we can test against a single column

```
$ awk ‘$2 == “Ideal”’ d.tab
$ awk ‘$2 != “Ideal”’ d.tab
$ awk ‘$3 ~ /[GFI]/’ d.tab     # reg exp
$ awk ‘$5 > 60’ d.tab
```

Logical Operators

|| Logical OR
&& Logical AND
! Logical NOT

These are used to string conditions together

(**<condition1> || <condition2>**) **&& ! <condition3>**

Conditional examples

```
$ awk ‘$1 > 1 && $7 < 5000’ d.tab
$ awk ‘$2 == “Premium” || $3 == “E”’ d.tab
$ awk ‘!/^#/ && ($1 > 1 || $2 == “Premium”)’ d.tab
```

Try a few other combinations

You can also use the full dataset, diamonds.tab

Resetting Field Separator

You may reset the separator with option (**-F**)

```
# set field separator to comma
$ awk -F’,’ ‘$2==“Ideal”’ d.csv
```

Warning about quotes

```
awk “$1 > 1” d.tab   # WRONG
```

Here AWK gets the ***shell*** *variable* $1 instead of a literal string ‘$1’

This shell variable, will usually be undefined

Procedures

Syntax

condition { procedure }

When condition is TRUE, do procedure
(implicit IF statements)

```
$2 == “Fred” { print $3 }
```

print command

awk ‘{print $2, $1}’

Prints 2nd and 1st fields

Commas are special, they are field separators

Procedures can be used alone

‘{‘ and ‘}’ are **NOT** optional

Comparison to sed

Problem: Print 2nd and 1st fields of input

```
# solution in awk
$ awk ‘{print $2, $1}’

# solution in sed
$ sed -r ‘s/([^ ]+) ([^ ]+).*/\2 \1/’
```

Mathematical Operators

AWK will interpret variables as numbers if you perform mathematical operations on them.

+ - * /	normal plus, minus, times, div
^ **	exponentiation
%	modulo operator

Math examples

```
$ echo ‘1.1␣4’ | awk ‘print $1, $2, $1 + $2’
1.1␣4␣5.1
$ echo ‘2␣8’ | awk ‘print $1 ** $2’
128
$ echo ‘1␣2␣5’ | awk ‘print ($1 + $2) ** $3’
243
```

String concatenation

- Adjacent strings are concatenated
- Spaces are ignored
- Mathematical operations have precedence over string concatenation

```
$ echo “1 5” | awk ‘{print $1 “+” $2 “=” $1 + $2}
1+5=6
```

AWK as a language

```
pi = 4 * atan2(1,1)
# Box-Muller transform: produces two normal random variables
function rnorm(pi, a, b){
            r1 = rand(); r2 = rand() # all variables are global
      a = sqrt(-2 * log(r1)) * cos(2 * pi * r2)
            b = sqrt(-2 * log(r1)) * sin(2 * pi * r2)
            return  # return takes no arguments
}
{rnorm(pi, a, b); print a “\n” b}
```

Exercise

Follow the instructions in script.sh

Supplementary
(extra if there is time)

AWK builtin variables (1)

AWK has several special, builtin variables

NR - current line number

Conditional examples (2)

```
# print the 5th line
$ awk ‘NR == 5’ a.tab
# like `head -5` or `sed 1,5`
$ awk ‘NR == 1, NR == 5’ a.tab
```

```
# fastq to fasta converter
$ awk 'NR % 4 ~ /[12]/' a.fq | tr ‘@’ ‘>’
```

AWK Variables

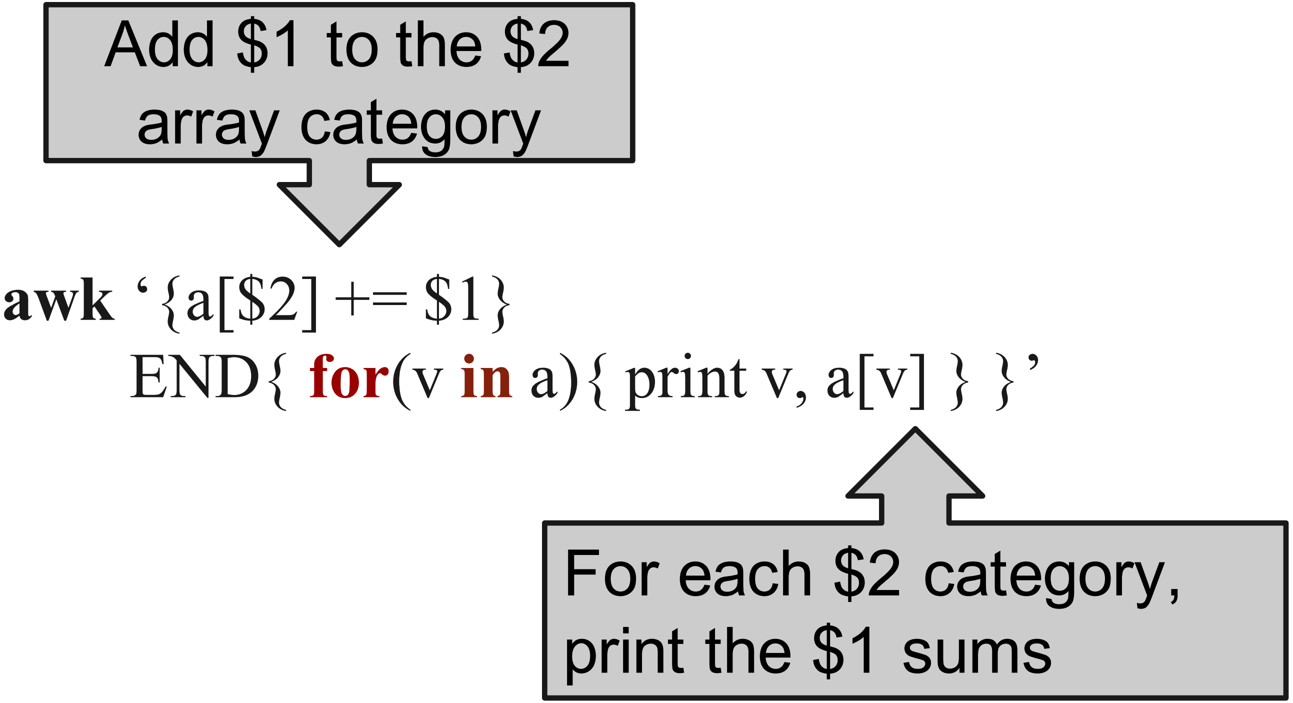
On each line, add $1 to **x**

```
awk ‘{x = x + $1} END {print x}’
```

Prints the sum of column 1

At the end, print **x**

AWK Arrays



Add $1 to the $2 array category

```
awk ‘{a[$2] += $1}
        END{ for(v in a){ print v, a[v] } }’
```

For each $2 category, print the $1 sums

Practice

Write an awk command to sum a column

Write a command to sum $7 across $2 in *a.tab*