awk-tober tutorial and tips and tricks

With applications on genomics files

Oct 18th 2024, Rob Bierman, Research Software Engineer

Goals for this presentation

- We'll start with an awk amuse-bouche
- History of the awk language
- Learn how awk works with a "quota" dataset
- awk as an inspiration for perl, python, and more
- The power of awk on .bed and .gtf files
- Using awk's associative arrays
- Two nice awk "tricks" for multiple files
- When to probably not use awk

Let's see some awk

Here's a data/quotas.tsv file included in this github repo

```
projects  scratch   User
 136.800    0.000   tcomi
  62.000   10.100   rb3242
   0.000   42.600   limingli
...
```

If we want to print just the User's column (the 3rd), we can use awk!

```
awk '{ print $3 }' data/quotas.tsv
```

Output:

```
User
tcomi
rb3242
limingli
...
```

(In this case, we could have also used `cut -f3 data/quotas.tsv`)

Comparable python code for `awk '{ print $3 }' data/quotas.tsv`

We saw how to print the 3rd column of the `data/quotas.tsv` file in `awk`

```
projects  scratch   User
 136.800    0.000   tcomi
  62.000   10.100   rb3242
   0.000   42.600   limingli
...
```

```
awk '{ print $3 }' data/quotas.tsv
```

Here is how we might accomplish the same thing with python:

```
with open("data/quotas.tsv") as f_in:
    for line in f_in:
        split_line = line.split("\t")
        print(split_line[2]) #indexed at 0
```

Everything that awk can do can be done in a more readable way in a "real" programming language.

There's a joke that awk is "write-only" because it can be so hard to read (there's a similar perl joke...)

What is awk?

- awk is a command line text-processing programming language
- awk is pre-installed on basically ALL unix-like OS's
  - linux, mac, windows subsystem for linux
  - awk is part of GNU coreutils
  - when you have ls you'll also have awk [citation-needed]
- awk is really great when working with tabular text like CSV or TSV
- awk is great for quick-and-dirty scripts
- awk is old, but not as old as sed
  - First appeared in 1977 (sed 1973)
  - Developed at AT&T Bell Laboratories by
    - Alfred Aho
    - Peter Weinberger
    - Brian Kernighan
- Brian Kernighan is a professor at Princeton!
  - https://www.cs.princeton.edu/~bwk/

Understanding the structure of an awk program: Column re-order

Let's return to the data/quotas.tsv dataset

```
projects  scratch   User
 136.800    0.000   tcomi
  62.000   10.100   rb3242
   0.000   42.600   limingli
...
```

and this time we want to create a new file where the order of the columns have User first. Does anyone have any suggestions? cut -f3,1,2 actually doesn't work.

```
awk '{ print $3,$1,$2 }' data/quotas.tsv
```

Output

```
User     projects   scratch
tcomi     136.800     0.000
rb3242     62.000    10.100
limingli    0.000    42.600
...
```

Understanding the structure of an awk program: Column re-order

Let's return to the data/quotas.tsv dataset

```
projects  scratch   User
 136.800    0.000   tcomi
  62.000   10.100   rb3242
   0.000   42.600   limingli
...
```

and this time we want to create a new file where the order of the columns have User first. Does anyone have any suggestions?

```
awk '{ print $3,$1,$2 }' data/quotas.tsv
```

Program structure

```
awk '{ COMMANDS }' INPUT
```

The COMMANDS get implicitly run on every line of the file, you don't have to write a for-loop! This helps keep awk very terse.

But what if we don't want to run on every line in the file?

Understanding the structure of an awk program: No header

Let's say we still want to re-order the columns, but we don't want the header row anymore. We'll start with:

```
projects  scratch   User
 136.800    0.000   tcomi
  62.000   10.100   rb3242
   0.000   42.600   limingli
...
```

and generate:

```
tcomi     136.800     0.000
rb3242     62.000    10.100
```

```
limingli    0.000    42.600
...
```

we could use a combination of `tail -n+2` to skip the first line and then pipe that to our previous `awk` command

```
tail -n+2 data/quotas.tsv | awk '{ print $3,$1,$2 }'
```

This shows that `awk` can be used in a pipe, which is nice, but `awk` can handle all of this itself!

Understanding the structure of an `awk` program: No header

Let's say we still want to re-order the columns, but we don't want the header row anymore. We'll start with:

```
projects  scratch   User
 136.800    0.000   tcomi
  62.000   10.100   rb3242
   0.000   42.600   limingli
...
```

and generate:

```
tcomi     136.800     0.000
rb3242     62.000    10.100
limingli    0.000    42.600
...
```

We can tell `awk` to skip the first line with the built in `NR` variable:

```
awk 'NR > 1 { print $3,$1,$2 }' data/quotas.tsv
```

NR is one of a handful of special variables! It keeps track of the current line number!

What if we wanted to calculate the sum quota of a user on scratch and projects?

Understanding the structure of an awk program: Sums

We'll start with the same file:

```
projects  scratch   User
 136.800    0.000   tcomi
  62.000   10.100   rb3242
   0.000   42.600   limingli
...
```

but now we'll make:

```
tcomi    136.8
rb3242    72.1
limingli  42.6
...
```

We can just add the 1st and 2nd columns!

```
awk 'NR > 1 { print $3,$1+$2 }' data/quotas.tsv
```

But what if we wanted a custom header?

Understanding the structure of an awk program: Custom sums

We'll start with the same file:

```
projects  scratch   User
 136.800    0.000   tcomi
  62.000   10.100   rb3242
   0.000   42.600   limingli
...
```

but now we'll make:

```
USER     SUM
tcomi    136.8
rb3242    72.1
limingli  42.6
...
```

We can specify a special instruction for just the first row

```
awk 'NR == 1 { print "USER SUM"} NR > 1 { print $3,$1+$2 }' data/quotas.tsv
```

We've already done a lot with awk, let's have a review.

Understanding the structure of an awk program: Review

- An awk program operates on one line at a time and takes the structure:

```
awk 'CONDITION { COMMAND } CONDITION { COMMAND } ... ' input.txt
```

- awk provides $1, $2, etc for working with columns
  - awk splits on whitespace by default, or specify with -F

- awk is happy to do numerical calculations for us!
  - It will try to perform implicit conversions

- No need to import libraries, awk is always installed!

Next we're going to see how awk inspired other famous tools!

awk as an inspiration

- perl (1987) was inspired from awk (1977) as well as other languages

> Perl borrows features from other programming languages
> including C, sh, AWK, and sed.[1]

> Perl takes hashes ("associative arrays") from AWK and
> regular expressions from sed.

- and since python (1991) was inspired by perl (1987)

> Many professional programmers are turning to Python,
> often as an alternative to Perl, or other scripting languages.

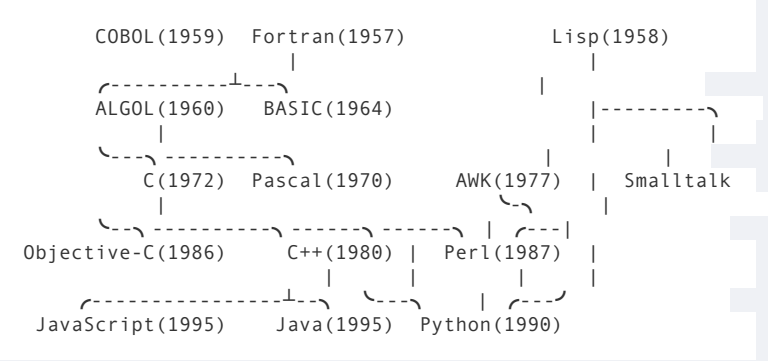
> Like Perl, Python is excellent for scripting,
> and string manipulation, yet its syntax is much less cryptic. [2]

Here's an example of a perl script that renames files from sample.1 to sample.001 etc

```perl
$old = $ARGV[0];
$new = $ARGV[1];
$start = $ARGV[2];
$stop = $ARGV[3];

for ($i=$start; $i <= $stop; $i++) {

    $num = $i;
    if($i<10) { $num = "00$i"; }
    elsif($i<100) { $num = "0$i"; }

    $cmd = "mv $old.$num $new.$num";
    print $cmd."\n";
    if(system($cmd)) { print "rename failed\n"; }
```

2. https://www.cs.ubc.ca/wccce/Program03/papers/Toby.html

Programming language family tree



(Stolen from https://tecky.io/en/blog/evolution-of-programming-languages/)

Let's do some more awk! This time on common genomics files!

Calculating total number of basepairs covered in a .bed file

Here's a data/tiny.bed file (with a header) included in this github repo

```
#chr start end
chr1 100 120
chr1 140 160
chr2 560 580
```

awk script:

```
awk '{ s+=$3-$2 } END { print s }' data/tiny.bed
```

result: 60

For pandas users, you could do the same thing with

```
import pandas as pd
df = pd.read_csv("data/tiny.bed", sep=" ")
(df['end']-df['start']).sum()
```

For R-tidyverse users, you could do the same thing with

```
library(tidyverse)
dat = read_csv("data/tiny.bed")
```

```
sum(dat['end']-dat['start'])
```

Understanding the running sum of .bed region spans

How is this script is working?

```
#chr start end
chr1 100 120
chr1 140 160
chr2 560 580
```

awk script:

```
awk '{ s+=$3-$2 } END { print s }' data/tiny.bed
```

We can make and use variables in awk! It's a programming language!

We create a variable s which defaults to 0 and then we add the value of the span, end minus start, or $3-$2.

At END of the program, we print out the value of s.

What about the header line?

awk couldn't convert start and end to numeric, so defaulted to 0! Quirky!

There's another file called data/tiny_bad_header.bed which will produce the same output!

```
chr1 100 120
chr1 140 160
#chr start end
```

```
chr2 560 580
```

awk arrays on Vernot 2016 S* calls

For our next awk example we'll use a file of S* called introgression regions from Vernot et al 2016.

I've made a copy of this file on della here:

```
/projects/AKEY/akey_vol2/rbierman/callset.bed
```

this is again a .bed file with chrom, start, end, but also with individual, category, and ancient-source:

```
1 1880284 1891263 1.NA18992.2 NA18992 EAS neand
1 2250695 2299059 1.HG01851.2 HG01851 EAS neand
1 2250695 2300081 1.HG00701.1 HG00701 EAS neand
1 2250695 2300081 1.HG01867.2 HG01867 EAS neand
1 2250695 2300081 1.HG01874.1 HG01874 EAS neand
1 2250695 2300081 1.NA18617.2 NA18617 EAS neand
...
1 1904910 1919381 1.UV500.1   UV500   PNG den
1 1927030 1959261 1.UV919.2   UV919   PNG den
1 1927030 1959261 1.UV929.2   UV929   PNG den
```

How many times does neand, den, or anything else come up as the last column?

Honestly, I'd normally do this with cut, sort, and uniq -c:

```
cut -f7 callset.bed | sort | uniq -c
```

Counting occurrences of neand, den, etc with awk

Piped approach:

```
cut -f7 callset.bed | sort | uniq -c
```

Dataset:

```
1 1880284 1891263 1.NA18992.2 NA18992 EAS neand
1 2250695 2299059 1.HG01851.2 HG01851 EAS neand
1 2250695 2300081 1.HG00701.1 HG00701 EAS neand
1 2250695 2300081 1.HG01867.2 HG01867 EAS neand
1 2250695 2300081 1.HG01874.1 HG01874 EAS neand
1 2250695 2300081 1.NA18617.2 NA18617 EAS neand
...
1 1904910 1919381 1.UV500.1   UV500   PNG den
1 1927030 1959261 1.UV919.2   UV919   PNG den
1 1927030 1959261 1.UV929.2   UV929   PNG den
```

Let's make an awk solution using arrays

```
awk '{ a[$7]+=1 } END { for(k in a){ print k,a[k] } }' callset.bed
```

We've created an array called a, but we could have used any name. Then for each line we increment the value of the a array using column 7 as the key.

Finally, at the end, we loop through the keys of a and print out the info.

How about calculating the per-person bed coverage of den?

Starting with:

```
1 1880284 1891263 1.NA18992.2 NA18992 EAS neand
1 2250695 2299059 1.HG01851.2 HG01851 EAS neand
...
1 1904910 1919381 1.UV500.1   UV500   PNG den
1 1927030 1959261 1.UV919.2   UV919   PNG den
1 1927030 1959261 1.UV929.2   UV929   PNG den
```

We want:

```
PERSONCODE1 SUM_DEN_SEQ_RANGES
PERSONCODE2 SUM_DEN_SEQ_RANGES
PERSONCODE3 SUM_DEN_SEQ_RANGES
...
```

We can make an array a keyed by person ($5) which stores a running total of the sequence span ($3-$2) ONLY for rows where the 7th column is den.

```
awk '$7=="den" {a[$5]+=$3-$2} END {for(k in a){print k,a[k]}}' callset.bed
```

with pandas, if you wanted an ugly one-liner (columns start from 0):

```
df = pd.read_table("callset.bed", header=None)
df[df[6].eq('den')].groupby(4).apply(lambda g: (g[2]-g[1]).sum())
```

awk trick to split one file into many

Starting with the same callset.bed:

```
1 1880284 1891263 1.NA18992.2 NA18992 EAS neand
1 2250695 2299059 1.HG01851.2 HG01851 EAS neand
...
1 1904910 1919381 1.UV500.1   UV500   PNG den
1 1927030 1959261 1.UV919.2   UV919   PNG den
1 1927030 1959261 1.UV929.2   UV929   PNG den
```

Let's say we wanted to put the neand, den, etc lines into separate files like neand_lines.bed, den_lines.bed?

with awk you can print to files, not just stdout!

```
awk '{ print > $7"_lines.bed" }' callset.bed
```

this is printing the entire current line to a file we are creating with a name that depends on the $7 column: $7"_lines.bed" (this is str concat).

Only one more example!

awk trick to concatenate files, keeping only first header

We have the GDP in millions of dollars for the 50 states split across 5 different files data/f1.tsv, data/f2.tsv, ... , data/f5.tsv

Here's what each file looks like, but with different states:

data/f1.tsv

data/f2.tsv

State	GDP_2022
Alabama	281,569
Alaska	65,699
Arizona	475,654
Arkansas	165,989
California	3,641,643
Colorado	491,289
Connecticut	319,345
Delaware	90,208

State	GDP_2022
Florida	1,439,065
Georgia	767,378
Hawaii	101,083
Idaho	110,871
Illinois	1,025,667
Indiana	470,324
Iowa	238,342
Kansas	209,326

If we wanted to create a single file, we could try `cat data/f*.tsv > all.tsv`, but then we'd get the header multiple times, and internally.

Instead we can use an awk trick

```
awk 'FNR > 1 || NR == FNR { print }' data/f*.tsv > all.tsv
```

This relies on awk's NR and FNR variables.

- NR : row number that DOESN'T start over between files
- FNR: row number that starts over between files
- so NR == FNR is true for all rows of just the first file

When not to use awk

- awk is best used for relatively simple scripts
- If your awk script starts getting to be large and ugly then maybe it's time to switch to R or python
- awk is great as "glue" between command line tools or scripts and can feel a bit "unprofessional"

Having said that, even Heng Li suggests using awk to process hifiasm output!

Here's an entry in the hifiasm FAQ

How do I get contigs in FASTA?

The FASTA file can be produced from GFA as follows:

```
awk '/^S/{print ">"$2;print $3}' test.p_ctg.gfa > test.p_ctg.fa
```

What is this command doing?

Here's a few rows of a .gfa file:

```
S  ptg000001l  TCCTGGTGAGGC...  ...
A  ptg000001l  0                ...
A  ptg000001l  271              ...
A  ptg000001l  1642             ...
...
```

Heng Li's use of awk in hifiasm

How do I get contigs in FASTA?

The FASTA file can be produced from GFA as follows:

```
awk '/^S/{print ">"$2;print $3}' test.p_ctg.gfa > test.p_ctg.fa
```

```
S  ptg000001l  TCCTGGTGAGGC...  ...
A  ptg000001l  0                ...
A  ptg000001l  271              ...
A  ptg000001l  1642             ...
...
```

Output would be:

```
>ptg000001l
TCCTGGTGAGGC...
```

The command operates only on lines that start with `S` (`/^S/`).

Then it prints the 2nd column, newline, then the 3rd column.

Summary and further resources

Here's what we spoke about today:

- We started with an awk amuse-bouche
- History of the awk language
- Learn how awk works with a "quota" dataset
- awk as the inspiration for perl, python, and more
- The power of awk on .bed and .gtf files
- Using awk's associative arrays
- Two nice awk "tricks" for multiple files

When to probably not use awk

If awk is something you want to commit to learning more of, then I'd suggest working with chatGPT.

It's really good at writing and explaining awk!