Six Essential awk Lessons for Computational Biologists

Ming Tommy Tang

4/12/2023

What is awk?

Text processing and data extraction are done using the scripting language and command known as AWK. Its name derives from the initials of its creators, Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan, who worked on it at AT&T Bell Laboratories in 1977. The main function of AWK is to look for text that matches a pattern in files and then conduct an action on that text. AWK is frequently used for text replacement, data formatting and printing, and string and integer manipulation.

The following are the three variations of AWK:

- AWK is the original AWK.
- NAWK is the new AWK.
- GAWK is GNU AWK. All Linux distributions come with GAWK. This is fully compatible with AWK and NAWK.

Lesson 1 Awk Command Syntax

```
awk -Fs '/pattern/ {action}' input-file
# or
awk -Fs '{action}' input-file
```

- `-F` is the field separator. It will use a space as the field delimiter if you don't specify.
- The `/pattern/` and the `{action}` should be enclosed inside single quotes.
- `/pattern/` is optional. If you don't provide it, awk will process all the records from the input file. If you specify a pattern, it will process only those records from the input-file that match the given pattern.
- `{action}` - These are the awk commands, which can be one or multiple. The whole action block, including all the awk commands together, should be closed between { and }

```
cat test.bed
```

```
## chr1 100 200
## chr1 300 500
## chr2 240 440
## chr2 400 600
## chr3 0   150
```

Print out the first column if the line matches `chr1`

```
awk -F '\t' '/chr1/ {print $1}' test.bed
```

```
## chr1
## chr1
```

Print out the second column if the line matches `chr2`

```
awk -F '\t' '/chr2/ {print $2}' test.bed
```

```
## 240
## 400
```

By default, `awk` use space/tab as the separator, so you can omit the -F `'\t'`:

```
awk '/chr2/ {print $2}' test.bed
```

```
## 240
## 400
```

$1 means the first column, $2 means the second column, etc etc.

Lesson 2 Awk Program Structure (BEGIN, body, END block)

An awk program has following three blocks: BEGIN, body and END.

```
awk 'BEGIN { awk-commands } \  #BEGIN
/pattern/ {action} \  #body
END { awk-commands }' input-file  #END
```

You can use \ to split a unix command line to multiple lines.

The begin block executes only once at the beginning, before awk starts executing the body block for all the lines in the input file.

- The begin block is a good place to print report headers, and initialize variables.
- You can have one or more awk commands in the begin block.
- The keyword BEGIN should be specified in upper case.
- Begin block is optional.

The body block gets executed once for every line in the input file.

- If the input file has 10 records, the commands in the body block will be executed 10 times (once for each record in the input file).
- There is no keyword for the body block. I discussed pattern and action previously.

The end block gets executed only once at the end, after awk completes executing the body block for all the lines in the input-file.

- The end block is a good place to print a report footer and do any clean-up activities.
- You can have one or more awk commands in the end block.
- The keyword END should be specified in upper case.
- End block is optional.

Let's see an example

```
awk 'BEGIN {FS='\t'; print "----header ----"} \
/chr2/ {print $0} \
END {print "----end------"}' test.bed
```

```
## ----header ----
## chr2 240 440
## chr2 400 600
## ----end------
```

$0 means the whole line.

The BEGIN and END block are optional

```
awk '/chr2/ {print $0} \
END {print "----end------"}' test.bed
```

```
## chr2 240 440
## chr2 400 600
## ----end------
```

```
awk 'BEGIN {FS='\t'; print "----header ----"} \
/chr2/ {print $0}' test.bed
```

```
## ----header ----
## chr2 240 440
## chr2 400 600
```

Lesson 3 AWK built-in variables

- FS - Input Field Separator

There are two ways to specify the field separator:

```
awk -F '\t' '{print $2, $3}' test.bed
```

```
## 100 200
## 300 500
## 240 440
## 400 600
## 0 150
```

```
awk 'BEGIN {FS="\t"} {print $2, $3}' test.bed
```

```
## 100 200
## 300 500
## 240 440
## 400 600
## 0 150
```

note that the default field separator is not just a single space. It actually matches one or more whitespace characters.

- OFS - Output Field Separator

You can reformat the output by:

```
awk -F '\t' '{print $1, ":" $2, ":", $3}' test.bed
```

```
## chr1 :100 : 200
## chr1 :300 : 500
## chr2 :240 : 440
## chr2 :400 : 600
## chr3 :0 : 150
```

```
awk -F '\t' '{print $2, "," $1, ",", $3}' test.bed
```

```
## 100 ,chr1 , 200
## 300 ,chr1 , 500
## 240 ,chr2 , 440
## 400 ,chr2 , 600
## 0 ,chr3 , 150
```

The output still has a tab between them. Now, use OFS

```
awk -F '\t' 'BEGIN { OFS=":" } \
{ print $1,  $2, $3 }' test.bed
```

```
## chr1:100:200
## chr1:300:500
## chr2:240:440
## chr2:400:600
## chr3:0:150
```

Now, the output is separated by ":".

note the subtle difference between including a comma vs not including a comma in the print statement (when printing multiple variables). When you specify a comma in the print statement between different print values, awk will use the OFS.

```
awk '{print $1, $2, $3 }' test.bed
```

```
## chr1 100 200
## chr1 300 500
## chr2 240 440
## chr2 400 600
## chr3 0 150
```

```
awk '{print $1 $2 $3 }' test.bed
```

```
## chr1100200
## chr1300500
## chr2240440
## chr2400600
## chr30150
```

Everything squeezes together.

```
awk '{print $1, "test", $2, "test", $3 }' test.bed
```

```
## chr1 test 100 test 200
## chr1 test 300 test 500
## chr2 test 240 test 440
## chr2 test 400 test 600
## chr3 test 0 test 150
```

- RS - Record Separator

```
cat test.bed | tr "\n" ":"
```

```
## chr1 100 200:chr1    300 500:chr2    240 440:chr2    400 600:chr3    0   150:
```

Let's specify : as the record separator

```
cat test.bed | tr "\n" ":" | \
awk 'BEGIN {RS=":" } {print $1,$2}'
```

```
## chr1 100
## chr1 300
## chr2 240
## chr2 400
## chr3 0
```

If we do not specify the RS:

```
cat test.bed | tr "\n" ":" | \
awk '{print $1,$2}'
```

```
## chr1 100
```

- ORS - Output Record Separator

```
awk 'BEGIN {ORS=","} {print $1,$2}' test.bed
```

```
## chr1 100,chr1 300,chr2 240,chr2 400,chr3 0,
```

```
awk 'BEGIN {ORS="\n---\n"} {print $1,$2}' test.bed
```

```
## chr1 100
## ---
## chr1 300
## ---
## chr2 240
## ---
## chr2 400
## ---
## chr3 0
## ---
```

- NR - Number of Records

When used inside the BODY block, this gives the line number. When used in the END block, this gives the total number of records in the file.

```
awk '{print "line",NR,"chromosome is",$1;} \
END {print "Total number of records:",NR}' test.bed
```

```
## line 1 chromosome is chr1
## line 2 chromosome is chr1
## line 3 chromosome is chr2
## line 4 chromosome is chr2
## line 5 chromosome is chr3
## Total number of records: 5
```

Lesson 4, more built-in variables

- FILENAME – Current File Name

```
cp test.bed test2.bed
```

```
awk '{print FILENAME}' test.bed test2.bed
```

```
## test.bed
## test.bed
## test.bed
## test.bed
## test.bed
## test2.bed
## test2.bed
## test2.bed
## test2.bed
## test2.bed
```

This is useful, when you want to combine multiple files but want to add an extra column to specify which file it is from:

```
awk 'BEGIN {OFS="\t"} {print $0, FILENAME}' test.bed test2.bed
```

```
## chr1 100 200 test.bed
## chr1 300 500 test.bed
## chr2 240 440 test.bed
## chr2 400 600 test.bed
## chr3 0   150 test.bed
## chr1 100 200 test2.bed
## chr1 300 500 test2.bed
## chr2 240 440 test2.bed
## chr2 400 600 test2.bed
## chr3 0   150 test2.bed
```

Merge all bed files and add a column for the filename:

```
awk '{print $0 "\t" FILENAME}' *bed
```

- FNR - File "Number of Record"

"NR" is "Number of Records" (or "Number of the Record"), which prints the current line number of the file that is getting processed. How will NR behave when we give have two input files? NR keeps growing between multiple files. When the body block starts processing the 2nd file, NR will not be reset to 1, instead it will continue from the last NR number value of the previous file.

```
awk '{print FILENAME ": record number",NR,"is",$1;} \
END {print "Total number of records:",NR}' test.bed test2.bed
```

```
## test.bed: record number 1 is chr1
## test.bed: record number 2 is chr1
## test.bed: record number 3 is chr2
## test.bed: record number 4 is chr2
## test.bed: record number 5 is chr3
## test2.bed: record number 6 is chr1
## test2.bed: record number 7 is chr1
## test2.bed: record number 8 is chr2
## test2.bed: record number 9 is chr2
## test2.bed: record number 10 is chr3
## Total number of records: 10
```

Let's use FNR instead:

```
awk '{print FILENAME ": record number",FNR,"is",$1;} \
END {print "Total number of records:",NR}' test.bed test2.bed
```

```
## test.bed: record number 1 is chr1
## test.bed: record number 2 is chr1
## test.bed: record number 3 is chr2
## test.bed: record number 4 is chr2
## test.bed: record number 5 is chr3
## test2.bed: record number 1 is chr1
## test2.bed: record number 2 is chr1
## test2.bed: record number 3 is chr2
## test2.bed: record number 4 is chr2
## test2.bed: record number 5 is chr3
## Total number of records: 10
```

Merge multiple files with the same header by keeping the header of the first file I usually do it in R, but like the quick solution.

```
awk 'FNR==1 && NR!=1{next;}{print}' *.csv
```

Lesson 5 Variables

Awk variables should begin with an alphabetic character; the rest of the characters can be numbers, or letters, or underscore. Keywords cannot be used as an awk variable name. Unlike other programming languages, you don't need to declare an variable to use it. If you wish to initialize an awk variable, it is better to do it in the BEGIN section, which will be executed only once. There are no data types in Awk. Whether an awk variable is a number or a string depends on the context in which the variable is used in.

```
awk 'BEGIN {total=0} \
{print $2; total=total+$2} \
END {print "total is "  total}' test.bed
```

```
## 100
## 300
## 240
## 400
## 0
## total is 1040
```

calculate total number of reads in a bam file:

```
samtools idxstats example.bam | cut -f3 | \
awk 'BEGIN {total=0} {total += $1} END {print total}'
```

Lesson 6 Arithmetic Operators

- + Addition
- - Subtraction
- * Multiplication / Division
- % Modulo Division

```
awk 'BEGIN {OFS="\t"} {print $1, $2 + 100, $3 }' test.bed
```

```
## chr1 200 200
## chr1 400 500
## chr2 340 440
## chr2 500 600
## chr3 100 150
```

This is useful. If you get a transcription start site (TSS) bed file, you can use

```
awk 'BEGIN {OFS="\t"} {print $1, $2 - 1000, $3 }' tss.bed
```

to get the upstream 1000 bp bed file.

- String Operator (space) is a string operator that does string concatenation. We have seen it in the previous examples

This operator is why you must separate the values in a print statement with a comma if you want to print the OFS in between. If you do not include a comma to separate the values, the values are concatenated instead.

```
awk 'BEGIN {OFS="\t"} {print "chromosome is "$1, "start is " $2 , "end is " $3 }' test.bed
```

```
## chromosome is chr1   start is 100    end is 200
## chromosome is chr1   start is 300    end is 500
## chromosome is chr2   start is 240    end is 440
## chromosome is chr2   start is 400    end is 600
## chromosome is chr3   start is 0  end is 150
```

- Comparison Operators
- > Is greater than
- >= Is greater than or equal to
- < Is less than
- <= Is less than or equal to == Is equal to
- != Is not equal to
- && Both the conditional expressions are true
- || Either one of the conditional expressions is true

A note on the following examples: If you don't specify any action, awk will print the whole record if it matches the conditional comparison.

```
awk '$1 == "chr1"' test.bed
```

```
## chr1 100 200
## chr1 300 500
```

```
awk '$2 > 100' test.bed
```

```
## chr1 300 500
## chr2 240 440
## chr2 400 600
```

```
awk '$2 > 100 && $3 >=500' test.bed
```

```
## chr1 300 500
## chr2 400 600
```

```
awk '$2 > 100 || $3 >=300' test.bed
```

```
## chr1 300 500
## chr2 240 440
## chr2 400 600
```

```
awk '$2 > 100 {print $2, $1}' test.bed
```

```
## 300 chr1
## 240 chr2
## 400 chr2
```

- Regular Expression Operators
- ~ Match operator
- !~ No Match operator

When you use the == condition, awk looks for a full match.

```
awk '$0 == "chr1"' test.bed
```

This prints nonthing because there is no whole line equals to chr1.

```
awk '$0 ~ "chr1"' test.bed
```

```
## chr1 100 200
## chr1 300 500
```

Bonus:

- AWK GTF! How to Analyze a Transcriptome Like a Pro
- bioawk
- seqkit