Awk

Bio724D: Spring 2025

2025-02-04

AWK

AWK is a programming language that is designed to efficiently work on tabular data based on pattern matching.

AWK is names after it’s three inventors, all of whom were computer scientists at Bell Labs in the 70s. Recall that Bell Lab’s is also where Unix was invented:

- Alfred V. Aho – also invented efficient algorithms for regular expressions that were critical for their adoption
- Peter J. Weinberger – contributed to AWK and modern Fortran; became head of computer science research at Bell Labs in the 80s
- Brian W. Kernighan – contributed to the development of Unix; helped to popularize the C programming language (Kernighan and Ritchie, The C Programming Language, 1978 is known simply as “K&R C”)

Why Awk

- Awk combines the text processing powers of many of the core Unix utilities such as `cut`, `grep`, and `sed` into a single programming language.
- Awk is concise and has a relatively simple syntax.
- Awk is fast and can handle GBs of data efficiently!

Kernighan’s concise description of Awk, part I

Structure of an AWK program:

An awk program is a sequence of pattern-action statements

```
    pattern { action }
    pattern { action }
```

A pattern is a regular expression, numeric expression, string expression or combination; an action is executable code, similar to C.

Operation:

```
    for each file
       for each input line
          for each pattern
             if pattern matches input line
                do the action
```

If there is no pattern, the action is performed on each input line. If there is no action, the line is printed if it matches the pattern. The special pattern BEGIN matches before any input has been read; the special pattern END matches after all input has been read.

Usage:

```
    awk 'program' [ file1 file2 ... ]
    awk -f progfile [ file1 file2 ... ]
```

Note parallels with `sed`

sed:

```
[addr]command[flags]
```

awk:

```
pattern {action}
```

- `pattern` in awk terms is equivalent to `[addr]` in sed terms
- `action` is equivalent to `command` but `action` can be any arbitrary computation, not just a single command.

Kernighan’s concise description of Awk, part II (modified)

```
AWK features:

    input is read automatically across multiple files
    lines are split into fields called $1, ..., $NF; $0 is the whole line
        default split is by white space
        changing FS to some other value (string or RE) affects split
        change FS by assigning to it, or by -F'...' on commandline
    The variable NR is the current record (line) number
    variables contain string or numeric values
        no declarations: using a variable declares it
        initialized to 0 and empty string
        type determined by context and use: the type is set by the
        last operation, and might be string or number or both.  for
        example, x = 1 makes x a number, x = "1" makes it a string.
    operators work on strings or numbers
        coerce the type according to context (e.g., to string for printing)
    built-in variables for frequently-used values; see below
    associative arrays (arbitrary subscripts): x["anything"]
    regular expressions in /.../ (like egrep)

    control flow statements are similar to C
        if-else, while, for, do (but no switch), break, continue
        for (i in array)
           sets i to each subscript of associative array in turn
        next: start next iteration of main loop
        exit: leave main loop, go to END block
```

Awk one-liners

Call these as `awk -F'<delimiter>' '<code>' input.txt`:

- `{print NR, $0}` – number each lines
- `{print $1, $3, $2, $5}` – write out fields of interest in new order
- `NF > 0` – print non-empty lines
- `/regexpr/` – print lines that match the regular expression
- `$0 !~ /regexpr/` – print lines that don’t match the regular expression
- `$3 ~ /regexpr/` – print lines where 3rd field matches the regular expression
- `{print $0, $5 - $4}` – add a new column to each line with a derived value

Multiline examples

Put these in a file and call them as `awk -f <filename.awk> input.txt`

- filter comment and empty lines

```
/^#/ { next }    # skip lines that start with #
NF > 0           # print non empty lines
```

- calculate number of lines and average line length

```
{ linesum += length($0)}
END { print NR, linesum/NR}
```

Awk program, basic template

```
# BEGIN rule evaluated before lines are processed
BEGIN {
    initial actions
}

# simple patttern action written as one line
pattern1 {action1}

# more complicated pattern-action statement
# might include flow control like if-else, for loop, etc
pattern2 {
    action2
}

# a pattern without an action (default to printing)
pattern3

# an action without a pattern, applies to all lines
{action4}

# END rule evaluated after all lines are processed
END {
    final actions
}
```

Awk program, concrete example

```
BEGIN {
    FS = "\t"    # Tab is the field delimiter in input files
    gene_ct = 0   # setup a counter variable
    size_sum = 0  # setup a variable to hold sum of gene lengths
}

/^##FASTA/ { exit }  # end of tabular GFF features,
                     # so exit and don't process any more lines

/^#/ { next } # metadata line

NF != 9  { next }  # validate number of fields (NF)

# Redirect rows to new CVS files based on their size, add a new field too
$3 == "gene" {
    gene_ct += 1
    gene_size = $5 - $4 + 1
    size_sum += gene_size
}

END {
    print "Number of genes: " gene_ct
    print "Average gene size: " size_sum / gene_ct
}
```

Kernighan’s concise description of Awk, part III

```
Basic AWK programs:

Operators include C operators like + - * / % = += -= *= /= %= && || !
  Expressions are almost the same as C.

x ~ /re/, s !~ /re/  string matches/does not match re.

Strings are concatenated by being adjacent:
    hw = "hello" "world" sets hw to "helloworld".
    Watch out; this often has surprising properties.

These are all one-liners:
  { print NR, $0 }  precede each line by its line number
  { $1 = NR; print }    replace first field by the line number
  { print $2, $1 }  print field 2, then field 1 (and nothing else)
  { temp = $1; $1 = $2; $2 = temp; print }   flip $1, $2, print whole line
  { $2 = ""; print }    zap field 2
  { print $NF }     print last field
  NF > 0        print non-empty lines
  NF > 4        print lines with more than 4 fields
  $NF > 4       print line if last field is greater than 4
  NF > 0        {print $1, $2}  print two fields of non-empty lines
  /regexpr/     print lines that match regepxr
  $1 ~ /regexpr/    print lines where first field matches regexpr
  END { print NR }  line count: print number of records at the end
```

Awk field separators, field and record counts

- Field separators – both the input field separator (`FS`) and the output field separator (`OFS`) can be specified in an Awk program. This is usually done in a `BEGIN` rule:

```
BEGIN {
  FS = "\t"  # input is tab-delimited
  OFS = ","  # output will be comma-delimited
}
```

- Record and field counts:
  - `NR` and `FNR` – `NR` gives the total number of records seen so far. FNR`gives the current record number in the current file (awk can process multiple files simultaneously); When processing a single file,`FNR==NR‘.
  - `NF` – gives the number of fields in the current record (line)

```
awk -F "\t" `{print "There are", NF, "fields in line", NR}` input.txt
```

Awk control flow statements

- if-else
- while
- do-while
- for
- switch
- break
- continue
- next
- nextfile
- exit

Redirection and piping in Awk

- Awk has a redirection operator, >
  - Overwrites file if it currently exists, but “smart” in that subsequent calls append to the file not overwrite
  - `print items > output-file`
- Awk has an append operator, >>
  - `print items >> output-file`
- Awk has a pipe operator: ‘|’
  - print items | command

```
awk -F"\t" '$3 == "gene" {print $1, $3, $5-$4  | \
"sort -nr -k3,3" }' yeast.gff
```

Note that the command in the pipe call needs to be quoted.

Example with flow control and redirection: Splitting a GFF file

```
# Process GFF file, removing all meta data lines and splitting the data into
# a tabular feature file and a FASTA seq file
BEGIN {
    FS = "\t"    # Tab is the field delimiter in input files
    in_seq = 0
}

/^##FASTA/ { in_seq = 1 }  # dealing with sequence part of file
/^#/ { next } # metadata lines

# NOTE: hard-coding file names is not a great idea!
{
    if (in_seq) {
        print $0 > "seq_data.fasta"
    } else {
        print $0 > "tabular_data.gff"
    }
}
```

Improving our GFF splitter with the awk -v option

Awk allows one to initialize variables at the command line using the -v option. This can be used to set values that may need ot change at run time. Here we use this approach to make our GFF splitter program a little more robust:

```
# EXAMPLE: awk -v prefix=out1 -f gffsplitter.awk input.gff
BEGIN {
    FS = "\t"    # Tab is the field delimiter in input files
    if (prefix == "") {
        print "Error: please specify a prefix for output file using -v prefix=XXXX."
        exit
    }
}

/^##FASTA/ { in_seq = 1 }  # dealing with sequence part of file
/^#/ { next }              # metadata lines

{
    if (in_seq) {
        print $0 > prefix ".fasta"
    } else {
        print $0 > prefix ".gff"
    }
}
```

Awk arrays

- “Associative arrays” (sometimes called maps or dictionaries in other languages) are created on the fly in awk
- Keys (indices) are used to set of get the values in an awk array. Keys can be strings or numbers

```
# Count the number of features per chromosome in a GFF file
# and output results as a CSV table
BEGIN {
    FS = "\t"
    OFS = ","
}
/^##FASTA/ { exit }
/^#/ { next }
$3 == "chromosome" { next } # don't count chromosomes themselves

{ chrom_ct[$1] += 1 }

END {
    print "seq", "ftr_count"
    for (i in chrom_ct) {
        print i, chrom_ct[i]
    }
}
```

Awk built-in functions (from Kernighan's Awk help)

```
Awk strings and string functions are 1-origin; be careful.

    length(s) length of a string
        length(array) returns number of elements
    n = index(s, f)
        returns index of f in s, or 0 if not there
    n = match(s, re)
        index where re matched in s, or 0 if no match
    nsub = sub(re, repl, target)
        replaces first instance of re in target by repl
        returns 0 if no match
    nsub = gsub(re, repl, target)
        replaces all instances of re in target by repl
        returns 0 if no match, number of replacements otherwise
    str = substr(s, start, length)
        returns substring of s starting at start, up to length
        characters (default is rest of string).  works sensibly
        if you go off the ends.  note: origin is 1.
    s = toupper(str)
    s = tolower(str)
        map case
    s = sprintf("...", exprlist)
        formats expressions, returns string result

There are also some of the usual math functions: int, sqrt, exp, log,
sin, cos, atan2, rand (uniform between 0 and 1), srand(new_seed).
```

GNU Awk (gawk) extends/adds built-in functions

See the GNU Awk Manual, Section 9.1 for a full list. Some useful ones include:

String manipulation

- `match(string, regexp [, array])` – extended version of `match` allowing you to capture regex groups into an array
- `split(string, array [, fieldsep [, seps ] ])` – split a string by defining a separator (`fieldsep`) (can be a regex)
- `patsplit(string, array [, fieldpat [, seps ] ])` – split a string by defining what's between the separators (`fieldpat`)

Others

- `system(command)` – run a command outside of awk and then return to the awk program.
- `strftime([format [, timestamp [, utc-flag] ] ])` – get current system time and return it as a formatted string