2016 年 06 月 24 日

計算機実習 III (2016 年度)
第 9 回: AWK その 3

(http://takeno.iee.niit.ac.jp/%7Eshige/
math/lecture/comp4/comp4.html)

目 次

1 行フィルタとしての AWK の概要

今回より、AWK の行単位のフィルタとしての動作について説明する。

この場合、今までのようなスクリプトファイル指定の後ろに、AWK に処理させるデータファイルを指定する:

```
Z:¥> gawk -f [スクリプトファイル] [データファイル]
```

このように実行すると、AWK は図 1 に示すフローチャートのように、データファイルのデータを 1 行ずつ読み込んで、そのそれぞれの行に対してスクリプトで書いた処理を順番に行う。このように、入力テキストデータを行単位で処理し、結果を画面 (標準出力) に出力するプログラムを **行フィルタ** と呼ぶ[1]。

AWK スクリプトの構造は、一般には「BEGIN ブロック」、「通常ブロック」、「END ブロック」の 3 種類の実行ブロックからなる。

[1]正確には、フィルタとは入力も標準入力であるものを指すが、AWK はそのような使い方も可能。詳しくは、後の回で説明する予定。

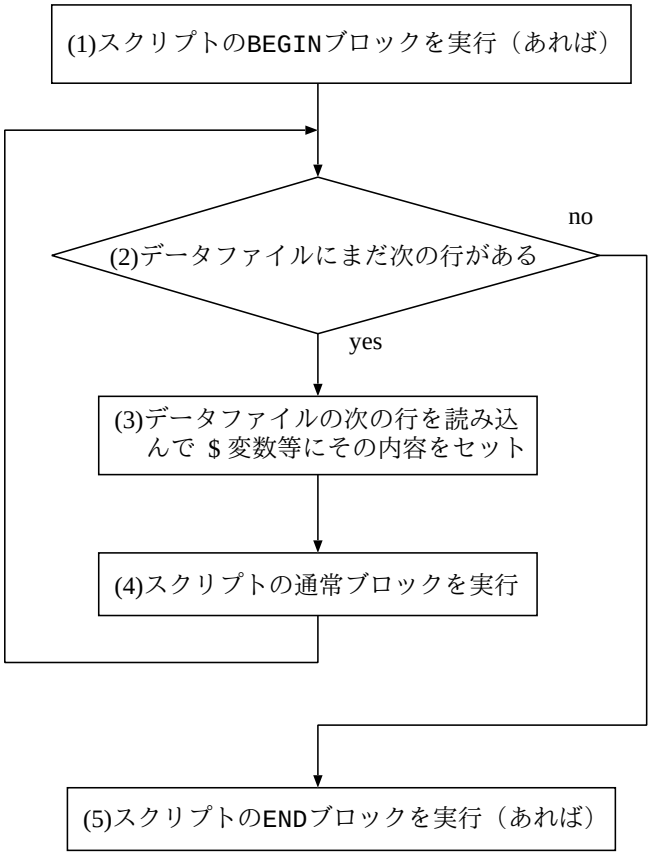

図 1: フィルタとしての AWK の処理の流れ

```
BEGIN {
    [BEGIN ブロックの実行内容]
}
[条件] {
    [通常ブロックの実行内容]
}
END {
    [END ブロックの実行内容]
}
```

BEGIN ブロック、通常ブロック、END ブロックはそれぞれ省略可能であり、BEGIN ブロックはデータファイルの読み込み前に実行され (図 1 の (1))、通常ブロックは、データファイルの行数分だけ繰り返し実行され (図 1 の (4))、END ブロックはデータファイルの最後の行に対する通常ブロック処理が終わった後に実行される (図 1 の (5))。

通常ブロックには、[条件] をつけることもできるが (省略も可)、これについては 4 節

で説明する。

2 フィールド変数

本節では、まず図 1 のフローチャートの (3) の部分にある「$ 変数等への内容のセット」について説明する。

AWK は、データファイルを 1 行読みこむと、それを空白やタブで区切られた **フィールド** (あるいは **列** とも言う) と呼ばれる単位に分割し、その結果を表 1 のような変数にそれぞれ保存する。

変数	意味
$[n]	その行の [n] 番目のフィールド文字列 (先頭が 1)
$0	その行全体の文字列
NF	その行のフィールド数
NR	その行の行番号 (先頭行が 1)

表 1: 行の読み込みで設定されるフィールド変数等

例えば、データファイル data1.txt が

```
2016 年 6 月 22 日 (水)
2016 年 6 月 23 日 (木)
2016 年 6 月 24 日 (金)
```

のように、3 行のデータで各行にはスペース区切りで 7 列のフィールドデータが書かれている場合、この 2 行目が読み込まれた時点 ((3) の処理のあと) での各フィールド変数の値は、

```
NF=7, NR=2, $1="2016", $2="年", $3="6", $4="月", $5="23",
$6="日", $7="(木)", $0="2016 年 6 月 23 日 (木)"
```

となる (NF と NR は数値、$ 変数は基本的に文字列値)。よって、例えば test1.awk が

```
BEGIN { print "日付は" }
{ print $3, $4, $5, $6 }
END { print "です。" }
```

というスクリプトであるとき、

```
Z:¥> gawk -f test1.awk data1.txt
```

とすると、

```
日付は
6 月 22 日
6 月 23 日
6 月 24 日
です。
```

と表示される。

```
{ x = $1 + $2; print x }
```

このスクリプトは、データの 1 列目と 2 列目を数値として足した結果を x に保存しそれを出力する (表示する[2])、という作業をすべての行に対して行う。なお、print の後ろには数式を書くこともできるので、

```
{ print $1 + $2 }
```

というスクリプトでも同じことになる。

```
{ printf "%4d: %s¥n", NR, $0 }
```

というスクリプトは、入力データファイルの行番号を 4 桁で表示し、その後ろに「:」とスペースを 1 つつけて、その後ろに行の内容をそのまま表示する。これにより、元のデータファイルの左に行番号が付加されたものが表示される。この結果を

```
Z:¥> gawk -f test2.awk hoge1.c > hoge1-c.txt
```

のように、出力リダイレクションを用いてファイルとして保存することももちろん可能である。

$ 変数を使う場合、$ の後ろには 1, 2, 3 などの定数以外に、「j=3; s=$j」のように値が整数である変数や、「s=$(2*j+3)」のように結果が整数となる数式を指定することもできる (ただし数式をかっこで囲む必要がある)。例えば、$NF はその行の最終フィールド文字列、$(NF-1) はその行の最後から一つ手前のフィールド文字列になる。

例えば、

```
{ for (j=1; j<=NF; j++) print $j }
```

[2]以後、print, printf による「表示」は、リダイレクトされることも考えて「出力」とも表現する。

は、データファイルのすべてのフィールド (単語) を、1 行に 1 つずつ出力する。for 文で j を 1 から NF まで変化させることで、$j はその行の最初のフィールド文字列から最後のフィールド文字列の値を順に取ることになる。

3 END ブロック

図 1 のフローチャートの (5) の END ブロックでは、すべての行の処理が済んだ後の処理を行うことができる。

例えば、

```
{ sum += $1 } # (4)
END { print sum/NR } # (5)
```

というスクリプトは、すべての行に対し 1 列目の値を sum という変数に追加していき ((4) の部分)、データの読み込みが終わったあとでそれを NR で割った値を出力する。すなわち、入力データの 1 列目のすべての行の値の平均値を表示することになる。

なお、第 7 回で説明したように、AWK では初期化してない変数の初期値は 0 (か空文字) になるので、この sum の初期値も 0 であり、初期値の設定は必要ない。

また、END ブロックでは一見 NR は意味がないようだが、実際には END ブロックでは NR の値は更新されず、入力データの最後の行でセットした行番号が残ったままになっているので、END ブロック内では NR は最後の行の行番号、すなわちデータファイル全体の行数を表すことになる。

```
{ h[NR] = $0 } # 各行を配列 h[ ] に保存
END { for (j=NR; j>=1; j--) print h[j] }
```

このスクリプトは、データファイルの行を逆順に出力する。通常ブロックで各行の内容全体 ($0) を、行番号を添字とする配列変数 (h[NR]) に保存しているが、行を逆順に出力するためには、最後の行までデータを読んだあとで既に読んだ行をすべて出力しないといけないので、このようにすべての行を一旦配列に保存して処理する必要があり、データによってはメモリを大量に消費する。

入力データファイルの最後の 5 行だけを出力するには、例えば上と同様にすべての行を配列に保存し、

```
{ h[NR] = $0 }
END { for (j=4; j>=0; j--) print h[NR - j] }
```

のようにすればよいが、この方法では行数の多いデータファイルの場合は大半の配列要素は無駄になる (配列要素を 5 つしか使用しない方法を考えてみよ)。

4 条件部分の使い方

1 節で書いたように、通常ブロックには [条件] をつけることもできるが、[条件] をつけると、データファイルの各行がその条件に適合する場合だけそのブロック部分を実行するようになる。例えば、

```
NR >= 11 && NR <= 15 { print $0 }
```

というスクリプトは、「NR >= 11 && NR <= 15」がその [条件] 部分であるが、これはデータファイルの 11 行目から 15 行目の間の行だけ「print $0」を実行し、その他の行には何も実行しない、ということになる。なお、この例のように、[条件] 部分には条件式だけを書き、if (や else) は書かない。

この形式のスクリプトは、外の [条件] を使わず if 文で書き換えることもでき、例えば上の例は、[条件] 部分を実行部分の if 文に変えた

```
{ if ( NR >= 11 && NR <= 15 ) print $0 }
```

と、動作は全く同じである。

なお、AWK では、「print $0」は「print」とだけ書いても同じことをすることになっているので、上の 2 つのスクリプトでは $0 を省略することもできる。

[条件] 部分、および通常ブロックの実行部分 ({ } 部分) はそれぞれ省略することができ、省略した場合は、

- [条件] 部分を省略 $\Longrightarrow$ すべての行に対して実行部分を実行
- 実行部分を省略 $\Longrightarrow$ [条件] に合う行をそのまま出力

となっている。例えば上のスクリプトも、条件に合う行を出力するだけなので、

```
NR >= 11 && NR <= 15
```

の条件部分だけでよい。

```
NR % 2 == 0
```

これも [条件] だけのスクリプトで、データファイルの偶数行のみを出力する。このようなスクリプトは「ワンライナー」で有用だが、それについては後の回で説明する。

5 行フィルタのサンプル

本節では、行フィルタとしての短いスクリプトのサンプルをいくつか紹介する。

```
{ print $2, $1 }
```

これは、各行の最初の 2 列を交換してスペース区切りで表示する。

```
{ print $2 $1 }
```

これは、同様の交換をするが、「$2 $1」は $2 と $1 を連結した文字列を意味するので、「区切りを入れずに」連結して表示することになる。

```
{ print $1 + x; x = $2 }
```

は、各行の 1 列目の値に、一つ前の行の 2 列目の値 (x に保持) を加えたものを出力し、

```
{ print $1 + x; x += $2 }
```

は、各行の 1 列目の値に、それ以前のすべての行の 2 列目の値 (x に保持) を加えたものを出力する。例えば、データファイルが

```
1 5
2 6
3 7
4 8
```

であった場合、上から 3 番目のスクリプトは 1,7,9,11 を順に出力し、その次のスクリプトは 1,7,14,22 を出力する。ことになる。

```
{ x = 0; for (j=1; j<=NF; j++) x += $j; print x }
```

これは、各行の列のデータのすべての和をそれぞれ出力する (横方向の和)。縦方向の

和を取るには、各列の和の値を配列に保存して、

```
{ if (NF > N) N = NF; for (j=1; j<=NF; j++) h[j] += $j }
END { for (j=1; j<=N; j++) print h[j] }
```

のようにすればよい。N は最大フィールド数を保持し、j 列目の和の値を配列 h[j] に保存していて、最後に END ブロックでそれらを表示している。なお、データのすべての行の列数が同じならこの N は不要で、代わりに NF を用いればよい。

```
{ print; if (NR % 10 == 0) print "" }
```

これは、データファイルをほぼそのまま出力するが、10 行毎に空行を 1 行ずつ出力する (print "" は改行のみを出力)。

```
NR == 1 { max = $1 } # (6)
NR > 1 && max < $1 { max = $1 } # (7)
END { print max }
```

これは、データファイルの 1 列目の最大値を出力する。(6) の行はデータの 1 行目でのみ実行される行で (max 初期設定)、(7) の行でデータの 2 行目以降での max の更新を行う。このように、実行ブロックは複数書くこともでき、その場合それらは、図 1 の (4) の部分で順に実行される。

このスクリプトを外の条件を使わず if 文で書き換えると、以下のようになる。

```
{ if ( NR == 1 ) max = $1
    else if ( max < $1 ) max = $1 }
END { print max }
```

else を使っているので、「NR > 1」という条件式はこの場合は必要ない。

(6), (7) の実行部分が同じなので、条件と実行ブロックは以下のようにもできる。

```
NR == 1 || max < $1 { max = $1 }
END { print max }
```

C や AWK では、「A || B」の B の部分は、A が真のときは無視され A が偽であるときだけ評価されるので、この場合も「NR > 1」という条件式は必要ない。

なお、実は元のスクリプトの (7) でも「NR > 1」は実は必要はない (理由を考えてみよ)。