2012 年 06 月 01 日

計算機実習 IV (2012 年度)
第 7 回: AWK その 2

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

目 次

1 行フィルタとしての動作

AWK は本来「行フィルタ」プログラムであり、入力データを 1 行ずつ処理する[1]。AWK スクリプトは、その処理方法 (AWK の命令) を書いたものであり、複数 (または一つ) の

[条件] { [命令文] }

の形式のブロックで構成される[2]。前回紹介した BEGIN ブロックもその一つである。命令文は、1 行でも複数行でもよい。

gawk は最後のオプションを入力データファイルと見なすが、オプションとして入力データを指定しなかった場合は、標準入力から入力データを読み込むので、入力データファイルと AWK スクリプトは以下のような数通りの方法で指定できる。

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

[1] AWK では 1 行のデータ単位を **レコード** と呼ぶ。
[2] GNU AWK の場合は、さらに関数定義部分もあるが、本講義では省略する。

スクリプトが 1 行で書ける場合には、ファイルに保存しなくても、

```
Z:> gawk "[1 行スクリプト]" [データファイル]
Z:> gawk "[1 行スクリプト]" < [データファイル]
Z:> type [データファイル] | gawk "[1 行スクリプト]"
```

のようにして実行できる。すなわち、スクリプトファイルは -f をつけて指定し、入力データファイルには何もつけずに最後に指定するか、標準入力にリダイレクションかパイプで渡せばよい。

なお、この 1 行スクリプト内で 2 重引用符 " を使う場合は、引用符の終わりと見なされないよう ¥" と書く必要がある。

まずは、スクリプトのブロックが一つの場合

```
[条件] { [命令文] }
```

の動作を説明する。

この場合 AWK は、入力データファイルから 1 行ずつ読みこみ、[条件] に適合する行に対して [命令文] に書かれた処理を実行する。[条件] を省略して

```
{ [命令文] }
```

とした場合はすべての行に対し [命令文] を実行し、逆に「{ [命令文] }」を省略して

```
[条件]
```

だけを書いた場合は [条件] に適合する行をそのまま出力する。

AWK は行データを **フィールド** 単位に分割して処理する。フィールドの区切りはデフォルトでは空白やタブになっている[3]。AWK は読み込んだ行のフィールドデータに応じて、以下のような変数に値を設定する。

- $1,$2,... : その行の 1 番目のフィールド文字列、2 番目のフィールド文字列、...
- $0 : 元々の行全体の文字列
- NF : その行のフィールド数
- NR : その行の行番号

[3]AWK のフィールドの区切りはオプションにより変更できる。

$ の後ろには定数以外に、「$(2*j+3)」のように整数値を持つ変数や式を指定することもできる。なお、以後データの横の並び (すなわちレコード) を行と呼び、縦の並び (すなわちフィールド) を列と呼ぶことにする。

例えば、

```
2012 年 4 月 5 日（水）
2012 年 4 月 6 日（木）
2012 年 4 月 7 日（金）
```

というデータ data1 の場合、3 行の各行に 7 列のデータがあり、2 行目が読み込まれた段階では、

```
NF=7, NR=2, $1="2012", $2="年", $3="4", $4="月",
$5="6", $6="日", $7="(木)"
```

となる。

このデータに対して、

```
Z:> gawk "{printf ¥"%s %s¥n¥", $3, $5}" data1
```

とすると、

```
4 5
4 6
4 7
```

と表示され、

```
Z:> gawk "{print $(NF-2), $NF}" data1
```

とすると

```
5（水）
6（木）
7（金）
```

と表示され、

```
Z:> gawk "$5 == 6 {print $0}" data1
```

とすると、

```
2012 年 4 月 6 日（木）
```

と表示される。なお print は、引数を省略すると $0 を引数としたことと同じになるので、これは

```
Z:> gawk "$5 == 6 {print}" data1
```

と書いても同じであり、さらに命令文を省略した場合はその行そのものを出力するので、

```
Z:> gawk "$5 == 6" data1
```

と書いても同じである。

課題 7-1. 各行が 2 列の数字からなるデータに対し、1 列目と 2 列目の和を出力する 1 行スクリプトを書け。

課題 7-2. 各行が 2 列の文字列のデータに対し、1 列目と 2 列目の順番を入れ換えて出力する 1 行スクリプトを書け。

課題 7-3. 各行が 2 列の数字のデータに対し、1 列目の値と、2 列目の値と、(1 列目 − 2 列目) の値、の 3 列を出力する 1 行スクリプトを書け。

課題 7-4. 各行の 1 列目が数字のデータに対し、1 列目の値が 5 以上である行を出力する 1 行スクリプトを書け。

各行の列の数は NF で取得できるので、例えば

```
Z:> gawk "NF>0" data
```

とすれば、ファイル data の空行 (フィールド数が 0) 以外の行だけを表示することになり、

```
Z:> gawk "{x=0; for(j=1;j<=NF;j++) x+=$j; print x}"
data （実際には 1 行で書く）
```

とすると、「x+=$j」は x に j 列目の値を追加することを意味するので、結局 data の各行で行内の列の値の和を表示することになる。

課題 7-5. 列が 2 つ以上ある行を出力する 1 行スクリプトを書け。

課題 7-6. 各行が 1 列以上の数字のデータに対し、すべての列の値を二乗した値の和を出力する AWK スクリプト kadai7-6.awk を作成せよ。

課題 7-7. 「dir C:¥Windows¥Fonts」の出力をパイプで gawk に流し、各行の最終列とその前の列だけを表示させてみよ。

各行の行番号は NR で取得できるので、例えば

```
Z:> gawk "NR<=10" data
```

とすれば、ファイル data の先頭 10 行が表示される。

課題 7-8. 5 行目から 20 行目までを出力する 1 行スクリプトを書け。

課題 7-9. 10 行毎に空行を 1 行追加して出力する AWK スクリプト kadai7-9.awk を作成せよ。

課題 7-10. 入力データファイルの各行の先頭に「0001:　[元の 1 行目の内容]」のように 4 桁の行番号を追加して出力する AWK スクリプト kadai7-10.awk を作成せよ。

2 複数のブロックがある場合

AWK スクリプトに複数のブロックがある場合

```
[条件 1] { [命令文 1] }
[条件 2] { [命令文 2] }
[条件 3] { [命令文 3] }
...
```

は、以下のように動作する。

1. 条件部分が「BEGIN」であるブロックがあれば、データを読む前にそのブロックの処理を行う。
2. 入力データから 1 行読みこみ、[条件 1] が真ならばその行に対して [命令文 1] を実行し、次に [条件 2] が真ならば [命令文 2] を実行し、という形で順に処理を行う。
3. 入力データから次の行を読みこみ、2. と同じことを行う。これを入力データの最後の行まで行う。

4. 最後に、条件部分が「END」であるブロックがあれば、そのブロックの処理を行う。

例えば、

```
BEGIN { N = 0; }
{ N ++; }
END { print N; }
```

というスクリプトの場合は、入力前に N に 0 をセットし、入力行を 1 行読み込む度に N の値を 1 つずつ増やし、データを読み終ったら N の値を表示するので、結果としてデータファイルの行数を出力するスクリプトになる。

なお、AWK では数変数は明示的に初期化しない場合は最初に 0 がセットされるので、一行で

```
{N++} END{print N}
```

と書いても同じことになる。なお、この例からわかるように、各ブロック同士も改行で区切られている必要はない。

また、例えば

```
{x+=$1} END{print x}
```

は、各行の 1 列目の値の合計を出力する。

課題 7-11. 2 列以上ある行の数を出力する 1 行スクリプトを書け。

課題 7-12. 「`dir /-c C:¥Windows¥Fonts`」(`/-c` はファイルサイズをカンマで区切らない) の出力をパイプで「`find /l "ttf"`」に渡し、さらにその出力をパイプで gawk に渡すことで、その 3 列目の値の合計 (= TrueType フォントのファイルサイズの合計) を求めよ。

課題 7-13. 各行の 1 列目が数字のデータに対し、その 1 列目のデータの平均値を出力する AWK スクリプト kadai7-13.awk を作成せよ。

3 簡単な統計処理

前節で見たように、AWK で簡単にデータの平均値は計算できるが、統計処理で重要なものに分散や平方和、積和などがあり、実験データの回帰分析にも

良く用いられる。この節ではそれらを AWK で計算する方法について考える。

「身長と体重」のように、2 つずつ組になった n 個のデータ (x_j, y_j) $(j = 1, 2, \ldots, n)$ に対し、平均を表す記号 $E[x]$ 等を以下のように定める。

$$
\begin{aligned}
E[x] &= \frac{1}{n}\sum_{j=1}^{n} x_j = \frac{1}{n}(x_1 + x_2 + \cdots + x_n) \quad (x \text{ の平均}) \\
E[y] &= \frac{1}{n}\sum_{j=1}^{n} y_j = \frac{1}{n}(y_1 + y_2 + \cdots + y_n) \quad (y \text{ の平均}) \\
E[x^2] &= \frac{1}{n}\sum_{j=1}^{n} x_j^2 = \frac{1}{n}(x_1^2 + x_2^2 + \cdots + x_n^2) \quad (x^2 \text{ の平均}) \\
E[y^2] &= \frac{1}{n}\sum_{j=1}^{n} y_j^2 = \frac{1}{n}(y_1^2 + y_2^2 + \cdots + y_n^2) \quad (y^2 \text{ の平均}) \\
E[xy] &= \frac{1}{n}\sum_{j=1}^{n} x_j y_j = \frac{1}{n}(x_1 y_1 + x_2 y_2 + \cdots + x_n y_y) \quad (xy \text{ の平均})
\end{aligned}
$$

なお、$E[x]$, $E[x^2]$, $E[xy]$ 等は、$\bar{x}$, $\overline{x^2}$, $\overline{xy}$ のように書かれることも多い。

このとき、

$$
S_{xx} = \sum_{j=1}^{n}(x_j - E[x])^2 = x \text{ の}\textbf{平方和} \tag{1}
$$

$$
S_{yy} = \sum_{j=1}^{n}(y_j - E[y])^2 = y \text{ の}\textbf{平方和} \tag{2}
$$

$$
S_{xy} = \sum_{j=1}^{n}(x_j - E[x])(y_j - E[y]) = x, y \text{ の}\textbf{積和} \tag{3}
$$

と呼び、S_{xx}/n を x の **標本分散**、$S_{xx}/(n-1)$ を x の **不偏分散**、S_{xy}/n を x, y の **共分散** と呼ぶ。

平方和 S_{xx} を (1) の式で計算するには、まずすべての x_j を参照して $E[x]$ を求め、再びすべての x_j を参照して (1) の式に代入する、ということになり、データを 2 回見る必要があるので (2 パス)、1 行ずつデータを参照する AWK では簡単には計算できない。

しかし、(1) は式変形により、

$$
S_{xx} = \sum_{j=1}^{n}(x_j^2 - 2x_j E[x] + E[x]^2) = \sum_{j=1}^{n} x_j^2 - 2E[x]\sum_{j=1}^{n} x_j + E[x]^2 n
$$

$$= \quad nE[x^2] - nE[x]^2 = \sum_{j=1}^{n} x_j^2 - \frac{1}{n}\left(\sum_{j=1}^{n} x_j\right)^2 \tag{4}$$

となり、この (4) を使えば平方和 S_{xx} は 1 パスでも計算できる。例えば、

```
{ A += $1*$1; B += $1; N++; }
END { Sxx = A - B*B/N }
```

とすれば、A には 1 列目の 2 乗の値の和、B には 1 列目の和が保存されるので、Sxx に 1 列目のデータの平方和が計算されることになる。

同様に、(2), (3) も

$$S_{yy} = nE[y^2] - nE[y]^2 = \sum_{j=1}^{n} y_j^2 - \frac{1}{n}\left(\sum_{j=1}^{n} y_j\right)^2, \tag{5}$$

$$S_{xy} = nE[xy] - nE[x]E[y] = \sum_{j=1}^{n} x_j y_j - \frac{1}{n}\left(\sum_{j=1}^{n} x_j\right)\left(\sum_{j=1}^{n} y_j\right) \tag{6}$$

のような 1 パス形の式に変形することができる。

課題 7-14. データの 1 列目の標本分散と 2 列目の標本分散を同時に求める AWK スクリプト kadai7-14.awk を作成せよ。

課題 7-15. 1 列目のデータと 2 列目のデータの共分散を求める AWK スクリプト kadai7-15.awk を作成せよ。

(4), (5), (6) は 1 パスで計算できるので AWK のような行フィルタには向いているが、しかしこれらの計算式は (1), (2), (3) に比べて桁落ちが起きやすいことも知られていて、実際は (1), (2), (3) で計算する方が良いらしい。

AWK で 2 パスの計算をするには、すべてのデータを一旦配列に保存して、END ブロックで計算するしかないが[4]、それほど大きくないデータに対しては有効な方法である。例えば、以下のようにすればよい。

[4] 他にも getline を使って明示的にデータを再オープンする方法もあるが、説明は省略する。

```
{ A += $1; N++; h[N] = $1 }
END {
    A = A/N; # 平均
    for (j=1; j<=N; j++)
        Sxx += (h[j] - A)*(h[j] - A);
}
```

A は平均の計算用、h[ ] は 1 列目のデータを全部保存する配列である。

課題 7-16. 適当に (rand() や sin() などを使って) 50 個程度のデータを作成し、上記の 1 パスの方法と 2 パスの方法の両者で平方和を計算し、両者に違いがあるかどうか確認せよ。

課題 7-17. (3) の式を用いて ((6) は使わずに) データの 1 列目と 2 列目の積和を求める AWK スクリプト kadai7-17.awk を作成せよ。

課題 7-18. 入力データの行を逆順に出力する AWK スクリプト kadai7-18.awk を作成せよ (すべての行の内容を配列に保存し END ブロックで処理すればよい)。

課題 7-19. 入力データの最後の 5 行だけを出力する AWK スクリプト kadai7-19.awk を作成せよ (どこからが最後の 5 行であるかは最後まで読み込まないとわからない)。

コラム: スクリプト言語

現在は、実はスクリプト言語全盛といってもいいくらいスクリプト言語がよく使われている。

AWK よりもむしろ有名なスクリプト言語に、Perl, PHP, Python, Ruby などがある。これらは Unix, MS–Windows など多くの環境で動作する。

MS–Windows 上では、VBScript や JScript, ASP など Microsoft が提供するスクリプト言語もあるし、フリーの MS–Windows 用のスクリプト言語としては、簡単なゲームなどを作るのによく使われている HSP や、日本語でプログラミングできる「なでしこ」などがある。

これらはいずれも汎用のスクリプト言語であり、広い目的で使用されるが、Web アプリケーション用の JavaScript や PHP, Flash アニメーションの記述を行う ActionScript、アプリケーションの GUI インターフェース部分のみを作成する Tcl/Tk、グラフ描画ソフト gnuplot の命令スクリプトなど、特定の用途専用のスクリプト言語もある。バッチファイル (Unix でいうシェルスクリプト) もスクリプト言語の一つと言えるだろう。

そもそも「スクリプト」(script) とは「台本」を意味する言葉で、バッチファイルのように命令を順次実行させるよう並べて書いたようなものをスクリプトと呼び、一般的には以下のような性質を持ったものをスクリプト言語と呼んでいるようである。

- 動作はインタプリタ型
- 通常のプログラミング言語のように型宣言などが厳密でない
- コンパイル型言語に比べて短い命令で一定の仕事ができる
- 長いプログラムにはあまり向かない
- 動作速度もあまり早くない

Perl や Python, Ruby は特に広く使われているスクリプト言語であるが、これに共通しているのは以下のような性質であり、そのために好まれているのだろう。

- フリーソフト (オープンソースソフトウェア)
- 多くの環境で動作する
- ライブラリを追加することで容易に機能拡張が行える
- 情報 (本) も多く出版されている
- AWK よりは高機能で C 言語よりは楽に多くのことが行える
- 正規表現をサポートしている

C 言語を一通り終えたら Perl や Ruby などのスクリプト言語をひとつ勉強し身につける便利だろうと思う。