AWK Users JP

awk 基礎文法最速マスター

この文書は書きかけです

ここでは nawk (new awk) で使用可能な命令で記述しています。gawk (GNU awk) を使うことで、さらに効率よい記述を行うことができますが、nawk の文法で覚えておくと全ての awk で使うことができます。

基礎

print 文

表示は print 文です。

```
print "Hello World!";
```

コメント

```
# 以降がコメントになります。
```

スクリプトの実行

以下のように '-f' に続いてスクリプトを指定します。

```
nawk -f foo.awk
```

複数の awk スクリプトを指定することもできます。

```
nawk -f foo1.awk -f foo2.awk -f foo3.awk
```

スクリプトを直接指定できます。

```
# "Hello World" と表示
nawk 'BEGIN { print "Hello World" }'
```

BEGIN, END, パターン + アクション

BEGIN ブロック

ファイルを読む前に実行するブロックです。

```
# ファイルを読む前に "Not Read" と表示
BEGIN { print "Not Read" }
```

END ブロック

```
# ファイルを読み終わったら "Already Read" と表示
END { print "Already Read" }
```

ファイルを読み終わった後に実行するブロックです。

パターン + アクション

特定のパターン (または条件式) にマッチする場合に、アクションを実行します。

```
パターン { アクション }
```

BEGIN, END もある種のパターンと思えば全て同じ作法で記述することができます。

```
# 正規表現 reg にマッチしたら "Matched!" と表示
$0 ~ /reg/ { print "Matched!" }
```

条件式でも問題ありません。

```
# 行の長さが 0 の場合に "Nothing" と表示
length($0) == 0 { print "Nothing" }
```

パターンがない場合

パターンは真であることを前提に動作します。

```
# 全ての行を表示
{ print $0 }
```

アクションがない場合

'{ print $0 }' が省略されたものとして動作します。

```
# 正規表現 reg にマッチする行のみ表示
$0 ~ /reg/
```

特殊変数

$0

特に指定のない限り 1 行 (レコード) に該当します。

```
# 全ての行を表示
{ print $0 }
```

$1 ～ $NF

一行をスペースまたはタブで分解し、それぞれの項目 (フィールド) を行頭から $1, $2 の順で割り当てます。最後の要素は $NF に格納されます。

```
# 最初と最後の要素を表示
{ print $1, $NF }
```

NR, FNR

NR には現在処理している通し行数が格納され、FNR にはファイル単位での処理行数が格納されます。

```
# 最初の 10 行を表示
NR <= 10 { print $0 }
```

数値

数値の表現

```
num1 = 10;                              # 整数
num2 = 3.14;                            # 小数
num3 = 1e10;
```

四則演算

四則演算、べき乗ならびに括弧は通常の中置記法で記述します。

```
num = ((1 + 2) * 3 / 4) ^ 5;                    # 57.665
```

商と余り

```
num1 = 3;
num2 = 2;
# num1 = num2 * div + mod
# 商
div = int(num1 / num2);                    # 1
# 余り
mod = num1 % num2;                         # 1
```

インクリメントとデクリメント

インクリメントとデクリメントは以下のようにします。

```
# インクリメント (1)
i = i + 1;
# インクリメント (2)
i++;
# インクリメント (3)
++i;

# デクリメント (1)
i = i - 1;
# デクリメント (2)
i--;
# デクリメント (3)
--i;
```

算術関数

sin, cos, atan2, log, exp, sqrt の算術関数が使えます。

tan (タンジェント) は以下のようにして求めることができます。

```
tan = sin(num) / cos(num);
```

例えば、円周率は以下のようにして求めることができます。

```
pi = 2 * atan2(1, 0);                      # 3.14159
```

文字列

文字列の表現

ダブルクォーテーションで囲んで代入します。

```
str = "abc";
```

文字列の操作

連接

文字列を繋げるには、そのまま繋げます。

```
str1 = "abc";
str2 = str1 "def";                              # abcdef
```

配列、連想配列

awk の配列は全て連想配列（ハッシュ）です。

配列の代入

awk にはリストがないため、直接代入します。

```
arr[1] = 123;
arr[2] = "abc";
arr["item1"] = 123;
arr["item2"] = "abc";
```

連続している場合には、split 関数を用いて分割することも可能です。

```
# arr[1] = 1; arr[2] = 2; arr[3] = 3; arr[4] = 4; arr[5] = 5; と同じ
str = "1 2 3 4 5";
# num_arr には配列の個数（5）が代入される
num_arr = split(str, arr);
```

配列の個数

配列を for 文で呼び出してカウントすることで配列の個数を調べることができます。

```
for (index in arr) {
    num_arr++;                                  # 配列の個数が加算される
}
```

多次元配列

awk では配列を ","（コンマ）で区切ることで擬似的な多次元配列を扱えます。

```
arr[1, 2] = 123;
```

制御文

if 文

C 言語の if 文と基本的に同じ文法です。

```
# 奇数なら "Odd Number" と表示
if (num % 2 == 1) {
    print "Odd Number";
}
```

if ～ else 文

C 言語の if ～ else 文と基本的に同じ文法です。

```
# 奇数なら "Odd Number" と表示、偶数なら "Even Number" と表示
if (num % 2 == 1) {
    print "Odd Number";
} else {
    print "Even Number";
}
```

FizzBuzz 問題は以下のように書くことができます。

```
if (num % 15 == 0) {                    # 3 の倍数かつ 5 の倍数
    print "FizzBuzz";
} else if (num % 5 == 0) {               # 5 の倍数
    print "Buzz";
} else if (num % 3 == 0) {               # 3 の倍数
    print "Fizz";
} else {
    print num;
}
```

while 文

C 言語の while 文と基本的に同じ文法です。

```
# 0 から 10 まで表示
while (i <= 10) {
    print i++;
}
```

for 文

C 言語の for 文と基本的に同じ文法です。

```
# 0 から 10 まで表示
for (i = 0; i <= 10; i++) {
    print i++;
}
```

比較演算子

数値と文字列での扱いの違いはありません。

- == 等しい
- != 異なる
- <= 左辺が右辺以下
- < 左辺が右辺より小さい
- >= 左辺が右辺以上
- > 左辺が右辺より大きい
- ~ 正規表現マッチする
- !~ 正規表現マッチしない

条件演算子

条件演算子は以下のように使います。

```
# 奇数なら "Odd Number" と表示、偶数なら "Even Number" と表示
print (num % 2 == 1) ? "Odd Number" : "Even Number";
```

ユーザー定義関数

ユーザー定義関数を用いることができます。

```
# タンジェントを返す関数
function tan(num,    val) {

```

```
    val = sin(num) / cos(num);

    return val;
}
```

上記引数 num は関数自体の引数で、呼び出された時に使われていない引数 val は局所変数として振る舞います。

ファイルの入出力

通常ファイルの入力はパターン + アクションで行い、出力はリダイレクトを用います。

```
$ seq 10 | nawk 'NR % 2 { print $0 }' > odd_number.txt
```

プログラム中でもリダイレクトでき、"/dev/strout"（標準出力）, "/dev/stderr"（標準エラー出力）等の特殊デバイスも扱えます。

```
if (err_num > 0) {
    # 標準エラー出力に "Error" と表示
    print "Error" > "/dev/stderr";
} else {
    # 標準出力に "Not Error" と表示
    print "Not Error" > "/dev/stdout";
}
```

コマンドライン引数

コマンドライン引数は配列 ARGV に格納されます。

```
# コマンドライン引数を全て表示
BEGIN {
    for (i in ARGV) {
        print i, ARGV[i];
    }
}
```

良く使う表現

- Google search

- Menu
  - Home
  - AWK ならどう書く?
  - 関数・命令 Index
  - 今日の一行野郎
  - 人気ページ

  コミュニティ
  - メーリングリスト
  - イベント
  - お問い合わせ

  Links
  - 関連書籍
  - リンク集
  - One true awk
  - mawk
  - gawk
  - AwkChannelWiki
  - xgawk
  - Blis
-
  -
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  - 
  - 

-
- 

Theme adapted from Relaxation, a WordPress theme by John Wrana.