テキスト処理'15 ＃1 – テキスト処理の基本

久野　靖 *

2015.5.26

1　はじめに

テキスト処理 (text processing) とは、プログラムによってテキスト情報を処理することを言います。テキスト情報とは、文字の集まりから成るような情報のことです。しかし、コンピュータがもともと得意とするのは**数値処理** (numerical processing) で、今日のデータ分析や統計処理やデータマイニングなどでも、扱うのはおもに数値データなはずです。ではなぜ、テキスト処理が重要なのでしょうか?

それは、この世の中に存在しているデータの多くはテキスト情報だからです。本もデータシートも「文字で」情報が記されていますし、今日では重要な情報源となっている Web サイトも「文字の」情報が掲載されています。実際にはそれらの中に数値が書かれていることも多いわけですが、数値だけが書かれているということはまずなく、説明や見出しなどの文字も一緒に入っています。

ということは、これらのデータが**機械可読** (machine readable — 直ちにプログラムで処理できる状態) だったとしても、そこには沢山の文字情報が連なっていて、必要な数値は埋め込まれていることが多いわけです。また、場合によっては数値データはそもそもなくて、文字データから (誰が特許の出願者であるかなどの) 情報を抜き出してきて、それを数えたりして数値化しなければならないこともあります。

では、その数値化は人間がテキストから抜き出して打ち込むのでしょうか? ごく少量ならそれもいいでしょう。しかし大量データの場合、とてもそんな手間は掛けられません。大変ですし、間違いも多くなります。ですから、「プログラムによってテキストを処理して」必要なデータを取り出して来ることがどうしても必要になります。そのために、テキスト処理を学ぶことが必要なわけです。

本科目では、Ruby 言語を用いて、実践的なテキスト処理の方法を学んで行きます。Ruby は、まつもとゆきひろ氏 (matz さん) が作り出したプログラミング言語であり、主要なプログラミング言語の中で日本人が生み出したものとしては唯一の言語だといえます。ただし、ここで取り上げているのはそれが理由ではなく、簡便で、プログラムが書きやすく、文字列処理に適した機能やライブラリが揃っているためです。また、複数の OS で利用可能 (マルチプラットフォーム) であることも利点です。

プログラムを書くので、当然プログラミングの知識は必要になりますが、基本的なところから学びますので、関連科目「プログラミング」と並行して学んでいただければ大丈夫かと思います。内容としては、こちらはテキスト処理に関連する機能を多く扱うので、重複はあまりありません。

その性質上、本科目では時間中に「紙や黒板で」プログラムを書くという演習をやっていただきます。これをきちんとやらないと、自分の頭で考えてプログラムを書くという技能は身につきにくい、というのが我々の考えです。また、毎回「次回までの課題」をいくつか提示しますので、こちらも各自で解いて結果を `gssm.k-kiso` で報告して頂きます。これは、授業時間に演習をするだけで 1 週間ほっておいたら、翌週にはせっかく学んだことを忘れてしまうので、そのようにしています。

本科目の成績は上記の出席ならびに課題の結果に応じてつけることになります。ではこれから、よろしくお願いします。

*経営システム科学専攻

2　ファイル処理の基本

計算機内部のデータはファイルという単位で保管されていますね。そして、我々が普段読み書きするファイルの場合、その内容はテキストである、ということです。そこでテキスト処理の基本は、ファイルを読んだり書いたりする処理ということになります。

過去においては、ファイルを処理するときは「ファイルを 1 行ずつ読みながら、その 1 行ごとに対して処理をして行く」のが基本でした。しかし今ではメモリが豊富にあるので、ファイルが大きくない場合は「ファイルの内容を全部まず読み込んでしまい、それから処理を始める」方が楽です。ここでもまずその方法を扱います。[1]

最初の例題は「ファイルを読み込み、そのまま出力する」というものです。

```
def readdata
  r = []
  open('test1.txt') do |f|
    f.each_line do |s| r.push(s.chop) end
  end
  return r
end
def writedata(a)
  a.each do |s| puts(s) end
end

$data = readdata
writedata($data)
```

ひととおり説明しましょう。

- 「def 名前 ... end」はメソッド (手続き/関数/サブルーチン) を定義します。メソッドは定義がすんだら、名前を指定して呼び出すことができます。呼び出す際にパラメタ (引数とも呼ぶ) を渡すことができます。ここでは、パラメタ無しのメソッド readdata とパラメタありのメソッド writedata を定義しています。
- パラメタ無しのメソッド readdata は、test1.txt というファイルの内容を読み込み、配列 (一連の値が並んだデータ構造) として返します。
  - r = [] は空っぽの配列を変数 r に入れます。配列は値の並びを保持するデータ構造です。r.push(…) で末尾に値を追加していけます。
  - 「open('ファイル名') do |f| ... end」は、Ruby の特徴である「ブロックを渡すメソッド」を利用しています。do...end がブロックの範囲で、ここでは f というパラメタを受け取ります。open はファイルを読める状態にしてからそのファイルを読むための IO オブジェクトをブロックに渡してくれます。[2]
  - 「f.each_line do |s| ... end」というのは、IO オブジェクト f に対応するファイルから 1 行ずつ内容を呼んでそれをパラメタ s としてブロックに渡して繰り返し実行します。この中で s.chop により行末の改行文字を削除したものを配列 r に追加していくことで、ファイルの内容を配列にすべて読み込みます。
  - 「return r」は、メソッドから r の値を結果として返させます。
- writedata は a という名前のパラメタを受け取るメソッドです。a には配列を渡すことが想定されています (プログラムにはそう書かれていないけど)。

[1] ただし、処理対象とするテキストファイルのサイズがメモリに読み込める程度のサイズである場合に限ります。大きなファイルの場合については後の回で触れます。

[2] Ruby ではすべてのデータはオブジェクト (object、さまざまな機能が付随したデータのこと) であり、付随している機能 (メソッド) を呼び出すことで処理を行わせたり多様な結果を返してもらったりできます。

- 「a.each do |s| ... end」というのも (配列に対する) ブロックを渡すメソッド呼び出しです。このメソッドは、配列に入っているそれぞれの値ごとにその値をパラメタとして渡してブロックを (繰り返し) 呼び出すものです。
- 「puts(s)」は、文字列を標準出力に書き出します。普通に実行した場合、その文字列は画面に表示されます。

- その後、最後の 2 行がプログラム本体です。メソッドの外側で変数を定義するときは (グローバル変数)、$をつけた名前を使用してください。ここでは、readdata で読み込んだ配列を$data に入れ、直ちにそれを writedata で表示しています。[3]

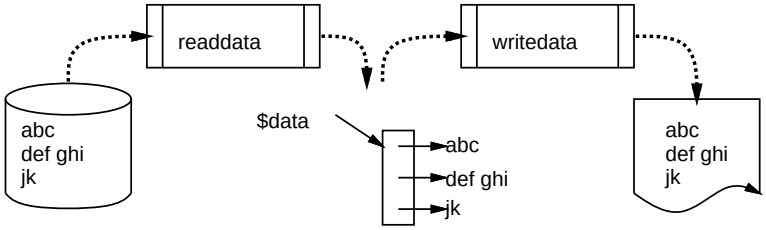


図 1: 最初の例題のデータの流れ

この例題のデータの流れを図 1 に示します。これを動かすには、次のようにしてください (もちろん、ファイル test1.txt も用意するのを忘れないように)。

(1) ファイルに上記のプログラムを入れる。たとえばそのファイル名を sam11.rb だとします。

(2) コマンド窓で「ruby sam11.rb」のように ruby コマンドを使うことで実行されます。

1 つ目の例題だけでは様子が分からないと思うので、もう 1 つ例題を示します。今度の例題は「データファイルが 2 行ずつで 1 件なので、その 2 行ずつを間に空白をはさんで 1 行に連結して出力する」というものです。ただし、行数が奇数個の場合は、最後の 1 行はそれ単独で出力します。[4]

これをやるためには、配列についてもう少し説明が必要なので説明します。まず a が配列のとき、入っているデータの数は a.size で参照できます。次に、「i 番目のデータ」を指定するには「a[i]」のように角かっこの中に番号を表す式を指定します。ただし、番号は 0 番から始めるので、たとえば a.size が 10 だったら、データは a[0], a[1], ..., a[9] に入っている、ということになります。a[10] を取ろうとすると空っぽを表す値 nil が入っているので注意。

次に、文字列の連結は「+」で表します。間に空白をはさむということなので、出力するのは a[0]+a[1]、a[2]+a[3]、…ということになります。先の例題で出て来た a.each は全部の行が順番に出て来るものだったのですが、今回はこのような (0、2、4…のような) 整数を順番に出してくるようなループが必要です。それには次のメソッドが使えます。

```
整数.step(上限, 増分) do |i| ...  end
```

このループでは、初期値が「整数」、そのあと「増分」ずつ増えて行く値が繰り返しブロックに渡され、上限を超えたらそこで終わります。たとえば「0.step(8, 2) とすれば「0、2、4、6、8」がブロックに渡されるループになるわけです (上限が 7 なら「0、2、4、6」で終わり)。では、これを使った例解を示します。

[3] 変数を使わず「writedata(readdata)」とすることもできます。ここでは 1 ステップずつの実行が分かりやすいように変数を使用しました。

[4] プログラミングではこのように境界部分のことをきちんと考えないと思わぬところで処理が失敗したりします。

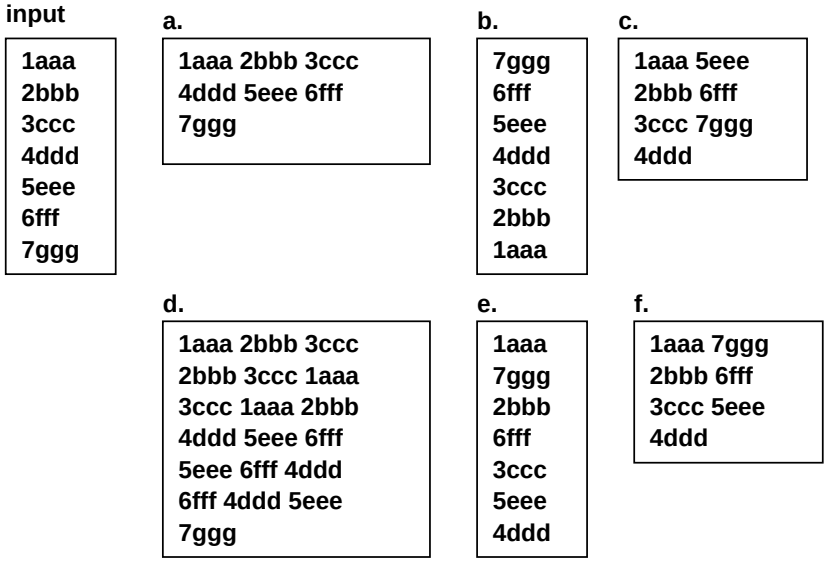
```
def readdata
  r = []
  open('test1.txt') do |f|
    f.each_line do |s| r.push(s.chop) end
  end
  return r
end
def writedata(a)
  a.each do |s| puts(s) end
end
def joinlines(a)
  r = []
  0.step(a.size-2, 2) do |i| r.push(a[i]+' '+a[i+1]) end
  if a.size % 2 > 0 then r.push(a[a.size-1]) end
  return r
end

$data1 = readdata
$data2 = joinlines($data1)
writedata($data2)
```

基本的な方針として、readdata、writedata は変更せず、読み込んだデータをもとにこれを加工するメソッド joinlines を追加し、これを使ってデータを加工してからその結果を書き出しています。

joinlines の中身ですが、空っぽの配列を変数 r に用意し、r.push(…) によって内容を追加して行き、最後にできあがった配列を返します。渡されたパラメタ a を処理するループについては、前述のように、a.size 番目は範囲を超えているので、1 つ手前を上限とします。しかしその後の if は何でしょうか? これは、データの数が奇数なら (%は「割った余り」を求める演算子)、最後の行を単独で出力する、という処理なのでした。

演習 1 2 つの例題をそのまま打ち込み、動かしてみて、動作を確認しなさい。

演習 2 次のようなデータの加工をおこなうメソッドを書きなさい (紙に書いて検討すること。実際に打ち込んで動かすのは後にする)。



a. 入力データを 3 行ずつくっつけるメソッド triplejoin。★

b. 入力データを前後反転するメソッド upsidedown。★

c. 入力データの前半分と後ろ半分をペアにしてくっつけるメソッド verticalpair。奇数の場合は前半が 1 行多くなり、その最後の行が単独のままになる。

d. 3 行ずつ組になりそれを巡回してくっつけた 3 行を出力する `rotate3`。最後に半端があればそれはそのまま。

e. 先頭、最後、先頭から 2 番目、最後から 2 番目、…のように順番に並べる `tobbottomalt`。

f. 先頭と最後、先頭から 2 番目と最後から 2 番目、…のように対にしてくっつける `topbottompair`。奇数なら中央の行は単独で。

演習 3 演習 2 で作成したプログラムを打ち込んで動かし、動作を確認しなさい。

3 パターンマッチと正規表現

前節では単に「何行目か」だけに基づいて行の並び替えや連結をしていましたが、当然ながら実際のデータ処理では「行の中身 (つまり文字列の内容)」も見ながら処理をするのが普通です。

では文字列の内容をどのようにして調べるのでしょうか。古典的なプログラミング言語だと、文字列に対してできることは「長さを調べる」「i 文字目を取り出す」「$i \sim j$ 文字目を取り出す」「文字 (列) が同じかどうかを調べる/(辞書順の) 大小を調べる」くらいで、これらを組み合わせて必要な処理をするのは結構大変でした。

これに対し、最近の言語では**正規表現** (regular expression) と呼ばれる形の**パターン**が使えるので、文字列の内容を調べるのがとても簡単になっています。具体的には、入力データがパターンにマッチするかどうかを判定できますし、マッチした時はついでにその必要な部分を抽出して以後の処理に使う、という形でデータの取り出しにも活用できます。Ruby では正規表現は「`/.../`」で囲んで表します (他の多くの言語でもそうです)。

```
if /パターン/ =~ 文字列      ←「=~」はマッチ演算子
    あてはまった場合の処理
else
    あてはまらなかった場合の処理
end
```

正規表現については計算機科学基礎でもやりましたが、簡単におさらいしておきましょう。正規表現 e は次のいずれかの形になります。

* c --- 通常の (特別な意味を持たない) 文字はその文字そのものにマッチ
* `.` --- 任意の 1 文字にマッチ
* `[`$c_1c_2c_3\ldots$`]` --- c_1, c_2, c_3 ... のいずれかにマッチ
* `[`c_1`-`c_2`]` --- コードが c_1 から c_2 の範囲の文字にマッチ
* `[^...]` --- ... で指定した文字群以外の文字にマッチ (文字クラス)
* e`*` --- e を 0 回以上繰り返したものにマッチ
* e`?` --- e を 0 回または 1 回繰り返したものにマッチ
* e`+` --- e を 1 回以上繰り返したものにマッチ
* e_1`|`e_2 --- e_1 または e_2 のいずれかにマッチ
* `^`e --- 行頭の e にマッチ
* e`$` --- 行末の e にマッチ
* `(`e`)` --- e にマッチし、なおかつマッチしたものを覚える

また、メタ文字列 (\+1 文字) も使用できます。

* \n --- 改行文字
* \r --- 復帰文字
* \s --- 空白文字全般 (文字クラス)
* \d --- 数字 (文字クラス)
* \w --- 英数字 (文字クラス)

なお、「[]|/()」などの特別な意味を持つ文字を普通の文字として使う場合は \ を前置します。\ を指定したい場合は「\\」です。

4 行内容のマッチに基づく処理

ではいよいよ、パターンマッチを使った処理の例を見ましょう。「継続行」機能を持つようなファイル形式があります。たとえば、行末に「\」があるとその行は次の行とつながっている (「\」と続く改行が無かったことになる)、というのが代表例です。たとえば次のようになります。

```
aaa bbb \          =>  aaa bbb ccc
ccc                    ddd eee fg
ddd ee\
e fg
```

このようなファイルを読み込み、継続行をくっつけて長い行にする処理を作成してみます。

```
def readdata
  r = []
  open('test2.txt') do |f|
    f.each_line do |s| r.push(s.chop) end
  end
  return r
end
def writedata(a)
  a.each do |s| puts(s) end
end
def contlines(a)
  r = []; l = ""
  a.each do |s|
    if s =~ /\\$/ then
      l += s.chop
    else
      l += s; r.push(l); l = ""
    end
  end
  if l != "" then r.push(l) end
  return r
end

$data1 = readdata
$data2 = contlines($data1)
writedata($data2)
```

contlines の中身ですが、変数 l は「現在作成中の行」に対応し、最初は空文字列です。渡された配列の各文字列について、「行末が\かどうか」を正規表現で判定しています。「\」だった場合は、その文字を削除した上で変数 l の後ろにくっつけます。そうでなかった場合は、そのまま l の後ろにくっつけ、l を r の末尾に追加してから空文字列にリセットします。なぜこれでいいか、よく考えてください。

ループが終わった後の if 文は何でしょうか。ループが終わって出て来たときに、l が空文字列でないのは、最後の行が「\」で終わっていた、ということです。この場合の処理はどうすべきか (次の行に続くはずなのにその次の行が無いので) 困るのですが、内容が失われるのも困りそうなので、残った l の内容を最後に追加しています。[5]

演習 4 上の例題をそのまま打ち込み、動かしてみて、動作を確認しなさい。

演習 5 次のようなデータの加工をおこなうメソッドを書きなさい (紙に書いて検討すること。実際に打ち込んで動かすのは後にする)。

a. 行の先頭が「#」になっている行をコメント行として削除する。★

b. 行の先頭が空白になっている行は前の行の続きとして前の行の後にくっつける。★[6]

c. a. のようにコメント行を削除しつつ、例題のように行末の「\」を次の行への継続として扱う。「\」で終わる行の後ろにコメント行があった場合、コメント行は無視し、次のコメントでない行をくっつけること。

d. a. のようにコメント行を削除しつつ、b. のように空白で始まる行を前の行の継続として扱う。コメント行の後に空白で始まる行があった場合はその行は通常の行として扱う (これにより、空白で始まる行が出力できるようになる)。

e. 数値の並びから成るファイルがあるので、その合計を求める。ただし a. のようなコメント行も含まれているので、それは無視する。

f. 数値の並びから成るファイルがあるので、その合計を求める。ただし a. のようなコメント行があったら、そこでそこまでの小計を出力し、小計は 0 とする。最後に残った小計と合計とを出力する。

演習 6 演習 5 で作成したプログラムを実際に動かせ。ただし、動かす前にテストデータと「そのテストデータを与えた場合どのような結果になるか (想定出力)」を用意し、動かした後想定出力との一致を確認すること。

5 第 1 回レポート課題

演習 5 の a〜f の問題のうちから 2 つ以上を選び、演習 6 を実行して結果を報告しなさい。各問題ごとに次の内容を含むこと。

- 学籍番号、氏名、提出日付 (これらは最初に 1 回でよい)
- 問題の再掲
- 作成したプログラムとその説明
- 予め用意したテストデータとその想定出力 (必要なだけ何組みでも全部)
- 実行結果と考察

とくに考察において、「十分なテストデータであるためには、どのような場合とどのような場合を含める必要があるか」をきちんと書くこと。レポート課題は次回授業開始時までに、`gssm.k-kiso` に投稿すること。ただし★がついている問題 (易しい問題) を含む場合はは本日中。

[5] ところで、このコードでは最後がこのような「\」で終わる行なのに無くなってしまう場合というのもあるのですが (ある意味ではバグ)、どういう場合か分かりますか。

[6] なお、文字列 s の 2 文字目以降を取り出すのは「`s[1..-1]`」でできます。