Eloquent Ruby by Russ Olsen,

「明解！Ruby」，ラス・オルセン著，(ピアソン桐原,2012).

第5章

正規表現で正しい文字列を見つける

問題を1つ提起したいと思います。文字列に格納されたテキストの中の09:24 AMのような時刻を探すプログラムを書く必要があるとします。どのように書きますか？ 前の章で説明したメソッドを使用して、数字がないか文字列をスキャンします。次にその後に続く別の数字、その後にコロン、さらに2つの数字が続いているか調べる…手順はよく分かっています。Stringメソッドでこれをすべて実行させることは、確かに可能です。可能ですが、面倒です。

問題が発生することもあります。この時刻検索問題は、間違ったツールを使用していることを教えようとしています。この問題の解決に必要な本当のツールは正規表現です。正規表現とはある文字列とマッチするまたはマッチしないパターンを作ることですが、作りがシンプルなほど強力です。残念ながら、正規表現は複雑であいまいだと思われていて、多くの技術者たちは正規表現を敬遠しています。しかし、正規表現を敬遠するのはまったく現実的ではありません。前の章で説明したように、Rubyプログラムはたくさんの文字列を処理しています。文字列のあるところには「正規表現あり」です。この章の前半では正規表現の基本を説明します。みなさんの正規表現アレルギーを解消してから、実際に正規表現をRubyコードで使用してみましょう。最後はいつも通りの実践に基づいた正規表現の使用例と避けるべきことで締めくくります。

1度に1文字のマッチング

正規表現について書かれた本はたくさんありますが[1]、この非常に便利なツールの基本は複雑ではありません。たとえば、正規表現の英字と数字は、実際の英字と数字にマッチします。例を示します。

- 正規表現xはxとマッチします。
- 正規表現aaaは連続する3個のaとマッチします。
- 正規表現123は最初の3個の数字とマッチします。
- 正規表現R2D2はあるSFロボットの名前とマッチします。

デフォルトでは、正規表現で大文字／小文字の区別は重要です。したがって最後の表現はr2d2もR2d2もマッチしません[2]。

英字や数字と異なり、ピリオド(.)やアスタリスク(*)のような句読文字は、正規表現で特別な意味を持ちます。たとえば、ピリオドすなわちドット文字は、任意の1文字とマッチします[3]。例を示します。

1) 付録に正規表現についての本を記載しています。参考にしてください。
2) 正規表現の大文字／小文字の区別はオフにできます。
3) 実際は、改行文字以外のすべての1文字です。先を読み続けてください。

- 正規表現での1個のピリオド(.)は、rおよび%および~を含む1文字とマッチします。
- 同様に、2個のピリオド(..)は、任意の2文字とマッチします。たとえば、xxや4Fだけでなく、[!ともマッチします。しかし、Qは1文字なので(2文字でない)マッチしません。

句読文字の特別な意味を打ち消すには、バックスラッシュを使用します。例を示します。

- 正規表現のバックスラッシュとドット(\.)は文字ドットとマッチします。
- 3\.14は、円周率(π) 3.14とマッチします。
- Mr\. OlsenはMr. Olsenとマッチします。

正規表現が美しいのは、さまざまな表現を組み合わせて、さらに複雑なマッチパターンを作り出せるところです。例を示します。

- 正規表現A.は、大文字のAで始まる任意の2文字とマッチします。たとえば、AM、An、At、さらにA=ともマッチします。
- 同様に、...Xは、末尾がXである任意の4文字とマッチします。たとえば、UVWXやXOOXとマッチします。
- 正規表現.r\. Smithは、Dr. SmithともMr. Smithともマッチしますが、Mrs. Smithとはマッチしません。

任意の1文字とマッチするというドットの機能がいかに正規表現の範囲を広げているかよくわかります。しかし。任意ではない何かとマッチさせたい場合はどうしたらよいでしょうか? 1文字の英字のみ、あるいは母音のみ、あるいは数字のみとマッチする表現を書きたい場合はどうしたらよいでしょうか?

セット、範囲、候補

セットを入力してください。セットは、一群の文字のどれか1つとマッチします。正規表現セットを作成するには、文字を角かっこで囲みます。したがって、正規表現[aeiou]は任意の1文字の小文字の母音とマッチします。ここでのキーワードは1文字です。[aeiou]は1文字のaまたは1文字のiとマッチしますが、aiとはマッチしません。aiが2文字だからです。同じ例を示します。

- 正規表現[0123456789]は任意の1文字の数字とマッチします。
- [0123456789abdef]は、任意の1文字の16進文字[4]、たとえば7やfとマッチします。

セットがどのように機能するか理解すれば、かなり複雑な正規表現を作ることができます。例を示します。

4) 英字の場合は小文字です。

- 正規表現[Rr]uss [Oo]lsenは、先頭が大文字であってもなくても、私の名前「Russ Olsen(russ olsen)」とマッチします。
- もっと実用的な例として、[0123456789abcdef][0123456789abcdef]は2桁の16進数とマッチします。たとえば、3eやffです。
- [aApP][mM]はamまたはPMとマッチします。また、中間のaMやPmなどにもマッチします。

シンプルなセットには問題が1つあります。入力が面倒なことです。誰も[0123456789abcdef]を入力したくないですし、ましてや[abcdefghijklmnopqrstuvwxyz]や[abcdefghijklmnopqrstuvwxyz0123456789]は間違わずに入力する自信がありません。しかし、この連続する文字で構成されたやっかいなセットのための特別な正規表現があります。それが範囲です。

範囲という名前が示すとおり、一連の文字の最初と最後をダッシュでつなぎ、指定することで範囲を定義します。したがって、範囲[0-9]は10進の数字(0から9)にマッチします。同様に[a-z]は小文字の英字(aからz)にマッチします。複数の範囲を組み合せたり、通常のセット表記と混在させることもできます。

- [0-9abcdef]は1個の16進文字にマッチします。
- [0-9a-f]も1個の16進文字にマッチします。
- [0-9a-zA-Z_]は、任意の英字(大文字、小文字)、数字、またはアンダースコア文字にマッチします。

[0-9]でもまだ面倒だと思うならば、一般的なセットのショートカットがあります。

- \dは任意の数字にマッチします。したがって\d\dは00から99の2桁の数字にマッチします。
- \wは、「word character」を表し、任意の英字、数字、またはアンダースコアにマッチします。
- \sは、普通のスペースを含む任意のホワイトスペース、タブ、改行にマッチします。

正規表現のパワーを拡張する別な方法は、選択肢の使用です。バーティカルバーは「or」を意味するため、バーティカルバーの前の部分または後の部分のいずれかにマッチします。

- A|Bは、AまたはBのどちらかにマッチします。
- AM|PMは、AMまたはPMのどちらかにマッチします。
- Batman|Spidermanは、2人のスーパーヒーローのどちらかの名前にマッチします。

候補には限界はありません。好きなだけ選択肢を指定できます。したがって、A\.M\.|AM|P\.M\.|PMは、A.M.またはAMまたはP.M.またはPMにマッチします。候補をかっこで囲み、パターンの他の部分と切り離して指定することもできます。

```
The (car|boat) is red
```

この例は、The car is redにもThe boat is redにもマッチします。
これまでに説明した正規表現を使用すれば、最初に提示した時刻を見つける問題は解決します。

```
\d\d:\d\d (AM|PM)
```

上の表現を翻訳すると、「2桁の数値で始まり、その後にコロンとさらに2桁の数字が続き、1個の空白があり、AMまたはPMのいずれかが続く文字列」を表しています。

正規表現のアスタリスク

ここまでに正規表現の基礎を説明しました。次にアスタリスクについて説明しますが、正規表現のアスタリスクは大変興味深い表現です。正規表現のアスタリスク（*）は、その直前の記述の0回以上の繰り返しにマッチします。アスタリスクの直前にある記述の0回以上とは・・・ちょっと考えてみましょう。たとえばA*であれば0個以上のAにマッチするということになりますね。次に例を見てみましょう。

- AB*はABとマッチします。すなわち、Aの後に1個のBが続いています。
- AB*はABBやABBBBBBBBにマッチします。1個のAの後に任意の数のBが続いています。
- AB*は、ただのAともマッチすることを忘れないでください。0個以上のBとはBがないことも含まれるからです。

ここまでの例では、アスタリスクは表現の末尾にありましたが、置く場所はどこでもかまいません。たとえばR*ubyは任意の数のRの後にubyが続くことを意味します。したがって、uby、Ruby、RRuby、RRRRRRRubyのどれにもマッチします。Rub*yの場合は、Ruy、Ruby、Rubbbbbbbbbyなどにマッチします。使用するアスタリスクは数に制限がありません。したがって、R*u*byもR*u*b*yも問題ありません。ちなみにR*u*byにマッチするのは、任意の数のRの後に任意の数のuが続き、その後にbyが続く記述です。
アスタリスクはセットと組み合わせて使用することもできます。

- [aeiou]*は任意の数の母音にマッチします。[aeiou]セット全体がアスタリスクの指示する繰り返しの対象です。
- 同様に[0-9]*は任意の数の数字にマッチします。
- [0-9a-f]*は任意の数の16進文字にマッチします。

最後に、ドットとアスタリスクの組合せは、すべての正規表現の中で最も広く使用される表現です。

```
.*
```

ドットとアスタリスクだけのこの小さな宝石は、任意の数の任意の文字、言い換えるとすべてのものとマッチします。つまり、アスタリスクはドットを0回以上繰り返し、ドットは任意の1文字を表わすので、.*はすべてのものにマッチすることになります。

5) Regexp

.*を他の正規表現と組み合わせると、非常に柔軟なパターンが作成できます。次に例を示します。

- 正規表現George.*は、ファーストネームがGeorgeであるすべてのフルネームにマッチします。
- 一方、.*Georgeはラストネームが Georgeであるすべての名前にマッチします。
- 最後に、.*George.*は名前の中のどこかにGeorgeを含むすべての名前にマッチします。

Rubyでの正規表現

Rubyでは、正規表現、あるいは短縮形Regexp[5]は、組み込みデータタイプの1つで、独自の特別なリテラル構文を持ちます。Rubyの正規表現を作成するには、パターンをスラッシュではさみます。したがって、前述の時刻を探す正規表現は次のようになります。

```
/\d\d:\d\d (AM|PM)/
```

正規表現が文字列にマッチするかどうかをテストするには、=~演算子を使用します。したがって、上の正規表現を実際の時刻にマッチさせるには、次のコードを実行します。

```
puts /\d\d:\d\d (AM|PM)/ =~ '10:24 PM'
```

次のように出力されます。

```
0
```

このゼロには、ふたつの意味があります。まず、nilではないので正規表現がマッチしたことを示しています。また、正規表現をマッチさせるときに、文字列をスキャンしてマッチした場所も示しています。次の例でスキャンの動作を見ることができます。

```
puts /PM/ =~ '10:24 PM'
```

このコードを実行すると、次のように出力されます。

```
6
```

6は、Rubyが文字列をスキャンした後でようやく、正規表現がマッチしたことを示しています。マッチしない場合は、nilが返されます。

```
/May/ =~ 'Sometime in June'
```

上の例はnilを返します。=~は、マッチが見つかったときは数値を、見つからないときはnilを返すため、正規表現マッチをブールとして使用することができます。

5) RegexpはRuby正規表現クラスの名前です。

```
the_time = '10:24 AM'
puts "It's morning!" if /AM/ =~ the_time
```

=~演算子は両手利きでもあります。文字列と正規表現のどちらが先にくるかは問いません。したがって、最後の例は次のように書き換えることができます。

```
puts "It's morning!" if '10:24 AM' =~ /AM/
```

前述したように、正規表現はデフォルトで大文字／小文字を区別します。/AM/は「am」とはマッチしません。大文字／小文字の区別をオフにするには、正規表現の末尾にiを置きます。次のようになります。

```
puts "It matches!" if /AM/i =~ 'am'
```

何かが出力されます。

正規表現と=~演算子は基本的に単体で使用するほかに、正規表現が検索に使用する文字列のメソッドに関与することを覚えておいてください。たとえば、gsubメソッドに正規表現を渡して、ドキュメントに含まれる時刻をすべて伏字にすることができます。

```
class Document
  # 大部分のクラスを省略・・・

  def obscure_times!
    @content.gsub!(/\d\d:\d\d (AM|PM)/, '**:** **' )
  end
end
```

文字列と正規表現を統合すると、非常に強力なテキスト処理ツールキットになります。

先頭と末尾

=~演算子は文字列のどこでもスキャンするのであれば、正規表現を文字列の先頭でのみマッチさせたい場合はどのようにしたらよいでしょうか？ たとえば、「Once upon a time」という言葉を探すには？ どちらも問題ありません。普通の正規表現で解決できます。

```
/Once upon a time/
```

しかし、この定型句で始まるおとぎ話を探す場合はどうしたらよいでしょうか？ この場合も、そのための特別な正規表現があります。\Aです。例を示します。

```
/\AOnce upon a time/
```

これは、文字列がおとぎ話のように始まればマッチします。\Aは、文字列の先頭とマッチしますが、文字列の先頭が非表示文字（改行文字などの特殊文字）の場合も検索対象となることに注意してください。同様に、\z(小文字である点に注意)は文字列の末尾とマッチします。

```
/and they all lived happily ever after\z/
```

これは、古くからおとぎ話の結び「そしていつまでも幸せに暮らしましたとさ」のフレーズにのみマッチします。

埋め込まれた複数の改行を持つ複数行文字列では、このような操作の際に、注意する問題がいくつかあります。文章が次のように複数行で格納されているとします。

```
content = 'The Princess And The Monkey

Once upon a time there was a princess...
...and they all lived happily ever after.

The End'
```

この文章がおとぎ話であることはわかっていますが、そのことがわかる正規表現はどのように書いたらよいでしょうか？　前の例の/\AOnce upon a time/は、上の文字列が「Once upon a time」で始まっていないので機能しません。今回、検索する文字列は、文字列の内部のフレーズです。このような場合に使用する正規表現に、サーカムフレックス（^）があります。サーカムフレックス文字は、文字列の先頭または文字列の中の任意の行の先頭とマッチします。次のようにします。

```
puts "Found it" if content =~ /^Once upon a time/
```

同様に、ドル記号$は文字列の末尾または文字列内の任意の行の末尾とマッチするので、次のように書くことができます。

```
puts "Found it" if content =~ /happily ever after\.$/
```

複数行文字列にはもう1つの問題があります。それはドットです。デフォルトでは、ドットは改行文字以外の任意の文字とマッチします。タイトルとThe Endを除いたおとぎ話のテキスト全体とマッチさせるために、次のようにするとどうなるでしょう。

```
/^Once upon a time.*happily ever after\.$/
```

この場合、.*は行をまたいでマッチしないため絶望的です。しかし方法はあります。mを追加するだけで、この動作をオフにできます。

```
/^Once upon a time.*happily ever after\.$/m
```

そして、いつまでも幸せに暮らしましたとさ。めでたし、めでたし。

実践

Rubyをインストールすれば、それに同梱されているtime.rbファイルで、正規表現の実用的な使い方のすばらしい例を見ることができます。time.rbのコードは、各種の標準フォーマット[6]で表された日時を含む文字列を解析し、それをTimeクラスのインスタンスに変換します。この章で見てきたシンプルな例とは異なり、time.rbでは、実際の複雑な日付と時刻を扱う必要があります。ファイルの前のほうにzone_offsetメソッドがあります。これは、日付と時刻の文字列のタイムゾーンセクションを解析して、文字列がいったいどこの12:01 AMを表しているのかを調べます。特にこのメソッドは'UTC'のような名前、または'-07:00'のような数値オフセットで表されるタイムゾーンを理解できなければなりません。

zone_offsetメソッドは操作が複雑にならないように、最初にzone.upcaseでゾーン文字列全体を大文字に変換します。

```
def zone_offset(zone, year=self.now.year)
  # ...
  zone = zone.upcase
  # ...
```

次に、ゾーンを多数の正規表現でテストし、形式の1つにマッチするかどうか調べます。

```
if /\A([+-])(\d\d):?(\d\d)\z/ =~ zone
```

上記の表現は、説明していない疑問符という正規表現機能を使用しています。疑問符は、その直前にある記述の0個または1個とマッチします。

したがって、:?により、正規表現はゾーンオフセットにコロンがあってもなくてもマッチします。

正規表現がマッチしない場合、zone_offsetメソッドは、ZoneOffsetコレクションに格納されている'GMT'や'EST'を探すための文字列比較処理を開始します。

```
elsif ZoneOffset.include?(zone)
```

zone_offsetメソッドは、うんざりする複雑な問題に対して非常に素晴らしいアプローチをします。正規表現とシンプルな文字列メソッドを上手に使い分けし、それぞれが一番良い状態で機能する場合に適宜使用します。

トラブルの回避

正規表現を使用したときに誰もが犯すミスがありますが、このミスは簡単に防止することができます。まず、例を見てみましょう。

```
puts /abc*/ == "abccccc"
```

6) Timeオブジェクトから文字列への変換、また逆方向の変換もできます。

このコードのミスは、思考と入力を指任せにした結果です。正規表現/abc*/は、少なくとも==演算子に従えば、"abccccccc"と等しいということはあり得ません。正規表現のマッチ演算子は=~であり、==ではないことを思い出してください。これを正しいコードで書くと次のようになります。

```
puts /abc*/ =~ "abcccccc"
```

ばかばかしいミスの2つ目は、特にCやC++の素養[7]がある場合に犯しやすいのですが、最後の例の出力に対する反応です。

```
0
```

これを見てマッチしなかったと思い込むことです。C++プログラマーには、ゼロは常にネガティブに見えます。この結果は、文字列の先頭、すなわち0番目のインデックスで始まる文字列に正規表現がマッチしたことを意味しています。正規表現マッチではマッチしない場合、nilが返されることを思い出してください。

まとめ

面倒な文字列処理に直面したときの唯一の解決策は、正規表現です。アスタリスク、ドル記号、サーカムフレックスを含む正規表現は、手ごわく見えますが、基本的考え方はどのプログラマーにも十分に理解できます。

7) 私自身がそうなので、自分が言うことのばかさ加減がわかります。