
正規表現 HOWTO

リリース 3.9.19

Guido van Rossum
and the Python development team

5月 01, 2024

目次

1	はじめに	2
2	単純なパターン	3
2.1	文字のマッチング	3
2.2	繰り返し	4
3	正規表現を使う	6
3.1	正規表現をコンパイルする	6
3.2	バックスラッシュ感染症	6
3.3	マッチの実行	7
3.4	モジュールレベルの関数	10
3.5	コンパイルフラグ	10
4	パターンの能力をさらに	13
4.1	さらなる特殊文字	13
4.2	グルーピング	14
4.3	取り出さないグループと名前つきグループ	16
4.4	先読みアサーション (Lookahead Assertions)	18
5	文字列を変更する	19
5.1	文字列の分割	20
5.2	検索と置換	21
6	よくある問題	22
6.1	文字列メソッドを利用する	23
6.2	match() 対 search()	23

6.3	貪欲 (greedy) 対非貪欲 (non-greedy)	24
6.4	re.VERBOSE の利用	25
7	フィードバック	25

著者 A.M. Kuchling <amk@amk.ca>

概要

このドキュメントは `re` モジュールを使って Python で正規表現を扱うための導入のチュートリアルです。ライブラリレファレンスの正規表現の節よりもやさしい入門ドキュメントを用意しています。

1 はじめに

正規表現 regular expressions (REs や regexes または regex patterns と呼ばれます) は本質的に小さく、Python 内部に埋め込まれた高度に特化したプログラミング言語で `re` モジュールから利用可能です。この小さな言語を利用することで、マッチさせたい文字列に適合するような文字列の集合を指定することができます; この集合は英文や e-mail アドレスや TeX コマンドなど、どんなものでも構いません。「この文字列は指定したパターンにマッチしますか?」「このパターンはこの文字列のどの部分にマッチするのですか?」といったことを問い合わせることができます。正規表現を使って文字列を変更したりいろいろな方法で別々の部分に分割したりすることもできます。

正規表現パターンは一連のバイトコードとしてコンパイルされ、C で書かれたマッチングエンジンによって実行されます。より進んだ利用法では、エンジンがどう与えられた正規表現を実行するかに注意することが必要になり、高速に実行できるバイトコードを生成するように正規表現を書くことになります。このドキュメントでは最適化までは扱いません、それにはマッチングエンジンの内部に対する十分な理解が必要だからです。

正規表現言語は相対的に小さく、制限されています、そのため正規表現を使ってあらゆる文字列処理作業を行なえるわけではありません。正規表現を使って行うことのできる作業もあります、ただ表現はとても複雑なものになります。それらの場合では、Python コードを書いた方がいいでしょう; Python コードは念入りに作られた正規表現より遅くなりますが、おそらくより読み易いでしょう。

2 単純なパターン

まずはできるだけ簡単な正規表現を学ぶことから始めてみましょう。正規表現は文字列の操作に使われるので、まずは最も一般的な作業である文字のマッチングをしてみます。

正規表現の基礎を成す計算機科学 (決定、非決定有限オートマトン) の詳細な説明については、コンパイラ作成に関するテキストブックをどれでもいいので参照して下さい。

2.1 文字のマッチング

多くの活字や文字は単純にそれ自身とマッチします。例えば、`test` という正規表現は文字列 `test` に厳密にマッチします。(大文字小文字を区別しないモードでその正規表現が `Test` や `TEST` にも同様にマッチすることもできます; 詳しくは後述します。)

この規則には例外が存在します; いくつかの文字は特別な **特殊文字** (*metacharacters*) で、それら自身にマッチしません。代わりに通常のマッチするものとは違うという合図を出したり、正規表現の一部に対して繰り返したり、意味を変えたりして影響を与えます。このドキュメントの中の多くは様々な特殊文字とそれが何をするかについて論じることになります。

ここに特殊文字の完全な一覧があります; これらの意味はこの HOWTO の残りの部分で説明します。

`. ^ $ * + ? { } [] \ | ()`

最初に扱う特殊文字は `[]` です。これらは文字クラスを指定します、文字クラスはマッチしたい文字の集合です。文字は個別にリストにしても構いませんし、二つの文字を `-` でつなげて文字を範囲で与えてもかまいません。たとえば `[abc]` は `a`, `b`, または `c` のどの文字列にもマッチします; これは `[a-c]` で同じ文字集合を範囲で表現しても全く同じです。小文字のアルファベットのみにマッチしたい場合、`[a-z]` の正規表現をつかうことになるでしょう。

Metacharacters (except `\`) are not active inside classes. For example, `[akm$]` will match any of the characters `'a'`, `'k'`, `'m'`, or `'$'`; `'$'` is usually a metacharacter, but inside a character class it's stripped of its special nature.

補集合を取る ことで、文字クラス内のリストにない文字に対してマッチさせられます。補集合は、クラスの最初の文字として `^` を含めることで表せます。例えば、`[^5]` は `'5'` 以外の文字にマッチします。キャレットが文字クラス以外の場所に現れた場合は、特別な意味は持ちません。例えば、`[5^]` は `'5'` や `^` にマッチします。

おそらく最も重要な特殊文字はバックスラッシュ `\` でしょう。Python の文字列リテラルのようにバックスラッシュに続けていろいろな文字を入力することでいろいろな特殊シーケンスの合図を送ることができます。また、バックスラッシュはすべての特殊文字をエスケープするのにも利用されます、つまり、特殊文字をマッチさせることができます; 例えば、`[または \ にマッチさせたい場合、それらをバックスラッシュに続けることで特殊な意味を除きます: \[または \\ 。`

'\' で始まるいくつかの特殊シーケンスは、数字、アルファベット、空白文字以外など、よく使う文字集合を表しています。

一つ例をお見せしましょう: `\w` は任意の英数字文字にマッチします。バイト列パターンに対しては、これは文字クラス `[a-zA-Z0-9_]` と等価です。ユニコードパターンに対しては、`\w` は `unicodedata` モジュールで提供されている Unicode データベースで `letters` としてマークされている全ての文字とマッチします。正規表現のコンパイル時に `re.ASCII` フラグを与えることにより、`\w` を、より制限された定義で使うことが出来ます。

以下に続く特別な文字列のリストは完全ではありません。特殊シーケンスと拡張クラスについてのユニコードパターンの定義についての完全なリストは、標準ライブラリリファレンスの 正規表現の構文 の最後の部分を参照してください。一般的にユニコードバージョンは、ユニコードデータベース内で相応しいカテゴリに属すればマッチします。

`\d` 任意の十進数とマッチします; これは集合 `[0-9]` と同じ意味です。

`\D` 任意の非数字文字とマッチします; これは集合 `[^0-9]` と同じ意味です。

`\s` 任意の空白文字とマッチします; これは集合 `[\t\n\r\f\v]` と同じ意味です。

`\S` 任意の非空白文字とマッチします; これは集合 `[^\t\n\r\f\v]` と同じ意味です。

`\w` 任意の英数文字および下線とマッチします; これは、集合 `[a-zA-Z0-9_]` と同じ意味です。

`\W` 任意の非英数文字とマッチします; これは集合 `[^a-zA-Z0-9_]` と同じ意味です。

これらのシーケンスは文字クラス内に含めることができます。例えば、`[\s,.]` は空白文字や `,` または `.` にマッチする文字クラスです。

この節での最後の特殊文字は `.` です。これは改行文字を除く任意の文字にマッチし、さらに改行文字に対してもマッチさせる代替モード (`re.DOTALL`) があります。`.` は「任意の文字」にマッチさせたい場合に利用されます。

2.2 繰り返し

さまざまな文字集合をマッチさせることは正規表現で最初に行えるようになることで、これは文字列に対するメソッドですぐにできることではありません。しかし、正規表現がより力を発揮する場面がこれだけだとすると、正規表現はあまり先進的とはいえません。正規表現の力をもう一つの能力は、正規表現の一部が何度も繰り返されるようなものを指定できることです。

最初にとりあげる繰り返しのための最初の特特殊文字は `*` です。`*` は文字リテラル `'*'` とはマッチしません; その代わりに、前の文字がぴったり 1 回ではなく 0 回以上繰り返されるパターンを指定します。

例えば、`ca*t` は `'ct'` (文字 `'a'` が 0 個)、`'cat'` (`'a'` が 1 個)、`'caaat'` (`'a'` が 3 個)、などにマッチします。

`*` のような繰り返しは 貪欲 (*greedy*) です; 正規表現を繰り返したいとき、マッチングエンジンは可能な限り何度も繰り返そうと試みます。パターンの後ろの部分にマッチしない場合、マッチングエンジンは戻ってより少ない繰り返しを再び試みます。

一歩ずつ例を進めていくとより明確にわかります。正規表現 $a[bcd]^*b$ を考えましょう。この正規表現は文字 'a'、文字クラス $[bcd]$ の 0 個以上の文字、最後に来る 'b' にマッチします。この正規表現が文字列 'abcbd' に対してマッチする流れを想像してみましょう。

ステップ	マッチした文字列	説明
1	a	正規表現の a がマッチ。
2	abcbd	正規表現エンジンが、文字列の終わりに向かってできるだけ遠くまで $[bcd]^*$ をマッチさせる。
3	失敗	正規表現エンジンが b でマッチを試みるが、現在の位置が文字列の最後なので失敗。
4	abcb	$[bcd]^*$ が一文字少なくマッチするように戻る。
5	失敗	再び b にマッチするか試みるが、現在の位置は最後の文字 'd' 。
6	abc	$[bcd]^*$ は bc のみにマッチするように再び戻る。
6	abcb	再び b を試みる。今回の現在位置の文字は 'b' なので成功。

正規表現の終端に達して、'abcb' にマッチしました。この説明は、マッチングエンジンが最初に到達できるところまで進みマッチしなかった場合、逐次戻って再度残りの正規表現とのマッチを次々と試みる様子を示しています。正規表現エンジンは $[bcd]^*$ の 0 回マッチを試すところまで戻り、その後続の正規表現とのマッチに失敗した場合には、エンジンは正規表現と文字列が完全にマッチしないと結論づけることになります。

別の繰り返しのメタ文字には + があり、この特殊文字は 1 回以上の繰り返しにマッチします。* と + に違いに対しては十分注意して下さい; * は 0 回 以上の繰り返しにマッチするので、繰り返す部分が全くなくても問題ありません。一方で + は少なくとも 1 回 は表われる必要があります。同様の例を使うと $ca+t$ は 'cat' ('a' 1 文字)、'caaat' ('a' 3 文字)、とマッチし、'ct' とはマッチしません。

2 回以上の繰り返しを制限する修飾子も存在します。クエスチョンマーク ? は 0 か 1 回のどちらかにマッチします; これはオプションな何かを示しているとも考えられます。例えば、 $home-?brew$ は 'homebrew' と 'home-brew' のどちらにもマッチします。

最も複雑な繰り返しの修飾子は $\{m,n\}$ で、ここで m と n は 10 進整数です。この修飾子は最低 m 回、最大で n 回の繰り返すことを意味しています。例えば、 $a/\{1,3\}b$ は 'a/b' と 'a//b' そして 'a///b' にマッチし、スラッシュの無い 'ab' や 4 つのスラッシュを持つ 'a////b' にはマッチしません。

m か n のどちらかは省略することができます; その場合は、省略された値は合理的な値が仮定されます。 m の省略は下限は 0 と解釈され、 n の省略は上限は無限として解釈されます。

還元主義者傾向のある読者は、3 つの修飾子がこの表記を使って表現できることに気付くでしょう。 $\{0, \}$ は * と、 $\{1, \}$ は + と、そして $\{0, 1\}$ は ? と同じです。できるなら *, +, ? を利用した方が賢明です。それは単に、短く読み易くなるからです。

3 正規表現を使う

これまででいくつかの単純な正規表現に触れてきました、実際に Python ではこれらをどう使えばいいのでしょうか？ `re` モジュールは正規表現エンジンに対するインターフェースを提供していて、それらを使うことで正規表現をオブジェクトにコンパイルし、マッチを実行することができます。

3.1 正規表現をコンパイルする

正規表現はパターンオブジェクトにコンパイルされます、パターンオブジェクトは多くの操作、パターンマッチの検索や文字列の置換の実行などのメソッドを持っています。

```
>>> import re
>>> p = re.compile('ab*')
>>> p
re.compile('ab*')
```

`re.compile()` はいくつかの *flags* 引数を受け付けることができます、この引数はさまざまな特別な機能を有効にしたり、構文を変化させたりします。利用できる設定に何があるかは後に飛ばすことにして、簡単な例をやることにしましょう:

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

正規表現は文字列として `re.compile()` に渡されます。正規表現は文字列として扱われますが、それは正規表現が Python 言語のコアシステムに含まれないためです、そのため正規表現を表わす特殊な構文はありません。(正規表現を全く必要としないアプリケーションも存在します、そのためそれらを含めて言語仕様を無駄に大きくする必要はありません) その代わりに、`re` モジュールは `socket` や `zlib` モジュールのような通常の C 拡張モジュールとして Python に含まれています。

正規表現を文字列としておくことで Python 言語はより簡素に保たれていますが、そのため 1 つの欠点があります、これについては次の節で話題とします。

3.2 バックスラッシュ感染症

先に述べたように、正規表現は特別な形式や特殊な文字の特別な意味を意味を除くことを示すためにバックスラッシュ文字 (`'\'`) を利用します。これは Python が文字列リテラルに対して、同じ文字を同じ目的で使うことと衝突します。

`\section` という文字列 (これは LaTeX ファイルでみかけます) にマッチする正規表現を書きたいとします。どんなプログラムを書くか考え、マッチして欲しい文字列をはじめに考えます。次に、バックスラッシュや他の特殊文字をバックスラッシュに続けて書くことでエスケープしなければいけません、その結果 `\\section` のような文字列となります。こうしてできた `re.compile()` に渡す文字列は `\\section` でなければいけません。しかし、

これを Python の文字列リテラルとして扱うにはこの二つのバックスラッシュを **再び** エスケープする必要があります。

文字	段階
<code>\section</code>	マッチさせるテキスト
<code>\\section</code>	<code>re.compile()</code> のためのバックスラッシュエスケープ
<code>"\\\\section"</code>	文字列リテラルのためのバックスラッシュエスケープ

要点だけをいえば、リテラルとしてのバックスラッシュにマッチさせるために、正規表現文字列として `'\\\\'` と書かなければいけません、なぜなら正規表現は `\\` であり、通常の Python の文字列リテラルとしてはそれぞれのバックスラッシュは `\\` で表現しなければいけないからです。正規表現に関してこのバックスラッシュの繰り返しの機能は、たくさんのバックスラッシュの繰り返しを生むことになり、その結果として作られる文字列は理解することが難しくなります。

この問題の解決策としては正規表現に対しては Python の raw string 記法を使うことです; `'r'` を文字列リテラルの先頭を書くことでバックスラッシュは特別扱いされなくなります、つまり `"\n"` は改行を含む 1 つの文字からなる文字列であるのに対して、`r"\n"` は 2 つの文字 `'\'` と `'n'` を含む文字列となります。多くの場合 Python コードの中の正規表現はこの raw string 記法を使って書かれます。

それに加えて、正規表現では有効であるものの Python の文字列リテラルとしては有効でない特殊文字のエスケープシーケンスは、現在では `DeprecationWarning` を引き起こし、最終的には `SyntaxError` となります。すなわち、そのようなシーケンスは raw string 記法を使うか、バックスラッシュによるエスケープを使わないかぎり無効になることを意味します。

通常の文字列	Raw string
<code>"ab*"</code>	<code>r"ab*"</code>
<code>"\\\\section"</code>	<code>r"\\section"</code>
<code>"\\w+\\s+\\1"</code>	<code>r"\\w+\\s+\\1"</code>

3.3 マッチの実行

一旦コンパイルした正規表現を表現するオブジェクトを作成したら、次に何をしますか? パターンオブジェクトはいくつかのメソッドや属性を持っています。ここでは、その中でも最も重要なものについて扱います; 完全なリストは `re` ドキュメントを参照して下さい。

メソッド/属性	目的
<code>match()</code>	文字列の先頭で正規表現とマッチするか判定します。
<code>search()</code>	文字列を操作して、正規表現がどこにマッチするか調べます。
<code>findall()</code>	正規表現にマッチする部分文字列を全て探しだしリストとして返します。
<code>finditer()</code>	正規表現にマッチする部分文字列を全て探しだし iterator として返します。

`match()` と `search()` はマッチするものが見つからなければ `None` を返します。成功すればそれらは `Match` オブジェクト のインスタンスを返します。このオブジェクトにはマッチした情報が含まれます: マッチの開始と終了位置、マッチした部分文字列、など。

`re` モジュールで対話的に実験することで学ぶこともできます。`tkinter` が利用できれば、Python に含まれるデモプログラム `Tools/demo/redemo.py` を見るといいかもしれません。このデモは正規表現と文字列を入力し、正規表現がマッチしたかどうかを表示します。`redemo.py` は複雑な正規表現のデバッグを試みる時にも便利に使うことができます。

この HOWTO では例として標準の Python インタプリタを使います。最初に Python インタプリタを起動して、`re` モジュールをインポートし、正規表現をコンパイルします:

```
>>> import re
>>> p = re.compile('[a-z]+')
>>> p
re.compile('[a-z]+')
```

さて、いろいろな文字列を使って正規表現 `[a-z]+` に対するマッチングを試してみましょう。空の文字列は全くマッチしません、なぜなら `+` は「1 回以上の繰り返し」を意味するからです。この場合では `match()` は `None` を返すべきで、インタプタは何も出力しません。明確にするために `match()` の結果を明示的に出力することもできます:

```
>>> p.match("")
>>> print(p.match(""))
None
```

では、今度はマッチするはずの文字列、例えば `tempo` を試してみましょう。このケースでは、`match()` は `match object` を返すので、後で使うために結果を変数に記憶しておくべきです。

```
>>> m = p.match('tempo')
>>> m
<re.Match object; span=(0, 5), match='tempo'>
```

これでマッチした文字列についての情報を `Match` オブジェクト に問い合わせることが出来ます。`Match` オブジェクトインスタンスはいくつかのメソッドと属性も持っていて、最も重要なのは次のものです:

メソッド/属性	目的
<code>group()</code>	正規表現にマッチした文字列を返す
<code>start()</code>	マッチの開始位置を返す
<code>end()</code>	マッチの終了位置を返す
<code>span()</code>	マッチの位置 (<code>start</code> , <code>end</code>) を含むタプルを返す

これらのメソッドを試せば、その意味はすぐに理解できます:

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

`group()` は正規表現でマッチした部分文字列を返します。`start()` と `end()` はそれぞれ、マッチの開始インデクスと終了インデクスを返します。`span()` は開始と終了のインデクスを一つのタプルにして返します。`match()` メソッドは正規表現が文字列の開始位置でマッチするかどうかだけをチェックするので、`start()` は必ずゼロを返します。しかし、`search()` メソッドではパターンを文字列全体について走査するので、マッチの開始はゼロにならないかもしれません。

```
>>> print(p.match('::: message'))
None
>>> m = p.search('::: message'); print(m)
<re.Match object; span=(4, 11), match='message'>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

実際のプログラムでは Match オブジェクト を変数に記憶しておき、その次に `None` なのか調べるのが一般的なスタイルです。普通このようにします:

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print('Match found: ', m.group())
else:
    print('No match')
```

あるパターンにマッチするもの全てを返す Pattern インスタンスのメソッドが2つあります。`findall()` はマッチした文字列のリストを返します:

```
>>> p = re.compile(r'\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```

この例では、文字列リテラルを raw string リテラルにするプレフィックス `r` が必要です。これは、正規表現とは異なり、通常の " 調理済み " 文字列リテラルにおけるエスケープシーケンスは Python では認識されないためであり、現在では `DeprecationWarning` を引き起こし、最終的には `SyntaxError` となります。詳しくは [バックスラッシュ感染症](#) を参照してください。

`findall()` は結果を返す前に完全なリストを必ず生成してしまいます。いっぽう `finditer()` メソッドはマッチオブジェクト インスタンスのシーケンスを iterator として返します:

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable_iterator object at 0x...>
>>> for match in iterator:
...     print(match.span())
...
(0, 2)
(22, 24)
(29, 31)
```

3.4 モジュールレベルの関数

パターンオブジェクトを作ってそのメソッドを呼び出す、とする必要は必ずしもありません。re モジュールはトップレベルの関数として `match()`、`search()`、`findall()`、`sub()` などを用意しています。これら関数は、対応するメソッドの最初の引数に RE が追加されただけで後は同じで、None か Match オブジェクト インスタンスを返すのも同じです:

```
>>> print(re.match(r'From\s+', 'Fromage amk'))
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<re.Match object; span=(0, 5), match='From ' >
```

内部的には、これら関数は単にあなたのためにパターンオブジェクトを生成し、対応するメソッドを呼び出すだけのことです。とともに、将来の呼び出しで同じ RE のパースが何度も何度も必要とならないよう、コンパイル済みオブジェクトはキャッシュされます。

これらモジュールレベル関数を使うのと、パターンを自身で作って自身で呼び出すのとでどちらを使うべきでしょう? 正規表現をループの内側で使うならば、プリコンパイルは関数呼び出しを減らします。ループの外側であれば、内部キャッシュのおかげで、どちらでも大差ありません。

3.5 コンパイルフラグ

コンパイルフラグは正規表現の動作をいくつかの側面から変更します。フラグは re モジュール下で二つの名前で見ることが出来ます、例えば長い名前は `IGNORECASE` で短い名前は 1 文字で `I` のようになっています。(1 文字形式は Perl のパターン修飾子と同じ形式を使います; 例えば `re.VERBOSE` の短かい形式は `re.X` です。) 複数のフラグが OR ビット演算で指定することができます; 例えば `re.I | re.M` は `I` と `M` フラグの両方を設定します。

ここに利用可能なフラグの表があります、それぞれについてのより詳細な説明が後に続きます。

Flag	意味
ASCII, A	\w, \b, \s, そして \d などそれぞれのプロパティをもつ ASCII 文字だけにマッチさせます。
DOTALL, S	. を改行を含む任意の文字にマッチするようにします。
IGNORECASE, I	大文字小文字を区別しないマッチを行います。
LOCALE, L	ロケールに対応したマッチを行います。
MULTILINE, M	^ や \$ の意味を変更し、複数行文字列に対するマッチングを行います。
VERBOSE, X ('X' は 'extended' の 'X')	冗長な正規表現を利用できるようにして、よりきれいで理解しやすくまとめることができます。

I

IGNORECASE

大文字と小文字を区別しないマッチングを実行します; 文字クラスと文字列リテラルは大文字か小文字かに関係なくパターンにマッチします。例えば、[A-Z] は小文字のアルファベットにもマッチします。ASCII フラグによって非 ASCII 文字のマッチングが無効化されていなければ、完全なユニコードのマッチングも可能です。ユニコードで [a-z] または [A-Z] が IGNORECASE フラグとともに使われると、52 個の ASCII 文字に加えて 次の 4 つの 非 ASCII 文字にマッチします: 'İ' (U+0130, ラテン語の大文字 I で、上部に点がついたもの), 'ı' (U+0131, ラテン語の小文字 i で上部に点がない), 'Ŧ' (U+017F, ラテン語の小文字 s), 'K' (U+212A, ケルビン記号)。Spam は 'Spam', 'spam', 'spAM', そして 'fpam' にマッチします (ただし最後の文字列はユニコードモードの場合のみマッチします)。この「小文字化」は現在のロケールを考慮しません; ただし LOCALE フラグをセットした場合はロケールを考慮します。

L

LOCALE

\w, \W, \b, \B と小文字大文字の区別を無視したマッチングを、Unicode データベースではなく現在のロケールに従って行います。

ロケールは言語の違いを考慮したプログラムを書くことを手助けすることを目的とした C ライブラリの機能です。例えば、エンコードされたフランス語のテキストを処理していて、\w+ を使って単語のマッチを行いたいとします。ですがこの場合、\w はバイトパターンにおいて文字クラス [A-Za-z] だけにマッチします; すなわち é や ç に対応するバイト列にはマッチしません。もしシステムが適切に設定されていて、ロケールがフランス語に設定されていれば、ある C 関数はプログラムに é に対応するバイト列も文字として考慮するべきであると伝えます。正規表現をコンパイルするときに LOCALE フラグを設定すると、コンパイルされたオブジェクトが \w に対してロケールを考慮する C 関数を使うようになります; これにより処理は遅くなりますが、\w+ を期待通りフランス語の単語にマッチさせることが可能になります。このフラグを Python 3 で利用することは推奨されません。なぜならロケールの仕組みは非常に信頼性が低く、同時にひとつの "文化" しか扱うことができず、また 8 ビットのロケールでしか正しく動作しないからです。Python 3 ではユニコード (文字列の) パターンに対してユニコードのマッチングがデフォルトで有効化されており、これにより異なるロケールまたは言語を同時に扱うことができます。

M

MULTILINE

(`^` と `$` についてはまだ説明していません; これらは [さらなる特殊文字](#) の節で説明します。)

通常 `^` は文字列の先頭にマッチし、`$` は文字列の末尾と文字列の末尾に改行 (があれば) その直前にマッチします。このフラグが指定されると、`^` は文字列の先頭と文字列の中の改行に続く各行の先頭にマッチします。同様に `$` 特殊文字は文字列の末尾と各行の末尾 (各改行の直前) のどちらにもマッチします。

S

DOTALL

特別な文字 `.` を改行を含む全ての任意の文字とマッチするようにします; このフラグが無しでは、`.` は改行 **以外** の全てにマッチします。

A

ASCII

`\w`, `\W`, `\b`, `\B`, `\s`, `\S` が、完全な Unicode マッチングではなく、ASCII のみのマッチングをするようにします。これは Unicode パターンにのみ意味があり、byte パターンには無視されます。

X

VERBOSE

このフラグはより柔軟な形式で正規表現を読み易く書けるようにします。このフラグを指定すると、正規表現の中の空白は無視されます、ただし、文字クラス内やエスケープされていないバックスラッシュに続く空白の場合は例外として無視されません; これによって正規表現をまとめたり、インデントしてより明確にすることができます。このフラグはさらにエンジンが無視するコメントを追加することもできます; コメントは `#` で示します、これは文字クラスやエスケープされていないバックスラッシュに続くものであってはいけません。

例えば、ここに `re.VERBOSE` を利用した正規表現があります; 読み易いと思いませんか?

```
charref = re.compile(r"""
    &[#]           # Start of a numeric entity reference
    (
        0[0-7]+    # Octal form
        | [0-9]+    # Decimal form
        | x[0-9a-fA-F]+ # Hexadecimal form
    )
    ;              # Trailing semicolon
""", re.VERBOSE)
```

冗長な表現を利用しない設定の場合、正規表現はこうなります:

```
charref = re.compile("&#(0[0-7]+"
                    "| [0-9]+"
                    "| x[0-9a-fA-F]+);")
```

上の例では、Python の文字列リテラルの自動結合によって正規表現を小さな部分に分割しています、それでも `re.VERBOSE` を使った場合に比べるとまだ難しくなっています。

4 パターンの能力をさらに

ここまでで、正規表現の機能のほんの一部を扱ってきました。この節では、新たにいくつかの特殊文字とグループを使ってマッチしたテキストの一部をどう取得するかについて扱います。

4.1 さらなる特殊文字

これまでで、まだ扱っていない特殊文字がいくつかありました。そのほとんどをこの節で扱っていきます。

残りの特殊文字の内いくつかは **ゼロ幅アサーション** *zero-width-assertions* に関するものです。これらは文字列に対してエンジンを進めません; 文字列を全く利用しない代わりに、単純に成功か失敗かを利用します。例えば、`\b` は現在位置が単語の境界であることを示します; `\b` によってエンジンの読んでいる位置は全く変化しません。つまり、これはゼロ幅アサーションは繰り返し使うことがありません、一度ある位置でマッチしたら、明らかに無限回マッチできます。

| 代替 (alternation) または "or" 演算子です。 *A* と *B* が正規表現の場合、*A|B* は *A* と *B* のどちらかにマッチするような文字列にマッチします。複数の文字からなる文字列による代替処理が適切に動作するために、| の優先度は非常に低く設定されています。`Crow|Servo` は 'Crow' か 'Servo' のどちらかにマッチするパターンであり、「'Cro' に続いて 'w' または 'S' があり、さらに 'ervo' が続く」という意味ではありません。

リテラル '|' にマッチするには、`\|` を利用するか、`[|]` のように文字クラス内に収めて下さい。

^ 行の先頭にマッチします。MULTILINE フラグが設定されない場合には、文字列の先頭にのみマッチします。MULTILINE モードでは文字列内の各改行の直後にマッチします。

例えば、行の先頭の `From` にのみマッチさせたい場合には `^From` 正規表現を利用します。

```
>>> print(re.search('^From', 'From Here to Eternity'))
<re.Match object; span=(0, 4), match='From'>
>>> print(re.search('^From', 'Reciting From Memory'))
None
```

リテラル '^' にマッチするには `\^` を利用してください。

\$ 行の末尾にマッチします、行の末尾は文字列の末尾と改行文字の直前として定義されます。

```
>>> print(re.search('}$', '{block}'))
<re.Match object; span=(6, 7), match='}'>
>>> print(re.search('}$', '{block} '))
None
>>> print(re.search('}$', '{block}\n'))
<re.Match object; span=(6, 7), match='}'>
```

リテラル '\$' にマッチするには、`\$` を利用するか、`[$]` のように文字クラス内に収めて下さい。

- `\A` 文字列の先頭にのみマッチします。MULTILINE モードでない場合には `\A` と `^` は実質的に同じです。MULTILINE モードでのこれらの違いは: `\A` は依然として文字列の先頭にのみマッチしますが、`^` は文字列内に改行文字に続く部分があればそこにマッチすることです。
- `\Z` 文字列の末尾でのみマッチします。
- `\b` 単語の境界。これはゼロ幅アサーションで、単語の始まりか終わりにのみマッチします。単語は英数文字のシーケンスとして定義されます、つまり単語の終わりは空白か非英数文字として表われます。

以下の例では `class` がそのものの単語のときのみマッチします; 別の単語内に含まれている場合はマッチしません。

```
>>> p = re.compile(r'\bclass\b')
>>> print(p.search('no class at all'))
<re.Match object; span=(3, 8), match='class'>
>>> print(p.search('the declassified algorithm'))
None
>>> print(p.search('one subclass is'))
None
```

この特殊シーケンスを利用するときには二つの微妙な点を心にとめておく必要があります。まずひとつめは Python の文字列リテラルと表現の間の最悪の衝突を引き起こすことです。Python の文字列リテラルでは `\b` は ASCII 値 8 のバックスペース文字です。raw string を利用していない場合、Python は `\b` をバックスペースに変換し、正規表現は期待するものとマッチしなくなります。以下の例はさきほどと同じ正規表現のように見えますが、正規表現文字列の前の `'r'` が省略されています。

```
>>> p = re.compile('\bclass\b')
>>> print(p.search('no class at all'))
None
>>> print(p.search('\b' + 'class' + '\b'))
<re.Match object; span=(0, 7), match='\x08class\x08'>
```

ふたつめはこのアサーションが利用できない文字列クラスの内部では Python の文字列リテラルとの互換性のために、`\b` はバックスペース文字を表わすことになるということです。

- `\B` 別のゼロ幅アサーションで、`\b` と逆で、現在の位置が単語の境界でないときにのみマッチします。

4.2 グルーピング

正規表現にマッチするかどうかだけでなく、より多くの情報を得なければいけない場合は多々あります。正規表現はしばしば、正規表現をいくつかのサブグループに分けて興味ある部分にマッチするようにして、文字列を分割するのに使われます。例えば、RFC-822 ヘッダ行は `:` を挟んでこのようにヘッダ名と値に分割されます:

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
```

(次のページに続く)

```
MIME-Version: 1.0
To: editor@example.com
```

これはヘッダ全体にマッチし、そしてヘッダ名にマッチするグループとヘッダの値にマッチする別のグループを持つように正規表現を書くことで扱うことができます。

グループは特殊文字 '(' , ')' で表わされます。 '(' と ')' は数学での意味とほぼ同じ意味を持っています; その中に含まれた表現はまとめてグループ化され、グループの中身を *, +, ? や {m,n} のような繰り返しの修飾子を使って繰り返すことができます。例えば、(ab)* は ab の 0 回以上の繰り返しにマッチします。

```
>>> p = re.compile('(ab)*')
>>> print(p.match('ababababab').span())
(0, 10)
```

'(' , ')' で指示されたグループは、マッチしたテキスト開始と終了位置もキャプチャします; group(), start(), end(), span() に引数を与えて取り出せます。グループはゼロ始まりの数値です。グループ 0 は常に使えます; それは RE でマッチした全体で、Match オブジェクト メソッドの全てはグループ 0 をデフォルト引数にしています。マッチするテキストの範囲をキャプチャしないグループの書き方はのちほど見ることにします。

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

サブグループは左から右へ 1 ずつ番号付けされます。グループはネストしてもかまいません; 番号を決めるには、単に開き括弧を左から右へ数え上げます。

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

group() には一回に複数の引数を渡してもかまいません、その場合にはそれらのグループに対応する値を含むタプルを返します。

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

groups() メソッドは 1 から全てのサブグループの文字列を含むタプルを返します。:

```
>>> m.groups()
('abc', 'b')
```

パターン中で後方参照を利用することで、前に取り出されたグループが文字列の中の現在位置で見つかるように指定できます。例えば、`\1` はグループ 1 の内容が現在位置で見つかった場合成功し、それ以外の場合に失敗します。Python の文字列リテラルでもバックスラッシュに続く数字は任意の文字を文字列に含めるために使われるということを心に留めておいて下さい、そのため正規表現で後方参照を含む場合には raw string を必ず利用して下さい。

例えば、以下の正規表現は二重になった単語を検出します。

```
>>> p = re.compile(r'\b(\w+)\s+\1\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

このような後方参照は文字列を検索するだけの用途では多くの場合役に立ちません。--- このように繰り返されるテキストフォーマットは少数です。--- しかし、文字列の置換をする場合には **とても** 有効であることに気づくでしょう。

4.3 取り出さないグループと名前つきグループ

念入りに作られた正規表現は多くのグループを利用します、その利用法には対象となる部分文字列を取り出す、正規表現自身をグループ化したり構造化する、という二つの方法があります。複雑な正規表現では、グループ番号を追っていくことは困難になっていきます。この問題の解決を助ける二つの機能があります。その両方が正規表現を拡張するための一般的な構文を利用します、まずはそれらをみてみましょう。

Perl 5 は標準正規表現にパワフルな拡張を加えたことでよく知られています。それらの新しい機能のために Perl 開発者たちは、Perl 正規表現と標準正規表現との混乱を招く違いなしには、新たな一文字メタキャラクタも \ ではじまる新たな特殊シーケンスもどちらも選択出来ませんでした。たとえば彼らがもし `&` を新たなメタキャラクタとして選んでいたら、`&` が通常文字とみなされていた古い正規表現は `\&` や `[&]` のように書くことでエスケープされなければならなかったでしょう。

解決策として Perl 開発者が選んだものは `(?...)` を正規表現構文として利用することでした。括弧の直後の `?` は構文エラーとなります、これは `?` で繰り返す対象がないためです、そのためこれは互換性の問題を持ち込みません。`?` の直後の文字はどの拡張が利用されるかを示しています、つまり、`(?=foo)` は一つの拡張を利用したもの(肯定先読みアサーション)となり、`(?:foo)` は別の拡張を利用した表現(`foo` を含む取り込まないグループ)となります。

Python は Perl の拡張のいくつかをサポートし、また、Perl の拡張に一つ拡張を加えています。クエスションマークに続く最初の文字が `P` のものは、そうです、Python 固有の拡張です。

一般化された拡張構文についてはわかりましたので、いよいよ複雑な正規表現内でのグループの扱いを単純化する機能に話を戻しましょう。

ときとしてあなたは、正規表現の一部として使いたいけれども、その内容を取り出すことに興味がないようなグループを記述する必要に迫られます。このためには、取り出さないグループ: (?:...) を使います。... 部分は任意の正規表現です。

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

マッチしたグループの内容を取得しないということを除けば、取り込まないグループは厳密に取り込むグループと同様に振る舞います; この中に何を入れてもかまいません、* のような繰り返しの特殊文字で繰り返したり、他のグループ (取り込むまたは取り込まない) の入れ子にすることもできます。(?:...) は特に、既にあるパターンを変更する際に便利です、なぜなら他の番号づけ新しいグループを変更することなく新しいグループを追加することができます。取り込むグループと取り込まないグループで検索のパフォーマンスに差がないことにも触れておくべきことです; どちらも同じ速度で動作します。

より重要な機能は名前つきグループです: 番号で参照する代わりに、グループに対して名前で参照できます。

名前つきグループのための構文は、Python 固有拡張の一つ: (?P<name>...) を使います。name は、もちろん、グループの名前です。名前つきグループは取り込むグループと完全に同じに振る舞い、加えて名前が関連付けられます。Match オブジェクト の取りこむグループを扱うメソッドは全て、番号によるグループ参照のための整数、名前によるグループ参照のための文字列、ともに許容しています。名前つきグループにも番号が振られますので、グループについての情報を、2つの方法で取り出せます:

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '(((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

さらに、名前付きのグループを groupdict() を使って辞書として取り出すこともできます:

```
>>> m = re.match(r'(?P<first>\w+) (?P<last>\w+)', 'Jane Doe')
>>> m.groupdict()
{'first': 'Jane', 'last': 'Doe'}
```

名前つきグループは、番号を覚える代わりに、簡単に覚えられる名前を利用できるので、簡単に扱うことができます。これは imaplib モジュールから正規表現の例です:

```
InternalDate = re.compile(r'INTERNALDATE "'
    r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
    r'(?P<year>[0-9][0-9][0-9][0-9])'
```

(次のページに続く)

```

r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9]) '
r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9]) '
r''')

```

取得する番号 9 を覚えるよりも、`m.group('zonem')` で取得した方が明らかに簡単にすみます。

後方参照のための構文 `(...)\1` はグループ番号への参照となっています。グループ番号の代わりに、グループ名を利用する変種があるのは当然でしょう。これはもう一つの Python 拡張です: `(?P=name)` は、`name` という名前のグループの内容が、現在の位置で再びマッチすることを示しています。同じ単語が 2 つ連なっているを見つめる正規表現 `\b(\w+)\s+\1\b` は `\b(?P<word>\w+)\s+(?P=word)\b` のように書けます:

```

>>> p = re.compile(r'\b(?P<word>\w+)\s+(?P=word)\b')
>>> p.search('Paris in the the spring').group()
'the the'

```

4.4 先読みアサーション (Lookahead Assertions)

他のゼロ幅アサーションは先読みアサーションです。先読みアサーションは肯定、否定の両方の形式が利用可能です、これを見てください:

`(?=...)` 肯定先読みアサーション。`...` で表わす正規表現が現在位置でマッチすれば成功し、それ以外の場合失敗します。しかし、表現が試行された場合でもエンジンは先に進みません; パターンの残りの部分はアサーションの開始時点から右に試行します。

`(?!...)` 否定先読みアサーション。これは肯定アサーションの逆で、正規表現が文字列の現在位置にマッチしなかった場合に成功します。

より具体的にするため、先読みが便利な場合をみてみましょう。ファイル名にマッチし、`.` で分けられた基本部分と拡張子に分離する単純なパターンを考えましょう。例えば、`news.rc` は `news` が基本部分で `rc` がファイル名の拡張子です。

マッチするパターンはとても単純です:

```
.*[.].*$
```

`.` はメタキャラクタですので特別に扱わなければなりませんから、文字クラス内に入れて、そのものだけマッチするようにしていることに注目です。末尾の `$` にも注目してください; これは残り全ての文字列が拡張子に含まれるべきであることを保障するために追加しています。この正規表現は `foo.bar`, `autoexec.bat`, `sendmail.cf`, `printers.conf` にマッチします。

さて、問題を少し複雑にしてみましょう; 拡張子が `bat` でないファイル名にマッチしたい場合はどうでしょう? 間違った試み:

`.*[.][^b].*$` この最初の `bat` を除く試みは、最初の文字が `b` でないことを要求します。これは誤っています、なぜなら `foo.bar` にもマッチしないからです。

`.*[.]([^b] | . [^a] | . [^t])$`

正規表現が混乱してきました。最初の解決策を取り繕って、以下の場合に合わせることを要求しています: 拡張子の最初の文字は `b` でなく; 二番目の文字は `a` でなく; 三番目の文字は `t` でない。これは `foo.bar` を受け付けますが、`autoexec.bat` は拒否します。しかし、三文字の拡張子を要求し、`sendmail.cf` のような二文字の拡張子を受け付けません。これを修正するのにパターンを再び複雑にすることになります。

`.*[.]([^b]? | . [^a]? | . [^t]?)$`

三番目の試みでは、`sendmail.cf` のように三文字より短い拡張子とマッチするために第二第三の文字を全てオプションにしています。

パターンはさらに複雑さを増し、読みにくく、理解が難しくなりました。より悪いことに、問題が `bat` と `exe` 両方を拡張子から除きたい場合に変わった場合、パターンはより複雑で混乱しやすいものになります。

否定先読みはこの混乱全てを取り除きます:

`.*[.](?!bat$)[^.]*$` 否定先読みは以下を意味します: この位置で拡張子 `bat` にマッチしない場合、残りのパターンが試行されます; もし `bat$` にマッチすればパターン全体が失敗します。`$` を続けることで、`sample.batch` のように `bat` で始まる拡張子を許容することを保証しています。このパターンで `[^.]` を使うことで、ファイル名に複数のドットがあったときにも上手くいくようになります。

他のファイル名の拡張子を除くことも簡単です; 単純にアサーション内に拡張子を代替 (or) で加えます。以下のパターンは `bat` や `exe` のどちらかで終わるファイル名を除外します:

`.*[.](?!bat$|exe$)[^.]*$`

5 文字列を変更する

ここまででは単純に静的な文字列に対する検索を実行してきました。正規表現は文字列を様々な方法で変更するのもよく使われます。変更には以下のパターンメソッドが利用されます:

メソッド/属性	目的
<code>split()</code>	文字列をリストに分割する、正規表現がマッチした全ての場所で分割を行う
<code>sub()</code>	正規表現にマッチした全ての文字列を発見し、別の文字列に置き換えます
<code>subn()</code>	<code>sub()</code> と同じことをしますが、新しい文字列と置き換えの回数を返します

5.1 文字列の分割

`split()` メソッドは文字列を正規表現にマッチした場所で分割し、リストで返却します。文字列の `split()` メソッドに似てはいますが、もっとずっと一般化したデリミタで分割出来ます; 文字列の `split()` メソッドは単に空白文字か固定文字列で分割出来るだけです。ご想像通り、モジュールレベルの `re.split()` 関数もあります。

```
.split(string[, maxsplit=0])
```

string を正規表現のマッチで分割します。正規表現内に取り込むための括弧が利用されている場合、その内容も結果のリストの一部として返されます。*maxsplit* が非ゼロの場合、最大で *maxsplit* の分割が実行されます。

maxsplit に値を渡すことで、分割される回数を制限することができます。*maxsplit* が非ゼロの場合、最大で *maxsplit* の分割が行なわれ、文字列の残りがリストの最終要素として返されます。以下の例では、デリミタは任意の英数文字のシーケンスです。

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

興味の対象がデリミタの間のテキストだけでなく、デリミタが何なのかということを知りたい場合はよくあります。取りこみ用の括弧を正規表現に使った場合、その値もリストの一部として返されます。以下の呼び出しを比較してみましょう:

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '...', 'is', ' ', 'a', ' ', 'test', '.', '']
```

モジュールレベル関数 `re.split()` は最初の引数に利用する正規表現を追加しますが、それ以外は同じです。

```
>>> re.split(r'[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split(r'([\W]+)', 'Words, words, words.')
['Words', ' ', ' ', 'words', ' ', ' ', 'words', '.', '']
>>> re.split(r'[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

5.2 検索と置換

もう一つのよくある作業は、パターンにマッチする全ての文字列を探し、異なる文字列に置換します。`sub()` メソッドは置換する値をとります、文字列と関数の両方をとることができ、文字列を処理します。

```
.sub(replacement, string[, count=0])
```

`string` 内で最も長く、他の部分と重複するところがない正規表現を `replacement` に置換した文字列を返します。パターンが見つからなかった場合 `string` は変更されずに返されます。

オプション引数 `count` はパターンの出現の最大置換回数です; `count` は非負の整数でなければいけません。デフォルト値 0 は全ての出現で置換することを意味します。

ここに `sub()` メソッドを使った単純な例があります。これは色の名前を `colour` に置換します:

```
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

`subn()` メソッドも同じ働きをしますが、新しい文字列と置換の実行回数を含む 2-タプルを返します:

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

空文字列とのマッチは、直前の空文字列とマッチした部分と隣接していない場合にのみ置換されます。

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b--d-'
```

`replacement` が文字列の場合、文字列内のバックスラッシュエスケープは処理されます。つまり、`\n` は改行文字に、`\r` はキャリッジリターンに、等となります。`\&` のような未知のエスケープシーケンスはそのまま残されます。`\6` のような後方参照は正規表現内の対応するグループにマッチする文字列に置換されます。これを使うことで元のテキストの一部を、置換後の文字列に組み込むことができます。

この例は単語 `section` に続く `{ }` で閉じられた文字列にマッチし、`section` を `subsection` に変更します:

```
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}', 'section{First} section{second}')
'subsection{First} subsection{second}'
```

(`?P<name>...`) 構文で定義された名前つきグループを参照するための構文もあります。`\g<name>` は `name` で名前づけされたグループにマッチする文字列を利用し、`\g<number>` は対応するグループ番号を利用します。つまり

\g<2> は \2 と等価ですが、\g<2>0 のような置換文字列に対しては明確に異なります。(\20 はグループ番号 20 への参照と解釈され、グループ 2 の後にリテラル文字 '0' が続くとは解釈されません。) 以下に示す置換は全て等価ですが、これらは文字列置換に全部で 3 種の変種を利用しています。

```
>>> p = re.compile('section{ (?P<name> [^]]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}','section{First}')
'subsection{First}'
```

より細かな制御を手中にするために *replacement* として関数を使うことが出来ます。*replacement* が関数であれば、その関数は重なり合わない *pattern* の発生のたびに呼び出されます。それぞれの呼び出しで、マッチした Match オブジェクト が引数として渡されるので、望みの置換と返却のためにこの情報を利用出来ます。

続く例では、置換関数は十進数文字列を十六進数文字列に変換しています:

```
>>> def hexrepl(match):
...     "Return the hex string for a decimal number"
...     value = int(match.group())
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffd2 for printing, 0xc000 for user code.'
```

モジュールレベルの `re.sub()` 関数を使うときには、パターンが最初の引数として渡されます。パターンはオブジェクトや文字列をとります; 正規表現フラグを指定する必要がある場合、パターンオブジェクトを最初の引数として使うか、修飾子を埋め込んだパターン文字列を使うかしなければいけません、例えば `sub("(?i)b+", "x", "bbbb BBBB")` は `'x x'` を返します。

6 よくある問題

正規表現はいくつかの応用に対して強力なツールですが、いくつかの部分でそれらの振る舞いは直感的ではなく、期待通りに振る舞わないことがあります。この節では最もよくある落とし穴を指摘します。

6.1 文字列メソッドを利用する

いくつかの場合 `re` モジュールを利用することは間違いである場合があります。固定文字列や単一の文字クラスにマッチさせる場合や、`IGNORECASE` フラグのような `re` の機能を利用しない場合、正規表現の全ての能力は必要とされていなのでしょう。文字列は固定文字列に対する操作を実行するメソッドを持っていて、大きな汎用化された正規表現エンジンではなく、目的のために最適化された単一の小さな C loop で実装されているため、大抵の場合高速です。

一つの例としては、単一の固定文字列を別の固定文字列に置き換える作業があるでしょう；例えば `word` を `deed` で置換したい場合です。`re.sub()` はこの目的で使う関数のように思えますが、`replace()` メソッドを利用することを考えた方がいいでしょう。`replace()` は単語内の `word` も置換し、`swordfish` を `sdeedfish` に変えますが、安直な正規表現 `word` も同様に動作することに注意して下さい。(単語の一部に対する置換の実行を避けるには、パターンを `\bword\b` として、`word` の両側に単語の境界が要求されるようにします。これは `replace()` の能力を越えた作業です。)

別のよくある作業は、文字列の中に出現する文字を全て削除することと、別の文字で置換することです。この作業を `re.sub('\n', ' ', S)` のようにして行うかもしれませんが、`translate()` は削除と置換の両方の作業をこなし、正規表現操作よりも高速に行うことができます。

要は、`re` モジュールに向う前に問題が高速で単純な文字列メソッドで解決できるか考えましょうということです。

6.2 `match()` 対 `search()`

`match()` 関数は文字列の先頭に正規表現がマッチするかどうか調べるだけですが、その一方 `search()` はマッチするために文字列の先の方まで走査します。この違いを覚えておくことは重要なことです。`match()` は開始位置 0 でマッチが成功したときのみ報告する；もし開始位置 0 でマッチしなければ、`match()` はそれを報告しない、ということを覚えておいてください。

```
>>> print(re.match('super', 'superstition').span())
(0, 5)
>>> print(re.match('super', 'insuperable'))
None
```

一方 `search()` は文字列の先の方まで走査し、最初に見つけたマッチを報告します。:

```
>>> print(re.search('super', 'superstition').span())
(0, 5)
>>> print(re.search('super', 'insuperable').span())
(2, 7)
```

しばしば、`re.match()` を使い、`.*` を正規表現の最初に付け加える誘惑にからされることがあるでしょう。この誘惑に打ち克って、代わりに `re.search()` を利用すべきです。正規表現コンパイラはマッチを探す処理の高速化のためにいくつかの解析を行います。そのような解析のうちのひとつはマッチの最初の文字が何であるか評価す

ることです; 例えば、Crow で始まるパターンは 'C' から始まらなければいけません。解析によってエンジンは速やかに開始文字を探して走査します、'C' が発見された場合にはじめて完全なマッチを試みます。

.* を追加することはこの最適化を無効にします、文字列の終端までの走査が必要となり、走査後には残りの正規表現とのマッチ部分を見つけるために引き返すことになります。代わりに `re.search()` を利用して下さい。

6.3 貪欲 (greedy) 対非貪欲 (non-greedy)

正規表現を繰り返す場合、たとえば `a*` のように、できるだけパターンの多くにマッチするように動作することになります。この動作は、例えば角括弧で囲まれた HTML タグのような左右対称のデリミタの対にマッチしようという場合に問題となります。単一の HTML タグにマッチする素朴な正規表現はうまく動作しません、なぜならば .* は貪欲に動作するからです。

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print(re.match('<.*>', s).span())
(0, 32)
>>> print(re.match('<.*>', s).group())
<html><head><title>Title</title>
```

正規表現は '<html>' 内の '<' にマッチし、.* は残りの文字列の全てにマッチします。しかし、正規表現には依然として残っている部分があって、> は文字列の終端にマッチしないので、正規表現エンジンは一文字ずつ > とマッチするまで引き返すことになります。最終的にマッチする領域は '<html>' の '<' から '</title>' の '>' まで広がりますが、これは望んだ結果ではありません。

この場合、解決法は非貪欲を示す修飾子 `*?`, `+?`, `??` または `{m,n}?` を利用することです、これらはテキストに可能な限り **少なく** マッチします。上の例では、> は最初の '<' とのマッチ後すぐに '>' を試みま、失敗した場合にはエンジンが文字を先に進め、> が毎ステップ再試行されます。この動作は正しい結果を生み出します:

```
>>> print(re.match('<.*?>', s).group())
<html>
```

(HTML や XML を正規表現でパースすることは苦痛を伴うものであることは記憶に留めておいて下さい。素早く、汚いパターンは大抵の場合うまく動作しますが、HTML と XML は正規表現が破綻する特別な例です; 全ての可能な場合にうまく動作する正規表現を書き上げたときには、パターンは **非常に** 複雑なものになります。そのような作業をする場合には HTML や XML パーサを利用しましょう。)

6.4 re.VERBOSE の利用

ここまでで、正規表現がとても簡潔な表記であることに気づいたでしょう、また、正規表現は読みやすいものでもないということにも気づいたことでしょう。そこそこに入り組んだ正規表現はバックスラッシュ、括弧、特殊文字が長く続いて、読みにくく、理解しづらいものになります。

そのような正規表現に対しては、正規表現をコンパイルする時に `re.VERBOSE` フラグを指定することが助けになります。なぜなら、より明確な書式で正規表現を書けるからです。

`re.VERBOSE` の効果はいくつかあります。正規表現内の文字クラス内に **無い** 空白は無視されます。これは、`dog | cat` のような表現が少々可読性の落ちる `dog|cat` と等価となるということです、しかし、`[a b]` は依然として `'a'`, `'b'`, または空白にマッチします。加えて、正規表現にコメントを入れることもできるようになります；`#` 文字から次の改行までがコメントの範囲です。三重クォートを利用することで、正規表現をきちんとフォーマットすることができます：

```
pat = re.compile(r"""
\s*           # Skip leading whitespace
(?:P<header>[^\:]+) # Header name
\s* :         # Whitespace, and a colon
(?:P<value>.*?) # The header's value -- *? used to
               # lose the following trailing whitespace
\s*$         # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

これは下よりはるかに読みやすいです：

```
pat = re.compile(r"\s*(?:P<header>[^\:]+)\s*:(?:P<value>.*?)\s*$")
```

7 フィードバック

正規表現は複雑な話題です。このドキュメントは助けになったでしょうか？ わかりにくかったところや、あなたが遭遇した問題が扱われていない等なかったでしょうか？ もしそんな問題があれば、著者に改善の提案を送って下さい。

O'Reilly から出版されている Jeffrey Friedl の *Mastering Regular Expressions* は正規表現に関するほぼ完璧な書籍です（訳注 日本語訳「詳説 正規表現」が出版されています）。不幸なことに、この本は Perl と Java の正規表現を集中して扱っていて、Python の正規表現については全く扱っていません、そのため Python プログラミングのためのリファレンスとして使うことはできません。（第一版はいまや削除された Python の `regex` モジュールについて扱っていましたが、これはあまり役に立たないでしょう。）図書館で調べるのを検討してみましょう。