正規表現とは何か(Python版)

どんなプログラム言語にも、ある文字列の中に「特定の文字列」があるかどうか、調べる関数が用意されている。

```
>>> a = "abcdabc"
>>> a.find("c")   # find関数は、文字列のなかにcがあれば、その位置を返す
2                 # 最初の文字を0として数え、2の位置にcがある
>>> a.rfind("c")  # rfindは後ろから検索し、最初に見つかった位置
6
>>> a.find("f")
-1                # ない場合は-1
# stringというモジュールにあるfind関数も同じ
>>> import string
>>> string.find(a, "c")
2
>>> a.replace("c", "SII") # replaceは文字列の置換
'abSIIdabSII'
```

正規表現とは、「特定の文字列」を「特定の文字列パターン」に拡張する機能で、さまざまなパターンマッチングを行う。 たとえば、「abc」を検索するだけでなく、「英字3文字に続く丸括弧内に３桁の数字がある文字列」を検索できるようになる。 正規表現(regular expression)という名前に惑わされてはいけない。「パターン表現法」だ。

ここではPythonで例文を書いているが、JavascriptにもJavaにもVBにもC++にもPerlにも正規表現がある。ただし、拡張機能が微妙に違う（方言がある）ので、困った時は、その言語の資料に眼を通す必要がある。

以下は、正規表現を使って、上例と同様に、文字列abcdabcからcを探す例。 Pythonの正規表現関数は、引数の順番が逆になっている事に注意。

```
>>> import re
>>> re.findall("c", a)      # findallは見つかったものをリストで返す
['c', 'c']
>>> for m in re.finditer("c", a):   # finditerは見つかったものをMatchObjectとして返す
...   print m.start(), m.end(), m.string[m.start():m.end()]
...         # 最初の位置、最後の位置、見つかった文字列
2 3 c
6 7 c
```

これだけなら、正規表現を使う意味はない。

ここから、UTF-8で日本語を扱うため、文字列をunicode文字列に変えることにする（uをつけるだけ）

```
>>> a = u"134-0023 東京都千代田区1-1-1 東京書籍　山田太郎　"
>>> for m in re.finditer(u"東京", a):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
9 11 東京
22 24 東京
```

郵便番号を検索したいとする。もし、「文字列は郵便番号から始まる」と保証されているなら、最初の8文字を抜き出せばよい。 保証されていない場合、「数字３文字と-と数字４文字」を検索したいと思うはずだ。

```
>>> for m in re.finditer(u"\d{3}-\d{4}", a):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
0 8 134-0023
```

この検索キーワードになった「\d{3}-\d{4}」は以下のように解釈される。

- \d 数字
- {3} ３文字
- - 通常の-

- \d 数字
- {4} 4 文字

このような検索ができる「特殊な文字列パターン」を指定するのが正規表現で、プログラム言語だけでなく、高機能エディタなどにも装備されている。

正規表現の特殊文字

正規表現で使う特殊文字は以下の通り。（完全には網羅していない）

正規表現の基本

1文字を表す			
[abc]	abcのどれか1文字	\w	word文字(アルファベットと数字)
[^abc]	abc以外の1文字	\W	word文字以外(ピリオドなど)
[a-z]	a-zの1文字	\d	数字
[^a-z]	a-z以外の1文字	\D	数字以外
.	改行(\n)以外の1文字	\s	空白文字
コントロール文字		\S	空白文字以外
\t	タブ文字	\p{}	名前付きブロック（いろいろある）
\n	改行	\P	名前付きブロック以外
\v,\b,\e,\r,\f,\aなどがある			
アンカー（位置を示す記号）			
^	行頭（文頭と改行直後の行頭）	\G	直前のマッチングが終わった場所
\A	文頭	\b	wordの境界
\z	文末	\B	wordの境界以外
\Zか$	行末（文末と改行直前）		
数量(?をつけるとLazy)			
*	0回以上	{n}	ちょうどn回
+	1回以上	{n,}	n回以上
?	0回か1回	{n,m}	n回以上、m回以下

以下の違いを見れば、\(バックスラッシュ、windowsでは￥)が後続するdの意味を決定的に変えてしまうことが分かる。

```
>>> for m in re.finditer(u"d", u"abcd1234"):   #「d」を検索
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
3 4 d

>>> for m in re.finditer(u"\d", u"abcd1234"):  #「数字」を検索
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
4 5 1
5 6 2
6 7 3
7 8 4
```

「.」は「改行以外のすべての文字」を表す。以下の処理は、要するにすべての文字に分解している。

```
>>> for m in re.finditer(u".", a):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
0 1 1
1 2 3
2 3 4
3 4 -
4 5 0
5 6 0
6 7 2
7 8 3
8 9
9 10 東
10 11 京
11 12 都
12 13 千
13 14 代
14 15 田
15 16 区
16 17 1
17 18 -
18 19 1
19 20 -
20 21 1
21 22
22 23 東
23 24 京
24 25 書
25 26 籍
26 27
27 28 山
28 29 田
29 30 太
30 31 郎
31 32
```

ここで、Python特有の、モード設定を導入する。**基本的に文字列の最初に置く**。

- (?i) 大文字・小文字を区別しない
- (?L) ロケール依存にする（日本語設定など）
- (?m) 複数行モード
- (?s) DOTALLモード
- (?u) UNICODE依存モード
- (?x) 冗長モード

「\w」はwordの意味で、英数字と下線、つまり、[a-zA-Z0-9_]と同じ意味。 UNICODE依存モードでは、通常の文字も示す。全角の「、」や「　」は文字扱いではないことに注目。 「\W」はその補集合。

```
>>> for m in re.finditer(u"\w", u"あいう、１２３　えお.123"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
11 12 1
12 13 2
13 14 3
>>> for m in re.finditer(u"(?u)\w", u"あいう、１２３　えお.123"):  # UNICODEモード
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
0 1 あ
1 2 い
2 3 う
4 5 1
5 6 2
6 7 3
8 9 え
9 10 お
11 12 1
```

```
12 13 2
13 14 3
```

「\d」は数字で、[0-9]と同じ意味。UNICODE依存モードでは、全角の洋数字にもマッチングする。「\D」はその補集合。

```
>>> for m in re.finditer(u"\d", u"あいう、１２３　えお.123"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
11 12 1
12 13 2
13 14 3
>>> for m in re.finditer(u"(?u)\d", u"百二十三あいう、１２３　えお.123"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
8 9 1
9 10 2
10 11 3
15 16 1
16 17 2
17 18 3
```

「\s」は空白文字で、[\t\n\r\f\v]と同じ意味。つまり、タブや改行も含む。UNICODE依存モードでは、全角の空白にもマッチングする。「\S」はその補集合。

```
>>> for m in re.finditer(u"\s", u"あい　う、１２３　えお.123"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
2 3     # 半角スペース
>>> for m in re.finditer(u"(?u)\s", u"あい　う、１２３　えお.123"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
2 3     # 半角スペース
8 9      # 全角スペース
```

数量を示す特殊文字。?をつけるとLazyモード（最短マッチ）になる。

- * 0個以上（最長マッチ）
- + 1個以上（最長マッチ）
- ? 0か1個
- *? 0個以上（最短マッチ）
- +? 1個以上（最短マッチ）
- {m} m個（mを超えている場合もm個だけマッチングする）
- {m,n} m以上でn以下（最長マッチ）
- {m,n}? m以上でn以下（最短マッチ）

```
>>> re.findall(u"<.*>", u"<h1>hello</h1>")
[u'<h1>hello</h1>']
>>> re.findall(u"<.*?>", u"<h1>hello</h1>")
[u'<h1>', u'</h1>']

>>> re.findall(u"b{3}", u"abbb")
[u'bbb']
>>> re.findall(u"b{3}", u"abbbb")
[u'bbb']

>>> re.findall(u"b{3,5}", u"bbbbb")
[u'bbbbb']
>>> re.findall(u"b{3,5}?", u"bbbbb")
[u'bbb']
```

Pythonのエスケープ文字

Pythonでは(他の言語でも同じだが)、\(バックスラッシュ)は、改行やタブ文字など、特殊な文字を示す時に使われる。エスケープ文字（回避文字）と呼ばれ、続く文字（tやn）を通常の文字だと理解することをエスケープ（回避）して、改行記号やタブ文字だと解釈する。

```
>>> a = u"改行は\nでタブは\tです"
>>> print a
改行は
でタブは	です
```

しかし、これでは、本当に「\t」と表示したい場合に困る。このため、\をエスケープ文字であると解釈すること自体をエスケープしなければならない。pythonでは、エスケープ文字をエスケープするためにも\を使う。だから、

```
>>> a = u"改行は\\nでタブは\\tです"
>>> print a
改行は\nでタブは\tです
```

これまでの正規表現（例えばu”\s”）で、\を使っても問題なかったのは、pythonでは\sを使わないからにすぎない。

この問題を回避するため、（Python特有だが）「raw文字列」がある。r”abc”や、ur”あいう”と指定すると、「pythonのエスケープ」がその内部では機能しない。

```
>>> a = ur"改行は\nでタブは\tです"
>>> print a
改行は\nでタブは\tです
```

正規表現の特殊文字の続き

raw文字列を紹介したのには理由がある。以下の「正規表現」は「pythonのエスケープ」と名前が衝突するからだ。

- 「\A」は、正規表現で、文字列先頭を表す。
- 「\Z」は、正規表現で、文字列末端を表す。
- 「\b」は、正規表現で、単語の先頭か末尾を表す。この場合の単語とは、\wの対象になる文字列の連続で、UNICODE依存モードだと漢字なども対象になる。
- 「\B」は、正規表現で、単語の先頭でもなく末尾でもない、つまり、単語の内部を表す。

```
>>> for m in re.finditer(ur"(?u)\w{2,3}", u"東京、大阪、名古屋、福岡"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
0 2 東京
3 5 大阪
6 9 名古屋
10 12 福岡

>>> for m in re.finditer(ur"(?u)\A\w{2,3}", u"東京、大阪、名古屋、福岡"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
0 2 東京

>>> for m in re.finditer(ur"(?u)\w{2,3}\Z", u"東京、大阪、名古屋、福岡"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
10 12 福岡
```

以下の記号は位置を示す。

- 「^」は、正規表現で、MULTILINEモードで行頭を表す。単行モードなら\Aと同じ。
- 「$」は、正規表現で、MULTILINEモードで行末を表す。単行モードなら\Zと同じ。

鉤括弧はどれか 1 文字を示す。

- []は、文字の集合のうちのどれか 1 文字を表す。たとえば、[amk] はaかmかkのどれか 1 文字。連続した文字

の範囲を、先頭と最後の2文字とその間に‘-‘を挟んだ形で指定できる。[a-z]はすべての小文字。[0-5][0-9]は00から59までの、すべての２桁数字。[0-9A-Fa-f]は任意の16進数の数字を表す。-そのものを対象にする場合はエスケープする(例: [a\-z]はaか-か)。ただし、先頭か末尾に置かれた場合は不要。[]内では特殊文字はその意味を失い、[(+*)]は、(か+か*か)かのいずれか１文字を意味する。最初に^を置くと、補集合を意味する。[^5]は5以外のすべての文字。

```
>>> for m in re.finditer(ur"(?u)[東西南北]", u"南北朝時代の東京極"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
0 1 南
1 2 北
6 7 東
```

「正規表現A|正規表現B」で、「|」はA or Bを意味する。最初に正規表現Aを評価し、何も見つからなければ正規表現Bを評価する。

```
>>> for m in re.finditer(ur"(?u)東西|南北", u"南北朝時代の東京極"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
0 2 南北
```

グループと後方参照

グループは、正規表現のある部分を切り分けて、あとで利用しやすくする仕組みで、()で表す。このため、(と)を検索したい場合には\(、\)のようにバックスラッシュでエスケープしなければならない。 グループはいくつあってもいい。1から始まる番号がつけられる。また、(?P<name>...)と書くと、名前を付けることができる（番号も自動で割り当てられる）。

では、グループは何に使うのか？

まず、グループはマッチした文字列から抜き出すことに使うことができる。

以下は電話番号を抜き出す正規表現になる。

```
>>> for m in re.finditer(ur"(?u)(\d{2,3})-(\d{4})-\d{4}", u"番号は03-7263-7812です"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...   print u"市外局番", m.group(1)
...   print u"市内局番", m.group(2)
...
3 15 03-7263-7812
市外局番 03
市内局番 7263
```

名前付きグループはもっとわかりやすい。

```
>>> r = ur"(?u)(?P<SHIGAI>\d{2,3})-(?P<SHINAI>\d{4})-\d{4}"
>>> a = u"番号は03-7263-7812です"
>>> for m in re.finditer(r, a):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...   print u"市外局番", m.group("SHIGAI")
...   print u"市内局番", m.group("SHINAI")
...
3 15 03-7263-7812
市外局番 03
市内局番 7263
```

グループの別の役割は、後方参照と呼ばれる「グループと同じ文字列」を検索対象にすること。「\数字」で、先行するグループの文字列を指定できる。

例えば、繰り返しを検索する場合、(\w{2})\1は、「文字が２文字と、その２文字が続いたもの」という正規表現を使う。

```
>>> for m in re.finditer(ur"(?u)(\w{2})\1", u"さまざま、まざまざ、さまさま"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...   print m.group(1)
...
5 9 まざまざ
まざ
10 14 さまさま
さま
```

先読みアサーション、後読みアサーション

いわば「条件付き正規表現」で、以下のようなバリエーションがある。

- 先読みアサーション： aaa(?=bbb)は、bbbが続くという条件を満たすaaa
- 否定先読みアサーション： aaa(?!bbb)は、bbbが続かないという条件を満たすaaa
- 後読みアサーション： (?<=bbb)aaaは、bbbが先行するという条件を満たすaaa
- 否定後読みアサーション： (?<=!bbb)aaaは、bbbが先行しないという条件を満たすaaa

これらは、条件付き検索を行う。条件は、条件として参照されるだけ。一体化した場合と比較してみる。

```
>>> m = re.search('(?<=abc)def', 'abcdef')
>>> m.group(0)
'def'
>>> m = re.search('abcdef', 'abcdef')
>>> m.group(0)
'abcdef'
```

特殊な条件付き正規表現

(?(id/name)yes-pattern|no-pattern)は、idもしくはnameのグループがあるとき、 yes-patternを行い、ないときはno-patternと行う。

例えば、(<)?(\w+@\w+(?:\.\w+)+)(?(1)>)は、グループ２がメールアドレスを表し、グループ１が最初の「<」があるかないかを示す。 グループ３は、「グループ１があるときに『>』でマッチングし、ないときはマッチング対象なし」を意味する。つまり、<user@host.com>とuser@host.comにはマッチし、(閉じる>が欠けた)<user@host.comにはマッチしない。

コンパイルオプション

正規表現の「ある文字のどれか」「n文字以上」などの機能は、プログラムそのもののような感じがする。実際、正規表現はプログラム言語内言語であり、コンパイルされる。

コンパイル結果はSRE_Patternオブジェクトになり、search関数などが付加される。

```
>>> re.compile('abcdef')
<_sre.SRE_Pattern object at 0x10add4390>
>>> reg_obj = re.compile('abcdef')
>>> for m in reg_obj.finditer("abcdefghijk"):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
0 6 abcdef
```

つまり、２通りの書き方ができる。

re.XかRe.VERBOSEをコンパイル時に指定すると、正規表現の中にコメントを書き込むことができる。

```
a = re.compile(r"""\d +  # 整数部分
                   \.    # 小数点
```

```
                    \d *  # 小数点以下の数字
                    """, re.X)
```

正規表現でできること

単純な検索を行うメソッド。

- re.search(pattern, string, flags=0) 検索してMatchObjectとして返す。ないならNoneを返す。
- re.match(pattern, string, flags=0) 文字列先頭だけでMatchObjectを返す。ないならNone。
- re.split(pattern, string, maxsplit=0) 正規表現で分割し、リストで返す。

```
>>> re.split('\W+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('(\W+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('\W+', 'Words, words, words.', 1)
['Words', 'words, words.']
>>> re.split('[a-f]+', '0a3B9', flags=re.IGNORECASE)
['0', '3', '9']
```

すべてのマッチングを見つけるメソッド。

```
* re.findall(pattern, string, flags=0)  全てのマッチを文字列のリストとして返す。
* re.finditer(pattern, string, flags=0) 全てのマッチをMatchObjectのリストとして返す。
```

re.sub関数（置換）の詳細

- re.sub(pattern, repl, string, count=0, flags=0) 一致部分を置換する。

文字列内にpatternが見つかった場合、置換replで置換して得られた文字列を返す。もし、パターンが見つからなければ、 stringを変更せずに返す。

replは、文字列でも関数でも可能で、文字列であれば、後方参照(backreference)も使うことができる。

```
>>> a = u"東京(01)、西京(03)、北京(04)、南京(08)、中京(09)"
>>> print re.sub(ur"(?u)\w京", u"都", a)
都(01)、都(03)、都(04)、都(08)、都(09)
>>> print re.sub(ur"(?u)[東西南北]京", u"都", a)
都(01)、都(03)、都(04)、都(08)、中京(09)
>>> print re.sub(ur"(?u)([東西南北])京", ur"\1都", a)  # replがur文字列になっていることに注意
東都(01)、西都(03)、北都(04)、南都(08)、中京(09)
```

もし、replが関数であれば、その関数は一つのマッチオブジェクト引数を取り、置換文字列を返せばよい。例えば、

```
>>> def replacer(matchObj):
...   print matchObj.string[matchObj.start():matchObj.end()]
...   print matchObj.start(),matchObj.end(),matchObj.group(1)
...   if matchObj.group(1) == u"南":
...     return u"ナンキン"
...   else:
...     return matchObj.string[matchObj.start():matchObj.end()]
...
>>> print re.sub(ur"(?u)([東西南北])京", replacer, a)
東京
0 2 東
西京
7 9 西
北京
14 16 北
南京
21 23 南
```

```
東京(01)、西京(03)、北京(04)、ナンキン(08)、中京(09)
```

正規表現の例

全ての全角記号：[、-〇] 全てのひらがな：[ぁ-ん] 全てのカタカナ：[ァ-ヶ] 第一水準の漢字：[亜-腕] 第二水準の漢字：[弋-熙] 漢数字：[一二三四五六七八九十百千万億兆京、] 携帯電話番号：

```
>>> a = u"""〒171-8504 東京都豊島区西池袋1-11-1 メトロポリタンプラザビル15F
... TEL:(03)3984-6731, 携帯:090-6173-6731, メール:info@nhk.co.jp
... ホームページ：http://www.nhk.co.jp/info.html"""
>>> r = re.compile(ur"""0     # 0
...                     [89]   # 8か9
...                     0      # 0
...                     -?     # -があるかもしれない（0個か1個）
...                     \d{4}  # 数字4桁
...                     -?     # -があるかもしれない（0個か1個）
...                     \d{4}  # 数字4桁
...                     """, re.X)
>>> re.findall(r,a)
['090-6173-6731']
```

メールアドレス：

```
>>> r = re.compile(ur"""[\w.\-]+   # 「文字か.か-」が1個以上
...                     @          #  @
...                     [\w\-]+    # 「文字か-」が1個以上（最初のドメイン）
...                     \.         #  .
...                     [\w.\-]+   # 「文字か.か-」が1個以上
...                   """, re.X)
>>>
>>> re.findall(r,a)
[u'info@nhk.co.jp']
```

ＵＲＬ：

```
>>> r = re.compile(ur"""http       # http
...                     s?         # sがあるかもしれない
...                     ://        # ://
...                     [\w/:%#\$&\?\(\)~\.=\+\-]+   # 「文字か/か:か（ 以下省略）」が1個以上
...                   """, re.X)
>>>
>>> re.findall(r,a)
[u'http://www.nhk.co.jp/info.html']
```

日付：（この例は万能ではない）

```
>>> r = re.compile(ur"""\d{4}  # 数字4桁
...                     [/\.年]    # /か.か「年」
...                     \d{1,2}    # 数字1個か2個
...                     [/\.月]    # /か.か「月」
...                     \d{1,2}    # 数字1個か2個
...                     日?        # 「日」があるかもしれない
...                   """, re.X)
>>> a = u"2000年1月23日,1973/01/01,2014-3-12,西暦６４５年６月１２日"
>>> re.findall(r,a)
[u'2000\u5e741\u670823\u65e5', u'1973/01/01']  #取りこぼしていることに注意
```

藤原氏の末裔：(完璧な検索は無理かも)

```
>>> a = u"""藤原氏の公家諸家は平安末期・鎌倉時代以降、公式文書以外で「藤原」を使わず
... 「近衛」「九条」「鷹司」「二条」「一条」など各家の名称を名乗り、維新後もそれを名字とした。
... そのため、現在は「藤原さん」や「藤のつく苗字の家」は貴族の家系では存在しない。
... 藤原氏に由来する16の苗字を特に「十六藤」といい、佐藤、伊藤、斎藤、加藤、後藤などがある。"""
```

```
>>> r = re.compile(ur"""(?u)[^あ-ん]  # ひらがな以外（正確には[^ぁ-ヴ]にするべき）
...                  藤           # 藤
...                  """, re.X)
>>> for m in re.finditer(r,a):
...   print m.start(), m.end(), m.string[m.start():m.end()]
...
28 30 「藤
92 94 「藤
99 101 「藤
123 125          # 改行も選んでしまっている
藤
142 144 六藤
149 151 佐藤
152 154 伊藤
155 157 斎藤
158 160 加藤
161 163 後藤
```

数字の入力ミス：（ゼロを英字のO、１を英字のlと間違えて入力している場合）

```
>>> def printAll(r,a):  # 正規表現のマッチングを表示する関数
...   for m in re.finditer(r,a):
...     print m.start(), m.end(), m.string[m.start():m.end()]
>>> corrent_text = u"0.01cm,0.02sv/h,0.03mm,0mV,0.040cm,0.015hp"
>>> bad_text = u"0.01cm,O.02sv/h,O.O3mm,OmV,0.O40cm,0.0l5hp" # 間違いが分かる?
>>> r = ur"[0-9\.]+"    # 数字と.が1個以上の連続
>>> printAll(r,corrent_text)
0 4 0.01
7 11 0.02
16 20 0.03
23 24 0
27 32 0.040
35 40 0.015     # 確かに単位を除いた数字が６つ引き出せる
>>> printAll(r,bad_text)
0 4 0.01
8 11 .02
17 18 .
19 20 3
27 29 0.
30 32 40
35 38 0.0
39 40 5
```

もし、単位に英字のOやlがないなら、置換すればよい。

```
>>> replaced_by_0_text = re.sub( ur"[Oo]", ur"0", bad_text )
>>> replaced_by_1_text = re.sub( ur"l", ur"1", replaced_by_0_text )
>>> printAll(r,replaced_by_1_text)
0 4 0.01
7 11 0.02
16 20 0.03
23 24 0
27 32 0.040
35 40 0.015
```

単位が「ml」などと決まっているのなら、最初に単位を除けばよい。

```
>>> bad_text = u"0.01ml,O.02ml,O.O3ml,Oml,0.O40ml,0.0l5ml"
>>> ml_removed_text = re.sub( ur"ml", ur"", bad_text )
>>> replaced_by_0_text = re.sub( ur"[Oo]", ur"0", ml_removed_text )
>>> replaced_by_1_text = re.sub( ur"l", ur"1", replaced_by_0_text )
>>> printAll(r,replaced_by_1_text)
0 4 0.01
5 9 0.02
10 14 0.03
15 16 0
17 22 0.040
```

```
23 28 0.015
```

grepを使う

プログラム以外の利用例を示す。高機能エディタにも正規表現検索がある。ただし、どの言語の正規表現（方言）か、確認しなければならない。

> Windowsの場合、<http://frippery.org/busybox/>をインストールする。 実行ファイルそのものなので、カレントディレクトリに置けば、当座の間は使うことができる。

grepは、ファイルの中の文字列を検索し、位置を表示してくれるコマンド。エディタの検索機能で十分だが、

- 正規表現を使うことができる
- ディレクトリ内部を一気に検索できる

基本は、

```
$ grep "hoge" targetfile.txt
```

ワイルドカードも使うことができる。

```
$ grep "hoge" *.html
```

正規表現を使う場合はEオプションを使う。

```
$ grep -E "\d{4}" targetfile.txt
```

ディレクトリを対象にする場合はrオプションを使う。

```
$ grep -r -E "\d{4}" targetDirectory
```

マッチした文字列の前後の行を表示するためにはAオプション(afterのこと)、Bオプション(beforeのこと)を使う。

```
$ grep -A 3 -B 3 -r -E "\d{4}" targetDirectory
```

このように、大量のファイルがあるとき、関係ある行を探す際に使われる。