@IT
atmarkIT

解決！ Python
正規表現編

かわさきしんじ，Deep Insider 編集部［著］

※ 本 eBook の制作の都合上、Python コード中のシングルクオートやダブルクオート、バックスラッシュ（円マーク）などの記号類が、コードの実行確認に使用した Python 処理系ではシングルクオートやダブルクオート、バックスラッシュなどとして解釈されない文字となっていることがあります。コードをコピー&ペーストして使う際にはご注意ください。

正規表現を使って文字列が数字だけで構成されているかどうかを判定するには

正規表現を扱うための re モジュールの fullmatch 関数などを使用して、文字列が数字だけで構成されているかどうかを判定する方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 02 月 16 日）

```
# 文字列が半角の10進数字のみで構成されているかどうかを判定する関数の例
import re

def isonlynum(s):
    return True if re.fullmatch('[0-9]+', s) else False

s1, s2 = '123', '１２３'
r1 = isonlynum(s1)
r2 = isonlynum(s2)
print(f'{s1}: {r1}, {s2}: {r2}')  # 123: True, １２３: False

# 正規表現を使って、文字列が半角の10進数字のみで構成されているかどうかを判定
import re

s1, s2 = '123', '１２３'
r1 = re.fullmatch('[0-9]+', s1)  # 1文字以上の半角数字
r2 = re.fullmatch('[0-9]+', s2)
print(f'{s1}: {r1}')  # 123: <re.Match object; span=(0, 3), match='123'>
print(f'{s2}: {r2}')  # １２３: None

r1 = re.fullmatch(r'¥d+', s1, re.ASCII)  # 1文字以上の10進数字+ASCIIの範囲内
r2 = re.fullmatch(r'¥d+', s2, re.ASCII)
print(f'{s1}: {r1}')  # 123: <re.Match object; span=(0, 3), match='123'>
print(f'{s2}: {r2}')  # １２３: None

# 文字列が全角を含む10進数字のみで構成されているかどうかを判定
s = '１２３'
result = re.fullmatch(r'¥d+', s)
print(result)  # <re.Match object; span=(0, 3), match='１２３'>
```


→目次に戻る

正規表現を使って、文字列が半角の 10 進数字のみで構成されているかを判定

文字列が半角の（ASCII に含まれる範囲の）10 進数字のみで構成されているかどうかを判定するには正規表現を使える。あるいは文字列が持つ各種メソッドを使ってもよい。後者については「文字列が数字だけで構成されているかどうかを判定するには（isdecimal ／ isdigit ／ isnumeric ／ isascii メソッド）」を参照されたい。

なお、符号や小数点を含んだ文字列を整数値または浮動小数点数値に変換できるかを判定する方法については「文字列が数値へ変換可能かどうかを判定するには（int ／ float 関数の例外、re.fullmatch 関数）」を参照されたい。

re モジュールの fullmatch 関数を使った例を以下に示す。

```
import re

s = '123'
result = re.fullmatch('[0-9]+', s)  # 1文字以上の文字クラス[0-9]
print(result)  # <re.Match object; span=(0, 3), match='123'>

result = re.fullmatch(r'¥d+', s, re.ASCII)
print(result)  # <re.Match object; span=(0, 3), match='123'>

s = '123foo'
result = re.fullmatch('[0-9]+', s)
print(result)  # None
```

最初の例で使っている正規表現は、1 文字以上の半角数字「0」～「9」の並びを表す。re.fullmatch 関数は第 1 引数に指定した正規表現（パターン）が文字列全体とマッチするかを調べるので、ここでは文字列の先頭や末尾を意味する「^」「$」「¥A」「¥Z」を省略している。

次の例で使用している文字クラス「¥d」は Unicode で 10 進数字を意味する General_Category プロパティの値「Nd」を持っている文字全てにマッチする。そのため、re.ASCII フラグを fullmatch 関数の第 3 引数に指定することで、半角の数字のみマッチするようにしている（このフラグを指定しなければ、全角の 10 進数字などもマッチするようになる。後述）。

最後はマッチしなかった場合の例だ。

上のコードに示したように、マッチしたときには re.Match クラスのインスタンスが返送されるが、マッチしなかったときには何も返されない（None 値が返される）。このことを使えば、if 文で条件分岐したり、半角数字だけで構成されているかどうかを判定する関数を定義したりできるだろう。

```
s = '123foo'

if re.fullmatch('[0-9]+', s):
    print(f'{s} includes only decimal numbers')
else:
    print(f'{s} includes non decimal characters')
# 出力結果:
# 123foo includes non decimal characters

def isonlynum(s):
    return True if re.fullmatch('[0-9]+', s) else False

s1, s2 = '123', '123'
r1 = isonlynum(s1)
r2 = isonlynum(s2)
print(f'{s1}: {r1}, {s2}: {r2}')  # 123: True, 123: False
```

なお、re モジュールの match 関数や search 関数を使っても同様な判定は可能だ。

```
s = '123'

result = re.match('[0-9]+$', s)
print(result)  # <re.Match object; span=(0, 3), match='123'>

result = re.search(r'¥A[0-9]+¥Z', s)
print(result)  # <re.Match object; span=(0, 3), match='123'>

s = 'foo123bar456'
result = re.match('[0-9]+$', s)
print(result)  # None

result = re.search(r'¥A[0-9]+¥Z', s)
print(result)  # None
```

re.fullmatch 関数は文字列全体をマッチの対象としたが、re.match 関数は正規表現が文字列の先頭からマッチするかどうかの判定を行い、re.search 関数は文字列中の任意の位置に指定した正規表現とマッチする部分があるかを検索するという違いがある。このため、先ほどとは正規表現が少し異なっていることに注意しよう（先頭を表す「^」と「¥A」および末尾を表す「$」と「¥Z」を上の例では同様に取り扱っているが、マルチラインモードでマッチを行う際には振る舞いが少し変わってくる。これについては本稿では取り上げない）。文字列全体が数字で構成されているかを判定するということで、本稿では re.fullmatch 関数を優先して取り上げた。

文字列が全角を含む 10 進数字のみで構成されているかどうかを判定

ここまで読めば既に答えは出たも同じだが、全角数字「１２３」なども含めて、文字列が 10 進数字のみで構成されているかどうかを判定するには以下の正規表現が使える。

```
# 文字列が全角を含む10進数字のみで構成されているかどうかを判定
s = '123456789'
result = re.fullmatch(r'¥d+', s)
print(result)  # <re.Match object; span=(0, 9), match='123456789'>
```

re.fullmatch 関数に正規表現として「¥d」を渡して、第 3 引数で re.ASCII フラグを指定しないようにすればよい。

正規表現を使って文字列から特定の文字以降の部分や特定の文字より後ろの部分を抽出するには

正規表現を使って、文字列から特定の文字より後ろを抽出するときには、抽出したいパターンをかっこ「()」でグルーピングしておくとよい。

かわさきしんじ，Deep Insider 編集部（2021 年 03 月 30 日）

```
# 正規表現を使って特定の文字(文字列)より後ろにある特定のパターンを抽出
import re

s = 'date: 2021/03/30, time: 05:30'

p = r'time: (.*)'  # 「time: 」の後ろにある時間だけを抽出したい
# p = r'time: ([¥d:]+)'
m = re.search(p, s)
print(m.group(1))  # 05:30

p = r'date: (.*),'  # 「date: 」の後ろにある日付だけを抽出したい
# p = r'date: ([¥d/]+)'
m = re.search(p, s)
print(m.group(1))  # 2021/03/30

p = r': ([¥d/:]+)'  # コロンの後ろにある日付と時刻を抽出したい
r = re.findall(p, s)
print(r)  # ['2021/03/30', '05:30']

p = r': ([¥d/]+).*: ([¥d:]+)'  # コロンの後ろにある日付と時刻を抽出したい
m = re.search(p, s)
print(m.groups())  # ('2021/03/30', '05:30')

p = r': (¥d{4})/(¥d{2})/(¥d{2}).*(¥d{2}):(¥d{2})'  # 数字をバラバラに抽出
m = re.search(p, s)
r = [int(m.group(x)) for x in range(1, 6)]
print(r)  # [2021, 3, 30, 5, 30]

from datetime import datetime

d = datetime(*r)
print(d)  # 2021-03-30 05:30:00
```

正規表現を使って特定の文字（文字列）より後ろにある特定のパターンを抽出

文字列中にあるコロンや半角空白文字など特定の文字あるいは文字列より後ろを抽出しようという場合、文字列の構造が簡単であれば、find メソッドとスライスを組み合わせたり、split メソッドまたは partition メソッドを使ったりすることで、これを実現できる（その方法については「文字列から特定の文字列以降や特定の文字列の前などを抽出するには」を参照のこと）。しかし、find メソッドとスライス、あるいは split メソッドで簡単に抽出できないときには正規表現を使う必要があるかもしれない。

例えば、変数 s に「'date: 2021/03/30, time: 05:30'」という文字列が代入されているとしよう。時刻であれば、find メソッドあるいは rfind メソッドを使えば、正規表現を使わずともそれほど手間をかけることなく抽出できる。

```
s = 'date: 2021/03/30, time: 05:30'

target = 'time: '  # 「time: 」の後ろにある時間だけを抽出したい
idx = s.find(target)
r = s[idx+len(target):]
print(r)  # 05:30

target = ': '  # 「: 」の後ろにある時間だけを抽出したい
idx = s.rfind(target)
r = s[idx+len(target):]
print(r)  # 05:30
```

取り出す対象が文字列末尾にあるので、find メソッドで「time: 」が始まるインデックスを取得して、スライスを使えば簡単に抽出できるだろう（取得したインデックスに検索した文字列の長さを加算するのがポイントだ）。あるいは「: 」を rfind メソッドで右側から検索してもよい。

一方、正規表現を使うと次のように書ける。

```
import re

s = 'date: 2021/03/30, time: 05:30'

p = r'time: (.*)'  # 「time: 」の後ろにある時間だけを抽出したい
# p = r'time: ([¥d:]+)'
m = re.search(p, s)
print(m.group(1))  # 05:30
#print(m.group())  # time: 05:30

r = re.findall(p, s)
print(r)  # ['05:30']
```

ポイントは、「time: 」より後ろにある抽出対象をかっこ「()」でグルーピングすることだ。これにより、re.search メソッドを使ったときには re.Match オブジェクトの group メソッドにグループの番号を指定することで、re.findall メソッドを使ったときには文字列リストの要素として、抽出した部分にアクセスできる（re.findall メソッドは正規表現パターンにグループが含まれている場合は、グループにマッチした部分を要素とするリストを返送する）。

かっこで囲むグループの中は上のコードでは「(.*)」のように任意の長さの任意の文字となっているが、より限定するのなら、コメントにあるように「([¥d:]+)」とすることも考えられる。時刻は数字とコロン「:」の組み合わせとなっているからだ（より厳密に時刻だけを抽出するのなら、さらに詳細な正規表現を記述すべきだが、ここでは割愛する）。

この文字列から今度は日付だけを抽出したいとする。find メソッドとスライスを使うと次のようになるだろう。

```
s = 'date: 2021/03/30, time: 05:30'

# 日付の前後のインデックスを得て、それらを指定してスライスする
target1 = 'date: '
idx1 = s.find(target1) + len(target1)
target2 = ','
idx2 = s.find(target2)
r = s[idx1:idx2]
print(r)  # 2021/03/30
```

split メソッドを使うなら次のようなコードになる。

```
# カンマ「,」で分割した前の要素を、半角空白文字で分割した後の要素が日付
r = s.split(',')[0].split(' ')[1]
print(r)  # 2021/03/30
```

これに対して正規表現を使えば次のように書けるだろう。

```
p = r'date: (.*),'  # 「date: 」の後ろにある日付だけを抽出したい
# p = r'date: ([¥d/]+)'
m = re.search(p, s)
print(m.group(1))  # 2021/03/30
```

これも先ほどと同じで、「date: 」より後ろにある日付にマッチするパターンをかっこでグルーピングしている。ここでは「.*」として任意の文字を指定して、マッチが終わる条件としてカンマ「,」をパターンの最後に足しているが、コメントにあるように「r'date: ([¥d/]+)'」というパターンでもよい（グループ内は数字か日付を区切るスラッシュ「/」の繰り返しであることを表すパターン）。

find メソッドとスライスを使う方法は、その意図はよく分かるが煩雑だ。split メソッドを使う方法は簡潔だが、意味は分かりにくい。これに対して、正規表現は（グルーピングした日付のパターンが貧弱かもしれないが）意図もよく分かるし、コードも簡潔といえる。そのため、筆者としては正規表現を使う方法をオススメしたいところだ。

この文字列から日付と時刻を一度に抽出したいとなったら、正規表現を使うしかないだろう。以下に例を示す。

```
p = r': ([¥d/:]+)'  # コロンの後ろにある日付と時刻を抽出したい
r = re.findall(p, s)
print(r)  # ['2021/03/30', '05:30']
```

この例では、「: 」の後に「数字か日付を区切るスラッシュか時刻を区切るコロン」が連続している部分にマッチするパターンを指定して、re.findall メソッドを呼び出している。これにより日付と時刻の両者が得られる。

re.search メソッドを使うのであれば、次のような書き方もある。

```
p = r': ([¥d/]+).*: ([¥d:]+)'
m = re.search(p, s)
print(m.groups())  # ('2021/03/30', '05:30')
```

正規表現を使えば、日付と時刻を構成する数字だけを抽出し、それらを使って datetime クラスのインスタンスを生成するといった処理も可能だ。以下に例を示す。

```
p = r': (¥d{4})/(¥d{2})/(¥d{2}).*(¥d{2}):(¥d{2})'  # 数字をバラバラに抽出
m = re.search(p, s)
r = [int(m.group(x)) for x in range(1, 6)]
print(r)  # [2021, 3, 30, 5, 30]

from datetime import datetime

d = datetime(*r)
print(d)  # 2021-03-30 05:30:00
```

文字列から特定の文字列以降や特定の文字列の前などを抽出するには（str.find ／ str.split ／ str.partition メソッド、正規表現）

文字列が提供する各種メソッドを使って、指定した文字（文字列）の前後を抽出する方法を紹介。正規表現を使うシンプルな例も取り上げる。

かわさきしんじ，Deep Insider 編集部（2023 年 10 月 04 日）

* 本稿は 2021 年 03 月 23 日に公開された記事を Python 3.12.0 で動作確認したものです（確認日：2023 年 10 月 04 日）。

```
# スライスを用いて特定の文字より後ろを抽出
s = '2021/03/23 05:30'

target = ' '
idx = s.find(target)
r = s[idx+1:]  # スライスで半角空白文字のインデックス+1以降を抽出

print(r)  # 05:30

# スライスを用いて特定の文字より前を抽出(特定の文字以降を削除)
s = '2021/03/23 05:30'

target = ' '
idx = s.find(target)
r = s[:idx]  # スライスで半角空白文字よりも前を抽出

print(r)  # 2021/03/23

# findメソッドに2文字以上の文字列を渡す例
s = 'date: 2021/03/23, time: 05:30'

target = 'time: '  # 「time: 」より後ろ(時刻)を抽出したい
idx = s.find(target)
r = s[idx+len(target):]

print(r)  # 05:30

# splitメソッドを使ってもよい
s = '2021/03/23 05:30'

sep = ' '
t = s.split(sep)  # 半角空白文字で文字列を分割

r = t[0]  # 日付はインデックス0に含まれている
print(r)  # 2021/03/23(特定の文字より前を抽出=特定の文字以降を削除)

r = t[1]  # 時間はインデックス1に含まれている
```

```
print(r)  # 05:30(特定の文字より後ろを抽出=特定の文字までを削除)

# partitionメソッドを使う方法もある
sep = ' '
t = s.partition(sep)  # 戻り値は(sepよりも前, sep, sepよりも後ろ)というタプル

r = t[0]  # 日付はインデックス0に含まれている
print(r)  # 2021/03/23(特定の文字より前を抽出=特定の文字以降を削除)

r = t[2]  # 時間はインデックス2に含まれている
print(r)  # 05:30(特定の文字より後ろを抽出=特定の文字までを削除)

# 正規表現を使って特定の文字(文字列)より後ろにある特定のパターンを抽出
import re

s = 'date: 2021/03/23, time: 05:30'
p = r'time: (.*)'  # 「time: 」の後ろにある時間だけを抽出したい
m = re.search(p, s)

print(m.group(1))  # 05:30

p = r'date: (.*),'  # 「date: 」の後ろにある日付だけを抽出したい
m = re.search(p, s)

print(m.group(1))  # 2021/03/23

p = r': ([¥d:/]+)'  # コロンの後ろにある日付と時刻を抽出したい
r = re.findall(p, s)

print(r)
```

スライスを用いて特定の文字より後ろを抽出

一定の形式で記述されている文字列から、必要な部分だけを取り出したいということはよくあるだろう。空白文字「 」やコロン「:」などの記号に続けて何らかのデータが記述されている場合が例として考えられる。単純な場合は、正規表現を使わずとも文字列のメソッドとスライスを使うだけでそれらを抽出できる。

以下に例を示す。ここでは「'2021/03/23 05:30'」という半角空白文字を挟んで日付と時刻が記述されている文字列を例とする（以下で使用している find メソッドは引数に指定した文字列が見つからないときには -1 を返す。本来はこの値をチェックして、処理を分岐させるべきだが、ここでは省略する。また、文字列の index メソッドを使っても同様に指定した文字のインデックスが得られるが、こちらは指定した文字列が見つからないときには ValueError 例外となる）。

```
s = '2021/03/23 05:30'

target = ' '
idx = s.find(target)  # 半角空白文字のインデックスを検索
r = s[idx+1:]  # スライスで半角空白文字のインデックス+1以降を抽出
# 見つからなかったときを考慮:「r = s[idx+1:] if idx != -1 else 'not found'」

print(r)  # 05:30
```

この文字列から時刻だけを取り出すのであれば、文字列の find メソッドを用いて、空白文字が格納されているインデックスを調べ、「文字列 [空白文字のインデックス +1:]」のようにして、指定した文字よりも右側を抽出すればよい。この場合は特定の文字を抽出する文字列に含みたくないので find メソッドで得たインデックスに 1 を加算しているが、find メソッドに渡した文字を含めて抽出したければ、インデックスはそのままでよい。

特定の文字より後ろを取り出すのではなく、それよりも前の部分を抽出するには「文字列 [: 不要な部分の先頭のインデックス]」とする。以下はその例だ（対象の文字列は最初の例と同じく「'2021/03/23 05:30'」とする）。

```
s = '2021/03/23 05:30'

target = ' '
idx = s.find(target)
r = s[:idx]  # スライスで半角空白文字よりも前を抽出

print(r)  # 2021/03/23
```

ここで覚えておきたいのは、文字列を抽出するとは、それ以外の部分を削除することに他ならないということだ。最初の例は「半角文字列までを削除する」とも考えられるし、上の例は「半角文字列以降を削除する」とも考えられる。

find メソッドには 1 文字だけではなく、任意の長さの文字列を渡してもよい。以下の例は最初の例よりは複雑な「'date: 2021/03/23, time: 05:30'」という文字列から時刻だけを抽出しようというものだ。

```
s = 'date: 2021/03/23, time: 05:30'

target = 'time: '  # 「time: 」より後ろ(時刻)を抽出したい
idx = s.find(target)
r = s[idx+len(target):]

print(r)  # 05:30
```

時刻を取得したいので、ここでは単独のコロン「:」や半角空白文字ではなく、より明確に「'time: '」を引数として find メソッドを呼び出している(コロンの後に半角空白文字が含まれている点に注意)。find メソッドの戻り値は「'time: '」の「't'」があるインデックスとなるので、これに「time: 」の長さを含めて、文字列をスライスすることで望みの結果が得られる。

これまでに見てきた方法で「'date: '」に続く日付を抽出するには、少し面倒くさくなる。こうしたときには、本稿最後に示すように正規表現を使った方がよいかもしれない。

```
s = 'date: 2021/03/23, time: 05:30'

target1 = 'date: '  # 「date: 」の後にある日付を抽出したい
idx1 = s.find(target1)
target2 = ','
idx2 = s.find(target2)
r = s[idx1+len(target1):idx2]

print(r)  # 2021/03/23
```

split ／ partition メソッドを使う方法

同様なことは文字列の split メソッドを使っても行える。split メソッドは引数に渡した文字列を区切りとして、文字列を区切っていく(このときには分割を行う回数も指定できるが、ここでは説明は省略する。詳細は「Python 入門」の「文字列の操作」にある「文字列の分割：split メソッド」を参照のこと)。

以下に例を示す。

```
s = '2021/03/23 05:30'

sep = ' '
t = s.split(sep)  # 半角空白文字で文字列を分割

r = t[0]  # 日付はインデックス0に含まれている
print(r)  # 2021/03/23(特定の文字より前を抽出=特定の文字以降を削除)

r = t[1]  # 時刻はインデックス1に含まれている
print(r)  # 05:30(特定の文字より後ろを抽出=特定の文字までを削除)
```

この例では、半角文字列を区切りとして、日付と時刻を含んだ文字列を分割している。この結果は文字列リストであり、その先頭要素が日付、次の要素が時刻となる。そのため、前者を取得すれば元の文字列から日付を、後者を取得すれば元の文字列から時刻を取得したことになる。

さらに文字列の partition メソッドを使うことも考えられる。このメソッドは引数に指定した文字列を区切り文字として、元の文字列内でその区切りが最初に登場する箇所で文字列を分割する。戻り値は「(区切りよりも前の部分 , 区切り , 区切りよりも後ろの部分)」という 3 要素のタプルとなる。

```
sep = ' '
t = s.partition(sep)  # 戻り値は(sepよりも前, sep, sepよりも後ろ)というタプル

r = t[0]  # 日付はインデックス0に含まれている
print(r)  # 2021/03/23(特定の文字より前を抽出=特定の文字以降を削除)

r = t[2]  # 時間はインデックス2に含まれている
print(r)  # 05:30(特定の文字より後ろを抽出=特定の文字までを削除)
```

正規表現を使って特定の文字（文字列）より後ろにある特定のパターンを抽出

上記のメソッド（やスライス）では簡単に文字列を抽出できないこともあるかもしれない。そうした場合には、正規表現を使うことになるだろう。これについては別稿で詳しく解説するとして、簡単な例を以下に示すだけとしておく。

```
import re

s = 'date: 2021/03/23, time: 05:30'
p = r'time: (.*)'  # 「time: 」の後ろにある時間だけを抽出したい
m = re.search(p, s)

print(m.group(1))  # 05:30

p = r'date: (.*),'  # 「date: 」の後ろにある日付だけを抽出したい
m = re.search(p, s)

print(m.group(1))  # 2021/03/23

p = r': ([¥d:/]+)'  # コロンの後ろにある日付と時刻を抽出したい
r = re.findall(p, s)

print(r)  # ['2021/03/23', '05:30']
```

文字列から特定の文字で囲まれている部分を抽出するには（re.findall 関数）

文字列から特定の文字で囲まれている部分を抽出するには「開き文字」「囲まれている部分」「閉じ文字」というパターンに注目して正規表現を組み立てるとよい。

かわさきしんじ，Deep Insider 編集部（2021 年 03 月 16 日）

```
import re

s = 'this is *sample string* for _extracting substring_.'

# アスタリスクで囲まれている部分を抽出
p = r'¥*.*¥*'  # アスタリスクに囲まれている任意の文字
#p = r'¥*[^*]*¥*'  # アスタリスクに囲まれているアスタリスク以外の文字
r = re.findall(p, s)  # パターンに当てはまるものを全て抽出
print(r)  # ['*sample string*']

# アスタリスクが不要ならグループを使って抽出する部分を指定する
p = r'¥*(.*)¥*'  # アスタリスクに囲まれている任意の文字（アスタリスクを除く）
r = re.findall(p, s)
print(r)  # ['sample string']

# 貪欲マッチと非貪欲マッチ（最小マッチ）
s = 'this is *sample string* for *extracting substring*'
p = r'¥*(.*)¥*'  # 貪欲マッチ（上と同じ）
r = re.findall(p, s)
print(r)  # ['sample string* for *extracting substring']

p = r'¥*(.*?)¥*'  # 非貪欲マッチ（最小マッチ）
r = re.findall(p, s)
print(r)  # ['sample string', 'extracting substring']

p = r'¥*([^*]*)¥*'  # アスタリスクに囲まれたアスタリスク以外の文字（貪欲マッチ）
r = re.findall(p, s)
print(r)  # ['sample string', 'extracting substring']

# アスタリスクかアンダースコアで囲まれている部分を抽出
s = 'this is *sample string* for _extracting substring_.'
p = r'¥*(.*?)¥*|_(.*?)_'  # バーティカルバー「|」で2つの条件を列記
r = re.findall(p, s)
print(r)  # [('sample string', ''), ('', 'extracting substring')]

r = [item[0] + item[1] for item in re.findall(p, s)]
# r = [item[0] if item[0] else item[1] for item in re.findall(p, s)]
print(r)  # ['sample string', 'extracting substring']

p = r'[*_](.*?)[*_]'  # 大かっこ「[]」にアスタリスクとアンダースコアを指定
r = re.findall(p, s)
```

```
print(r)  # ['sample string', 'extracting substring']

# Markdown記法のリンクからテキストとURLを抽出する
s = '[Deep Insider](https://www.atmarkit.co.jp/ait/subtop/di/)は' + ¥
'[@IT](https://www.atmarkit.co.jp/ "@IT")のフォーラムの一つです。'

p = '¥¥[(.*?)¥¥]¥¥((.*?)¥¥s*(?:"(.*?)")?¥¥)'
r = re.findall(p, s)
print(r)
# 出力結果
# [('Deep Insider', 'https://www.atmarkit.co.jp/ait/subtop/di/', ''), ('@IT',
# 'https://www.atmarkit.co.jp/', '@IT')]
```

特定の文字や文字列に囲まれている部分を抽出する基本

文字列から特定の文字（クオート文字や何らかの記号類）や文字列（タグなど）に囲まれている部分を抽出するには、「開き文字」＋「囲まれている文字」＋「閉じ文字」というパターンに注目する。今回は特定の文字に話題を絞って、その考え方や正規表現の書き方を紹介する。

ここでは、Python の正規表現モジュールである re が提供する re.findall 関数を例に取る。なお、re.findall 関数の基本的な使い方については「re.findall 関数と正規表現を使って文字列から部分文字列を抽出するには」を参照されたい。

以下に簡単な例を示す。ここでは Markdown 記法を用いて記述されたテキストから強調を意味するアスタリスク「*」で囲まれている部分を抽出したいものとしよう（以下の変数 s には同じく強調を意味するアンダースコア「_」も含まれているが、これについては後の例で取り上げる）。

```
import re

s = 'this is *sample string* for _extracting substring_.'

p = r'¥*.*¥*'  # アスタリスクに囲まれている任意の文字
r = re.findall(p, s)
print(r)  # ['*sample string*']
```

ここでは「開き文字」と「閉じ文字」はもちろんアスタリスクだ。ただし、正規表現ではアスタリスクは「直前の正規表現の 0 文字以上の繰り返し」という意味を持つ特殊な文字となっているので、正規表現中にアスタリスクを含めるには「¥*」のようにエスケープする必要がある。

「囲まれている文字」は「0 文字以上の任意の文字」と考えられる。この場合、正規表現は「.*」となる。あるいは「0 文字以上のアスタリスク以外の文字」とも考えられる。こちらなら正規表現は「[^*]*」となる（大かっこ「[]」で文字種を指定する場合にはアスタリスクをエスケープする必要はない点に注意）。どちらを採用するかは好みといえるだろう。ここでは前者を採用する。もちろん、「囲まれている文字」について何らかの条件を付加するのであれば、そうした正規表現を書く必要がある（が、本稿では基本的には「.*」とする）。

よって、「開き文字」「囲まれている文字」「閉じ文字」を表す文字列は「r'¥*.*¥*'」という raw 文字列として表現できる。これをパターンとして、re.findall 関数を呼び出すと上の例に示した通り、「['*sample string*']」というアスタリスクで囲まれている部分が無事に抽出できた。

「囲まれている文字」を「0 文字以上のアスタリスク以外の文字」と考えた場合の例も以下に示す。上と同様な文字列が抽出できている点に注目しよう。

```
p = r'¥*[^*]*¥*'  # アスタリスクに囲まれている任意の文字
r = re.findall(p, s)  # パターンに当てはまるものを全て抽出
print(r)  # ['*sample string*']
```

開き文字と閉じ文字は抽出したくない場合

開き文字と閉じ文字は不要というときには、正規表現中でかっこ「()」を使ってグループを指定すればよい。

以下に例を示す。

```
p = r'¥*(.*)¥*'  # アスタリスクに囲まれている任意の文字（アスタリスクを除く）
r = re.findall(p, s)
print(r)  # ['sample string']
```

開き文字と閉じ文字である「¥*」の内部をかっこで囲んでいることに注意しよう。これにより、「['sample string']」のようにアスタリスク以外の部分が抽出できた。

後読みアサーションと先読みアサーションと呼ばれる機能を使う方法もあるが本稿では省略する。

貪欲マッチと非貪欲マッチ（最小マッチ）

ここで、元の文字列が「'this is *sample string* for *extracting substring*'」と強調したい部分をどちらもアスタリスクで囲んであるものとしよう。この場合、先ほどのパターンではうまく文字列を抽出できない。

```
s = 'this is *sample string* for *extracting substring*'

p = r'¥*(.*)¥*'  # 貪欲マッチ(上と同じ)
r = re.findall(p, s)
print(r)  # ['sample string* for *extracting substring']
```

「.*」という正規表現は「貪欲マッチ」といわれ、文字列でマッチする部分が可能な限り長くなるようになっている。そのため、「string」の直後にある「*」なども「0 文字以上の任意の文字」の一つと見なして抽出する。これを回避して、「sample string」と「extracting substring」を抽出するには、「非貪欲マッチ」（または「最小マッチ」）が行われるように指定する。これを行うコードを以下に示す。

```
p = r'¥*(.*?)¥*'  # 非貪欲マッチ(最小マッチ)
r = re.findall(p, s)
print(r)  # ['sample string', 'extracting substring']
```

非貪欲マッチを実行するには、対象の正規表現に続けてクエスチョンマーク「?」を記述すればよい。これにより、なるべく短い文字列にマッチするようになる。そのため、上のコードでは、「sample string」と「extracting substring」が別個に抽出できている。

なお、「囲まれている文字」を「0 文字以上のアスタリスク以外の文字」とした場合には、非貪欲マッチにしなくても問題ない。これはもちろん、アスタリスクが出てきた時点で「[^*]*」のマッチが終わるからだ（アスタリスクは閉じ文字の「¥*」にマッチする）。

```
p = r'¥*([^*]*)¥*'
r = re.findall(p, s)
print(r)  # ['sample string', 'extracting substring']
```

アスタリスクかアンダースコアで囲まれている部分を抽出

ここで元の文字列を先ほどと同じく「'this is *sample string* for _extracting substring_.'」として、ここからアスタリスクまたはアンダースコアで囲まれている部分を抽出する方法を考える。

「開き文字」「閉じ文字」はアスタリスク（¥*）とアンダースコア（_）と異なるが、「囲まれている文字」はこれまでに見た「(.*?)」が使えるだろう（非貪欲マッチ）。このパターンの記述方法は幾つかある。

1 つはアスタリスクで囲まれた文字というパターン「¥*(.*?)¥*」と、アンダースコアで囲まれた文字というパターン「_(.*?)_」をバーティカルバー「|」でつなぐ方法だ。これは、バーティカルバーで区切られた正規表現のいずれかにマッチすることを意味する。以下がそのコードだ。

```
s = 'this is *sample string* for _extracting substring_.'

p = r'¥*(.*?)¥*|_(.*?)_'  # バーティカルバー「|」で2つの条件を列記
r = re.findall(p, s)
print(r)  # [('sample string', ''), ('', 'extracting substring')]
```

この場合、re.findall 関数はバーティカルバーで区切られた正規表現でマッチしたものについては該当部分の文字列を、マッチしなかったものについては空文字列を要素とするタプルを格納するリストを返す。そのため、上では「('sample string', '')」と「('', 'extracting substring')」を要素とするリストが返されている。前者は「¥*(.*?)¥*」に「*sample string*」がマッチして（グループがあるので返値にはアスタリスクは含まれない）、「_(.*?)_」にはマッチしなかったことを意味する（マッチがあった時点で他の正規表現とのマッチは試されない）。後者についても同様だ。

そのため、タプルから要素を取り出すには例えば次のようなコードを書く必要があるだろう。

```
r = [item[0] + item[1] for item in re.findall(p, s)]
# r = [item[0] if item[0] else item[1] for item in re.findall(p, s)]
print(r)  # ['sample string', 'extracting substring']
```

ここでは、タプルの一方の要素は空文字列ではなく、もう一方が空文字列であることを利用して、単純に 2 つの要素を連結し、それを要素とするリストを作成している（コメントアウトしてあるが、if 式（三項演算子）を使っても同様なことは行える）。

「開き文字」と「閉じ文字」を「[*_]」のように表現する方法もある。「囲まれている文字」は「(.*?)」で同じだ。

```
p = r'[*_](.*?)[*_]'  # 大かっこ「[]」にアスタリスクとアンダースコアを指定
r = re.findall(p, s)
print(r)  # ['sample string', 'extracting substring']
```

こちらではバーティカルバーを使って複数のパターンを列挙していないので、戻り値がタプルとなるようなことはない。ただし、非貪欲マッチを行うようにしていないと次のような結果となるので注意が必要だ（アスタリスクとアンダースコアで囲まれている範囲を抽出してしまう）。

```
p = r'[*_](.*)[*_]'  # 貪欲マッチとなっていることに注意
r = re.findall(p, s)
print(r)  # ['sample string* for _extracting substring']
```

最後に応用例として、Markdown のリンク表記からテキストと URL を抽出する例を示す。

```
s = '[Deep Insider](https://www.atmarkit.co.jp/ait/subtop/di/)は' + ¥
'[@IT](https://www.atmarkit.co.jp/ "@IT")のフォーラムの一つです。'

p = r'¥[(.*?)¥]¥((.*?)¥s*(?:"(.*?)")?¥)'
r = re.findall(p, s)
print(r)
# 出力結果
# [('Deep Insider', 'https://www.atmarkit.co.jp/ait/subtop/di/', ''), ('@IT',
# 'https://www.atmarkit.co.jp/', '@IT')]
```

これについては深くは説明しないので、どういうパターンになっているかを読み解いてみてほしい。

文字列から正規表現を使って数字だけを抽出するには（re.findall ／ re.sub ／ re.search 関数）

Python の正規表現モジュールが提供する関数を使って、文字列から数字だけを抽出する方法を紹介する。

かわさきしんじ，Deep Insider 編集部（2021 年 03 月 12 日）

```
import re

s = '2021年03月12日 15時30分'

m = re.findall(r'¥d+', s)  # 文字列から数字にマッチするものをリストとして取得
print(m)  # ['2021', '03', '12', '15', '30']
r = ''.join(m)
print(r)  # 202103121530

r = re.sub(r'¥D', '', s)  # 元の文字列から数字以外を削除=数字を抽出
print(r)  # 202103121530

# re.search関数は最初にマッチするものだけを返送する
m = re.search(r'¥d+', s)
r = m.group()
print(r)  # 2021

# re.search関数で全ての数字を抽出する
m = re.search(r'¥D''(¥d+)¥D+(¥d+)¥D+(¥d+)¥D+ +(¥d+)¥D+(¥d+)¥D+', s)
r = m.groups()
print(r)  # ('2021', '03', '12', '15', '30')
r = m.group(2)
print(r)  # 03
```

文字列を抽出する基本的な方法については「インデックスやスライスを使って文字列から一部を抽出するには」「re.findall 関数と正規表現を使って文字列から部分文字列を抽出するには」「re.search ／ re.match 関数と正規表現を使って文字列から部分文字列を抽出するには」を参照のこと。

文字列から数字部分だけをリストとして取得：re.findall 関数

文字列から数字のみを抽出するには、Python が標準で提供する正規表現モジュール「re」の findall 関数を使うのが簡単だ。数字（Unicode の General_Category プロパティの値が「Nd」である文字）を表す正規表現は「¥d」であり、1 文字以上の繰り返しを意味する「+」と組み合わせて「¥d+」とすることで「1 文字以上の数字」を表せる。これと対象の文字列を re.findall 関数に渡せば、その文字列で「1 文字以上の数字」にマッチする全ての部分を要素とするリストが取得できる。

以下に例を示す。

```
import re

s = '2021年03月12日 15時30分'

m = re.findall(r'¥d+', s)  # 文字列から数字にマッチするものをリストとして取得
print(m)  # ['2021', '03', '12', '15', '30']
```

この例では、数字で構成される部分が 5 つあるが、それらがリストの要素となっていることに注目されたい。マッチした部分を個別に取り出せるので、後から必要な部分を利用するのも簡単だし、ひとまとめにすることも簡単だ。

```
from datetime import datetime

d = datetime(*[int(item) for item in m])  # datetimeクラスのインスタンス生成
print(d)  # 2021-03-12 15:30:00

r = '-'.join(m)  # 取得した文字列を結合
print(r)  # 2021-03-12-15-30
```

1 つ目の例では、取得した文字列要素を整数に変換したリストを作成し、それを展開したものを datetime クラスのコンストラクタに渡すことで、このクラスのインスタンスを生成している。2 つ目の例では単純に文字列を要素とするリストを文字列の join メソッドに渡すことで、新たな文字列を作成している。

文字列に含まれる数字を 1 つの文字列として取得：re.sub 関数

re.findall 関数のように数字にマッチする部分を個別の要素として取り出す必要はなく、全てをひとまとめにした文字列を 1 個だけ取り出せればよいという場合もあるかもしれない。そのときには、re.sub 関数を使って、「数字以外を削除した」＝「数字だけを抽出した」文字列を作成するという方法もある。

以下に例を示す。

```
r = re.sub(r'¥D', '', s)  # 元の文字列から数字以外を削除=数字を抽出
print(r)  # 202103121530
```

数字以外を表す正規表現は「¥D」であり、これを空文字列「''」に置換するように、re.sub 関数に伝えれば、元の文字列から数字以外の文字が削除された（数字だけが抽出された）文字列が得られる。文字列に数字が含まれているのが 1 カ所だけの場合など、こちらの方法で十分ということもあるだろう。

なお、re.search 関数は指定したパターンに最初にマッチした部分を表す re.Match オブジェクトを返送する。

```
m = re.search(r'¥d+', s)
r = m.group()
print(r)  # 2021
```

だが、パターン中でかっこ「()」を使ってグループを指定することで、複数の部分文字列を取得することも可能だ。

```
m = re.search(r'¥D''(¥d+)¥D+(¥d+)¥D+(¥d+)¥D+ +(¥d+)¥D+(¥d+)¥D+', s)

r = m.group()  # マッチ全体
print(r)  # 2021年03月12日 15時30分

r = m.group(2)  # 2つ目のグループの値を取得
print(r)  # 03

r = m.groups()  # 全てのグループの値をタプルとして取得
print(r)  # ('2021', '03', '12', '15', '30')
```

この例では、re.search 関数に渡したパターンにはかっこで囲んだグループが 5 つある。それらは re.Match オブジェクトの group メソッドなどでは 1 ～ 5 の番号を使って特定できる。マッチ全体は引数なしで group メソッドを呼び出す（グループにない部分も含めてマッチした全体が得られる）。特定のグループに対応する文字列を得るには、今述べた番号を group メソッドに渡せばよい。マッチした個々の文字列を全て取得するには、引数なしで groups メソッドを呼び出す。

re.search ／ re.match 関数と正規表現を使って文字列から部分文字列を抽出するには

Python の正規表現モジュール（re）が提供する re.search ／ re.match 関数を使って、文字列からパターンにマッチした部分を抽出する方法を紹介する。re.Match オブジェクトも簡単に取り上げる。

かわさきしんじ，Deep Insider 編集部（2024 年 01 月 08 日）

* 本稿は 2021 年 3 月 9 日に公開された記事を Python 3.12.1 で動作確認したものです（確認日：2024 年 1 月 8 日）。

```
import re

s = 'id: deep, mail: deep@foo.com, tel: 03-0123-4567'

# re.search関数は文字列にパターンとマッチする部分があるかを調べる
m = re.search(r'tel: [-¥d]+', s)
print(m)  # <re.Match object; span=(30, 47), match='tel: 03-0123-4567'>
r = m.group()  # re.Matchオブジェクトのgroupメソッドでマッチ全体を抽出
print(r)  # tel: 03-0123-4567

# re.match関数は文字列の先頭とパターンがマッチするかを調べる
m = re.match(r'tel: [-¥d]+', s)  # 「tel: 03-……」は先頭にはないので
print(m)  # これはNoneとなる

m = re.match(r'¥w+: [-¥w@.]+', s)  # 文字列先頭が「¥w+: [-¥w@.]+」にマッチするか
print(m)  # <re.Match object; span=(0, 8), match='id: deep'>
r = m.group()
print(r)  # id: deep

# re.search関数が返すのは最初にマッチした部分を表すre.Matchオブジェクト
m = re.search(r'¥w+: [-¥w@.]+', s)
print(m)  # <re.Match object; span=(0, 8), match='id: deep'>
print(m.group())  # id: deep

# 全てのre.Matchオブジェクトを取得するにはre.finditer関数を使う
m_iter = re.finditer(r'¥w+: [-¥w@.]+', s)
for m in m_iter:
    print(m.group())
# 出力結果:
# id: deep
# mail: deep@foo.com
# tel: 03-0123-4567

# re.Matchオブジェクトのメソッドを使って抽出する文字列をカスタマイズする
m = re.match(r'id: ¥w+', s)

start = m.start()  # マッチ全体の開始位置を取得
end = m.end()  # マッチ全体の終了位置を取得
```

```
print(f'start: {start}, end: {end}')  # start: 0, end: 8
print(f'span of match: {s[start:end]}')  # span of match: id: deep

start, end = m.span()  # マッチ全体の開始位置と終了位置を取得
print(f'span of match: {s[start:end]}')  # span of match: id: deep

m = re.match(r'id: (¥w+)', s)  # グループを指定
r = m.group()  # マッチ全体を取得
print(r)  # id: deep
r = m.group(1)  # 最初のグループを取得
print(r)  # deep
```

re.search 関数と re.match 関数

Python が標準で提供する正規表現モジュール（re）には、文字列から指定したパターンにマッチする部分を抽出するのに使える関数が幾つか用意されている。本稿ではその中から re.search 関数と re.match 関数を主に取り上げる。なお、re.findall 関数を使って、文字列から部分文字列を抽出する方法については「re.findall 関数と正規表現を使って文字列から部分文字列を抽出するには」を参照されたい。

re.search 関数と re.match 関数はいずれも指定したパターンにマッチする部分が文字列にあるかを調べて、マッチしたらそれを表す re.Match オブジェクトを返す。ただし、re.search 関数はマッチする最初の場所を文字列全体から探すのに対して、re.match 関数は文字列先頭から探す（文字列先頭とパターンがマッチするかどうかを調べる）点が異なる。文字列が特定のパターンを含んでいるかどうか調べて該当する部分を抽出するなら re.search 関数を使い、文字列の先頭が特定のパターンとマッチするかを調べるなら re.match 関数を使うようにするとよい。

以下に例を示す。

```
import re

s = 'id: deep, mail: deep@foo.com, tel: 03-0123-4567'

# re.search関数は文字列にパターンとマッチする部分があるかを調べる
m = re.search(r'tel: [-¥d]+', s)
print(m)  # <re.Match object; span=(30, 47), match='tel: 03-0123-4567'>
r = m.group()  # re.Matchオブジェクトのgroupメソッドでマッチ全体を抽出
print(r)  # tel: 03-0123-4567

# re.match関数は先頭から調べるので以下はNoneが返される
m = re.match(r'tel: [-¥d]+', s)
print(m)  # None
```

この例では、re.search 関数と re.match 関数に同じパターン「tel: [-¥d]+」を渡している。これは「tel: 」に続けて連続する数字とマイナス記号を表す。検索対象の文字列は「'id: deep, mail: deep@foo.com, tel: 03-0123-4567'」なので、「tel: 03-1234-5678」はこれにマッチする。マッチした部分の文字列は re.Match オブジェクトの group メソッドで抽出できる。

re.search 関数は文字列全体で最初にマッチする部分を探すので、これを見つけられる。しかし、re.match 関数は文字列先頭とパターンを比較するので、None が返される。

以下は、re.match 関数で先頭と「¥w+: [-¥w@.]+」というパターンがマッチする例だ。

```
m = re.match(r'¥w+: [-¥w@.]+', s)
print(m)  # <re.Match object; span=(0, 8), match='id: deep'>
r = m.group()
print(r)  # id: deep
```

このパターン「¥w+: [-¥w@.]+」は、「連続する英数字：連続するマイナス記号か英数字か @ か .」を表すので、「id: deep」「mail: deep@foo.com」「tel: 03-1234-5678」にマッチする。ただし、re.match 関数は文字列の先頭部分がパターンにマッチするかどうかを調べるので、これが返送するのはあくまでも「id: deep」を表す re.Match オブジェクトとなる。

同じパターンを re.search 関数に渡すと、やはり「id: deep」を表す re.Match オブジェクトが返される。

```
m = re.search(r'¥w+: [-¥w@.]+', s)
print(m)  # <re.Match object; span=(0, 8), match='id: deep'>
r = m.group()
print(r)  # id: deep
```

re.search 関数ではある文字列の中にマッチする箇所が複数あっても、最初にマッチする部分を表す re.Match オブジェクトしか返されないことには注意しよう。複数のマッチを re.search 関数や re.match 関数と同じく、re.Match オブジェクトの形で取得するには re.finditer 関数を使用する必要がある（複数のマッチを文字列として取得するには re.findall 関数を使用できる）。

以下に例を示す。

```
m_iter = re.finditer(r'¥w+: [-¥w@.]+', s)
for m in m_iter:
    print(m.group())
# 出力結果:
# id: deep
# mail: deep@foo.com
# tel: 03-0123-4567
```

この例では先ほどと同じパターンを、re.finditer 関数に渡している。戻り値はマッチした部分に対応する re.Match オブジェクトを反復するイテレータとなることには注意しておこう。

抽出する文字列のカスタマイズ

これまでに見てきたように、マッチした部分の文字列を抽出するには re.Match オブジェクトの group メソッドを使うのが簡単だが、正規表現でグループを指定して、その部分だけを抽出したいこともあるだろう。こうしたときには、group メソッドに引数を渡したり、re.Match オブジェクトの他のメソッドを使って、マッチした箇所のインデックスを取得したりしてもよい。

ここでは詳しい説明は省略するが、re.Match オブジェクトには以下のようなメソッドがある（一部を抜粋）。

- start メソッド：マッチの開始位置を取得。引数に整数を指定すると対応するグループの開始位置を取得
- end メソッド：マッチの終了位置を取得。引数に整数を指定すると対応するグループの終了位置を取得
- span メソッド：マッチの開始位置と終了位置をタプルで取得。引数に整数を指定すると対応するグループの開始位置と終了位置をタプルで取得
- group メソッド：マッチした部分を文字列として取得。引数に整数を 1 つ指定すると、対応するグループを文字列として取得。整数を複数指定すると、対応するグループの文字列を格納するタプルを取得
- groups メソッド：全てのグループの文字列を格納するタプルを取得

この他にもさまざまなメソッドや属性があるが、それらについては Python の公式ドキュメントにある「マッチオブジェクト」を参照されたい。

以下に例を示す。

```
m = re.match(r'id: ¥w+', s)

start = m.start()  # マッチ全体の開始位置を取得
end = m.end()  # マッチ全体の終了位置を取得
print(f'start: {start}, end: {end}')  # start: 0, end: 8
print(f'span of match: {s[start:end]}')  # span of match: id: deep

start, end = m.span()  # マッチ全体の開始位置と終了位置を取得
print(f'span of match: {s[start:end]}')  # span of match: id: deep
```

最初の例では、start ／ end メソッドを使って、マッチ全体の開始位置と終了位置を取得し、その値を使って文字列のスライスとしてマッチした部分を取り出している。次の例では開始位置と終了位置の取得に span メソッドを使っている。どちらの例でも、メソッドには引数を与えていないので、マッチした全体が得られている。

以下は正規表現にグループを指定している場合の例だ。

```
m = re.match(r'id: (¥w+)', s)  # グループを指定
r = m.group()  # マッチ全体を取得
print(r)  # id: deep
r = m.group(1)  # 最初のグループを取得
print(r)  # deep
```

この例では「(¥w+)」をいうグループ 1 つだけ正規表現中で指定している。group メソッドを引数なしで呼び出した結果は、これまでと同様にマッチ全体となっているが、group メソッドに「1」を渡すと、グループ「(¥w+)」にマッチした「deep」だけが抽出できている点に注目されたい。なお、上で見た start ／ end ／ span の各メソッドでもグループに対応する整数を引数に指定すると、そのグループの開始位置や終了位置を取得できる。

```
m = re.match(r'id: (¥w+)', s)  # グループを指定
r = m.group(1)  # 最初のグループを取得
print(r)  # deep
start, end = m.span(1)
print(s[start:end])  # deep
```

最後に正規表現に複数のグループが含まれている場合の例を示す。これについてはコードの説明は不要だろう。

```
m = re.match(r'(¥w+): (¥w+)', s)  # 複数のグループがある場合
r = m.group()  # マッチ全体を取得
print(r)  # id: deep
r = m.group(2)  # 2つ目のグループを取得
print(r)  # deep
r = m.groups()  # 全てのグループを取得
print(r)  # ('id', 'deep')
```

re.findall 関数と正規表現を使って文字列から部分文字列を抽出するには

Python の正規表現モジュール（re）が提供する re.findall 関数を使って、文字列からパターンにマッチした部分を抽出する方法を紹介する。同様な処理を行う re.finditer 関数も取り上げる。

かわさきしんじ，Deep Insider 編集部（2024 年 02 月 01 日）

* 本稿は 2021 年 3 月 5 日に公開された記事を Python 3.12.1 で動作確認したものです（確認日：2024 年 2 月 1 日）。

```
import re

s = 'id: deep, mail: deep@foo.com, tel: 03-0123-4567'

r = re.findall(r'¥w+:', s)  # 「英数字:」を抽出
print(r)  # ['id:', 'mail:', 'tel:']

r = re.findall('ba[rz]', s)
print(r)  # []  # マッチする部分がなければ空のリストが返される

# パターンにかっこで囲んだグループがあれば、それらにマッチしたものを
# 要素とするリストが返される
r = re.findall(r'(¥w+):', s)  # 「英数字:」の「英数字」だけを抽出
print(r)  # ['id', 'mail', 'tel']

# パターンに複数のグループがあれば、タプルのリストが返される
r = re.findall(r'(¥w+): ([-¥w¥s@.]+)', s)  # キーと値を抽出
print(r)  # [('id', 'deep'), ('mail', 'deep@foo.com'), ('tel', '03-0123-4567')]
d = dict(r)
print(d)  # {'id': 'deep', 'mail': 'deep@foo.com', 'tel': '03-0123-4567'}

# グループ(かっこ)をネストさせる
r = re.findall(r'((¥w+)@([-¥w.]+))', s)
print(r)  # [('deep@foo.com', 'deep', 'foo.com')]

# re.finditer関数はMatchオブジェクトを反復するイテレータを返す
r = re.finditer(r'(¥w+): ([-¥w¥s@.]+)', s)
for m in r:
    print(f'group(0): {m.group(0)}, ', end='')
    print(f'group(1): {m.group(1)}, group(2): {m.group(2)}')
# 出力結果:
# group(0): id: deep, group(1): id, group(2): deep
# group(0): mail: deep@foo.com, group(1): mail, group(2): deep@foo.com
# group(0): tel: 03-0123-4567, group(1): tel, group(2): 03-0123-4567
```

re.findall 関数

Python が標準で提供している re モジュールを使うと正規表現のマッチング処理を行える。このうち、文字列から部分文字列を抽出するには幾つかの関数が使える。本稿では、re.findall 関数と re.finditer 関数を紹介する。

re.findall 関数を呼び出す際には、第 1 引数に正規表現パターンを、第 2 引数に文字列を指定する。文字列中でパターンにマッチした文字列（やそれらを格納するタプル）を要素とするリストとなる。

以下に例を示す。

```
import re

s = 'id: deep, mail: deep@foo.com, tel: 03-0123-4567'

r = re.findall(r'¥w+:', s)  # 「英数字:」を抽出
print(r)  # ['id:', 'mail:', 'tel:']
```

この例では、re.findall 関数の第 1 引数に「¥w+:」というパターンを、第 2 引数に「id: deep, mail: deep@foo.com, tel: 03-0123-4567」という文字列を渡している。「¥w+:」は「1 文字以上の英数字」に続けてコロン「:」を意味するので、文字列の中では「id:」「mail:」「tel:」にマッチする。そのため、戻り値はこれらを要素とするリストとなっている。

なお、パターンにマッチする部分がないときには、空のリストが返される。

```
r = re.findall('ba[rz]', s)
print(r)  # []
```

パターン中にかっこ「()」が含まれている場合には、そのかっこによって形成されるグループを要素とするリストが返送される。以下に例を示す。

```
r = re.findall(r'(¥w+):', s)  # 「英数字:」の「英数字」だけを抽出
print(r)  # ['id', 'mail', 'tel']
```

この例では第 1 引数に渡すパターンが「(¥w+):」となっている。このときには、かっこ内の正規表現「¥w+」にマッチする部分がリストの要素となる。コロンは要素には含まれていないことに注目しよう。

パターンに複数のグループが含まれている場合には、それらにマッチする部分を要素とするタプルが戻り値であるリストの要素となる。以下に例を示す。

```
r = re.findall(r'(¥w+): ([-¥w¥s@.]+)', s)  # キーと値を抽出
print(r)  # [('id', 'deep'), ('mail', 'deep@foo.com'), ('tel', '03-0123-4567')]
```

この例では、かっこで囲まれた部分（グループ）が 2 つあり、パターンが少し長くなっている。1 つ目のグループは「(¥w+)」であり、これは上で見た通り、「id」「mail」「tel」にマッチする。これらのグループに続けてコロンと半角空白文字があり、その後に次のパターン「([-¥w¥s@.]+)」がある。こちらはマイナス記号／英数字／空白文字／アットマーク「@」／ピリオド「.」のいずれかが 1 文字以上連続する部分にマッチする。簡単にいえば、「id: 」の後の英字（deep）、「mail: 」の後のメールアドレス（deep@foo.com）、「tel: 」の後の電話番号（03-1234-5678）が 2 つ目のパターンにマッチする（全体としては「id: deep」といった部分にマッチする）。

パターンが複数あるときには、マッチした各文字列を格納するタプルを要素とするリストが返送されるので、その結果は「[('id', 'deep'), ('mail', 'deep@foo.com'), ('tel', '03-0123-4567')]」となる。こうして文字列からキーと値を抽出できたら、以下のようにして、その情報を辞書に登録するといった使い方も可能だ。

```
d = dict(r)
print(d)  # {'id': 'deep', 'mail': 'deep@foo.com', 'tel': '03-0123-4567'}
```

グループがネストしているときには、外側のグループの内容と内側のグループの内容を格納するタプルを要素とするリストが返される。以下に例を示す。

```
r = re.findall(r'((¥w+)@([-¥w.]+))', s)
print(r)  # [('deep@foo.com', 'deep', 'foo.com')]
```

この例ではパターンが「((¥w+)@([-¥w¥d.]+))」となっている。内側のグループは「(¥w+)」と「([-¥w.]+」の 2 つで、それらがアットマークでつながっている。もちろん、これらはメールアドレスを表す簡素なパターンであり、最初のパターンがユーザー名（ローカル部）に、次のパターンがドメインにマッチするだろう。そして、これらを囲むようにかっこでグループが形成されている。これはメールアドレス全体にマッチする。そのため、戻り値は「(ユーザー名 @ ドメイン , ユーザー名 , ドメイン)」というタプルを要素とするリストとなっている。

re.finditer 関数

これまでに見てきたように、re.findall 関数はリストを戻り値とする。これに対して、re.finditer 関数は同様な処理を行うが Match オブジェクトを反復するイテレータを戻り値とする点が異なる。文字列内からパターンにマッチする全ての Match オブジェクトを取り出すにはこの関数が便利だ（メモリ効率の面で、こちらを呼び出す方が有利なときもあるかもしれない）。

以下に簡単な例を示す。

```
r = re.finditer(r'(¥w+): ([-¥w¥s@.]+)', s)
for m in r:
    print(f'group(0): {m.group(0)}, ', end='')
    print(f'group(1): {m.group(1)}, group(2): {m.group(2)}')
# 出力結果:
# group(0): id: deep, group(1): id, group(2): deep
# group(0): mail: deep@foo.com, group(1): mail, group(2): deep@foo.com
# group(0): tel: 03-0123-4567, group(1): tel, group(2): 03-0123-4567
```

この例では、re.finditer 関数に渡しているパターンは「(¥w+): ([-¥w¥s@.]+)」となっているが、これについては既に述べた通り、全体としては「id: deep」のような部分にマッチする。

ただし、re.findall 関数とは異なり、その戻り値は Match オブジェクトとなる。Match オブジェクトには group メソッドがあり、その引数に「0」を指定した場合（または何も引数を指定しなかった場合）にはマッチ全体が、引数に「1」「2」……を指定した場合はかっこで囲まれたグループが文字列として得られる。そのため、上の例ではマッチした全体と、「id」などのキーに当たる文字列、「deep」などの値に当たる文字列が出力されている。

なお、戻り値はイテレータなので、取り出した Match オブジェクトを遡って取り出すことはできない点には注意しよう。

```
r = re.finditer(r'(¥w+): ([-¥w¥s@.]+)', s)
for m in r:
    print(f'group(0): {m.group(0)}')
# 出力結果:
# group(0): id: deep
# group(0): mail: deep@foo.com
# group(0): tel: 03-0123-4567

for m in r:
    print(m)  # 何も出力されない
```

編集：@IT 編集部
発行：アイティメディア株式会社