02

Expressões regulares

DOWNLOAD

Segue o link (https://s3.amazonaws.com/caelum-online-public/python/07-python.zip) com os arquivos utilizados nesta aula.

Introdução

No capítulo anterior criamos uma aplicação com funcionalidades como criar, alterar, remover e procurar. São as operações típicas também conhecidas como CRUD (Create-Read-Update-Delete). Neste capítulo vamos melhorar nossa funcionalidade de busca.

Antes de mais nada, vamos imaginar a seguinte situação. Temos milhares de perfis cadastrados, bom né? Aí eu te pergunto: quantas vezes na sua rede social favorita você buscou aquela pessoa que você só lembrava do primeiro ou último nome e ficou muito feliz em tê-la encontrado? Será que não podemos dar essa mesma satisfação para nossos usuários? Sim, podemos! Mas primeiro precisaremos adequar nossa busca, que atualmente só funciona se passarmos o nome completo do perfil.

Expressões regulares

Pesquisar por algum critério numa string não é exclusividade de redes sociais sendo uma prática muito difundida. Ela é tão difundida que foi criada uma sintaxe própria com esta finalidade chamada **Expressões Regulares** ou simplesmente **regex**. Continuaremos usando Python, mas o que aprenderemos neste capítulo poderá ser utilizado com outras linguagem que dão suporte à regex. Vamos dar uma olhada neste poderoso padrão!

O módulo RE

O Python não carrega por padrão toda a parafernália para trabalharmos com expressões regulares, pois nem sempre queremos utilizar este recurso. É por isso que precisamos explicitar seu carregamento através da instrução `import`. Em nosso caso, queremos importar o módulo `re`, responsável pelas expressões regulares:

```
>>> import re
```

Agora que já temos o módulo carregado, que tal entendermos um pouco sobre expressão regular? Não esqueça que nosso objetivo é entender o suficiente para conseguirmos melhorar a busca de nosso programa.

A função match

O modulo `re` possui uma série de funções para trabalharmos com expressões regulares. Independente do nome da função e da quantidade de parâmetros, temos de um lado o **padrão** que queremos encontrar e do outro o **pedaço de texto** que iremos vasculhar utilizando esse padrão.

Vamos começar pela função `re.match`. Ela recebe como primeiro parâmetro o padrão e como segundo uma string que será vasculhada. Vejamos o exemplo no qual procuramos o padrão `Py` dentro da string `Python`:

```
>>> resultado = re.match('Py','Python')
```

Então, `Py` é nossa expressão! Mas qual é o valor do resultado? Vamos imprimi-lo:

```
>>> resultado = re.match('Py','Python')
>>> resultado
<_sre.SRE_Match object at 0x108bd48b8>
```

Hum, ele é um objeto. Sabemos que objetos são dotados de dado e comportamento, então deve haver algum atributo ou método que nos informe o resultado da pesquisa. Claro que há, vamos chamar o método `group` :

```
>>> resultado = re.match('Py','Python')
>>> resultado.group()
'Py'
```

Encontrou o `Py` dentro do `Python` ! Faz sentido. E se tentamos procurar por `py` em minúsculo?:

```
>>> resultado = re.match('py','Python')
>>> resultado.group()
AttributeError: 'NoneType' object has no attribute 'group'
```

O erro é um pouco estranho, não? Vamos tentar decifrá-lo. Bem, parece que o objeto do tipo `None` não contém o método `group` . É verdade. Quando `match` não encontra o padrão procurado, ela retorna um objeto do tipo `None` que significa "nenhum" em português.

```
>>> type(resultado)
<type 'NoneType'>
```

Agrupando caracteres

Nem sempre queremos levar em consideração se o padrão procurado está em maiúscula ou minúscula, como é o caso da pesquisa por perfis de outras redes sociais. Nossa sorte é que podemos definir uma faixa de caracteres e assim podemos dizer que estamos procurando por `p` ou `P` . Vamos testar isso e criar um grupo na expressão regular usando colchetes `[..]` :

```
>>> resultado = re.match('[pP]y','Python')
>>> resultado.group()
'Py'
>>> resultado = re.match('[pP]y','python')
>>> resultado.group()
'py'
```

Perfeito! E se a quisermos achar todas as sílabas que começam com qualquer letra seguida de y? Basta alterar a nossa faixa e deixá-la mais genérica. Podemos definir todas as caracteres de A até Z pela expressão:

```
>>> resultado = re.match('[A-Za-z]y','Python')
>>> resultado.group()
'Py'
```

Ah, mas só temos uma palavra em nossa string, que tal adicionar mais uma e verificar se encontramos nosso padrão nas duas palavras?

```
>>> resultado = re.match('[A-Za-z]y','Python ou jython')
>>> resultado.group()
'Py'
```

Estamos procurando a expressão `[A-Za-z]y` dentro do texto `Python ou jython`. As palavras 'Python' e 'jython' atendem nosso critério, no entanto, só recebemos o `Py` como resposta. O problema é que a função `match` só encontrou a primeira ocorrência, parando logo em seguida. Nesse exemplo queremos achar `Py` E `jy`.

A função findall

Podemos utilizar no lugar da função `match` a função `findall`. Esta função devolve uma lista de resultados:

```
>>> resultados = re.findall('([A-Za-z]y)','Python ou jython')
>>> resultados
['Py','jy']
```

Repare que ela retorna uma lista com os padrões encontrados. Diferente da função `match` que retornava um objeto.

Meta caracteres

Perfeito, mas agora queremos a palavra completa como retorno. Queremos todas as palavras que começam com expressão `[A-Za-z]y`. O truque é definirmos uma faixa ainda maior de caracteres, indicado pelo operador `+`. Este operador significa um ou mais caracteres. Vamos adicionar a expressão `[A-Za-z]+` após a já existente:

```
>>> resultados = re.findall('([A-Za-z]y[A-Za-z]+)','Python ou jython ou PyPy')
>>> resultados
['Python', 'jython', 'PyPy']
```

Caso queiramos buscar qualquer caracter, inclusive considerando números, podemos para isso podemos usar `[A-Za-z0-9]` como faixa. Porém, por ser uma necessidade tão comum, podemos utilizar o atalho `\w`. Reescrevendo a expressão:

```
>>> resultados = re.findall('(\wy\w+)','Python ou jython ou PyPy')
>>> resultados
['Python', 'jython', 'PyPy']
```

Novamente, o `\w` também incorpora números. Não acredita? Vamos alterar nossa string:

```
>>> resultados = re.findall('(\wy\w+)','Python3 ou jython2 ou PyPy')
>>> resultados
```

```
['Python3', 'jython2', 'PyPy']
```

Mas só uma observação: `\w` não leva em consideração acentos. Infelizmente devemos configurar esses caracteres separadamente, como por exemplo: `\w|á|é` (\w ou á ou é)*

Existem outros atalhos como o `\d` que identifica apenas números ou `\s` para *whitespaces* como espaço ou tabulação. No exemplo a seguir, procuraremos por qualquer palavra que contenha um `y` como segunda letra, mas agora com um número no final:

```
>>> resultados = re.findall('(\wy\w+\d)','Python3 ou jython2 ou PyPy')
>>> resultados
['Python3','jython2']
```

Também poderíamos quantificar o número no final. Caso o número seja opcional podemos usar o operador `?` que significa **zero ou um**. Se quisermos *zero ou mais* números no final podemos usar o asterisco `*`.

Por exemplo, a expressão `[A-Za-z]+\d?` pega qualquer palavra com as letra de A-Z independente de minuscula ou maiúscula contendo opcionalmente um número:

```
>>> resultados = re.findall('[A-Za-z]+\d?','Python3 ou jython ou PyPy44')
>>> resultados
['Python3', 'ou', 'jython', 'ou', 'PyPy4']
```

Raw String

Repare que usamos muitos caracteres especiais para definir uma expressão regular. Existem vários outros como o **.** (ponto) que indica qualquer caracter, ou **()** (parênteses) para definir grupos de expressões. Alguns dos caracteres possuem um significado especial dentro de uma string e pode ocorrer um conflito entre a definição da string e a expressão regular. Ou seja, antes que o módulo `re` interprete a expressão a string já fez uma alteração da mesma!

Para fugir desses problemas devemos usar uma string crua, ou em inglês **raw string**. Para definir uma *raw string* devemos prefixar a string com a letra `r`, por exemplo `r'[A-Z]+'`. Não confunda o `r` com regex, `r` significa `raw`. A partir de agora sempre vamos usar *raw strings* para declarar expressões regulares!

Mais quantificadores

Repare com os regex que vimos aqui já podemos executar uma busca mais poderosa encontrarmos o nome de um perfil. Tendo quatro nomes, por exemplo: 'Nico Flavio Fabiana Romulo', podemos facilmente buscar todos os nomes que começam com a letra F:

```
>>> resultados = re.findall('([fF]\w+)','Nico Flavio Fabiana Romulo')
>>> resultados
['Flavio', 'Fabiana']
```

Vamos dificultar um pouco mais. Queremos buscar todos os nomes que começam com uma determinada letra, mas devem ter uma quantidade mínima de caracteres. Digamos que todos os nomes que começam com a letra F têm pelo menos 6 caracteres! Já vimos os quantificadores como +,? e *, mas existe uma forma exata de dizer quantos caracteres um resultado

deveria ter. Para isso usaremos as `{}` (chaves). Por exemplo, a expressão seguinte pega todos os nomes que começam com `F` e possuem mais de 6 caracteres: `r'[fF]\w{6}'` :

```
>>> resultados = re.findall(r'[fF]\w{6}','Nico Flavio Fabiana Romulo')
>>> resultados
['Fabiana']
```

Para buscar nomes entre 6 e 20 caracteres basta colocar: `r'[fF]\w{6,20}'` .

Buscar no início e fim da string

Para terminar, veremos rapidamente como buscar algo pelo início ou pelo fim da string. Quando usarmos as funções `match` ou `findall` vimos que elas procuram alguma ocorrência ou todas as ocorrências da expressão em um texto. Isso nem sempre é suficiente.

Por exemplo, quando realizamos leitura de um arquivos, muitas vezes queremos uma linha que comece com uma palavra ou símbolo especial. Neste caso, faz sentido lermos linha a linha e verificarmos se a linha inicia com aquele símbolo ou não. Não queremos saber se a linha contém a expressão, e sim se ela começa com a expressão.

Com expressões regulares procuramos pelo início através do caracter **^** (circunflexo). Por exemplo, para pegar um texto que começa com `#` (tralha) usaremos a expressão `r'^#'` .

```
>>> resultado = re.match(r'^#','#comentarios começam com tralha')
>>> resultado is None
False
```

Analogamente podemos usar o caracter **$** para buscar pelo final da string. Para saber se uma string termina com `br` podemos usar a expressão: r'*.br$' Repare o $ no final da expressão.* `br` *deve estar no final (* `$` *) e antes do* `br` *podem vir quaisquer caracteres, zero ou mais vezes (*`.`*)*:

```
>>> resultado = re.match(r'.*br$','http://alura.com.br')
>>> resultado.group()
'http://alura.com.br'
```

Há ainda muito para aprender sobre expressões regulares e este capítulo representa apenas uma introdução, mas já é o suficiente para praticarmos. Mãos à obra!