Universidade Federal de Uberlândia

Faculdade de Engenharia Elétrica

Disciplina: Engenharia de Software

Tutorial sobre o capítulo 7 do livro:

Automate the Boring Stuff with Python: Practical Programming for Total Beginners

Autor:
Gustavo Machado

Professor:
Dr. Marcelo Rodrigues

26 de abril de 2016

Sumário

1 O que são expressões regulares?

Expressões regulares, muita das vezes abreviadas por “regexes”, são cadeias de caracteres que associam sequências de caracteres em um texto. Essas expressões nos permitem especificar um padrão para encontrá-lo no texto.

No Brasil , os números de telefone são da forma XX-XXXXX-XXXX. Por exemplo, 34-99042-9878 é um número de telefone, enquanto 34,99042,9878 não é um número de telefone. Nesse caso, as expressões regulares podem ser bem úteis para fazer essa verificação.

2 Encontrando padrões sem a utilização de expressões regulares:

Vamos criar uma função chamada eNumeroDeTelefone() para verificar esse padrão citado anteriormente, que retorna True(verdadeiro) quando o número for de telefone ou False(falso) quando o número não for de telefone. Para isso, abra a sua IDE do Python e crie um novo arquivo com o nome que desejar, com o código a seguir:

```
def eNumeroDeTelefone(texto):
        if len(texto) != 13:
                return False
        for i in range(0,2):
                if not texto[i].isdecimal():
                        return False
        if text[2] != '-':
                return False
        for i in range(3, 8):
                if not text[i].isdecimal():
                        return False
        if text[8] != '-':
                return False
        for i in range(9, 13):
                if not text[i].isdecimal():
                        return False
        return True

print('34-99042-9878 é um número de telefone?')
print(eNumeroDeTelefone('34-99042-9878'))
print('34,99042,9878 é um número de telefone?')
print(eNumeroDeTelefone('34,99042,9878'))
```

Observe que, quando você compilar o código, a saída na tela será:

```
34-99042-9878 é um número de telefone?
True
34,99042,9878 é um número de telefone?
False
```

Primeiramente, dentro da função eNumeroDeTelefone(), verificamos se o texto digitado contém 13 caracteres. Posteriormente, verificamos, com um laço for, se os dois primeiros dígitos são números decimais e depois verificamos se o próximo digito é um hífen. Seguindo essa lógica até o último caractere, se nenhum dos casos retornar falso, a função retornará verdadeiro, comprovando que o texto digitado é um número de telefone.

3 Encontrando padrões com a utilização de expressões regulares:

Observe que o código utilizado anteriormente funciona perfeitamente, porém toma muitas linhas para realizar uma tarefa relativamente fácil. Veremos agora como as expressões regulares podem simplificar essa tarefa.

Por exemplo, um \d em uma expressão regular pode significar qualquer número natural entre 0 e 9. A regex \d\d-\d\d\d\d\d-\d\d\d\d é usada pelo Python determinando como é o formato em que um número de telefone se encaixa, simplificando bastante a função eNumeroDeTelefone(). Além disso, podemos simplificar ainda mais o código, colocando o número de dígitos que deseja entre chaves, após o \d, da seguinte forma \d{2}-\d{5}-\d{4}.

4 Criando objetos regex:

A maioria das funções regex do Python estão contidas no módulo re, portanto é necessário que você importe esse módulo toda vez que for utilizar uma função regex. Veja:

```
>>> import re
```

A função re.compile(), retorna um objeto regex padrão, por exemplo:

```
NumeroDeTelefoneRegex = re.compile(r'\d\d\-\d\d\d\d\d-\d\d\d\d')
```

Dessa forma, a variável NumeroDeTelefoneRegex contém um objeto regex.

5 Combinando objetos regex:

O código seguinte, por meio do método search() irá retornar o objeto regex correspondido caso o padrão estabelecido na variável NumeroDeTelefoneRegex seja encontrado, caso contrário, retornará None(vazio). O método group() irá retornar o texto atual correspondido da string pesquisada. A variável é denominada "mo"de forma genérica apenas para exemplificar "Match object".

```
>>> NumeroDeTelefoneRegex = re.compile(r'\d\d\-\d\d\d\d\d-\d\d\d\d')
>>> mo = NumeroDeTelefoneRegex.search('Meu número de telefone é 34-99042-9878.')
>>> print('Número de telefone encontrado: ' + mo.group())
```

Esse código exibirá na tela o seguinte:

```
Número de telefone encontrado: 34-99042-9878
```

6 Agrupando por parênteses:

Adicionando parênteses pode-se criar grupos regex, como por exemplo, separando o DDD do número de telefone dos demais dígitos, da seguinte forma: (\d\d)-(\d\d\d\d\d-\d\d\d\d).

Assim , pode-se utilizar o método group() para pegar o grupo desejado. Veja o código abaixo:

```
>>> NumeroDeTelefoneRegex = re.compile(r'(\d\d)-(\d\d\d\d\d-\d\d\d\d)')
>>> mo = NumeroDeTelefoneRegex.search('Meu número de telefone é 34-99042-9878.')
>>> mo.group(1)
 '34'
>>> mo.group(2)
'99042-9878'
>>> mo.group(0)
'34-99042-9878'
>>> mo.group()
'34-99042-9878'
```

7 Combinando vários grupos com o caractere Pipe (|):

Você pode utilizar o caractere | para encontrar uma string desejada em um texto. Por exemplo, a expressão regular r'Sheldon|Leonard , caso encontrado somente um dos nomes em um texto Sheldon ou Leonard, será retornado esse nome, caso os dois sejam encontrados, será retornado o primeiro que for encontrado. Veja o exemplo a seguir:

```
>>> heroRegex = re.compile(r'Sheldon|Leonard')
>>> mo1 = heroRegex.search('Sheldon e Leonard.')
>>> mo1.group()
'Sheldon'
>>> mo2 = heroRegex.search('Leonard e Sheldon.')
>>> mo2.group()
'Leonard'
```

8 Correspondência opcional com ponto de interrogação:

No código abaixo, a parte (wo)? da expressão regular significa que o padrão "wo"é um grupo opcional. Veja:

```
>>> batRegex = re.compile(r'Bat(wo)?man')
>>> mo1 = batRegex.search('As aventuras do Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('As aventuras da Batwoman')
>>> mo2.group()
'Batwoman'
```

9 Correspondência de zero ou mais com asterisco:

No exemplo a seguir, para 'Batman', a parte (wo)* da regex corresponde a zero casos de "wo"na string, para 'Batwoman', a parte (wo)* corresponde a um caso de "wo"na string, já para 'Batwowowowoman', (wo)* corresponde a quatro casos de "wo"na string. Veja:

```
>>> batRegex = re.compile(r'Bat(wo)*man')
>>> mo1 = batRegex.search('As aventuras do Batman')
>>> mo1.group()
'Batman'
>>> mo2 = batRegex.search('As aventuras da Batwoman')
>>> mo2.group()
'Batwoman'
>>> mo3 = batRegex.search('As aventuras da Batwowowowoman')
>>> mo3.group()
'Batwowowowoman'
```

10 Correspondência de um ou mais com o caractere de adição (+) :

Esse caso é bem parecido com o anterior, a diferença é que nesse, a parte (wo) deve corresponder a pelo menos um caso na string, caso contrário retornará None(nada). Veja:

```
>>> batRegex = re.compile(r'Bat(wo)+man')
>>> mo1 = batRegex.search('As aventuras da Batwoman')
>>> mo1.group()
'Batwoman'
>>> mo2 = batRegex.search('As aventuras da Batwowowowoman')
>>> mo2.group()
'Batwowowowoman'
>>> mo3 = batRegex.search('As aventuras do Batman')
>>> mo3 == None
True
```

11 Combinando repetições específicas com chaves:

São utilizadas para repetir strings em um número desejado de vezes. Por exemplo, a regex (Ha){3} corresponderá a string 'HaHaHa'.

Você também pode especificar um intervalo para fazer repetições. Por exemplo, a regex(Ha){3,5} corresponderá as strings 'HaHaHa', 'HaHaHaHa', e 'HaHaHaHaHa'. Veja o código a seguir:

```
>>> haRegex = re.compile(r'(Ha){3}')
>>> mo1 = haRegex.search('HaHaHa')
>>> mo1.group()
'HaHaHa'
>>> mo2 = haRegex.search('Ha')
>>> mo2 == None
True
```

12 O método findall():

Diferentemente do método search(), o método findall() retorna todos os objetos correspondidos do texto. Veja os seguintes exemplo:

12.1 O método search():

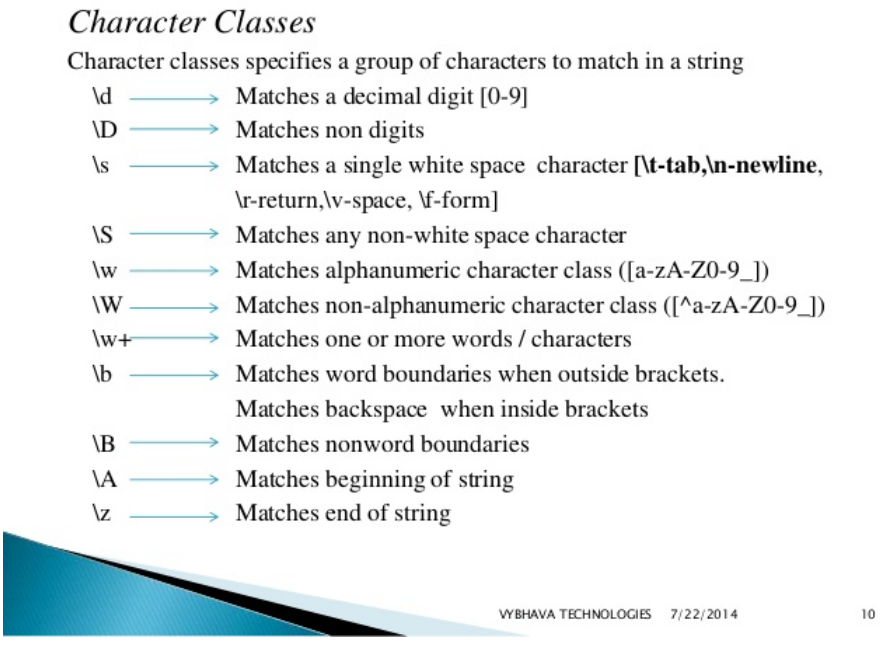
```
>>> NumeroDeTelefoneRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> mo = NumeroDeTelefoneRegex.search('Telefone pessoal: 34-99098-3465  Telefone do trabalho: 34-99092-2345')
>>> mo.group()
'34-99098-3465'
```

12.2 O método findall():

```
>>> NumeroDeTelefoneRegex = re.compile(r'\d\d\d-\d\d\d-\d\d\d\d')
>>> NumeroDeTelefoneRegex.findall('Telefone pessoal: 34-99098-3465  Telefone do trabalho: 34-99092-2345')
['34-99098-3465', '34-99092-2345']
```

13 Classes de caracteres:

Anteriormente, vimos que o \d pode corresponder a qualquer número natural entre 0 e 9. Esse \d corresponde, portanto, a uma classe de caracteres. As classes de caracteres são ótimas para diminuir o tamanho do código e facilitar a vida do programador em seus algoritmos. Na tabela abaixo, você verá algumas outras classes de caracteres:



Exemplo de código usando algumas das classes de caracteres da tabela:

```
>>> xmasRegex = re.compile(r'\d+\s\w+')
>>> xmasRegex.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids, 7 swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '6 geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

14 Criando suas próprias classes de caracteres:

Você pode definir suas própias classes de caracteres utilizando colchetes, como no exemplo a seguir, no qual a classe de caractere [aeiouAEIOU] irá corresponder a toda vogal que a string conter, sendo minúscula ou maiúscula. Veja:

```
>>> VogalRegex = re.compile(r'[aeiouAEIOU]')
>>> VogalRegex.findall('You are a good PROGRAMMER.')
['o', 'u', 'a', 'e', 'a', 'o', 'o', 'O', 'A', 'E']
```

15 Os caracteres dólar($) e circunflexo(ˆ):

O caractere circunflexo(ˆ) no início de uma regex indica o início do objeto a ser correspondido, enquanto que o caractere dólar($) no fim de uma regex indica o fim do objeto a ser correspondido. Quando juntos, os caracteres circunflexo(ˆ) e dólar($) indicam a string completa que irá ser correspondida. Veja o exemplo a seguir:

```
>>> wholeStringIsNum = re.compile(r'^\d+$')
>>> wholeStringIsNum.search('1234567890')
<_sre.SRE_Match object; span=(0, 10), match='1234567890'>
>>> wholeStringIsNum.search('12345xyz67890') == None
True
>>> wholeStringIsNum.search('12  34567890') == None
True
```

16 O caractere curinga:

O caractere .(ponto) em uma expressão regular é chamado de curinga. Esse caractere irá corresponder a um(somente) qualquer caractere com exceção à uma nova linha. Veja o exemplo a seguir:

```
>>> atRegex = re.compile(r'.at')
>>> atRegex.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

17 O caractere ponto-asterisco(.*):

É utilizado para corresponder a todo o texto que vier depois da string pré-estabelecida. Veja o exemplo a seguir:

```
>>> nameRegex = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> mo = nameRegex.search('First Name: Donald Last Name: Knuth')
>>> mo.group(1)
'Donald'
>>> mo.group(2)
'Knuth'
```

18 Letras maiúsculas ou minúsculas (Case-insensitive):

Quando precisa-se, em geral, apenas do significado de uma palavra, sem se preocupar com letras maiúsculas ou minúsculas, você pode passar como um segundo argumento re.IGNORECASE ou re.I para re.compile(). Veja o exemplo a seguir:

```
>>> robocop = re.compile(r'robocop', re.I)
>>> robocop.search('O RoboCop é legal!').group()
'RoboCop'
>>> robocop.search('O ROBOCOP é legal!').group()
'ROBOCOP'
>>> robocop.search('O robocop é legal!').group()
'robocop'
```

19 Substituindo strings com o método sub():

Expressões regulares também podem ser utilizadas para substituir um novo texto no lugar do padrão estabelecido. Para usar o método sub(), você deve passar dois argumentos, sendo o primeiro a string a ser substituída, e o segundo argumento a string para a expressão regular. Veja o exemplo a seguir:

```
>>> nomesRegex = re.compile(r'João \w+')
>>> nomesRegex.sub('Pedro', 'João é uma boa pessoa, mas às vezes João comete erros.')
'Pedro é uma boa pessoa, mas às vezes Pedro comete erros.'
```

20 Expressões regulares complexas:

Quando for necessário trabalhar com expressões regulares maiores e com um maior número de termos, você pode passar a variável re.VERBOSE() como segundo argumento para a re.compile() para organizar melhor o seu código. Por exemplo, veja o seguinte código:

```
NumeroDeTelefoneRegex = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-|\.)\d{4} (\s*(ext|x|ext.)\s*\d{2,5})?)')
```

Como você pode notar, esse código está um pouco confuso. Para ficar mais "organizado", você pode fazer o seguinte:

```
phoneRegex = re.compile(r'''(
(\d{3}|\(\d{3}\))?            # area code
(\s|-|\.)?                    # separator
\d{3}                         # first 3 digits
(\s|-|\.)                     # separator
\d{4}                         # last 4 digits
(\s*(ext|x|ext.)\s*\d{2,5})?  # extension
)''', re.VERBOSE)
```

21 Combinando re.IGNORECASE, re.DOTALL e re.VERBOSE:

Você pode escrever comentários na regex usando a re.VERBOSE e também utilizar o re.IGNORECASE para ignorar as letras maiúsculas e minúsculas. Além de que você também pode utilizar a re.DOTALL juntamente com as outras, para isso você deve utilizar o caractere pipe(|). Veja:

```
>>> someRegexValue = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

22 PROJETO: Encontrando padrões de números de telefone e endereços de email.

Suponha que você more nos Estados Unidos, você deve saber que os números de telefone de lá são da forma XXX-XXX-XXXX. Vamos criar um programa em Python para encontrar esse padrão em um texto. Além disso,

vamos também encontrar o padrão de um email em um texto.

22.1 Passo 1 - Criando uma expressão regular para pesquisar números de telefone:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.
import pyperclip, re
phoneRegex = re.compile(r'''(
(\d{3}|\(\d{3}\))?               # area code
(\s|-|\.)?                       # separator
(\d{3})                          # first 3 digits
(\s|-|\.)                        # separator
(\d{4})                          # last 4 digits
(\s*(ext|x|ext.)\s*(\d{2,5}))?   # extension
)''', re.VERBOSE)
# Passo 2: Create email regex.
# Passo 3: Find matches in clipboard text.
# Passo 4: Copy results to the clipboard.
```

22.2 Passo 2 - Criando uma expressão regular para pesquisar endereços de email:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.
import pyperclip, re
phoneRegex = re.compile(r'''(
--snip--
# Create email regex.
emailRegex = re.compile(r'''(
     [a-zA-Z0-9._%+-]+      # username
     @                      # @ symbol
   [a-zA-Z0-9.-]+         # domain name
(\.[a-zA-Z]{2,4})      # dot-something
 )''', re.VERBOSE)
# Passo 3: Find matches in clipboard text.
# Passo 4: Copy results to the clipboard.
```

22.3 Passo 3 - Encontrando todas as correspondências na área de transferência de um texto:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.
import pyperclip, re
phoneRegex = re.compile(r'''(
--snip--
# Find matches in clipboard text.
text = str(pyperclip.paste())
matches = []
for groups in phoneRegex.findall(text):
        phoneNum = '-'.join([groups[1], groups[3], groups[5]])
        if groups[8] != '':
            phoneNum += ' x' + groups[8]    matches.append(phoneNum)
 for groups in emailRegex.findall(text):
        matches.append(groups[0])
# Passo 4: Copy results to the clipboard.
```

22.4 Passo 4 - Juntando tudo:

```
#! python3
# phoneAndEmail.py - Finds phone numbers and email addresses on the clipboard.
--snip-
for groups in emailRegex.findall(text):
    matches.append(groups[0])
# Copy results to the clipboard.
 if len(matches) > 0:
     pyperclip.copy('\n'.join(matches))
     print('Copied to clipboard:')
     print('\n'.join(matches))
 else:
 print('No phone numbers or email addresses found.')
```

22.5 Passo 5 - Compilando o código :

Finalmente, abra pelo seu navegador, a página http://www.nostarch.com/contactus.htm, pressione ctrl-A para selecionar todo o texto que está contido na página. Depois disso, pressione ctrl-C para copiar isso para a clipboard. Quando você rodar esse programa, você verá os seguintes números e emails na tela:

```
800-420-7240
415-863-9900
415-863-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
help@nostarch.com
```

Referências

[Sweigart, 2015] Sweigart, A. (2015). *Automate the Boring Stuff with Python: Practical Programming for Total Beginners*. No Starch Press.