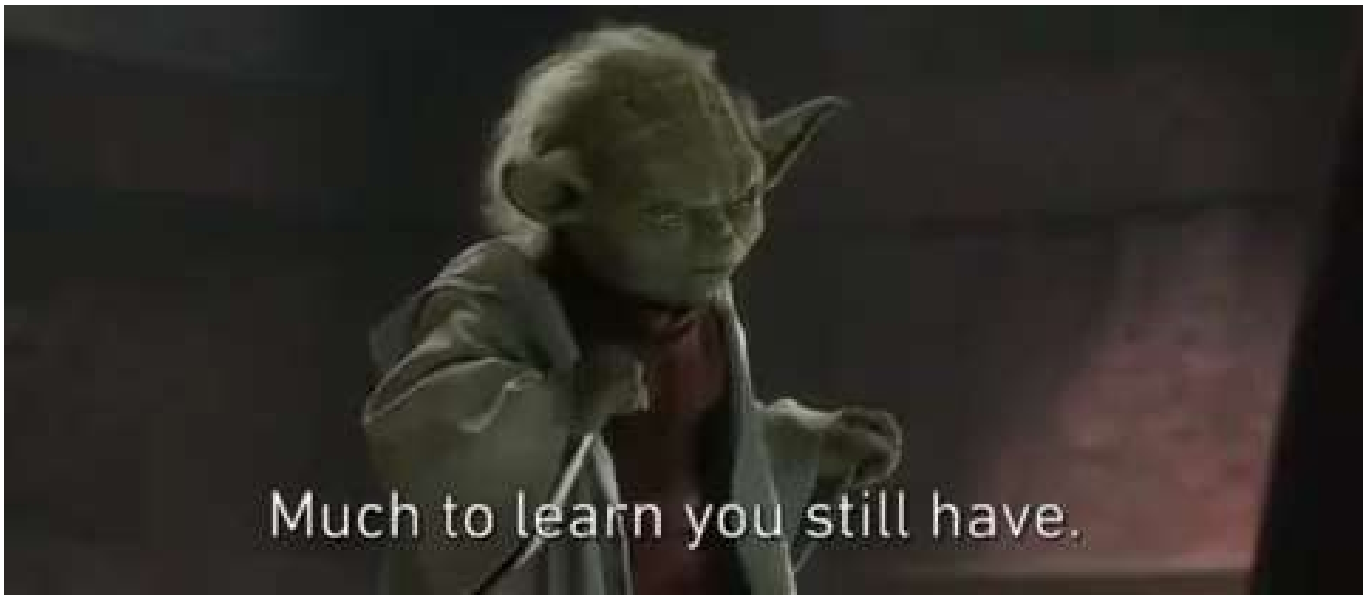
RegEx básico em Python

Naiara Cerqueira Follow
Feb 9 · 5 min read

RegEx é uma das coisas mais legais e úteis que aprendi ano passado! E toda vez que uso aprendo alguma coisa nova! Aqui tem um notebook com os exemplos que utilizei.



1. O que é RegEx?

Uma expressão regular (Regex) é uma **notação** que serve para **caracterizar um conjunto de strings**. Tem aplicações em diversas linguagens e programas (no Sheets e no Data Studio, por exemplo). Com ela podemos:

- Encontrar todos os links em um documento;
- Buscar emails, telefones;
- Remover/substituir caracteres indesejados;
- Normalizar/padronizar um texto;

- Tratar um *corpus* para realizar processamento de linguagem natural.

2. Biblioteca

No python, o pacote que permite o uso de expressões regulares é o `re`.

```
import re
```

Nesse pacote existem alguns métodos:

- **re.match**: determina se a expressão regular combina com o início da string. Retorna a posição da string buscada.
- **re.search**: varre toda a string, procurando o primeiro local onde esta expressão regular tem correspondência. Retorna a posição da string buscada.
- **re.findall**: encontra todas as substrings que têm correspondência com a expressão regular, e as retorna como uma lista. Retorna uma lista com as strings encontradas.
- **re.finditer**: encontra todas as substrings que têm correspondência com a expressão regular, e as retorna como um iterador.
- **re.sub**: procura onde esta expressão regular tem correspondência e faz as substituições desejadas. Retorna a string com a substituição realizada.

Vamos fazer alguns testes:

```
string = "frase que será analisada pelo regex porque regex é legal e
útil"

re.match('frase', string)
```

Isso me retorna *<_sre.SRE_Match object; span=(0, 5), match='frase'>*, o que significa que a palavra buscada "frase" está localizada nas posições de 0 a 5.

```
re.match('regex', string)
```

Não me retorna nada, já que a palavra ‘regex’ não está na primeira posição da frase.

```
re.search('regex', string)
```

Testando com o search, ele me retorna a posição da primeira palavra ‘regex’:
<_sre.SRE_Match object; span=(30, 35), match=’regex’>

```
re.findall('que', string)
```

Isso me retorna uma lista *[‘que’, ‘que’]*, sendo o segundo relativo a palavra ‘porque’.

```
re.finditer('regex', string)
```

Já o finditer me retorna um objeto, que pode ser iterado:

```
for i in re.finditer('regex', string):
     print(i)
```

E isso sim, me retorna as posições de ambas as palavras ‘regex’: *<_sre.SRE_Match object; span=(30, 35), match=’regex’>*
<_sre.SRE_Match object; span=(43, 48), match=’regex’>

```
re.sub('analisada', 'avaliada', string)
```

Por fim, isso substitui a palavra ‘analisada’ por ‘avaliada’, tendo como resultado: ‘frase que será avaliada pelo regex porque regex é legal e útil’

Ok, sabemos o básico dos métodos. Vamos um pouco mais longe:

3. Padrões

Existem diversos padrões para facilitar nossas buscas:

Padrão	Significado
^	Nega a expressão OU marca o início de uma string
$	Marca o final de uma string
+	Prolonga o caractere anterior
*	Prolonga o caractere anterior, mas ele também pode não existir
?	Diz que há dúvida se o caractere anterior a ele existe
.	Substitui qualquer caractere (apenas um)
[A-Z]	Procura qualquer caractere maiúsculo de **A** a **Z**
[a-z]	Procura qualquer caractere minúsculo de **a** a **z**
[0-9]	Procura qualquer dígito de **0** a **9**
[125]	Procura os dígitos **1**, **2** ou **5**
[^0-9]	O ^ nega a expressão a seguir, logo, procura tudo que não for número.
[abc]	Procura os caracteres **a**, **b** ou **c**
[^A-Z]	Procura tudo que não for letra maiúscula
[a-zA-Z]	Procura tudo que for letra
\s	Procura espaços
\w	Procura caracteres, exceto os especiais
\d	Também procura qualquer dígito de **0** a **9**

Resumo

Vamos trabalhar com uma lista de strings para ficar mais claro.

```
lista = [ 'www.google.com', 'https://www.google.com', 'google.com.br'
]

for string in lista:
     print(re.search("^www", string))
```

Foi realizada uma busca na lista para encontrar qual das strings se inicia (^) com ‘www’. Como resultado temos que apenas a primeira string da lista se inicia com ‘www’.

<_sre.SRE_Match object; span=(0, 3), match='www'>
None
None

```
for string in lista:
     print(re.search("com$", string))
```

Agora, buscamos quais strings terminam com ‘com’ e foram encontradas duas correspondências:

<_sre.SRE_Match object; span=(11, 14), match='com'>
<_sre.SRE_Match object; span=(19, 22), match='com'>
None

```
lista= [ 'naiara', 'naiaraaaaaa', 'naaaaaaiara', 'naiar', 'naiars' ]

for string in lista:
     print(re.search("naiar.", string))
```

Aqui, encontramos três correspondências: as duas primeiras e a última string. Isso ocorre pois ele busca ‘naiar’ mais **um** caractere qualquer. Porém, observe que no segundo caso ele encontrou correspondência apenas com o *primeiro caractere* após o ‘r’.

<_sre.SRE_Match object; span=(0, 6), match='naiara'>
<_sre.SRE_Match object; span=(0, 6), match='naiara'>
None
None
<_sre.SRE_Match object; span=(0, 6), match='naiars'>

```
for string in lista:
     print(re.search("naiara?", string))
```

Nesse caso, não encontramos correspondência apenas no ‘naaaaaaiara’. A interrogação diz que o último ‘a’ pode ou não existir.

<_sre.SRE_Match object; span=(0, 6), match='naiara'>
<_sre.SRE_Match object; span=(0, 6), match='naiara'>
None

<_sre.SRE_Match object; span=(0, 5), match='naiar'>
<_sre.SRE_Match object; span=(0, 5), match='naiar'>

```
for string in lista:
     print(re.search("naiara+", string))
```

Nesse último caso, encontramos correspondência com os dois primeiros

<_sre.SRE_Match object; span=(0, 6), match='naiara'>
<_sre.SRE_Match object; span=(0, 11), match='naiaraaaaaa'>

Complicando um pouquinho:

```
string = "essa é uma frase com espaços demais no final.           "

re.sub("\s+$", "", string)
```

Nesse último caso, pedi uma busca por todos (+) os espaços (\s), no final da frase ($). Encontrando eles, substituir por nada, ou seja, remover. O resultado é ‘essa é uma frase com espaços demais no final.’

Para estudarmos melhor alguns padrões, vou pegar uma frase do wikipedia sobre PLN:

```
string = "A história do PLN começou na década de 1950, quando Alan
Turing publicou o artigo Computing Machinery and Intelligence, que
propunha o que agora é chamado de teste de Turing como critério de
inteligência. Em 1954, a experiência de Georgetown envolveu a
tradução automática de mais de sessenta frases russas para o inglês.
Os autores afirmaram que dentro de três ou cinco anos a tradução
automática seria um problema resolvido.[2] No entanto, os avanços
reais foram muito mais lentos do que o previsto e, após o relatório
ALPAC em 1966, que constatou que a pesquisa de dez anos não conseguiu
satisfazer as expectativas, o financiamento para este estudo em
tradução automática foi reduzido drasticamente. Poucas pesquisas em
tradução automática foram conduzidas até o final dos anos 80, quando
os primeiros sistemas estatísticos de tradução foram desenvolvidos."
```

Okay, vamos buscar caracteres:

```
re.findall(‘[A-Z]’,string)
```

Isso vai listar todas as letras maiúsculas: [‘A’, ‘P’, ‘L’, ’N’, ‘A’, ‘T’, ‘C’, ‘M’, ‘I’, ‘T’, ‘E’, ‘G’, ‘O’, ’N’, ‘A’, ‘L’, ‘P’, ‘A’, ‘C’, ‘P’]

```
re.findall('[0-9]+',string)
```

Isso lista todos os caracteres numéricos: [‘1950’, ‘1954’, ‘2’, ‘1966’, ‘80’]. Sem o “+” ele me retornaria cada número separadamente.

```
string = 'ABCDEFabcdef_123450&%@#!'
re.findall('\w',string)
```

Isso encontra caracteres não especiais: ['A', 'B', 'C', 'D', 'E', 'F', 'a', 'b', 'c', 'd', 'e', 'f', ‘_‘, '1', '2', '3', '4', '5', '0']

```
string = '[codigo] titulo'

re.findall("\[.*\]", string)
```

Vamos supor que nessa string a gente quer encontrar o que está dentro dos colchetes ([código]). Nesse caso, como o colchete é usado pelo próprio método, temos que utilizar um caractere de *scape* (\) antes de cada colchete para contar para o algoritmo que esse caractere é um caractere comum. Assim, eu digo para o algoritmo que quero qualquer caractere (.) quantas vezes for (*), ou seja, tudo, que estiver dentro dos colchetes. O resultado é: [‘[codigo]’], já que o findall me retorna uma lista como resultado.

```
re.findall("\[(.*)\]", string)
```

Nesse caso eu queria saber o que está *dentro* desse colchete, e, para isso, ultilizei os parêntesis, tendo como resultado: [‘codigo’]

4. Bônus

Aqui vai um bônus que eu gostei muito de descobrir: eu sempre recebo dados com datas em formatos que alguns programas não gostam de receber. Assim, aprendi a alterar esse formato:

```
data = "28/11/2020"
re.sub(r"(..)/(..)/(....)",r"\3-\2-\1", data)
```

O que isso faz? Busca dois caracteres (..), dois caracteres (..), quatro caracteres (….), todos separados por uma barra, mas foca apenas nos caracteres (por isso o uso dos parêntesis). Ele inverte a ordem desses caracteres encontrados (3–2–1) e coloca “-” entre os valores. O resultado final é: ‘2020–11–28’.

Esse resultado também pode ser alcançado com:

```
re.sub(r"(\d{1,2})/(\d{1,2})/(\d{2,4})",r"\3-\2-\1", data)
```

Que busca um ou dois ({1,2}) dígitos (\d), seguidos por uma barra e etc., focando apenas nos dígitos, e inverte a ordem, exatamente como o exemplo anterior. A vantagem desse formato é que o formato inicial da data pode ser 5/1/21.

Espero que tenham aprendido um pouquinho sobre RegEx! Fica a dica que RegEx é prática: quanto mais a gente usa, mais a gente vê usos para ele e mais fácil fica de encontrar os padrões!

Para facilitar seus estudos, esse site aqui é ótimo para fazer os testes de correspondências.

Thanks to Peo Pinheiro.

Python Regex