ACH5531

Introdução à Computação

Expressões Regulares

Prof. Dr. Grzegorz Kowal

grzegorz.kowal@usp.br

https://sites.google.com/usp.br/ach5531

1° sem 2019 – sexta-feira, 14h00-15h45 – CB, Bloco 3, 2° andar, Lab. 6

Expressões Regulares

Expressões regulares (*regex*, *regexp*) é uma técnica para procurar, de forma bem específica, um texto abrangente. Em prática, é uma sequência de caracteres que define um padrão de pesquisa, principalmente para encontrar padrões em um texto, ou detectar uma sequência de caracteres, ou seja, como operações de "localizar e substituir".

A expressão regular relaciona todas as ocorrências (**matches**) de um padrão (**pattern**) em um trecho de texto (**subject**).

O exemplo mais simples possível de uma expressão regular seria a busca pelo "termo exato". Imagine que queremos procurar a palavra "*exemplo*" dentro da frase "*O seguinte exemplo mostra como usar expressões.*".

O assunto (**subject**) é "*O seguinte exemplo mostra como usar expressões.*" e a **expressão regular** é "*exemplo*", logo o resultado seria:

Expressão regular: **/exemplo/**

Texto: O seguinte **exemplo** mostra como usar expressões.

Expressões regulares

Outro exemplo, é a expressão regular ca.a. O sinal "." (ponto) é um metacaractere que representa qualquer coisa, ou melhor, qualquer caractere. Veja os resultados (matches) da expressão abaixo:

Expressão regular: /**ca.a**/

Texto: casa, castanha, carpinteiro, cana de açúcar, cama, casar, cavalo

A expressão /e.tendido/ pode representar: estendido, extendido, entendido, e1tendido, etc.

Para especificar quais exatamente caracteres podem ocorrer usamos **colchetes []**:

/e[ns]tendido/ → estendido e entendido

/[Tt]eclado/ → Teclado e teclado

/digito[0-9]/ → digito0, digito1, digito2, digito23, etc.

/bal[a-z]/ → bala, balb, balc, bale, balz, etc.

Expressões regulares

Quando queremos excluir alguns caracteres da lista podemos usar lista negada dentro dos colchetes **[^]**. Por exemplo, **[^a-c]** é equivalente **[d-z]** ou **[^0-9]** significa qualquer caractere menos os números.

O quantificador opcional "**?**" indica que pode ter ou não a ocorrência da entidade anterior, pois ele a repete 0 ou 1 vez.

A expressão /fala[r!]?/ corresponde palavras: fala, fala! e falar.

O asterisco "*****" repete em qualquer quantidade.

A expressão /6*0/ corresponde 0, 60, 660, 6660, 6666666666660, etc.

O mais "**+**" tem funcionamento idêntico ao do asterisco, a única diferença é que o mais não é opcional, então a entidade anterior deve ocorrer pelo menos uma vez, e pode ter várias.

O quantificador chaves "**{}**" funciona como uma repetição mais controlada, onde **{n,m}** significa de **n** até **m** vezes.

A expressão /n{1,4}/ corresponde "n", "nn", "nnn" e "nnnn" (exatamente isso; nem mais, nem menos).

Expressões regulares

O metacaractere circunflexo “**^**” marca o começo de uma linha, serve para procurar no começo da linha.

A expressão /^[0-9]/ corresponde qualquer linha que começa com número.

Analogicamente, o metacaractere cifrão “**$**” marca o fim de uma linha e só é válido no final de uma expressão regular.

A expressão /[0-9]$/ corresponde qualquer linha que termina com número.

O metacaractere “ou” **|** serve para os casos em que temos mais de uma alternativa possível.

A expressão regular boa-tarde|boa-noite casa com a duas opções.

Para agrupar usamos parênteses **()**. O exemplo da linha anterior pode ser escrito como boa-(tarde|noite), que corresponde boa-tarde e boa-noite.

Expressões regulares

Regular Expression Basics	
.	Any character except newline
a	The character a
ab	The string ab
a|b	a or b
a*	0 or more a's
\	Escapes a special character

Regular Expression Quantifiers	
*	0 or more
+	1 or more
?	0 or 1
{2}	Exactly 2
{2, 5}	Between 2 and 5
{2,}	2 or more
(,5}	Up to 5

Default is greedy. Append ? for reluctant.

Regular Expression Groups	
(...)	Capturing group
(?P<Y>...)	Capturing group named Y
(?:...)	Non-capturing group
\Y	Match the Y'th captured group
(?P=Y)	Match the named group Y
(?#...)	Comment

Regular Expression Character Classes	
[ab-d]	One character of: a, b, c, d
[^ab-d]	One character except: a, b, c, d
[\b]	Backspace character
\d	One digit
\D	One non-digit
\s	One whitespace
\S	One non-whitespace
\w	One word character
\W	One non-word character

Regular Expression Assertions	
^	Start of string
\A	Start of string, ignores m flag
$	End of string
\Z	End of string, ignores m flag
\b	Word boundary
\B	Non-word boundary
(?=...)	Positive lookahead
(?!...)	Negative lookahead
(?<=...)	Positive lookbehind
(?<!...)	Negative lookbehind
(?()|)	Conditional

Regular Expression Flags	
i	Ignore case
m	^ and $ match start and end of line
s	. matches newline as well
x	Allow spaces and comments
L	Locale character classes
u	Unicode character classes
(?iLmsux)	Set flags within regex

Regular Expression Special Characters	
\n	Newline
\r	Carriage return
\t	Tab
\YYY	Octal character YYY
\xYY	Hexadecimal character YY

Regular Expression Replacement	
\g<0>	Insert entire match
\g<Y>	Insert match Y (name or number)
\Y	Insert group numbered Y

Como usar Expressões Regulares?

Para usar expressões regulares no Python, precisamos primeiro importar a biblioteca **re**.

```
import re
```

Os usos mais comuns são:

- buscar um padrão de texto (funções search() e match())
- achar todas as ocorrências de um padrão (função findall())
- quebrar um texto em subtextos (função split())
- substituir parte de um texto (função sub())

Função re.match()

Esse método encontra equivalência se ela ocorrer no início do texto.

```
re.match('expressão', 'texto que será procurado')
```

Por exemplo, chamar match() no texto "Meu nome é Pedro" e buscar por um padrão "Meu" vai retornar uma equivalência. Contudo, se buscarmos por "Pedro", por exemplo, não encontrará um padrão equivalente.

Exemplo 1:

```
import re
texto = "Meu nome é Pedro"
resultado = re.match(r'Meu', texto)
print(resultado)
# <_sre.SRE_Match object; span=(0, 3), match='Meu'>
print(resultado.group(0))
# Meu
```

Observação: O prefixo *r* antes da expressão regular evita o pré-processamento da expressão regular pela linguagem. Colocamos o modificador *r* (do inglês "raw", crú) imediatamente antes das aspas, assim: (r'\bdia\b'). Nem sempre é preciso a presença do prefixo.

Função re.search()

É similar a match() mas não nos restringe a encontrar equivalência apenas no começo do texto que será procurado. Diferente de métodos anteriores, aqui a busca pelo padrão “Pedro” irá retornar resultado positivo.

```
re.search('expressão', 'texto que será procurado')
```

Exemplo 2:

```
import re
texto = “Meu nome é Pedro. Pedro é Piotr em Polonês.”
resultado = re.search(r'Pedro', texto)
print(resultado)
# <_sre.SRE_Match object; span=(11, 16), match='Pedro'>
print(resultado.start(), resultado.end())
# 11 16
```

Aqui podemos ver que o método **search**() consegue encontrar um padrão em qualquer posição do texto mas que somente retorna a primeira ocorrência do padrão de busca. Usando métodos **start()** e **end()** podemos encontrar a posição do começo e fim da ocorrência.

Função re.findall()

É útil obter uma lista de todos os padrões encontrados. Não há restrições em buscar do começo ou do fim. Se usarmos o método **findall()** para buscar por "Pedro" num dado texto, irá retornar todas ocorrências de expressão procurada. Quando efetuar buscas num texto, é recomendado usar sempre **re.findall()**, funciona como ambas **re.search()** e **re.match()**.

```
re.findall('expressão', 'texto que será procurado')
```

Exemplo 3:

```
import re
texto = "Meu nome é Pedro. Pedro é Piotr em Polonês."
resultado = re.findall(r'Pedro', texto)
print(resultado)
# ['Pedro', 'Pedro']
for m in re.finditer(r'Pedro', texto):
  print(m.start(), m.end())
```

Assim podemos encontrar o número de ocorrências de uma expressão regular, calculando o número de elementos na lista usando função **len()**.

Usando **re.finditer()** em vez de **re.findall()** podemos usar **start()** e **end()** para encontrar a posição de cada ocorrência.

Função re.split()

Este método ajuda a dividir a string pela ocorrências do padrão dado, parecido com a função **split()** já conhecida, mas mais podero.

```
re.split('expressão', 'texto que será procurado')
```

Exemplo 4:

```
import re
texto = "Meu nome é Pedro."
resultado = re.split(r'e', texto)
print(resultado)
# ['M', 'u nom', ' é P', 'dro.']
```

Acima, dividimos nosso texto "Meu nome é Pedro" por "e". O método split() tem um argumento chamado "maxsplit". Seu valor padrão é zero. Nesse caso, faz o máximo de divisões possíveis.

Função re.sub()

Às vezes é útil buscar um padrão de texto e substituí-lo por uma novo sub-texto. Se o padrão não for encontrado, o texto original é retornado sem mudanças.

```
re.sub('expressão', 'texto a ser inserido', 'texto')
```

Exemplo 5:

```
import re
texto = "Meu nome é Pedro."
resultado = re.sub(r'Pedro', r'Thiago', texto)
print(resultado)
# Meu nome é Thiago.
```

Função re.compile()

Se as expressões regulares são constantes e vão ser usadas muitas vezes, vale a pena construir um objeto expressão regular invocando a função **re.compile()**. O objeto devolvido tem métodos que correspondem às funções de mesmo nome.

```
re.compile('expressão')
```

Exemplo 6:

```
import re
texto = "Meu nome é Pedro."
er = re.compile(r'Pedro')
resultado = er.sub(r'Thiago', texto)
print(resultado)
# Meu nome é Thiago.
texto = "Pedro é meu amigo."
resultado = er.sub(r'Thiago', texto)
print(resultado)
# Thiago é meu amigo.
```

Exemplo 7

Solicite o usuário fornecer o endereço e-mail dele, verifique se o endereço fornecido está no formato correto, ou seja, o usuário começa com uma letra e o endereço possui símbolo '@', e imprime separadamente o usuário e domínio processando o e-mail fornecido.

```
import re

while True:
  email = input("Forneça seu endereço e-mail: ")
  if re.search(r'@', email) == None:
    print("Formato de endereço incorreto!")
  else:
    usuario = re.search(r'^[A-Za-z]+', email)
    dominio = re.search(r'@[A-Za-z.]+', email)
    print("Usúario: ", usuario.group())
    print("Domínio: ", dominio.group())
    break
```

Exemplo 8

Cria uma variável de texto com seu nome completo e dos dois colegas do seu lado separando os nomes com vírgula. Por exemplo:

```
nomes = “Grzegorz Kowal, Lucas Aguiar de Moura, Pedro Paulo Diniz Lucinda”
```

Use re.split() para criar uma lista de nomes e imprime cada nome em formato “Sobrenome, Iniciais de primeiros nomes” usando expressões regulares. É suficiente usar funções **re.search()** e **re.findall()**.

```
import re

nomes = "Grzegorz Kowal, Lucas Aguiar de Moura, Pedro Paulo Diniz Lucinda"

lista_nomes = re.split(r', ', nomes)

for nome in lista_nomes:
  lnome = re.search(r'(de )*[A-Za-z]+$', nome)
  inome = re.findall(r'[A-Z]+', nome)
  rnome = lnome[0] + ', '
  for i in inome[0:-1]:
    rnome = rnome + i + '. '
  print(nome, ‘:’, rnome)
```

Exemplo 9

Usando a sequência DNA do exercício 13 no arquivo ‘dna.dat’, escreva um programa que acha todas as ocorrências de sequencias ‘CAA’, ‘CCA’, ‘CGA’, ‘CTA’ usando expressões regulares. Use **re.finditer()** para achar a posição de cada ocorrência.

```
import re

f = open("dna.dat")
dna = f.read()
f.close()

for m in re.finditer(r'C.A', dna):
  print("%s: %d - %d" % (m.group(), m.start(), m.end()))
```