MC-102 — Aula 20
Expressões Regulares

Instituto de Computação – Unicamp

16 de Maio de 2018

Roteiro

Expressões regulares

- Expressões regulares são formas concisas de descrever um conjunto de strings que satisfazem um determinado padrão.
- Por exemplo, podemos criar uma expressão regular para descrever todas as strings na forma `dd/dd/dddd` onde `d` é um dígito qualquer (é um padrão que representa datas).
- Dada uma expressão regular podemos resolver por exemplo este problema: existe uma sequência de caracteres numa string de entrada que pode ser interpretada como um número de telefone? E qual é ele?
- Note que números de telefones podem vir em vários "formatos"
  - 19-91234-5678
  - (019) 91234 5678
  - (19)912345678
  - 91234-5678
  - etc.

Expressões regulares

- Expressões regulares são uma mini-linguagem que permite especificar as regras de construção de um conjunto de strings.
- Essa mini-linguagem de especificação é muito parecida entre as diferentes linguagens de programação que contém o conceito de expressões regulares (também chamado de RE ou REGEX).
- Assim, aprender a escrever expressões regulares em Python será útil para descrever REs em outras linguagens de programação.

Uma expressão regular

- Um exemplo de expressão regular é:

  ```
  '\d+'
  ```

- Essa RE representa uma sequência de 1 ou mais dígitos.
- Vamos ver algumas regras de como escrever essas REs mais tarde na aula - no momento vamos ver como usar uma RE.
- É conveniente escrever a string da RE com um `r` na frente para especificar uma **raw string** (onde coisas como `'\n'` são tratados como 2 caracteres e não uma quebra de linha).
- Assim a RE é:

  ```
  r'\d+'
  ```

Usando REs

- Expressões regulares em Python estão na biblioteca **re**, que precisa ser importada.
- Segue o link para a documentação da biblioteca **re**
  `https://docs.python.org/3/library/re.html`

re.search

- A principal função da biblioteca é a **re.search**: dada uma RE e uma string, a função busca a primeira ocorrência de uma substring especificada pela RE.

```
>>> import re
>>> a=re.search(r'\d+','Ouviram do Ipir723anga margens 45')
>>> a
<_sre.SRE_Match object; span=(15, 18), match='723'>
```

- O resultado de **re.search** é do tipo **match** que permite extrair informação sobre qual é a substring que foi encontrada (o *match*) e onde na string ele foi encontrado (o *span*).

```
>>> b=re.search(r'\d+','Ouviram do Ipiranga margens')
>>> b
>>>
```

- Neste último exemplo nenhum *match* é encontrado.

re.search

- Se nenhum *match* foi encontrado, o **re.search** retorna o valor **None**.
- Assim, depois de usar o método **re.search** deve-se verificar se algo foi encontrado:

```
b=re.search(r'\d+','Ouviram do Ipiranga margens')
if b:
    ....
```

- O valor **None** se comporta como um **False** em expressões booleanas.

Objetos do tipo match

- O método **span** de um objeto match retorna a posição inicial e final+1 de onde a substring foi encontrada.
- O método **group** retorna a substring encontrada.

```
>>> a=re.search(r'\d+','Ouviram␣do␣Ipir723anga␣margens␣45')
>>> a.span()
(15, 18)
>>> a.group()
'723'
```

- Note que o método **re.search** acha apenas a *primeira* instância da RE na string (o número 45 também satisfaz a RE).

Outras funções da biblioteca **re**

- A função **re.match** é similar a **re.search**, mas a RE deve estar no **começo** da string.

```
>>> a =  re.match(r'\d+','Ouviram␣do␣Ipir723anga␣margens␣45')
>>> a
>>> a =  re.match(r'\d+','1234Ouviram␣do␣Ipir723anga␣margens␣45')
>>> a
<_sre.SRE_Match object; span=(0, 4), match='1234'>
>>>
```

- A função **re.sub** substitui na string todas as REs por uma outra string,

```
>>> re.sub(r'\d+','Z','Ouviram␣do␣Ipir723anga␣margens␣45')
'Ouviram␣do␣IpirZanga␣margens␣Z'
>>> re.sub(r'\d+','Z','Ouviram␣do␣Ipiranga␣margens')
'Ouviram␣do␣Ipiranga␣margens'
```

Outras funções da biblioteca **re**

- A função **re.findall** retorna uma lista de todas as ocorrências da RE:

```
>>> re.findall(r'\d+','Ouviram␣do␣Ipir723anga␣margens␣45')
['723', '45']
>>> re.findall(r'\d+','Ouviram␣do␣Ipiranga␣margens')
[]
```

- A função **re.split** funciona como a função **split** para strings, mas permite usar uma RE como separador:

```
>>> re.split(r'\d+','ab␣1␣cd34efg␣h␣56789␣z')
['ab␣', '␣cd', 'efg␣h␣', '␣z']
```

Compilando REs

- Procurar uma RE numa string pode ser um processamento custoso e demorado. É possível "compilar" uma RE de forma que a procura seja executada mais rápida.

```
>>> zz=re.compile(r'\d+')
>>> zz.search('Ouviram do Ipir723anga margens 45')
<_sre.SRE_Match object; span=(15, 18), match='723'>
```

- As funções vistas anteriormente funcionam também como métodos de REs compilados, e normalmente permitem mais alternativas.
- O método **search** de um RE compilado permite dizer a partir de que ponto da string começar a procurar a RE.

```
>>> zz.search('Ouviram do Ipir723anga margens 45', 20)
<_sre.SRE_Match object; span=(31, 33), match='45'>
```

- O método **search** começou a procurar a RE a partir da posição 20.

Regras básicas para Escrita de uma RE

- As letras e números numa RE representam a si próprios.
- Assim a RE `r'wb45p'` representa apenas a substring `'wb45p'`.
- Os caracteres especiais (chamados de meta-caracteres) são:

  `. ^ $ * + ? { } [ ] \ | ( )`

Repetições

- O meta-caractere `.` representa qualquer caractere.
- Por exemplo, a RE `r'.ao'` representa todas as strings de 3 caracteres cujos 2 últimos são ao.

```
>>> r = re.compile(r'.ao')
>>> r.search("O cao")
<_sre.SRE_Match object; span=(2, 5), match='cao'>
>>> r.search("O caocao")
<_sre.SRE_Match object; span=(2, 5), match='cao'>
>>> r.search("O pao")
<_sre.SRE_Match object; span=(2, 5), match='pao'>
>>> r.search("O 3ao")
<_sre.SRE_Match object; span=(2, 5), match='3ao'>
>>> r.search("ao")
>>>
```

- Apenas no último exemplo não há um *match*.

Classe de Caracteres

- A notação [ ] representa uma classe de caracteres, de forma que deve-se ter um *match* com algum dos caracteres da classe.
- Por exemplo, `r'p[aã]o'` significa todas as strings de 3 caracteres que começam com p seguido de um a ou ã e terminam com o.

```
>>> r = re.compile(r"p[aã]o")
>>> r.search("O pão")
<_sre.SRE_Match object; span=(2, 5), match='pão'>
>>> r.search("O puo")  #Não acha nada
>>> r.search("O ampao")
<_sre.SRE_Match object; span=(4, 7), match='pao'>
```

- O caractere - dentro do [] representa um intervalo. Assim [1-7] representa um dos dígitos de 1 a 7.
- De forma parecida [a-z] e [0-9] representam as letras minúsculas e os dígitos, respectivamente.

Classe de Caracteres

- O caractere ^ no início de [] representa a negação da classe. Assim r'ab[^h-z]' representa qualquer string começando com ab e terminando com qualquer caractere exceto os de h até z.

```
>>> r = re.compile(r'ab[^h-z]')
>>> r.search("Oi abg")
<_sre.SRE_Match object; span=(3, 6), match='abg'>
>>> r.search("Oi abh")  #Não acha nada
>>> r.search("Oi ab6")
<_sre.SRE_Match object; span=(3, 6), match='ab6'>
```

Classe de Caracteres

Qualquer caractere de palavra poderia ser descrito como a classe `r'[a-zA-Z0-9]'`, mas Python fornece algumas classes pré-definidas que são úteis.

- `\d` - Qualquer número decimal, i.e, [0-9].
- `\D` - É o complemento de `\d`, equivalente a `[^0-9]`, i.e, faz *match* com um caractere não dígito.
- `\s` - Faz *match* com caracteres *whitespace* , i.e, equivalente a `[\t\n\r\f\v]`.
- `\S` - O complemento de `\s`.
- `\w` - Faz o *match* com um caractere alfanumérico, i.e, equivalente a `[a-zA-Z0-9]`.
- `\W` - O complemento de `\w`.

Opcional

- O meta-caractere ? significa que o caractere que o precede pode ou não aparecer. Nos dois exemplos abaixo há um match de `r'ab?c'` tanto com abc quanto com ac.

```
>>> r = re.compile(r'ab?c')
>>> r.search("abc")
<_sre.SRE_Match object; span=(0, 3), match='abc'>
>>> r.searhc("ac")
<_sre.SRE_Match object; span=(0, 2), match='ac'>
```

- Pode-se criar um grupo incluindo-se uma string entre parênteses. Por exemplo, se quisermos detectar ocorrências de Fev 2016, Fevereiro 2016 ou Fevereiro de 2016, etc, podemos usar a RE `r'Fev(ereiro)? (de)? ?2016'`

```
>>> r = re.compile(r'Fev(ereiro)? (de)? ?2016')
>>> r.search("Fevereiro 2016")
<_sre.SRE_Match object; span=(0, 14), match='Fevereiro 2016'>
>>> r.search("Fev 2016")
<_sre.SRE_Match object; span=(0, 8), match='Fev 2016'>
>>> r.search("Fevereiro de 2016")
<_sre.SRE_Match object; span=(0, 17), match='Fevereiro de 2016'>
```

Repetições

- O meta-caractere + representa uma ou mais repetições do caractere ou grupo de caracteres imediatamente anterior.
- O meta-caractere * representa 0 ou mais repetições do caractere ou grupo de caracteres imediatamente anterior.

```
>>> r = re.compile(r'abc(de)+')
>>> r2 = re.compile(r'abc(de)*')
>>> r.search("abc")  #Não acha pois tem que ter pelo menos um 'de' no
>>> r2.search("abc") #Acha pois tem que ter 0 ou mais 'de's no final
<_sre.SRE_Match object; span=(0, 3), match='abc'>
>>> r.search("abcdede")
<_sre.SRE_Match object; span=(0, 7), match='abcdede'>
>>> r2.search("abcdede")
<_sre.SRE_Match object; span=(0, 7), match='abcdede'>
```

Outros meta caracteres

- | representa um OU de diferentes REs.
- \b indica o separador de palavras (pontuação, branco, fim da string).
- `r'\bcasa\b'` é a forma correta de procurar a palavra "casa" numa string.

```
>>> re.search(r'\bcasa\b','a casa')
<_sre.SRE_Match object; span=(2, 6), match='casa'>
>>> re.search(r'\bcasa\b','casa')
<_sre.SRE_Match object; span=(0, 4), match='casa'>
>>> re.search(r'\bcasa\b','o casamento')
>>>
```

Exemplo: buscando um email

Uma RE para buscar emails:

- O userid é uma sequência de caracteres alfanuméricos `\w+` separado por `@`.
- O host é uma sequência de caracteres alfanuméricos `\w+`.

```
>>> re.search(r'\w+@\w+','bla bla bla abc@gmail.com bla')
<_sre.SRE_Match object; span=(12, 21), match='abc@gmail'>
```

- O host não foi casado corretamente. O ponto não é um caractere alfanumérico.
- Vamos tentar `r'\w+@\w+\.\w+'` (note que `\.` serve para considerar o caractere `\.` e não o meta-caractere).

```
>>> re.search(r'\w+@\w+\.\w+', 'bla bla bla abc@gmail.com bla')
<_sre.SRE_Match object; span=(12, 25), match='abc@gmail.com'>
>>> re.search(r'\w+@\w+\.\w+', 'bla bla bla abc@gmail.com.br bla'
<_sre.SRE_Match object; span=(12, 25), match='abc@gmail.com'>
>>>
```

- Note que no último exemplo não foi casado corretamente o `.br`.

Exemplo: buscando um email

- Podemos tentar `r'\w+@\w+\.\w+(\.\w+)?'`. Criamos um grupo no final `(\.\w+)?` que é um ponto seguido de caracteres alfanuméricos, porém opcional.

```
>>> re.search(r'\w+@\w+\.\w+(\.\w+)?','bla bla bla abc@gmail.com
<_sre.SRE_Match object; span=(12, 25), match='abc@gmail.com'>
>>> re.search(r'\w+@\w+\.\w+(\.\w+)?','bla bla bla abc@gmail.com.
<_sre.SRE_Match object; span=(12, 28), match='abc@gmail.com.br'>
>>>
```

- Agora a RE casa com os dois tipos de endereços de email.
- Há muito mais coisas sobre como escrever REs - veja por exemplo `https://docs.python.org/3/howto/RE.html#RE-howto`

Exercício 1

Escreva uma RE para encontrar numeros de telefone do tipo

- (019)91234 5678
- 19 91234 5678
- 19-91234-5678
- (19) 91234-5678

Exercício 2

Faca uma função que recebe um string e retorna o string com os números de telefones transformados para um unico formato (19) 91234 5678