MC-102 — Aula 22
Expressões Regulares

Instituto de Computação – Unicamp

7 de Outubro de 2015

Roteiro

Expressões regulares

- Expressões regulares são padrões que descrevem um conjunto grande (ou mesmo infinito) de substrings que voce quer encontrar dentro de um srting maior
- Por exemplo: existe uma sequencia de caracteres que pode ser interpretada como um número de telefone dentro do string? E qual é ele?
- note que numeros de telefones podem vir em varios "formatos"
  - 19-91234-5678
  - (019) 91234 5678
  - (19)912345678
  - 91234-5678
  - etc.
- e note que os digitos nao precisam ser os assima, qualquer digito pode ocupar quase qualquer posição

Expressões regulares

- Expressões regulares são uma mini-linguagem que permite especificar as regras de construção do conjunto de substrings.
- essa especificação é em si um string
- essa mini-linguagem de especificação é muito parecida através de diferentes linguagens de programação que contem o conceito de expressões regulares (tambem chamado de RE ou REGEX).
- assim aprender a escrever expressões regulares em Python será util para REGEXs em outras linguagens de programação.

Uma expressão regular

- Uma expressão regular é:

  `'\d+'`
- essa regex representa uma sequencia de 1 ou mais digitos (de 0 a 9)
- vamos ver alguns aspectos de como escrever essas regex mais tarde na aula - no momento vamos ver como usar essa regex.
- é conveiente escrever o string da regex com um `r` na frente. Assim a regex é:

  `r'\d+'`

Usando regex

- Expresssões regulares em Python estao no pacote `re`, que precisa ser importado.
- link para a documentação do `re`
  `https://docs.python.org/3/library/re.html`

re.search

- a principal função é a `re.search` que dado uma regex e um string tenta encontrar algums dos substrings especificados pela regex no string.

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

- o resultado do `re.search` é um objeto do tipo `match` que permite extrair informação sobre qual é o substring que foi encontrado (o `match`) e onde no string ele foi encontrado (o `span`)

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

re.search

- se o substring não foi encontrado o `re.search` o valor `None`
- assim depois de usar um `re.find` deve-se verificar se algo foi encontrado

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

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

Objetos do tipo match

- o metodo `span` de um objeto match retorna a posição inicial e final+1 de onde substring foi encontrado
- o metodo `group` retorna o substring encontrado

```
>>> a.span()
(15, 18)
>>> a.group()
'723'
```

- note que o find acha apenas a **primeira** instancia do regex no string (o 45 tambem satisfaz o regex)

Outras funções do re

- o `re.match` é similar ao `re.search` mas a regex deve estar no **começo** do string.

```
>>> re.match(r'\d+','Ouviram do Ipir723anga margens 45')
>>>
```

- o `re.sub` substitui no string todas as regex por um outro 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 do re

- `re.findall` retorna uma lista de todos so matchs

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

- o `re.split` é como o `split` mas permite usar um regex como separador

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

Compilando regex

- procurar um regex num string pode ser um processamento custoso e demorado (dependendo do regex). Se poucos regex serao usados, é possivle "compilar" esses regex de forma que a procura seja mais facil.

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

- As funções vistas acima funcionam também como métodos de regex compilados, e normalmente permitem mais alternativas.
- o `search` como metodo de um regex compilado permite dizer a partir de que ponto do string começar a procurar o regex

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

- o `search` começou a procurar o regex apartir da posição 20.

Escrevendo regex

- As letras e números num regex representam a si proprio
- assim o regex `r'wb45p'` representa apenas o substring `'wb45p'`.
- os caracteres especiais (chamados de meta-caracteres) são:

  `. ^ $ * + ? { } [ ] \ | ( )`
- para usar um meta-caracter como um caracter normal, usa-se o antes.

o []

- o [ ] representa um conjunto de caracteres assim `r'[abc]45p'` representa os strings `a45p` `b45p` e `c45p`
- o - dentro do [] representa um intervalo. Assim `[1-7]` representa os digitos de 1 a 7
- `[a-z]` e `[0-9]` representam as letras minusculas e os digitos respectivamente
- `[\^ ...]` representa o complemento dos caracteres
- dentro do `[]` a maioria dos meta-caracteres representam o proprio caracter

Abreviações para o []

- `\d` é uma abreviação para `[0-9]` - um digito
- `\D` é uma abreviação para `[^ 0-9]` - algo que não é um digito
- `\s` representa qualquer caracter "branco" inclusive mudança de linha
- `\S` qualquer caracter não "branco"
- `\w` qualquer caracter alfanumerico (`[a-zA-Z0-9_]`)
- `.` representa **qualquer** caracter

Repetições

- `+` representa uma ou mais repetições do caracter anterior.
- assim nossa regex `r'\d+'` e qualquer numero de repetições de algum digito
- `*` representa 0 ou mais repetições do caracter anterior

Outros meta caracteres

- | representa um OU de diferentes regex
- \b indica o separador de palavras (pontuação, banco, fim do string)
- r'\bcasa\b' é a forma correta de procuara a palavra "casa" num 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

um regex para buscar emails:

- o userid é uma sequencia de caracteres alfanuméricos `\w+`
- separado por `@`
- o host é uma sequencia de caracteres alfanumericos `\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 correntamente. Ponto nao é um caracter alphanumerico
- vamos tentar `[\w.]+` uma sequencia qualquer de alfanumericos e pontos

```
>>> re.search(r'\w+@[\w.]+','bla bla bla abc@gmail.com bla
<_sre.SRE_Match object; span=(12, 25), match='abc@gmail.co
>>> re.search(r'\w+@[\w.]+','bla bla bla abc@gmail4.com.br
<_sre.SRE_Match object; span=(12, 29), match='abc@gmail4.c
```

Exemplo: buscando um email

Mas:

```
>>> re.search(r'\w+@[\w.]+','bla bla bla abc@gmail4.com..br bl
<_sre.SRE_Match object; span=(12, 30), match='abc@gmail4.com.
```

Fazer a regex certa para todos os casos é normalmente bem complicado. A Internet é sua amiga nisso - alguem já resolveu o problema de uma regex para emails!

Há muito mais coisas sobre como escrever regex - veja por exemplo `https://docs.python.org/3/howto/regex.html#regex-howto`

Em particular é possivel usar parantesis nas regex. eles criam algo chamado grouping que permite recuperar pedacos dos match, por exemplo o userid do email.

Exercício 1

Escreva uma regex 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