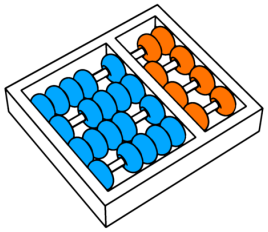

MC102 – Aula 13
Expressões Regulares

Algoritmos e Programação de Computadores

Zanoni Dias

2019

Instituto de Computação

Roteiro

Expressões Regulares

Exercícios

Expressões Regulares

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 que representam datas no formato `dd/dd/dddd`, onde `d` é um dígito qualquer.
  - Podemos verificar se uma string contém um número de telefone, descrito por uma expressão regular.

Expressões Regulares

- Note que números de telefones e datas podem ser escritos em vários formatos diferentes.
- Números de telefones:
  - 19-91234-5678
  - (019) 91234 5678
  - (19)912345678
- Datas:
  - 09/10/2019
  - 09-10-19
  - 2019-10-09

Expressões Regulares

- Expressões regulares constituem 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 possuem o conceito de expressões regulares (também chamado de RE, REGEX ou RegExp).
- Assim, aprender a escrever expressões regulares em Python será útil para descrever expressões regulares em outras linguagens de programação.
- Expressões regulares são frequentemente utilizadas para encontrar ou extrair informações de textos (*text parsing*).

Expressões Regulares

- Exemplo de expressão regular:

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

- Essa expressão regular representa uma sequência de um ou mais dígitos seguidos por uma contrabarra (\).
- Vamos aprender regras de como escrever e usar expressões regulares.
- Geralmente escrevemos expressões regular iniciando com um caractere `r` para indicar uma **`raw string`**, ou seja, uma string onde o caractere \ é tratado como um caractere normal.
- Assim, a expressão regular resultante seria:

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

Expressões Regulares - Regras

- Letras e números em uma expressão regular representam a si próprios.
- Assim a expressão regular `r'Python'` representa apenas a string `'Python'`.
- Os caracteres especiais (chamados de meta-caracteres) são:

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

Expressões Regulares - Regras

- `.` um qualquer caractere.
- `^` o início da string.
- `$` o fim da string.
- `?` repetir zero ou uma vez.
- `*` repetir zero ou mais vezes.
- `+` repetir uma ou mais vezes.
- `\` usado para indicar caracteres especiais.

Expressões Regulares - Regras

`[]` indica um conjunto de caracteres.

- `r'[0-9]'`: um dígito.
- `r'[^0-9]'`: um caractere que não é um dígito.
- `r'[a-z]'`: uma letra minúscula de a até z.
- `r'[A-Z]'`: uma letra maiúscula de A até Z.
- `r'[a-zA-Z]*'`: zero ou mais letras.
- `r'[ACTG]+'`: uma sequência de DNA.

`{}` indica a quantidade de vezes que o padrão será repetido.

- `r'[0-9]{2}'`: dois dígitos.
- `r'[a-z]{3}'`: três letras minúsculas.
- `r'[A-Z]{2,3}'`: duas ou três letras maiúsculas.
- `r'.{4,5}'`: quatro ou cinco caracteres quaisquer.
- `r'[01]{3,}'`: pelo menos três bits.
- `r'[0-9]{,6}'`: no máximo seis dígitos.

Expressões Regulares - Regras

`( )` indica um grupo em uma expressão regular.

- `r'([0-9]{3}\.){2}[0-9]{3}-[0-9]{2}'`: um CPF.
- `r'([a-z]+, )*[a-z]+'`: uma sequência de uma ou mais palavras separadas por vírgulas (e espaços).

`|` similar ao operador lógico `or` para expressões regulares.

- `r'U(nicamp|nesp|SP)'`: uma das 3 universidades paulistas.
- `r'([0-9]{3}|[a-z]{4})'`: uma sequência de três dígitos ou uma sequência de quatro letras minúsculas.

Expressões Regulares - Classes de Caracteres

- Python possui algumas classes pré-definidas de caracteres:
  - `\d` um dígito, ou seja, `[0-9]`.
  - `\D` o complemento de `\d`, ou seja, `[^0-9]`.
  - `\s` um espaço em branco, ou seja, a `[ \t\n\r\f\v]`.
  - `\S` o complemento de `\s`, ou seja, `[^ \t\n\r\f\v]`.
  - `\w` um caractere alfanumérico, ou seja, `[a-zA-Z0-9]`.
  - `\W` o complemento de `\w`, ou seja, `[^a-zA-Z0-9]`.

Expressões Regulares - Biblioteca `re`

- Em Python, expressões regulares são implementadas pela biblioteca **`re`**.
- Sendo assim, para usar expressões regulares precisamos importar a biblioteca **`re`**:

```
1 import re
```

- Documentação da biblioteca **`re`**: `https://docs.python.org/3/library/re.html`

Expressões Regulares

- A principal função da biblioteca `re` é a `search`.
- Dada uma expressão regular e uma string, a função `search` busca na string a primeira ocorrência de uma substring com o padrão especificado pela expressão regular.
- Se o padrão especificado pela expressão regular for encontrado, a função `search` retornará um objeto do tipo `Match`, caso contrário retornará `None`.
- Objetos do tipo `Match` possuem dois métodos:
  - `span`: retorna uma tupla com o local na string (posição inicial, posição final) onde a expressão regular foi encontrada.
  - `group`: retorna a substring encontrada.

Expressões Regulares

- Exemplo de uso da função `search`:

```python
import re
texto = "Algoritmos e Programação de Computadores"
result = re.search(r'(\w*)ama(\w*)', texto)
print(type(result))
# <class 're.Match'>
print(result.group())
# Programação
print(result.span())
# (13, 24)
print(re.search(r'^\w*', texto))
# <re.Match object; span=(0, 10), match='Algoritmos'>
print(re.search(r'\w*$', texto))
# <re.Match object; span=(28, 40), match='Computadores'>
print(re.search(r'(^|\s)\w{3,9}(\s|$)', texto))
# None
```

Expressões Regulares

- Outro exemplo utilizando a função **search**:

```
1 import re
2 texto = "Algoritmos e Programação de Computadores"
3 result = re.search(r'\w+', texto)
4 print(result.group())
5 # Algoritmos
6 print(result.span())
7 # (0, 10)
```

- Note que a função **search** retorna apenas a primeira ocorrência do padrão especificado.

Expressões Regulares

- Dada uma expressão regular e uma string, a função `findall` retorna uma lista com todas as ocorrências do padrão especificado pela expressão regular.
- Exemplo:

```python
import re
texto = "Algoritmos e Programação de Computadores"
result = re.findall(r'\w+', texto)
print(result)
# ['Algoritmos', 'e', 'Programação', 'de', 'Computadores']
telefone = "(019) 91234-5678"
result = re.findall(r'[0-9]+', telefone)
print(result)
# ['019', '91234', '5678']
```

Expressões Regulares

- Podemos construir uma expressão regular concatenando duas ou mais strings.
- Podemos usar o resultado das funções `search` e `findall` em expressões condicionais: `None` e `[]` são considerados `False`.

```
1  import re
2  texto = "Algoritmos e Programação de Computadores"
3  ini = "Algo"
4  meio = "ação"
5  fim = "dores"
6  regexp = r'^' + ini + r'.*' + meio + r'.*' + fim + r'$'
7  if re.search(regexp, texto):
8    print("OK")
9  else:
10   print("ERRO")
11 # OK
```

Expressões Regulares

- Podemos construir uma expressão regular concatenando duas ou mais strings.
- Podemos usar o resultado das funções `search` e `findall` em expressões condicionais: `None` e `[]` são considerados `False`.

```
1  import re
2  texto = "Algoritmos e Programação de Computadores"
3  ini = "Algo"
4  meio = "ação"
5  fim = "dores"
6  regexp = r'^' + ini + r'.*' + meio + r'.*' + fim + r'$'
7  if re.findall(regexp, texto):
8    print("OK")
9  else:
10   print("ERRO")
11 # OK
```

Expressões Regulares

- Expressões regulares podem ser utilizadas para dividir strings, similar ao método **`split`** visto na aula de strings.
- Dada uma expressão regular e uma string, a função **`split`** retorna uma lista com a divisão da string conforme especificado pela expressão regular.
- Exemplo:

```
1 import re
2 texto = "f1i1b2o3n5a8c13c21i"
3 letras = re.split(r'\d+', texto)
4 print(letras)
5 # ['f', 'i', 'b', 'o', 'n', 'a', 'c', 'c', 'i']
6 números = re.split(r'\D+', texto)
7 print(números)
8 # ['', '1', '1', '2', '3', '5', '8', '13', '21', '']
```

Expressões Regulares

- Expressões regulares podem ser utilizadas para substituir substrings, similar ao método **`replace`** visto na aula de strings.
- Dados dois padrões (strings ou expressões regulares) e uma string, a função **`sub`** retorna uma string com a subtituição na string de toda ocorrência do primeiro padrão pelo segundo padrão.

```python
import re
texto = "f1i1b2o3n5a8c13c21i"
letras = re.sub(r'\d+', "", texto)
print(letras)
# fibonacci
números = re.sub(r'(\D+)', ":", texto)
print(números)
# :1:1:2:3:5:8:13:21:
```

Expressões Regulares

- Usando a função **sub**, podemos utilizar expressões regulares para indicar como a string será modificada, com base nos grupos da expressão regular (\1, \2, etc).
- Exemplo:

```
1  import re
2  data = "19/09/1975"
3  antigo = r'(\d{2})/(\d{2})/(\d{4})'
4  novo1 = r'\1-\2-\3'
5  data1 = re.sub(antigo, novo1, data)
6  print(data1)
7  # 19-09-1975
8  novo2 = r'\3/\2/\1'
9  data2 = re.sub(antigo, novo2, data)
10 print(data2)
11 # 1975/09/19
```

Expressões Regulares

- Podemos referenciar os grupos dentro da própria expressão regular para construir padrões mais complexos.
- Exemplo:

```
1  import re
2  dna = "AGTTAGTGCACACACTGAGGTTC"
3  print(re.search(r'(G[ACTG]{2})(.*)\1', dna).group())
4  # GTTAGTGCACACACTGAGGTT
5  # 11122222222222222111
6  print(re.search(r'([ACTG]{2})(.*)\1(.*)\1', dna).group())
7  # AGTTAGTGCACACACTGAG
8  # 11221133333333333311
9  print(re.sub(r'([ACTG]{2})(.*)\1(.*)\1', r'\1\3\1\2\1',
10       dna))
11 # AGTGCACACACTGAGTTAGGTTC
12 # 113333333333112211----
```

Expressões Regulares

- Podemos recuperar cada um dos grupos de uma expressão regular com a função **group**.
- Exemplo:

```python
import re
texto = "Data de Nascimento: 19/09/1975"
result = re.search(r'(\d{2})/(\d{2})/(\d{4})', texto)
print(result.group())
# 19/09/1975
print("Dia:", result.group(1))
# Dia: 19
print("Mês:", result.group(2))
# Mês: 09
print("Ano:", result.group(3))
# Ano: 1975
print(result.group(1, 2, 3))
# ('19', '09', '1975')
```

Expressões Regulares

- Por padrão, os operadores `+`, `*`, `?` e `{,}` são executados de forma gulosa, ou seja, eles tentam casar com o maior número possível de caracteres.
- Usando o caractere `?` na frente daqueles operadores, eles são executados de forma não gulosa.
- Exemplo:

```
1 import re
2 texto = "Algoritmos e Programação de Computadores"
3 print(re.search(r'o(.*)e(.*)o', texto).group())
4 # oritmos e Programação de Computado
5 print(re.search(r'o(.*)e(.*?)o', texto).group())
6 # oritmos e Programação de Co
7 print(re.search(r'o(.*?)e(.*?)o', texto).group())
8 # oritmos e Pro
```

Expressões Regulares

- Por padrão, os operadores `+`, `*`, `?` e `{,}` são executados de forma gulosa, ou seja, eles tentam casar com o maior número possível de caracteres.
- Usando o caractere `?` na frente daqueles operadores, eles são executados de forma não gulosa.
- Exemplo:

```
1 import re
2 texto = "Removendo as <em>marcas</em> do <pre>texto</pre>."
3 print(re.sub(r'<.*>', "", texto))
4 # Removendo as .
5 print(re.sub(r'</.*>', "", texto))
6 # Removendo as <em>marcas.
7 print(re.sub(r'<.*?>', "", texto))
8
```

Expressões Regulares

- Por padrão, os operadores `+`, `*`, `?` e `{,}` são executados de forma gulosa, ou seja, eles tentam casar com o maior número possível de caracteres.
- Usando o caractere `?` na frente daqueles operadores, eles são executados de forma não gulosa.
- Exemplo:

```
1 import re
2 texto = "Removendo as <em>marcas</em> do <pre>texto</pre>."
3 print(re.sub(r'<.*>', "", texto))
4 # Removendo as .
5 print(re.sub(r'</.*>', "", texto))
6 # Removendo as <em>marcas.
7 print(re.sub(r'<.*?>', "", texto))
8 # Removendo as marcas do texto.
```

Exercícios

Exercícios

1. Escreva um programa para determinar se uma string representa um número (inteiro ou real) válido. Exemplos de números válidos: 10, $+5$, -3, -10.3, 0.80, 2.8033.
2. Escreva um programa para determinar se uma string representa um número de telefone (fixo ou celular) válido. Exemplos de números de telefones válidos:
   - (19) 3123-4567
   - 193123-4567
   - (019)3123-4567
   - (19)31234567
   - 1931234567
   - (019) 91234 5678
   - (019)91234 5678
   - 019912345678
   - 19 91234 5678
   - 1991234 5678

Exercícios

3. Com base no exercício anterior, escreva uma função que recebe como parâmetro uma string que representa um número de telefone (fixo ou celular). Caso o número não seja válido, sua função deve retornar `None`. Caso contrário, ela deve retornar uma string no formato `(XX) XXXX-XXXX` (no caso de telefone fixo) ou `(XX) XXXXX-XXXX` (no caso de telefone celular), onde X representa um dígito do telefone.
4. Escreva um programa que, dada uma palavra e uma frase, verifique se as letras da palavra aparecem na frase, na mesma ordem, mas não necessariamente de forma consecutiva. Exemplos:
    - “palavra” e “capa, lata, livro e caderno”
    - “escondida” e “mar, pesca, ondas e bebidas”

Exercício 1

```
1  import re
2  regexp = r'^[+-]?[0-9]+(\.[0-9]+)?$'
3
4  while True:
5    número = input()
6
7    if not(número):
8      break
9
10   if re.search(regexp, número):
11     print("OK")
12   else:
13     print("ERRO")
```

Exercício 2

```python
import re
ddd = r'^(0?[1-9]{2}[- ]?|\(0?[0-9]{2}\) ?)'
tel = r'[2-9]?[0-9]{4}[- ]?[0-9]{4}$'
regexp = ddd + tel

while True:
  telefone = input()

  if not(telefone):
    break

  if re.search(regexp, telefone):
    print("OK")
  else:
    print("ERRO")
```

Exercício 3

```python
import re

def padroniza_telefone(telefone):
  ddd = r'^(0?[1-9]{2}[- ]?|\(0?[0-9]{2}\) ?)'
  tel = r'([2-9]?[0-9]{4})[- ]?([0-9]{4})$'
  regexp = ddd + tel

  if re.search(regexp, telefone):
    return None

  dígitos = re.sub(r'[^0-9]', "", telefone)
  grupos = r'^0?([0-9]{2})([0-9]{4,5})([0-9]{4})$'
  formato = r'(\1) \2-\3'

  return re.sub(grupos, formato, dígitos)
```

Exercício 4

```python
import re

palavra = input()
frase = input()

regexp = ".*".join(list(palavra))

if re.search(regexp, frase):
  print("OK")
else:
  print("ERRO")
```