pyladies
Campinas

Atendimento!!

Quando? 27/05

Horário? 18h às 19h

Onde? Sala SI03

Algoritmos e Programação de Computadores

Matrizes e Vetores Multidimensionais

Profa. Sandra Avila
Instituto de Computação (IC/Unicamp)

MC102, 24 Maio, 2019

Agenda

- Matrizes
  - Matrizes e vetores multidimensionais
  - Criando matrizes
  - Acessando dados de uma matriz
  - Declarando vetores multidimensionais
- Exemplo com matrizes
- Exercícios
- **Informações extras: NumPy**
  - **O tipo Array**

NumPy

NumPy

- `NumPy` é uma biblioteca para Python que contém tipos para representar vetores e matrizes juntamente com diversas operações, dentre elas operações comuns de álgebra linear.

- `NumPy` é implementado para trazer maior eficiência do código em Python para aplicações científicas.

NumPy

- Primeiramente deve-se instalar o `NumPy` baixando-se o pacote de http://www.numpy.org

- Para usar os itens deste pacote deve-se importá-lo inicialmente com o comando: `import numpy`

NumPy

- O objeto mais simples da biblioteca é o **array** que serve para criar vetores homogêneos multidimensionais.
- Um **array** pode ser criado a partir de uma lista:

```
import numpy as np
a = np.array([1,2,3])
print(a.ndim)
print(a.size)
a
```

```
1
3
array([1,2,3])
```

NumPy

- O objeto mais simples da biblioteca é o **array** que serve para criar vetores homogêneos multidimensionais.
- Um **array** pode ser criado a partir de uma lista:

```
import numpy as np
b = np.array([[1,2,3],[4,5,6]])
print(b.ndim)
print(b.size)
b
```

```
2
6
array([[1, 2, 3],
       [4, 5, 6]])
```

NumPy

- Um **array** pode ser criado com mais do que uma dimensão utilizando as funções **arange** e **reshape**.

```
a = np.arange(10)
a
a = np.arange(10).reshape(2,5)
a
```

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

NumPy

- NumPy oferece a função **zeros** que cria um **array** contendo apenas zeros. Seu argumento de entrada é uma tupla.

```
np.zeros((3))
np.zeros((3,4))
```

```
array([ 0., 0., 0.])
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

NumPy

- Os operadores `*`, `-` , `+` , `/`, `**`, quando utilizados sob arrays, são aplicados em cada posição do **array**.

```
m = np.ones((2,3))
m+1
```

```
array([[ 2.,  2.,  2.],
       [ 2.,  2.,  2.]])
```

NumPy

- Os operadores `*`, `-` , `+` , `/`, `**`, quando utilizados sob arrays, são aplicados em cada posição do **array**.

```
m*4
```

```
array([[ 4.,  4.,  4.],
       [ 4.,  4.,  4.]])
```

NumPy

- Os operadores `*`, `-` , `+` , `/`, `**`, quando utilizados sob arrays, são aplicados em cada posição do **array**.

```
m = m + 1
m**3
```

```
array([[ 8.,  8.,  8.],
       [ 8.,  8.,  8.]])
```

NumPy

- Os operadores `*`, `-` , `+` , `/`, `**`, quando utilizados sob arrays, são aplicados em cada posição do **array**.

```
A = np.array([[1,1], [0,1]])
B = np.array([[2,0], [3,4]])
A*B
```

```
array([[2, 0],
       [0, 4]])
```

- Multiplicação de elemento por elemento.

NumPy

- Os operadores `*`, `-` , `+` , `/`, `**`, quando utilizados sob arrays, são aplicados em cada posição do **array**.

```
A = np.array([[1,1], [0,1]])
B = np.array([[2,0], [3,4]])
np.dot(A,B) # multiplicação de matrizes
```

```
array([[5, 4],
       [3, 4]])
```

NumPy

- Na biblioteca existe uma variedade de outras funções como funções para calcular autovalores e autovetores, resolução de um sistema de equações lineares, etc.

Referências & Exercícios

- Os slides dessa aula foram baseados no material de MC102 do Prof. Eduardo Xavier (IC/Unicamp).

- https://panda.ime.usp.br/aulasPython/static/aulasPython/aula11.html
- http://www.galirows.com.br/meublog/programacao/exercicios-resolvidos-python/ (Exercícios resolvidos, explicação com vídeo)

Algoritmos e Programação de Computadores

Expressões Regulares

Profa. Sandra Avila
Instituto de Computação (IC/Unicamp)

MC102, 24 Maio, 2019

Agenda

- Expressões regulares
- Usando REs
- Regras básicas para Escrita de uma RE
- Exercícios

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).

Expressões Regulares

- 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

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).

Uma Expressão Regular

- Um exemplo de expressão regular é: `'\d+'`
- Essa RE representa uma sequência de 1 ou mais dígitos.

- 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: **`import`** `re`

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

`re.search`

- **`re.search`**: dada uma RE e uma string, a função busca a **primeira** ocorrência de uma substring especificada pela RE.

```
import re
digitos = re.search(r'\d+','MC102 é importante para rac3ocín30 lógic0')
digitos
```

```
<_sre.SRE_Match object; span=(2, 5), match='102'>
```

re.search

- 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*).

```
import re
digitos = re.search(r'\d+','É importante para o raciocínio lógico.')
digitos
```

```
#Não acha nada
```

- 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:

```
import re
digitos = re.search(r'\d+','É importante para o raciocínio lógico.')
if digitos:
    print("Sim", digitos)
else:
    print("Não", digitos)
```

```
None Não
```

- 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.

```
import re
digitos = re.search(r'\d+','MC102 é importante para rac3ocín30 lógic0')
```

```
digitos.span()
(2,5)
```

```
digitos.group()
102
```

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.

```
import re
digitos = re.search(r'\d+','MC102 é importante para rac3ocín30 lógic0')
```

```
digitos.span()
(2,5)
```

```
digitos.group()
102
```

Note que o método `re.search` acha apenas a **primeira instância** da RE na string (os números 3, 30 e 0 também satisfazem 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.

```
import re
digitos = re.match(r'\d+','MC102 é importante para rac3ocín30 lógic0')
digitos
```

```
#Não acha nada
```

```
digitos = re.match(r'\d+','102MC é importante para rac3ocín30 lógic0')
digitos
```

```
<_sre.SRE_Match object; span=(0, 3), match='102'>
```

Outras Funções da Biblioteca re

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

```
import re
digitos = re.sub(r'\d+','S','MC102 é importante para rac3ocín30 lógic0')
digitos
```

```
'MCS é importante para racSocínS lógicS'
```

```
digitos = re.sub(r'\d+','Z','MCS é importante para racSocínS lógicS')
digitos
```

```
#Não tem dígito para substituir
```

Note que não temos nenhum dígito `r'\d+'` para substituir.

Outras Funções da Biblioteca re

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

```
import re
digitos = re.findall(r'\d+','MC102 é importante para rac3ocín30 lógic0')
digitos
```

```
['102', '3', '30', '0']
```

Outras Funções da Biblioteca re

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

```
import re
digitos = re.split(r'\d+','MC102 é importante para rac3ocín30 lógic0')
digitos
```

```
['MC', ' é importante para rac', 'ocín', ' lógic', '']
```

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.

```
import re
dd = re.compile(r'\d+')
dd.search('MC102 é importante para rac3ocín30 lógic0')
```

```
<_sre.SRE_Match object; span=(2, 5), match='102'>
```

Compilando REs

- 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.

```
import re
dd = re.compile(r'\d+')
dd.search('MC102 é importante para rac3ocín30 lógic0', 10)
```

```
<_sre.SRE_Match object; span=(27, 28), match='3'>
```

- O método **`search`** começou a procurar a RE a partir da posição 10.

Regras Básicas para Escrita de uma REs

- 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'.ão'` representa todas as strings de 3 caracteres cujos 2 últimos são ão.

```
import re
regex = re.compile(r'.ão')
```

```
regex.search("O cão")
```

```
<_sre.SRE_Match object; span=(2, 5), match='cão'>
```

```
regex.search("O cão são")
```

```
<_sre.SRE_Match object; span=(2, 5), match='cão'>
```

Repetições

```
regex.search("O pão")
```

```
<_sre.SRE_Match object; span=(2, 5), match='pão'>
```

```
regex.search("O 3cão")
```

```
<_sre.SRE_Match object; span=(3, 6), match='cão'>
```

```
regex.search("ão")
```

```
#Não acha nada
```

Note que não temos uma string de 3 caracteres cujos 2 últimos são ão.

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`.

```
import re
regex = re.compile(r'p[aã]o')
```

```
regex.search("O pão")
```

```
<_sre.SRE_Match object; span=(2, 5), match='pão'>
```

Classe de Caracteres

```
regex.search("O po")
```

```
#Não acha nada
```

```
regex.search("O seu pao")
```

```
<_sre.SRE_Match object; span=(6, 9), match='pao'>
```

```
regex.search("paão")
```

```
#Não acha nada
```

Classe de Caracteres

- O caractere – dentro do `[]` representa um intervalo. Assim `[1-8]` representa um dos dígitos de `1` a `8`.
- De forma parecida `[a-z]` e `[0-9]` representam as letras minúsculas e os dígitos, respectivamente.

```
import re
regex = re.compile(r'MC[0-9]')
```

```
regex.search('MC102 é importante para o rac3ocín30 lógic0')
```

```
<_sre.SRE_Match object; span=(0, 3), match='MC1'>
```

Classe de Caracteres

- O caractere `^` no início de `[]` representa a negação da classe.
- Assim `r'MC[^4-9]'` representa qualquer string começando com `MC` e terminando com qualquer caractere exceto os de `4` até `9`.

```
import re
regex = re.compile(r'MC[^4-9]')
```

```
regex.search('MC202 é importante para o rac3ocín30 lógic0')
```

```
<_sre.SRE_Match object; span=(0, 3), match='MC1'>
```

```
regex.search('MC402 é importante para o rac3ocín30 lógic0')
```

```
#Não acha nada
```

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* (espaços em branco), i.e., equivalente a `[\t\n\r\f\v]`.
  - `\S`: O complemento de `\s`.

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.
  - ...
  - `\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**.
- A seguir há um *match* de `r'ab?c'` tanto com `abc` quanto com `ac`.

```
import re
regex = re.compile(r'ab?c')
```

```
regex.search('abc')
```

```
<_sre.SRE_Match object; span=(0, 3), match='abc'>
```

```
regex.search('ac')
```

```
<_sre.SRE_Match object; span=(0, 3), match='abc'>
```

Opcional

- Pode-se criar um grupo incluindo-se uma string entre parênteses. Se quisermos detectar ocorrências de Maio 2019, Mai 2019 ou Maio de 2019, podemos usar a RE `r'Mai(o)? (de)? ?2019'`

```
import re
regex = re.compile(r'Mai(o)? (de)? ?2019')
```

```
regex.search('O mês é Maio de 2019')
```

```
<_sre.SRE_Match object; span=(8, 20), match='Maio de 2019'>
```

```
regex.search('Quando será o evento? Maio 2019')
```

```
<_sre.SRE_Match object; span=(22, 31), match='Maio 2019'>
```

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.

```
import re
r1 = re.compile(r'abc(de)+')
r2 = re.compile(r'abc(de)*')
```

```
r1.search('abc')
```

```
#Não acha nada
```

```
r2.search('abc')
```

```
<_sre.SRE_Match object; span=(0, 3), match='abc'>
```

Repetições

```
import re
r1 = re.compile(r'abc(de)+')
r2 = re.compile(r'abc(de)*')
```

```
r1.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.

```
import re
re.search(r'\bcasa\b', 'a casa')
```

```
<_sre.SRE_Match object; span=(2, 6), match='casa'>
```

```
re.search(r'\bcasa\b', 'o casamento')
```

```
#Não acha nada
```

Exemplo

Exemplo: Buscando um email

Uma RE para buscar emails:

- O userid (nome do usuário) é uma sequência de caracteres alfanuméricos `\w+` separado por @.
- O host (ex., gmail) é uma sequência de caracteres alfanuméricos `\w+`.

```
import re
re.search(r'\w+@\w+', 'bla bla abc@gmail.com bla')
```

```
<_sre.SRE_Match object; span=(8, 17), match='abc@gmail'>
```

Exemplo: Buscando um email

Uma RE para buscar emails:

- 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 abc@gmail.com bla')
```

```
<_sre.SRE_Match object; span=(8, 21), match='abc@gmail.com'>
```

Exemplo: Buscando um email

Uma RE para buscar emails:

```
re.search(r'\w+@\w+\.\w+', 'bla bla abc@gmail.com.br bla')
```

```
<_sre.SRE_Match object; span=(8, 21), match='abc@gmail.com'>
```

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

Exemplo: Buscando um email

Uma RE para buscar emails:

- 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 abc@gmail.com.br bla')
```

```
<_sre.SRE_Match object; span=(8, 24), match='abc@gmail.com.br'>
```

```
re.search(r'\w+@\w+\.\w+(\.\w+)?', 'bla bla abc@gmail.com bla')
```

```
<_sre.SRE_Match object; span=(8, 21), match='abc@gmail.com'>
```

Exemplo: Buscando um email

Uma RE para buscar emails:

```
re.search(r'\w+@\w+\.\w+(\.\w+)?', 'O email da professora é sandra@ic.unicamp.br')
```

```
<_sre.SRE_Match object; span=(24, 44), match='sandra@ic.unicamp.br'>
```

- Há muito mais coisas sobre como escrever REs: https://docs.python.org/3/howto/RE.html#RE-howto

Exercícios

Exercício 1

- Escreva uma RE para encontrar números de telefone do tipo:

  (019)91234 5678

  19 91234 5678

  19-91234-5678

  (19) 91234-5678

Exercício 2

- Faça uma função que recebe um string e retorna o string com os números de telefones transformados para um único formato:

  (19) 91234 5678

Referências & Exercícios

- Os slides dessa aula foram baseados no material de MC102 do Prof. Eduardo Xavier (IC/Unicamp).

- https://www.vooo.pro/insights/tutorial-sobre-expressoes-regulares-para-iniciantes-em-python/
- https://docs.python.org/3/library/re.html (em inglês)