NumPy

Thiago Teixeira Santos
<thiago.santos@embrapa.br>

17 de novembro de 2017

1 NumPy

Previamente na Seção 2, *A Linguagem Python*, vimos que listas podem armazenar qualquer tipo de dado. Tal flexibilidade acarreta perdas em desempenho, uma vez que, internamente, o interpretador Python precisa lidar com questões de posicionamento em memória e tipagem de dados. Em computação científica, a maior parte dos dados consiste em valores numéricos, geralmente de mesmo tipo (inteiro ou real). Ganhos de desempenho consideráveis podem ser obtidos se houver uniformidade no tipo de dado que está sendo considerado.

Um **array NumPy** é um coleção multidimensional e uniforme de elementos, isto é, todos os elementos ocupam o mesmo número de bytes em memória (VAN DER WALT et al., 2011). Ele é a representação padrão para arrays *n*-dimensionais na SciPy Stack e é comumente utilizada na representação de vetores, matrizes, observações, sinais e imagens (e imagens (*arrays* de até 32 dimensões podem ser representados).

- NumPy é um módulo Python que fornece **arrays multidimensionais**
- Internamente, executa código **nativo** e **otimizado**
- Foi desenvolvido especialmente para computação científica

NumPy pode ser importado no ambiente com o comando `import numpy`, como no exemplo abaixo:

```
In [1]: import numpy as np
        A = np.array([0,1,2,3])
        A

Out[1]: array([0, 1, 2, 3])

In [2]: A.shape = 2,2
        A

Out[2]: array([[0, 1],
               [2, 3]])
```

Se o interpretador IPython foi iniciado com a opção `pylab`, o módulo NumPy já estará carregado e não haverá necessidade, como no exemplo acima, do prefixo `np`.

```
In [4]: %pylab inline
        A = array([0,1,2,3])
        A
```

```
Populating the interactive namespace from numpy and matplotlib
```

```
Out[4]: array([0, 1, 2, 3])
```

NumPy evita iterações custosas do interpretador Python quando opera com arrays, utilizando, no lugar, eficientes operações **vetorizadas** implementadas em código de máquina. Como um exemplo simples, suponha que tenhamos um vetor com 10000 valores e desejemos computar o quadrado de cada valor. Utilizando uma lista, o interpretador Python iria usar iterações para realizar a computação para cada membro da lista:

- NumPy utiliza operações **vetorizadas**
- São executadas de modo paralelo, aproveitando recursos do processador

```
In [5]: v = range(10000)
        %timeit [i**2 for i in v]
```

```
1000 loops, best of 3: 564 µs per loop
```

A função `range` produz uma **lista** com 10000 valores, de 0 a 9999. A computação levou em média, para a máquina em questão, cerca de 500 *microssegundos*. Abaixo, utilizamos a função `arange` do módulo NumPy, que devolve um **array** com os mesmos 10000 valores:

```
In [7]: v = arange(10000)
        %timeit v**2
```

```
The slowest run took 11.34 times longer than the fastest. This could mean that an intermediate r
100000 loops, best of 3: 7.67 µs per loop
```

Utilizando código nativo e vetorização, NumPy levou cerca de 7 microssegundos para realizar toda a computação. Nos exemplos acima, a função "mágica" do IPython `%timeit` foi utilizada para calcular o desempenho do código. A operação `v**2` computa um novo array com todos os elementos de `v` ao quadrado (mais detalhes na Seção *Operações com arrays*).

1.1 Criação de arrays

Arrays podem ser facilmente criados com a função `array`, que recebe como argumento uma lista contendo os dados que deverão ser armazenados. O número de dimensões de um array pode ser obtido pelo atributo `ndim` enquanto que as dimensões em si ficam armazenadas no atributo `shape`:

```
In [8]: v = array([0,1,2,3])
        v
```

```
Out[8]: array([0, 1, 2, 3])
```

```
In [9]: v.ndim

Out[9]: 1

In [10]: v.shape

Out[10]: (4,)
```

A lista com os dados pode conter uma estrutura que permita ao método `array` inferir as dimensões pretendidas. Por exemplo, uma matriz 2D pode ser inicializada a partir de uma **lista de listas**:

```
In [11]: A = array([ [0,1,2], [3,4,5] ])
         A

Out[11]: array([[0, 1, 2],
                [3, 4, 5]])

In [12]: A.ndim

Out[12]: 2

In [13]: A.shape

Out[13]: (2, 3)
```

Se for mais conveniente, é possível passar apenas uma lista com todos os elementos encadeados e as dimensões do array podem ser redefinidas posteriormente:

```
In [14]: data = range(12)
         data

Out[14]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

In [15]: B = array(data)
         B

Out[15]: array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])

In [16]: B.ndim

Out[16]: 1

In [17]: B.shape

Out[17]: (12,)

In [19]: B[6]

Out[19]: 6

In [20]: B.shape = 3,4
```

```
In [21]: B.ndim

Out[21]: 2

In [22]: B.shape

Out[22]: (3, 4)

In [23]: B

Out[23]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

In [24]: # Linha 1, coluna 2
         B[1,2]

Out[24]: 6
```

1.1.1 Utilitários para criação de *arrays*

Para criar um array contendo uma sequência com n elementos de 0 a $n-1$, podemos utilizar a função `arange`:

```
In [25]: v = arange(5)
         v

Out[25]: array([0, 1, 2, 3, 4])
```

A função `arange` também permite definir os elementos inicial e final e um **passo** entre elementos sucessivos. Por exemplo, para produzir uma sequência de 1 (inclusivo) até 8 (exclusivo) na qual os números inteiros são tomados a cada 2 elementos, pode-se utilizar:

```
In [26]: v = arange(1,8,2)
         v

Out[26]: array([1, 3, 5, 7])
```

A função `arange` trabalha com números em $\mathbb{Z}$. Para uma sequência de números reais, igualmente espaçada dentro de um intervalo, pode-se utilizar `linspace`. Por exemplo, para obter 6 números reais igualmente espaçados no intervalo $[0, 1]$, utiliza-se:

```
In [27]: v = linspace(0, 1, 6)
         v

Out[27]: array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

`zeros`, `ones`, `identity` Para produzir matrizes de dimensões variadas com um valor fixo, podemos utilizar as funções `zeros` e `ones`, informando as dimensões desejadas na forma de uma tupla como argumento:

```
In [28]: A = zeros((4,5))
         A

Out[28]: array([[ 0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.],
                [ 0.,  0.,  0.,  0.,  0.]])

In [29]: B = ones((3,3))
         B

Out[29]: array([[ 1.,  1.,  1.],
                [ 1.,  1.,  1.],
                [ 1.,  1.,  1.]])
```

Matrizes identidade são particularmente importantes em computação envolvendo álgebra linear. A função `identity` produz uma matriz quadrada do tamanho desejado, com todos os elementos em sua diagonal principal apresentando o valor 1, zero para todos os demais:

```
In [30]: I = identity(3)
         I

Out[30]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])
```

`random` Certas aplicações podem necessitar de valores aleatórios, por exemplo em testes com ruído sintético adicionado ao sinal. Arrays aleatórios podem ser produzidos utilizando-se o submódulo `random` da NumPy. Valores obtidos de uma distribuição uniforme no intervalo [0,1] são produzidos com `rand`:

```
In [31]: A = random.rand(4)
         A

Out[31]: array([ 0.98374794,  0.64589646,  0.01020435,  0.08243475])
```

Outra opção é utilizar uma distribuição Gaussiana através da função `randn`:

```
In [32]: A = np.random.randn(4)
         A

Out[32]: array([ 0.03603625, -2.03178797, -0.69134411, -0.89009602])
```

1.2 Tipos de dados

Diferente de listas, que podem armazenar elementos de tipos diferentes, arrays devem conter elementos de **mesmo tipo**, como especificado em `dtype`.

```
In [33]: v = array([1,2,3])
         v.dtype

Out[33]: dtype('int64')
```

Acima, vemos que o tipo padrão para inteiros adotado pelo sistema em uso no exemplo utiliza 64 bits. Similarmente, vemos que o tipo padrão para ponto flutuante também se baseia em 64 bits:

```
In [34]: v = array([1., 2., 3.])
         v.dtype

Out[34]: dtype('float64')
```

Alternativamente, podemos especificar exatamente o tipo desejado, dentre os disponíveis na NumPy. O exemplo abaixo cria um array com elementos em ponto flutuante com 32 bits, apesar do argumento de entrada consistir em uma lista de inteiros:

```
In [37]: v = array([1,2,3], dtype=float32)
         v

Out[37]: array([ 1.,  2.,  3.], dtype=float32)

In [38]: v.dtype

Out[38]: dtype('float32')
```

Utilitários para criação de arrays também possibilitam especificar o tipo pretendido:

```
In [39]: I = identity(3, dtype=np.uint8)
         I

Out[39]: array([[1, 0, 0],
                [0, 1, 0],
                [0, 0, 1]], dtype=uint8)
```

1.3 Carga de dados armazenados em arquivos

Na maior parte de seu trabalho, o pesquisador não irá inserir manualmente o conteúdo dos arrays, nem terá todas suas necessidades atentidas pelos utilitários de criação. São necessários mecanismos para a carga de dados armazenados em arquivo. Considere por exemplo o conteúdo do arquivo `populations.txt`:

```
In [41]: !cat ./data/populations.txt
```

```
# year        hare        lynx        carrot
1900        30e3        4e3        48300
1901        47.2e3        6.1e3        48200
1902        70.2e3        9.8e3        41500
1903        77.4e3        35.2e3        38200
1904        36.3e3        59.4e3        40600
1905        20.6e3        41.7e3        39800
1906        18.1e3        19e3        38600
1907        21.4e3        13e3        42300
1908        22e3        8.3e3        44500
1909        25.4e3        9.1e3        42100
1910        27.1e3        7.4e3        46000
1911        40.3e3        8e3        46800
1912        57e3        12.3e3        43800
1913        76.6e3        19.5e3        40900
1914        52.3e3        45.7e3        39400
1915        19.5e3        51.1e3        39000
1916        11.2e3        29.7e3        36700
1917        7.6e3        15.8e3        41800
1918        14.6e3        9.7e3        43300
1919        16.2e3        10.1e3        41300
1920        24.7e3        8.6e3        47300
```

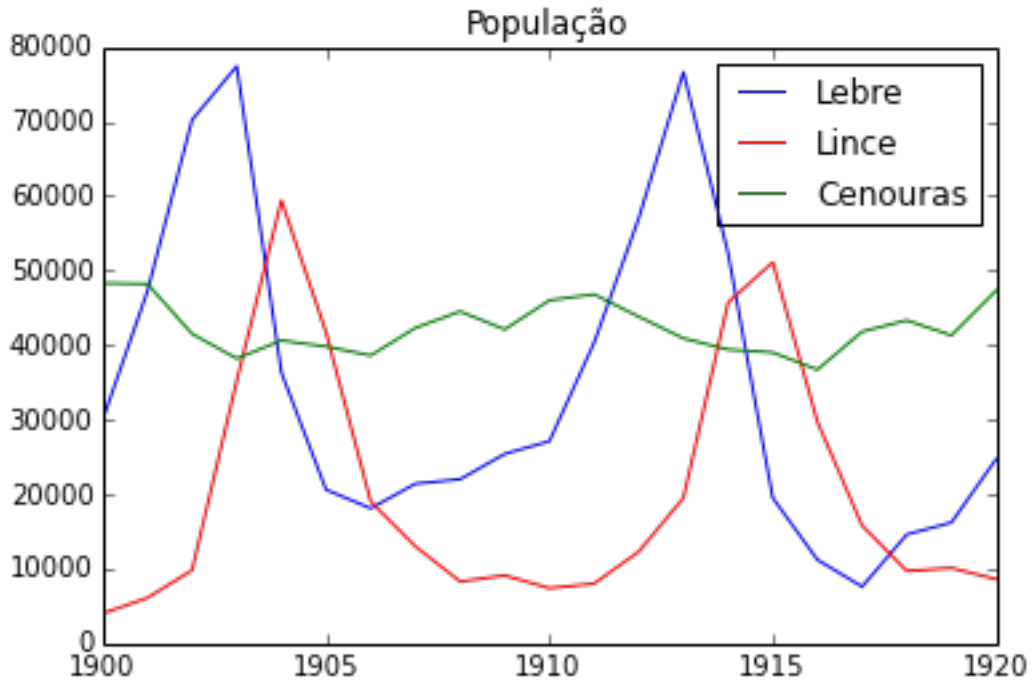
Esta tabela pode ser facilmente carregada pelo ambiente como um array NumPy com o uso da função `loadtxt`:

```
In [42]: populations = loadtxt('./data/populations.txt')
         populations

Out[42]: array([[  1900.,  30000.,   4000.,  48300.],
                [  1901.,  47200.,   6100.,  48200.],
                [  1902.,  70200.,   9800.,  41500.],
                [  1903.,  77400.,  35200.,  38200.],
                [  1904.,  36300.,  59400.,  40600.],
                [  1905.,  20600.,  41700.,  39800.],
                [  1906.,  18100.,  19000.,  38600.],
                [  1907.,  21400.,  13000.,  42300.],
                [  1908.,  22000.,   8300.,  44500.],
                [  1909.,  25400.,   9100.,  42100.],
                [  1910.,  27100.,   7400.,  46000.],
                [  1911.,  40300.,   8000.,  46800.],
                [  1912.,  57000.,  12300.,  43800.],
                [  1913.,  76600.,  19500.,  40900.],
                [  1914.,  52300.,  45700.,  39400.],
                [  1915.,  19500.,  51100.,  39000.],
                [  1916.,  11200.,  29700.,  36700.],
                [  1917.,   7600.,  15800.,  41800.],
```

```
[  1918.,  14600.,   9700.,  43300.],
[  1919.,  16200.,  10100.,  41300.],
[  1920.,  24700.,   8600.,  47300.]])
```

Como ilustração, abaixo segue uma representação gráfica do desenvolvimento das populações de cenouras, lebres e linces ao longo do tempo, contida no array `populations`. A produção de gráficos será apresentada na Seção *Matplotlib*.

```
In [43]: year = populations[:,0]
         hare = populations[:,1]
         lynx = populations[:,2]
         carrot = populations[:,3]
         plot(year, hare, 'b-')
         plot(year, lynx, 'r-')
         plot(year, carrot, 'g-')
         title(u'População')
         legend(('Lebre', 'Lince', 'Cenouras'))

Out[43]: <matplotlib.legend.Legend at 0x7fc36dd46650>
```



De modo similar, `savetxt` pode ser utilizado para salvar um array em um **arquivo texto**. NumPy também possui uma funções `save` e `load` que respectivamente salvam e carregam arrays em um **formato binário** próprio.

1.3.1 Leitura de arquivos CSV

Considere que os dados estão no formato CSV (*Comma Separated Values*):

```
In [45]: !cat data/populations.csv
```

```
year,hare,lynx,carrot
1900,30000,4000,48300
1901,47.2e3,6.1e3,48200
1902,70.2e3,9.8e3,41500
1903,77.4e3,35.2e3,38200
1904,36.3e3,59.4e3,40600
1905,20.6e3,41.7e3,39800
1906,18.1e3,19000,38600
1907,21.4e3,13000,42300
1908,22000,8.3e3,44500
1909,25.4e3,9.1e3,42100
1910,27.1e3,7.4e3,46000
1911,40.3e3,8000,46800
1912,57000,12.3e3,43800
1913,76.6e3,19.5e3,40900
1914,52.3e3,45.7e3,39400
1915,19.5e3,51.1e3,39000
1916,11.2e3,29.7e3,36700
1917,7.6e3,15.8e3,41800
1918,14.6e3,9.7e3,43300
1919,16.2e3,10.1e3,41300
1920,24.7e3,8.6e3,47300
```

Novamente, podemos utilizar `loadtxt`

```
In [47]: populations = loadtxt('data/populations.csv', delimiter=',', skiprows=1)
         populations
```

```
Out[47]: array([[  1900.,  30000.,   4000.,  48300.],
                [  1901.,  47200.,   6100.,  48200.],
                [  1902.,  70200.,   9800.,  41500.],
                [  1903.,  77400.,  35200.,  38200.],
                [  1904.,  36300.,  59400.,  40600.],
                [  1905.,  20600.,  41700.,  39800.],
                [  1906.,  18100.,  19000.,  38600.],
                [  1907.,  21400.,  13000.,  42300.],
                [  1908.,  22000.,   8300.,  44500.],
                [  1909.,  25400.,   9100.,  42100.],
                [  1910.,  27100.,   7400.,  46000.],
                [  1911.,  40300.,   8000.,  46800.],
                [  1912.,  57000.,  12300.,  43800.],
                [  1913.,  76600.,  19500.,  40900.],
                [  1914.,  52300.,  45700.,  39400.],
                [  1915.,  19500.,  51100.,  39000.],
                [  1916.,  11200.,  29700.,  36700.],
                [  1917.,   7600.,  15800.,  41800.],
```

```
                [  1918.,  14600.,   9700.,  43300.],
                [  1919.,  16200.,  10100.,  41300.],
                [  1920.,  24700.,   8600.,  47300.]])
```

Porém, é comum situações em que tabelas de dados apresentam alguns dados ausentes. Considere, por exemplo, o seguinte arquivo CSV:

```
In [48]: !cat data/populations-missing.csv
```

```
year,hare,lynx,carrot
1900,30000,4000,48300
1901,,5.1e3,48200
1902,70.2e3,9.8e3,41500
1903,77.4e3,35.2e3,38200
1904,36.3e3,59.4e3,40600
1905,,41.7e3,39800
1906,18.1e3,19000,38600
1907,21.4e3,13000,42300
1908,22000,,44500
1909,25.4e3,9.1e3,42100
1910,27.1e3,,46000
1911,40.3e3,8000,46800
1912,57000,12.3e3,43800
1913,76.6e3,19.5e3,
1914,52.3e3,45.7e3,
1915,19.5e3,51.1e3,39000
1916,11.2e3,29.7e3,36700
1917,7.6e3,15.8e3,41800
1918,14.6e3,9.7e3,43300
1919,,10.1e3,41300
1920,24.7e3,8.6e3,47300
```

Para essa situação, podemos utilizar `genfromtxt`:

```
In [51]: populations = genfromtxt('data/populations-missing.csv', delimiter=',', skiprows=1)
         populations
```

```
Out[51]: array([[  1900.,  30000.,   4000.,  48300.],
                [  1901.,     nan,   5100.,  48200.],
                [  1902.,  70200.,   9800.,  41500.],
                [  1903.,  77400.,  35200.,  38200.],
                [  1904.,  36300.,  59400.,  40600.],
                [  1905.,     nan,  41700.,  39800.],
                [  1906.,  18100.,  19000.,  38600.],
                [  1907.,  21400.,  13000.,  42300.],
                [  1908.,  22000.,     nan,  44500.],
                [  1909.,  25400.,   9100.,  42100.],
                [  1910.,  27100.,     nan,  46000.],
```

```
[  1911.,  40300.,   8000.,  46800.],
[  1912.,  57000.,  12300.,  43800.],
[  1913.,  76600.,  19500.,     nan],
[  1914.,  52300.,  45700.,     nan],
[  1915.,  19500.,  51100.,  39000.],
[  1916.,  11200.,  29700.,  36700.],
[  1917.,   7600.,  15800.,  41800.],
[  1918.,  14600.,   9700.,  43300.],
[  1919.,     nan,  10100.,  41300.],
[  1920.,  24700.,   8600.,  47300.]])
```

1.4 Sub-arrays

```
In [52]: import itertools as itts
         data = [10 * i + j for i, j in itts.product(range(6), range(6))]
         A = np.array(data)
         A.shape = 6,6
```

Considere a matriz `A` abaixo (representa como um array NumPy):

```
In [54]: A

Out[54]: array([[ 0,  1,  2,  3,  4,  5],
                [10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35],
                [40, 41, 42, 43, 44, 45],
                [50, 51, 52, 53, 54, 55]])
```

A matriz é bi-dimensional. Seus elementos podem ser indexados através de duas coordenadas, separadas por uma vírgula:

```
In [55]: A[2,3]

Out[55]: 23
```

Fragmentos do array podem ser obtidos pelo mesmo tipo de *slicing* visto anteriormente para listas Python. Suponha que desejemos obter apenas as colunas 3 e 4 na linha 4. Podemos utilizar slicing nas colunas de `A`:

```
In [56]: A[4,3:6]

Out[56]: array([43, 44, 45])
```

Suponha agora que desejemos todas as linhas a partir da linha 4 e todas as colunas a partir da coluna 3:

```
In [57]: A[4:,3:]

Out[57]: array([[43, 44, 45],
                [53, 54, 55]])
```

Outros exemplos:

```
In [58]: A[:,2]

Out[58]: array([ 2, 12, 22, 32, 42, 52])

In [59]: A[2::2,::2]

Out[59]: array([[20, 22, 24],
                [40, 42, 44]])
```

Essas operações de *slicing* criam um **visão** (*view*) do *array* original, não uma cópia. Quando a visão é modificada, o array original é modificado também:

```
In [44]: B = A[0,3:5]
         B[0] = 99
         print B
         print A

[99  4]
[[ 0  1  2 99  4  5]
 [10 11 12 13 14 15]
 [20 21 22 23 24 25]
 [30 31 32 33 34 35]
 [40 41 42 43 44 45]
 [50 51 52 53 54 55]]
```

Para evitar sobreescrita, se necessário, podemos criar uma **cópia do array**. No exemplo abaixo, as matrizes `A` e `B` são representadas por arrays que não compartilham memória, não havendo sobre-escrita:

```
In [60]: A = array(data)
         A.shape = 6,6
         # Cópia
         B = A[0,3:5].copy()
         B[0] = 99
         print B
         print A

[99  4]
[[ 0  1  2  3  4  5]
 [10 11 12 13 14 15]
 [20 21 22 23 24 25]
 [30 31 32 33 34 35]
 [40 41 42 43 44 45]
 [50 51 52 53 54 55]]
```

1.5 *Fancy indexing*

Podemos utilizar **máscaras** para selecionar os elementos de um *array*:

```
In [46]: A = array(data)
         A.shape = 6,6
         A

Out[46]: array([[ 0,  1,  2,  3,  4,  5],
                [10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35],
                [40, 41, 42, 43, 44, 45],
                [50, 51, 52, 53, 54, 55]])
```

No exemplo abaixo, `mask` é um **array booleano** no qual cada elemento recebe o valor "Verdadeiro" (*True*) se e somente se seu elemento equivalente no array `A` for divisível por 3:

```
In [61]: mask = A % 3 == 0
         mask

Out[61]: array([[ True, False, False,  True, False, False],
                [False, False,  True, False, False,  True],
                [False,  True, False, False,  True, False],
                [ True, False, False,  True, False, False],
                [False, False,  True, False, False,  True],
                [False,  True, False, False,  True, False]], dtype=bool)
```

O array booleano `mask` pode ser utilizado para selecionar elementos de `A` (no caso, apenas os elementos múltiplos de 3):

```
In [62]: A[mask]

Out[62]: array([ 0,  3, 12, 15, 21, 24, 30, 33, 42, 45, 51, 54])
```

Podemos também aplicar a máscara booleana diretamente:

```
In [63]: B = A[A % 3 == 0]
         B # importante: B é uma cópia

Out[63]: array([ 0,  3, 12, 15, 21, 24, 30, 33, 42, 45, 51, 54])
```

Fancy indexing também pode ser realizado a partir de um array ou uma sequência de índices (na forma de listas ou tuplas). No exemplo abaixo, utilizamos a lista `[1,5,6,8]` como índice, recuperando assim os elemento nas posições 1, 5, 6 e 8 no array `v`:

```
In [64]: v = arange(0,100,10)
         v

Out[64]: array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

```
In [65]: v[[1,5,6,8]]

Out[65]: array([10, 50, 60, 80])
```

No exemplo abaixo, duas sequências são fornecidas, na forma de tuplas. A primeira indexa as linhas da matriz A, enquanto a segunta indexa as colunas:

```
In [66]: A

Out[66]: array([[ 0,  1,  2,  3,  4,  5],
                [10, 11, 12, 13, 14, 15],
                [20, 21, 22, 23, 24, 25],
                [30, 31, 32, 33, 34, 35],
                [40, 41, 42, 43, 44, 45],
                [50, 51, 52, 53, 54, 55]])

In [67]: A[(0,1,2,3,4),(1,2,3,4,5)]

Out[67]: array([ 1, 12, 23, 34, 45])
```

1.6 Operações com arrays

Escalar e array:

```
In [68]: v = arange(10)
         v

Out[68]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [73]: v + 1

Out[73]: array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10])

In [74]: 2**v

Out[74]: array([  1,   2,   4,   8,  16,  32,  64, 128, 256, 512])
```

Entre arrays, elemento por elemento:

```
In [75]: A = ones((3,3))
         B = 2 * ones((3,3))
         print A
         print B

[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]
[[ 2.  2.  2.]
 [ 2.  2.  2.]
 [ 2.  2.  2.]]
```

```
In [76]: A + B

Out[76]: array([[ 3.,  3.,  3.],
                [ 3.,  3.,  3.],
                [ 3.,  3.,  3.]])

In [77]: A * B

Out[77]: array([[ 2.,  2.,  2.],
                [ 2.,  2.,  2.],
                [ 2.,  2.,  2.]])
```

Importante: multiplicação de *arrays* não equivale à multiplicação de matrizes. A multiplação de matrices, como definida em álgebra, é obtida com a função `dot`:

```
In [78]: dot(A, B)

Out[78]: array([[ 6.,  6.,  6.],
                [ 6.,  6.,  6.],
                [ 6.,  6.,  6.]])
```

Transposição:

```
In [79]: A = arange(12).reshape(3,4)
         A

Out[79]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11]])

In [80]: A.T

Out[80]: array([[ 0,  4,  8],
                [ 1,  5,  9],
                [ 2,  6, 10],
                [ 3,  7, 11]])
```

Comparações:

```
In [81]: u = random.rand(5)
         u

Out[81]: array([ 0.77096045,  0.42731738,  0.73113641,  0.9227403 ,  0.12002013])

In [82]: v = random.rand(5)
         v

Out[82]: array([ 0.05566669,  0.15759572,  0.36029857,  0.01489561,  0.69945966])

In [83]: u > v

Out[83]: array([ True,  True,  True,  True, False], dtype=bool)
```

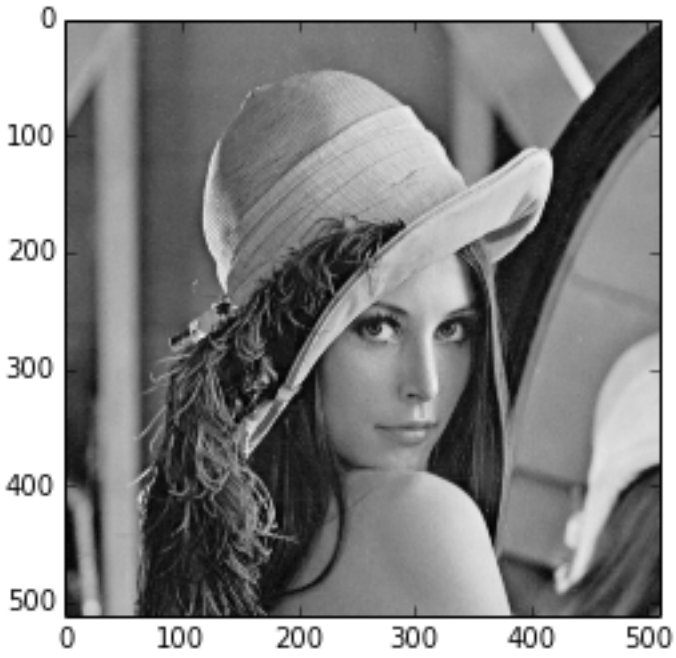
1.6.1 Álgebra linear

Operações de álgebra linear são implementadas em `numpy.linalg`. Porém, no geral, essas implementações **não são eficientes** e o módulo `scipy.linalg` deve ser preferido, por ser melhor otimizado.

1.7 Exemplo: imagens representadas como arrays

Imagens podem ser pensadas como arrays nos quais cada elemento representa uma intensidade luminosa naquela posição:

```
In [84]: from scipy.misc import lena

In [85]: I = lena()
         I

Out[85]: array([[162, 162, 162, ..., 170, 155, 128],
                [162, 162, 162, ..., 170, 155, 128],
                [162, 162, 162, ..., 170, 155, 128],
                ...,
                [ 43,  43,  50, ..., 104, 100,  98],
                [ 44,  44,  55, ..., 104, 105, 108],
                [ 44,  44,  55, ..., 104, 105, 108]])

In [86]: imshow(I, cmap=cm.gray)

Out[86]: <matplotlib.image.AxesImage at 0x7fc36c662350>
```

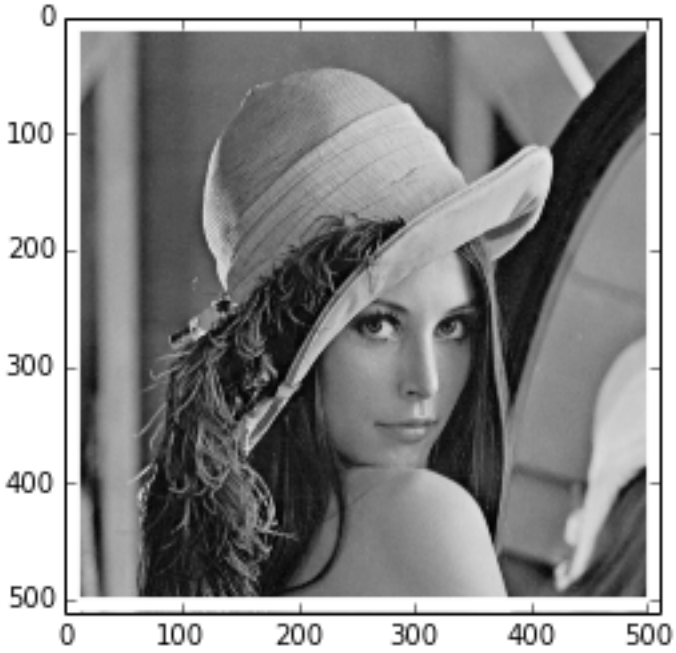


1.7.1 Exercício 5.1

As intensidades luminosas estão representadas inteiros entre 0 (preto) e 255 (branco). Utilize *slicing* para criar um **borda branca** na imagem, isto é, modificar as bordas da matriz para que assumam o valor 255 (branco).

Solução

```
In [87]: B = I.copy()
         B[0:13, :] = 255
         B[:, 0:13] = 255
         B[-13:-1, :] = 255
         B[:, -13:-1] = 255
         imshow(B, cmap=cm.gray)

Out[87]: <matplotlib.image.AxesImage at 0x7fc3615f4590>
```

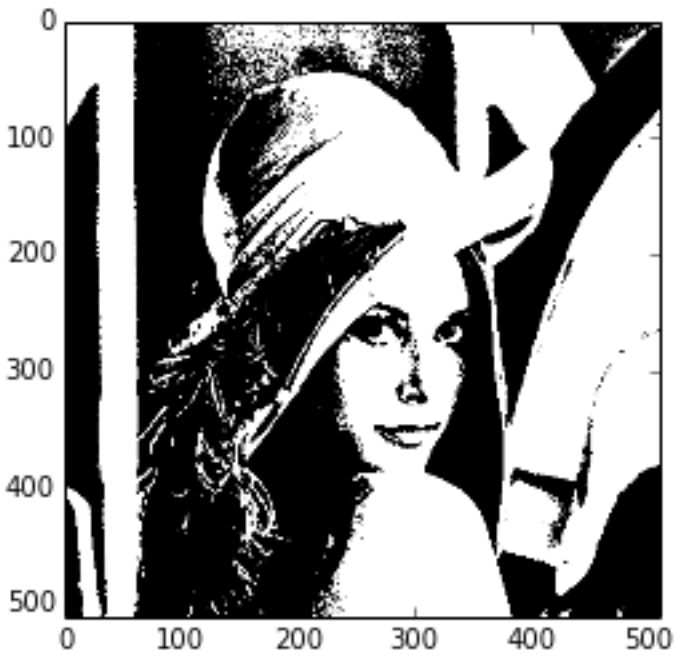


Exemplo **Limiarização (*thresholding*)** - uma operação comum em processamento de imagens, chamada de limiarização, consiste em realizar o seguinte procedimento em cada pixel:

$$I'[x,y] \leftarrow 255; \text{sse } I[x,y] > k, 0 \text{ caso contrário}$$

o valor k é chamado de **limiar** (*threshold*). Tal operação pode ser facilmente implementada utilizando-se os recursos da NumPy.

```
In [88]: R = zeros_like(I) # Cria um array de zeros com as mesmas dimensões de I
         R[I > 128] = 255
         imshow(R, cmap=cm.gray)

Out[88]: <matplotlib.image.AxesImage at 0x7fc361523750>
```



1.8 Exemplo: Cadeias de Markov e um modelo de clima muito simples

Considere um modelo bastante simplificado de predição climática. As probabilidades do dia ser *chuvoso* ou *ensolarado*, dado o clima do dia anterior, podem ser representadas a partir de uma **matriz de transição**:

```
In [89]: P = array([[0.9, 0.1],
                    [0.5, 0.5]])
```

A matriz `P` representa um modelo de clima no qual há uma chance de 90% de um dia ensolarado se seguido por outro dia de sol e uma chance de 50% de um dia chuvoso se seguir a outro dia de chuva. As *colunas* podem ser rotuladas como *ensolarado* e *chuvoso* e as linhas na mesma ordem. Assim, `P[i,j]` é a probabilidade de um dia do tipo i ser seguido por um dia do tipo j (note que as linhas somam 1).

Considere que o tempo no primeiro dia foi ensolarado. Podemos representar este dia através de um **vetor de estado** $\mathbf{x}_0$ no qual o campo *ensolarado* é 100% e o campo *chuvoso* é 0:

```
In [90]: x0 = array([1., 0])
```

A previsão do tempo para o dia seguinte pode ser dada por:

```
In [91]: x1 = x0.dot(P)
         x1

Out[91]: array([ 0.9,  0.1])
```

E no próximo:

```
In [92]: x2 = x1.dot(P)
         x2

Out[92]: array([ 0.86,  0.14])
```

E no seguinte:

```
In [93]: x3 = x2.dot(P)
         x3

Out[93]: array([ 0.844,  0.156])
```