Fundamentos de Programação

António J. R. Neves

Departamento de Electrónica, Telecomunicações e Informática

Universidade de Aveiro

an@ua.pt

http://elearning-projetos.ua.pt/

Resumo

- NumPy
- PyPlot

NumPy

- NumPy is the fundamental package for scientific computing with Python. It contains among other things:
  - a powerful N-dimensional array object
  - sophisticated functions
  - tools for integrating C/C++ and Fortran code
  - useful linear algebra, Fourier transform, and random number capabilities
- Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

Basics (1)

- NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In Numpy dimensions are called axes. The number of axes is the rank.
- The coordinates of a point in 3D space [1, 2, 1] is an array of rank 1, because it has one axis. That axis has a length of 3.
- Numpy's array class is called `ndarray`. The most important attributes are:
  - `ndarray.ndim` - the number of axes (dimensions) of the array.
  - `ndarray.shape` - the dimensions of the array. For a matrix with n rows and m columns, shape will be (n,m).
  - `ndarray.size` - the total number of elements of the array.
  - `ndarray.dtype` - type of the elements in the array.

Array creation (1)

- An array can be created from a regular Python list or tuple using the array function. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Array creation (2)

- Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content.
- The function `zeros` creates an array full of zeros, the function `ones` creates an array full of ones, and the function `empty` creates an array whose initial content is random and depends on the state of the memory. By default, the `dtype` of the created array is `float64`.

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )
>>>np.empty( (2,3) )
```

Array creation (3)

- To create sequences of numbers, NumPy provides a function `arange` (similar to `range`).

```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])
```

- The function `linspace` is used with floating point arguments and receives as an argument the limits and number of elements that we want.

```
>>> np.linspace( 0, 2, 9 )  #9 numbers from 0 to 2
array([ 0.  ,  0.25,  0.5 ,  0.75,  1.  …  2.  ])
```

- When you print an array, NumPy displays it in a similar way to nested lists.

```
>>> a = np.arange(6)  # 1d array
>>> print(a)
[0 1 2 3 4 5]
```

Basic Operations (1)

- Arithmetic operators on arrays apply elementwise. A new array is created and filled with the result.
- The product operator `*` operates elementwise in NumPy arrays. The matrix product can be performed using the `dot` function (`A.dot(B)`).
- When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting).

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 ) # array([0, 1, 2, 3])
>>> c = a-b
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
```

Basic Operations (2)

- Some operations, such as `+=` and `*=`, modify an existing array.

```
>>> a = np.ones((2,3), dtype=int)
>>> a *= 3
array([[3, 3, 3],
       [3, 3, 3]])
```

- Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the `ndarray` class.

```
>>> a = np.random.random((2,3))
>>> a.sum()
2.5718191614547998
>>> a.min()
0.1862602113776709
>>> a.max()
0.6852195003967595
```

Basic Operations (3)

- By default, the unary operations apply to the array considering it as a list of numbers, regardless of its shape.
- However, by specifying the `axis` parameter you can apply an operation along the specified axis of an array.

```
>>> b = np.arange(12).reshape(3,4)
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>> b.sum(axis=0)                  # sum of each column
array([12, 15, 18, 21])
>>> b.min(axis=1)                  # min of each row
array([0, 4, 8])
>>> b.cumsum(axis=1)     # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Universal Functions¶

- NumPy provides familiar mathematical functions such as `sin`, `cos`, and `exp`. In NumPy, these are called “universal functions”(ufunc).
- Within NumPy, these functions operate elementwise on an array, producing an array as output.

```
>>> B = np.arange(3)
array([0, 1, 2])
>>> np.exp(B)
array([ 1.        ,  2.71828183,  7.3890561 ])
>>> np.sqrt(B)
array([ 0.        ,  1.        ,  1.41421356])
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([ 2.,  0.,  6.])
```

Indexing, Slicing and Iterating (1)

- One-dimensional arrays can be indexed, sliced and iterated over, like lists and other Python sequences.

```
>>> a = np.arange(10)**3
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
>>> a[2]
8
>>> a[2:5]
array([ 8, 27, 64])
>>> a[:6:2] = -10
array([-10, 1, -10, 27, -10, 125, 216, 343, 512, 729])
>>> a[ : :-1]                                    # reversed a
array([  729, 512, 343, 216, 125, -10, 27, -10, 1, -10])
>>> for i in a:
...     print(i)
```

Indexing, Slicing and Iterating (2)

- Multidimensional arrays can have one index per axis. These indices are given in a tuple separated by commas.

```
>>> b[2,3]
23
>>> b[0:5, 1]                  # each row in the second column of b
array([ 1, 11, 21, 31, 41])
>>> b[ : ,1]                   # equivalent to the previous example
array([ 1, 11, 21, 31, 41])
>>> b[1:3, : ]
```

- When fewer indices are provided than the number of axes, the missing indices are considered complete slices.

```
>>> b[-1]                     # the last row. Equivalent to b[-1,:]
array([40, 41, 42, 43])
```

- `x[1,2,...]` is equivalent to `x[1,2,:,:,:]`
- `x[...,3]` to `x[:,:,:,:,3]`

Indexing, Slicing and Iterating (3)

- Iterating over multidimensional arrays is done with respect to the first axis.

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
...
[40 41 42 43]
```

- However, if one wants to perform an operation on each element in the array, one can use the flat attribute which is an iterator over all the elements of the array.

```
>>> for element in b.flat:
...     print(element)
0
...
43
```

Shape Manipulation (1)

- The shape of an array can be changed with various commands: `ravel`, `T`, `reshape`, `resize`.

```
>>> a = np.floor(10*np.random.random((3,4)))
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.shape
(3, 4)
>>> a.ravel() # flatten the array
array([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.,  8.,  9.,  3.,
6.])
>>> a.shape = (6, 2)
>>> a.T
array([[ 2.,  0.,  4.,  1.,  8.,  3.],
       [ 8.,  6.,  5.,  1.,  9.,  6.]])
```

Shape Manipulation (2)

- The `reshape` function returns its argument with a modified shape, whereas the `resize` method modifies the array itself.
- If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated.

```
>>> a
array([[ 2.,  8.],
       [ 0.,  6.],
          ...
       [ 3.,  6.]])
>>> a.resize((2,6))
>>> a
array([[ 2.,  8.,  0.,  6.,  4.,  5.],
       [ 1.,  1.,  8.,  9.,  3.,  6.]])
>>> a.reshape(3,-1)
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
```

Stacking and splitting

- Several arrays can be stacked together along different axes.
- The function `column_stack` stacks 1D arrays as columns into a 2D array. It is equivalent to `vstack` only for 1D arrays.
- Using `hsplit`, you can split an array along its horizontal axis, either by specifying the number of equally shaped arrays to return, or by specifying the columns after which the division should occur.

```
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
```

```
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

Copy

- When operating and manipulating arrays, their data is sometimes copied into a new array and sometimes not.
  - Simple assignments make no copy of array objects or of their data.
  - Different array objects can share the same data. The view method creates a new array object that looks at the same data.
  - The `copy` method makes a complete copy of the array and its data.

```
>>> a = np.arange(12)
>>> b = a               # no new object is created
>>> b is a              # a and b are the same ndarray object
True
>>> c = a.view()
>>> c.shape = 2,6
>>> a.shape             # a's shape doesn't change
(3, 4)
>>> c[0,4] = 1234       # a's data changes
```

Fancy indexing

- NumPy offers more indexing facilities than regular Python sequences. In addition to indexing by integers and slices, arrays can be indexed by arrays of integers and arrays of booleans.
- When we index arrays with arrays of (integer) indices we are providing the list of indices to pick.

```
>>> a = np.arange(12)**2        # the first 12 square numbers
>>> i = np.array( [ 1,1,3,8,5 ] )  # an array of indices
>>> a[i]                        # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
```

- When the indexed array `a` is multidimensional, a single array of indices refers to the first dimension of `a`.
- We can also give indexes for more than one dimension. The arrays of indices for each dimension must have the same shape.

Indexing with Boolean Arrays

- With boolean indices the approach is different; we explicitly choose which items in the array we want and which ones we don't.
- The most natural way one can think of for boolean indexing is to use boolean arrays that have the same shape as the original array.

```
>>> a = np.arange(12).reshape(3,4)
>>> b = a > 4
>>> b                    # b is a boolean with a's shape
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)
>>> a[b]                 # 1d array with the selected elements
array([ 5,  6,  7,  8,  9, 10, 11])
```

PyPlot

- Is a collection of command style functions that make `matplotlib` work like MATLAB.
- Each `pyplot` function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.
- Some states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes.

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
plt.show()
```

plot()

- If you provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you.
- `plot()` is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

- There is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB.

```
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

Lines

- Generally, you will use `numpy` arrays with pyplot. In fact, all sequences are converted to `numpy` arrays internally.

```
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)
# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

- Lines have many attributes that you can set: linewidth, dash style, antialiased, etc. There are several ways to set line properties:
  - Use keyword args:

```
plt.plot(x, y, linewidth=2.0)
```

Figures and axes

- pyplot have the concept of the current figure and the current axes. All plotting commands apply to the current axes.

```
t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)
plt.figure(1)
plt.subplot(2,1,1)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
plt.subplot(2,1,2)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
plt.show()
```

- You can create multiple figures by using multiple `figure()` calls with an increasing figure number.
- The `subplot()` command specifies `numrows`, `numcols`, `fignum` where `fignum` ranges from 1 to `numrows*numcols`.

Working with text

- The `text()` command can be used to add text in an arbitrary location, and the `xlabel()`, `ylabel()` and `title()` are used to add text in the indicated locations.

```
plt.xlabel('Smarts')
plt.ylabel('Probability')
plt.title('Histogram of IQ')
plt.text(60, .025, r'$\mu=100,\ \sigma=15$') #r is important
```

- All of the `text()` commands return an `matplotlib.text.Text` instance. Just as with lines, you can customize the properties by passing keyword arguments into the text functions or using `setp()`.

```
t = plt.xlabel('my data', fontsize=14, color='red')
```

- `matplotlib` accepts TeX equation expressions in any text expression.

Logarithmic and other nonlinear axis

- pyplot supports not only linear axis scales, but also logarithmic and logit scales. This is commonly used if data spans many orders of magnitude.
- Changing the scale of an axis is done using the `xscale` function.

```
plt.subplot(1,2,1)
plt.plot(x, y)
plt.yscale('linear')
plt.title('linear')
plt.subplot(1,2,2)
plt.plot(x, y)
plt.yscale('log')
plt.title('log')
```