Numpy Basics

Rui Leite

FEP

Nov 2022

Numpy Basics

Why Numpy is Usefull

- to perform a wide variety of mathematical operations on arrays
- has powerful data structures to Python that guarantee efficient calculations with arrays and matrices
- enormous library of high-level mathematical functions that operate on these arrays and matrices
- NumPy arrays are faster and more compact than Python lists
- uses vectorized code which is more concise and easier to read
- fewer lines of code generally means fewer bugs

The n-dimensional arrays

The core data structure (ndarray)

Numpy provides the ndarray data
Much more efficient than the list of lists way to represent arrays available in python base system.
More sofisticated indexing which helps our implementations
Elementwise application of functions (the universal functions) and operators.

Defining Arrays

- **converting a list of lists representation** M=np.array([[2,3],[1,7]])
- **by filling with some data pattern and specifying dimensions**
  - np.zeros(<shape>) *np.ones() do the same but with 1's*

    x=np.zeros((2,3)) # an array of 0's with 2 rows and 3 columns

    <shape> is an int or tuple of ints that represent the dimension
  - np.zeros_like(<otherArr>) *np.ones_like() the same with 1's*

    Creates an array with the same geometry (shape) as <otherArr> but filled with 0's

    x=np.zeros_like(b) # x would get the same shape as b
- **by filling with some 1-D data pattern and then reshaping**
  - np.arange(...).reshape(...)

    np.arange(1,7).reshape(3,2)

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

    arange([start],stop[,step],...) build a vec. specified by the args

Defining Arrays ...Properties of Arrays

- **by filling with some 1-D data pattern and then reshaping**
  - np.linspace(...).reshape(...)
    np.linspace(1,5,12).reshape(3,4)

```
array([[1.  , 1.36363636, 1.72727273, 2.09090909],
       [2.45454545, 2.81818182, 3.18181818, 3.54545455],
       [3.90909091, 4.27272727, 4.63636364, 5.  ]])
```

    linspace build a vector of evenly spaced numbers by specifying an interval and number of elements

- **by filling with random 1-D data and then reshaping**
  - np.random.<RandomDist>(...).reshape(...)
    np.random.normal(10,2,size=15).reshape(3,5)

```
array([[13.34524443, 10.19829843, 12.79599275, 9.45750402, 11.22640837],
       [ 9.46536562, 8.90138197, 10.26541659, 9.04771597, 12.61694616],
       [10.39002656, 10.80041998, 9.32473533, 12.51294453, 8.536061 ]])
```

    (see other methods like randint, uniform, choice, chisquare, ...)

Properties of Arrays

- **ndim** gives the number of axes(dimensions)
  `np.array([[[1,2]],[[2,4]],[[5,7]]]).ndim` gives 3
- **shape** gives the dimensions of the array, the size on each dimension(axis)
  `np.array([[[1,2]],[[2,4]],[[5,7]]]).shape` gives (3,1,2)
- **size** gives the number of elements
- **dtype** gives the type of of elements in the array
- **itemsize** gives the size in bytes of each element of the array

Methods of Arrays

all	any	argmax	argmin
argpartition	argsort	astype	byteswap
choose	clip	compress	conj
conjugate	copy	cumprod	cumsum
diagonal	dot	dump	dumps
fill	flatten	getfield	item
itemset	max	mean	min
newbyteorder	nonzero	partition	prod
ptp	put	ravel	repeat
reshape	resize	round	searchsorted
setfield	setflags	sort	squeeze
std	sum	swapaxes	take
tobytes	tofile	tolist	tostring
trace	transpose	var	view

Indexing Arrays

The indexing with numpy arrays is different from what we use in list of lists.
Here we can have more than one index inside the "[" "]" term.

Example: Consider that M is an array with shape (3,5) - 3 rows and 5 columns
To index element in the 2nd row and 3rd column we do M[1,2] (remember that index positions starts with 0)
while using list of lists construction we do M[1][2]

If we want we can decide to select everything in some of dimensions while for others use indexing to select specific positions say X is an array with shape (3,4,6) 3-dimensional (3 planes 4 rows and 6 columns)

X[1:2, :, 4] means position 1 of dimension 1 , all from dimension 2 and position 4 from dimension 3

Indexing Arrays cont.

As we notice on the previous slide we can use slices and so also negative positions
We can directly use list of the positions for any dimension that we want (we can have repetitions on the list of positions)

Example:
a=np.arange(24).reshape(2,4,3)
a[0,[1,2],:] # means position 0 of dimension 1, positions 1 and 2 on 2nd dim and all on the 3rd

Using list of bool to index
Consider the same **a** from the previous example
a[[True,False],:,2] # means position 0 of dim 1 (we have True on position 0), all from 2nd dim and position 2 on 3rd dimension

Consider and reflect about the following expression
a[1,:,a[1,1,:]>2]

Operations with Arrays - Concept of Broadcasting

- The operations using arrays that have the same shape (eg. x+y , with x and y arrays) is made elementwise.
  Each element of x on the position (i,j,k, ...) is operated with the element of y on the same position
  Consider x=np.array([[2,3,1],[4,1,7]]) and
  y=np.array([[6,1,4],[5,3,9]])
  x+y is array([[8, 4, 5], [9, 4, 16]])
- However if they have different shapes the operation can be Broadcasted if the following condition hold
  - from the last dimension to the first we check if either they have the same number on that dimension or one of it is = 1.
  - examples:
    Suppose arrays a and b with dimensions (2,1,3) and (2,3,4) we cannot do a+b because the operation cannot be broadcasted
    Consider now the dimensions (2,1,3) and (2,4,1) can be broadcasted and the result will have dimension (2,4,3) (the max for each dimension)

Concept of Broadcasting cont.

- Sometimes we want to operate elements of an array two each element of the same array we we don't want of course to apply elementwise
- the solution in these case is to expand the array to have more dimensions this is done using **np.newaxis**
- say you want to compute the difference of each element of 1-dimension array to every other element of the same array We need to operate over 2-dimensional arrays and to broadcast every 1 sized dimension to every value on the correspponding
  Suppose we have x=np.array([1,3,4,5])
  We can do x[np.newaxis,:] to get an array with dim (1,5)
  and x[:,np.newaxis] to get an array with dim (5,1)
  and so x[np.newaxis,:] - x[:,np.newaxis] would be an array with dim (5,5)

Operations with Arrays cont. - Universal Functions

- We have all the other common operators to apply on array expressions
  +, -, /, %, //, **, >, ...
- The only thing that we need is that the dimensions respect the condition to broadcasting

Universal Functions

- Numpy has some functions known as universal that when applyied to an array it is applyed elementwise to each element, and, therefore result in an array with the same shape
- We can consider all functions of one input and one output as universal for sure, but mainly the functions with one output (explore the functions available - list it using dir(np))

Universal Functions - Examples

Consider

`a=np.array([[1,3,4,25], [9,12,1,49]])`

- rounding the square root applyed to each element of **a**

  `np.round(np.sqrt(a), 2)`

  ```
  array([[1.  , 1.73, 2.  , 5.  ],
  [3.  , 3.46, 1.  , 2.  ]])
  ```

- computing the square of each element of **a**

  `np.square(a)`

  ```
  array([[ 1, 9, 16, 625],
  [ 81, 144, 1, 16]], dtype=int32)
  ```

- computing the of log2 of each element (round to 1 decimal)

  `np.round(np.log2(a), 1)`

  ```
  array([[0.  , 1.6, 2.  , 4.6],
  [3.2, 3.6, 0.  , 2.  ]])
  ```

Functions that Sumarizes Data (axis parameter)

- Some functions sumarizes data - transforms several values into just 1
  examples min, sum , prod, ...
- numpy have their version of these functions with something very usefull, the axis parameter
- We can using this get the min value of each column of a matrix (array with 2 dim)
  We do np.min(a,axis=0) # 0 means columns, 1 rows,...

  Considering **a** from last slide we get
  array([1, 3, 1, 4])
- **exercise** Compute the sum of each row of a 3-dimensional array (note that we are going to sum every element for each row - an entire plane for each)
- see also np.apply_over_axes()