NumPy_quick_tour

January 26, 2021

[9]:
```
# My standard magic !  You will see this in almost all my notebooks.

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

# Reload all modules imported with %aimport
%load_ext autoreload
%autoreload 1

%matplotlib inline
```

1 NumPy

VandePlas Chapter 2, Geron notebook

1.1 Python lists

Lists are *heterogeneous*: can contain elements of mixed type

[1]:
```
l = list( range(0,10) )
print(l)
```

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

[2]:
```
l[2] = "two"
print(l)
```

```
[0, 1, 'two', 3, 4, 5, 6, 7, 8, 9]
```

Heterogeneity == *slow* - Python interpreter has to constantly examine types

1.2 NumPy ndarray

[3]:
```
import numpy as np
```

NumPy n-dimensional arrays (`ndarray`) are *homogenous* - Can be faster because don't waste time examining type of each element - Can be treated as vectors - Vector arithmetic via compiled code = *fast*

[4]:
```
l = list( range(0,10))
l_plus_1 = [ e+1 for e in l]
print(l_plus_1)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

[5]:
```
l_np = np.array( np.arange(0,10))
print(l_np +1)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
```

1.2.1 Speed comparison

[6]:
```
list_len = 1000
l = list( range(0, list_len))
%timeit [ e+1 for e in l]
```

```
32.9 µs ± 132 ns per loop (mean ± std. dev. of 7 runs, 10000 loops each)
```

[7]:
```
l_np = np.array( np.arange(0, list_len) )
%timeit l_np +1
```

```
1.04 µs ± 6.71 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

When dealing with large datasets, you need NumPy

1.3 Basics of NumPy arrays

VanderPlas

Vandeplas YouTube: Losing your loops - slides

The most operation on ndarrays is indexing.

- ndarray indices are 0-based (i.e, first row/col is numbered 0, not 1)

[10]:
```
x = np.arange(0,6)

x

x[2]

M = np.arange(0,6).reshape(2,3)
M

M[1,1]
```

[10]:
```
array([0, 1, 2, 3, 4, 5])
```

[10]:
```
2
```

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

[10]: 4

1.3.1 Slicing

- Python (not just NumPy) upper bound of index is NOT inclusive

[11]:
```
print("x: ", x)
print("x tail: ", x[2:])
print("x head: ", x[:2])
```

```
x:  [0 1 2 3 4 5]
x tail:  [2 3 4 5]
x head:  [0 1]
```

1.3.2 Strides

`x[start:stop:step]`

[12]:
```
x[1:5:2]
```

[12]: array([1, 3])

1.3.3 Reshaping

[12]:
```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Add dimensions

[13]:
```
x = np.arange(0,6)
print("x: ", x)
print("x shape: ", x.shape)

print("x re-shaped: ", x.reshape(1,-1))
print("x re-shaped shape: ", x.reshape(1,-1).shape)

print("x w/newaxis: ", x[ np.newaxis,:])
print("x w/newaxis sja[e: ", x[ np.newaxis,:].shape)
```

```
x:  [0 1 2 3 4 5]
x shape:  (6,)
x re-shaped:  [[0 1 2 3 4 5]]
```

```
x re-shaped shape:  (1, 6)
x w/newaxis:  [[0 1 2 3 4 5]]
x w/newaxis sja[e:  (1, 6)
```

1.3.4 Concatentation, splitting

[14]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
x
y

np.concatenate([x, y])
```

[14]:
```
array([1, 2, 3])
```

[14]:
```
array([3, 2, 1])
```

[14]:
```
array([1, 2, 3, 3, 2, 1])
```

You can concatenate multi-dimensional ndarrays:

[15]:
```
M1 = np.array([ [1, 2, 3],
                [4, 5, 6]
              ])

M2 = np.array([ [ 7,  8,  9 ],
                [ 10, 11, 12]
              ])

M1
M2

np.concatenate([ M1, M2 ])
```

[15]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[15]:
```
array([[ 7,  8,  9],
       [10, 11, 12]])
```

[15]:
```
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])
```

You can also specify the dimension on which to concatenate

[16]:
```
M1
M2

np.concatenate([ M1, M2 ], axis=1)
```

[16]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

[16]:
```
array([[ 7,  8,  9],
       [10, 11, 12]])
```

[16]:
```
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

You can also use `vstack` (vertical stack) and `hstack` (horizontal stack)

[17]:
```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])

y = np.array( [ [100],
                [200]
              ])
x
grid

print("vstack:")
# vertically stack the arrays
np.vstack([x, grid])

print("hstack:")
y
grid
np.hstack( [y, grid])
```

[17]:
```
array([1, 2, 3])
```

[17]:
```
array([[9, 8, 7],
       [6, 5, 4]])
```

```
vstack:
```

[17]:
```
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

```
hstack:
```

[17]:
```
array([[100],
       [200]])
```

[17]:
```
array([[9, 8, 7],
       [6, 5, 4]])
```

[17]:
```
array([[100,   9,   8,   7],
       [200,   6,   5,   4]])
```

1.4 Ufuncs

Vandeplass

Math

- element-wise operations
- vectorized for speed
- operator overloading
  - +, -, *, /
  - <, ==, >
  - provides natural syntax
    - `1 + 1`
    - ‘np.add(1,1)

[18]:
```
x = np.array( np.arange(0,10))
print("x: ", x)
print("+1: ", x + 1)
print("+1 verbose: ", np.add(x,1))
print("-1: ", x -1)
```

```
x:  [0 1 2 3 4 5 6 7 8 9]
+1:  [ 1  2  3  4  5  6  7  8  9 10]
+1 verbose:  [ 1  2  3  4  5  6  7  8  9 10]
-1:  [-1  0  1  2  3  4  5  6  7  8]
```

1.4.1 Aggregates

Vanderplass

- Aggregation: taking a one-dimensional slice of an ndarray and reducing it to a scalar
  - also known as **reduce**

Best illustrated with an example

[19]:
```
x = np.arange(1, 6)
print("x: ", x)
print("x reduced by add: ",np.add.reduce(x))

# Less verbose synonym
print("x reduced by add, via sum", x.sum())
```

```
x:  [1 2 3 4 5]
x reduced by add:  15
x reduced by add, via sum 15
```

1.4.2 Aggregates on multi-dimensional ndarray: choose your dimension

[20]:
```python
x = np.arange(1,7).reshape(2,3)
print("x: ", x)

print("x reduced on first dimension: ", x.sum(axis=0))

print("x reduced on second dimension: ", x.sum(axis=1))
```

```
x:  [[1 2 3]
 [4 5 6]]
x reduced on first dimension:  [5 7 9]
x reduced on second dimension:  [ 6 15]
```

1.4.3 Cumulative

Closely related to `reduce`: `accumulate` - running operations, e.g, running sum

[21]:
```python
print("x: ", x)
print("x running sum: ", np.add.accumulate(x))  # NOTE: not a method ON x; x is␣
↪a parameter

# Less verbose synonym. n.b., WITHOUT an axis arg,, it will flatten x before␣
↪summing
print("x running sum via cumsum: ", x.cumsum(axis=0))
```

```
x:  [[1 2 3]
 [4 5 6]]
x running sum:  [[1 2 3]
 [5 7 9]]
x running sum via cumsum:  [[1 2 3]
 [5 7 9]]
```

1.5 Broadcasting

Vanderplass

You hopefully intuitively understand what NumPy does when a binary operator is applied to 2 identically-shaped arguments

[22]:
```python
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

[22]:
```
array([5, 6, 7])
```

But what happens if the two arguments have different shape ? Simplest case: one argument is dimension 0 or 1:

[23]:
```
print("a: ", a)
print("a + 1: ", a+1)
```

```
a:  [0 1 2]
a + 1:  [1 2 3]
```

Next case: what if one argument is identical to the other EXCEPT is missing a dimension:

[24]:
```
M = np.arange(1,10).reshape(3,3)

print("a shape (", a.shape, "): ", a)
print("M shape (", M.shape, "):\n", M)
print("a + M shape(", (a+M).shape, "):\n", a + M)
```

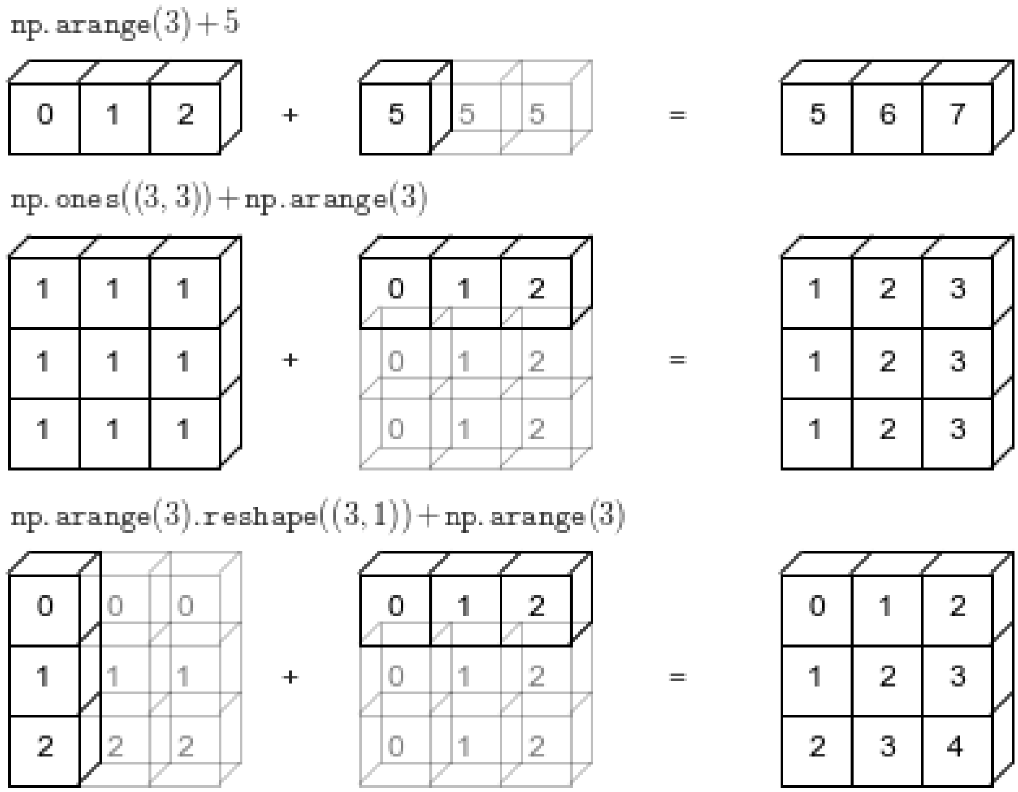
```
a shape ( (3,) ):  [0 1 2]
M shape ( (3, 3) ):
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
a + M shape( (3, 3) ):
 [[ 1  3  5]
 [ 4  6  8]
 [ 7  9 11]]
```

NumPy took a one dimensional ndarray `a` and treated it like a 2-d ndarray by repeated it's rows

This is called **broadcasting**

Broadcasting follows some simple rules (quoted from Vanderplass):

> Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.
>
> Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
>
> Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.



1.6 Boolean arrays and masks

Vanderplass

In discussing ufuncs, we stated that logical operators work on ndarrays

[25]:
```
x = np.array([1, 2, 3, 4, 5])

x < 3
```

[25]:
```
array([ True,  True, False, False, False])
```

What happens if you use a logical array to index into an ndarray ? It serves as a mask

[26]:
```
x[ x < 3 ]
```

[26]:
```
array([1, 2])
```

What happens when you apply a mask to a higher dimensional ndarray ? Notice what happens to the shape

[27]:
```
rng = np.random.RandomState(0)
x = np.arange(0,12).reshape(3,4)
print("x:\n", x)
```

```
print("x masked shape: ", x[ x < 3 ].shape)
print("x masked:\n", x[ x < 3 ])
```

```
x:
 [[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
x masked shape:  (3,)
x masked:
 [0 1 2]
```

The shape of the result is the shape of the indexing array.

1.7 Fancy indexing

1.7.1 Fancy indexing

[28]:
```
print("Done")
```

```
Done
```