Relevant Python modules: Pandas

AM

Motivations

Python does not cover the data structures normally used in science and technology work.

Numpy comes in to support data manipulation of n-dimensional arrays.

Extensive library of functions to reshape data.

Comprehensive collection of mathematical operations.

```
pip install numpy
```

default with Anaconda

Arrays

A computer version of vectors and matrices: sequence of uniform-type values with indexing mechanism by integers.

Numpy arrays have methods, applied element-wise, and functions that take into account the position of each element in the array.

```python
import numpy as np
```

```python
# nr from 2 to 20 (excl.) with step 2

b = np.arange(2, 20, 2)

b
```

```
array([ 2,  4,  6,  8, 10, 12, 14, 16, 18])
```

```
# element-wise operations

2*b
```

```
array([ 4,  8, 12, 16, 20, 24, 28, 32, 36])
```

```
# cumulative step-by-step sum
b.cumsum()
```

```
array([ 2,  6, 12, 20, 30, 42, 56, 72, 90])
```

Lists vs. Arrays

Same indexing notation:

```
mylist[0]

mylistoflists[0][1]
```

A list is a generic sequence of heterogenous objects.

So, strings, numbers, characters, file name, URLs can be all mixed up!

An array is a sequence of strictly-homogenous objects, normally `int` or `float`

```
myarray[1]

mymatrix[1][3]
```

Notation

1-dimension: an array (a line of numbers): `[1, 23, …]`

2-dimensions: a matrix (a table of numbers) `[ [1, 23, …], [14, 96, …], ...]`

3-dimensions: a tensor (a box/cube/cuboid) of numbers: `[ [ [1, 23, …], [14, 96, …], …], ...]`

2-D Numpy Arrays

```
c = np.arange(8)

c
```

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

```
# build a 2-dimensional array from a 1-d one
d = np.array([c, c*2])

d
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 0,  2,  4,  6,  8, 10, 12, 14]])
```

```
# count elements

d.size
```

```
16
```

```
#  size along each dimension

d.shape
```

```
(2, 8)
```

Axes

Numpy arrays can have multiple dimensions.

Unlike Pandas, not specifying the axis will apply a function to the entire array.

```
# operations along columns
d
```

```
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 0,  2,  4,  6,  8, 10, 12, 14]])
```

```
# operations along columns
d.sum(axis=0)
```

```
array([ 0,  3,  6,  9, 12, 15, 18, 21])
```

```
# summing by row
d.sum(axis=1)
```

```
array([28, 56])
```

```
# sum the whole content
d.sum()
```

```
84
```

Shapes

Using information about the shape we can create/manipulate (or reshape, or transpose) Numpy variables.

```
# Create 2x3 Numpy array and initialise it to 0s
e = np.zeros((2, 3), dtype = 'i')

e
```

```
array([[0, 0, 0],
       [0, 0, 0]], dtype=int32)
```

```
# Change the shape
e.reshape(3, 2)
```

```
array([[0, 0],
       [0, 0],
       [0, 0]], dtype=int32)
```

```
# Take another array to infer shape
f = np.ones_like(e, dtype = 'i')

f
```

```
array([[1, 1, 1],
       [1, 1, 1]], dtype=int32)
```

```
# Transposition

f.T
```

```
array([[1, 1],
       [1, 1],
       [1, 1]], dtype=int32)
```

Stacking

2-D arrays with the same dimensions can be merged

```
# Create an identity matrix of order 5
i = np.eye(5)

i
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

```
# stacking combines two 2-d arrays: vertically
np.vstack((i, i))
```

```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.],
       [1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])
```

```
# stacking combines two 2-d arrays: horizontally
np.hstack((i, i))
```

```
array([[1., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1., 0., 0., 0., 0., 1.]])
```

Detour: N-dimensional arrays

Numpy can handle multiple dimensions.

This is useful when dealing with multivariate data, from time series to documents.

```python
# N-dimensional array

g = np.zeros((2, 3, 4))

g
```

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

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

Two samples, each with three rows and four columns.

Slicing by Boolean filters

Data can be selected according to specific conditions.

The Boolean filter itself can be represented by a Numpy array

```python
l = np.array([np.arange(9)])

l.reshape((3, 3))

l
```

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

```python
# Let's apply a high-pass filter

l[l>4]
```

```
array([5, 6, 7, 8])
```

```
# Generate a Boolean array (False=0, True=1)

(l>4).astype(int)
```

```
array([[0, 0, 0, 0, 0, 1, 1, 1, 1]])
```

From Numpy to Pandas: `where()`

Even though Pandas is built on Numpy, `where()` has a distinct semantics

Numpy allows specifying the respective action associated to `True` and `False`

```
l = np.array([np.arange(9)])

l = l.reshape((3, 3))

l
```

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

```
#  If True then make it double, else halve it

np.where(l<5, l*2, l/2)
```

```
array([[0. , 2. , 4. ],
       [6. , 8. , 2.5],
       [3. , 3.5, 4. ]])
```

In Pandas, when False we assign `n/a`

Numpy func. to Pandas objects

```python
import pandas as pd

# l is a Numpy matrix which readily interoperates with Pandas
my_df = pd.DataFrame(l, columns=['A', 'B', 'C'])

my_df
```

	A	B	C
0	0	1	2
1	3	4	5
2	6	7	8

```python
# Extract the square root of each el. of column B (NB: my_df remains unchanged)
np.sqrt(my_df.B)
```

```
0    1.000000
1    2.000000
2    2.645751
Name: B, dtype: float64
```

Back and Forth b/w Pandas and Numpy

```python
# Extract the values back into a Numpy object

m = my_df.values

m
```

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