01_python_tools_ds

December 9, 2020

1 Python tools for data science

(last updated 12-05-2020)

1.1 The PyData Stack

The Python Data Stack comprises a set of packages that makes Python a powerful data science language. These include

- Numpy: provides arrays and matrix algebra
- Scipy: provides scientific computing capabilities
- matplotlib: provides graphing capabilities

These were the original stack that was meant to replace Matlab. However, these were meant to tackle purely numerical data, and the kinds of heterogeneous data we regularly face needed more tools. These were added more recently.

- Pandas: provides data analytic structures like the data frame, as well as basic descriptive statistical capabilities
- statsmodels: provides a fairly comprehensive set of statistical functions
- scikit-learn: provides machine learning capabilities

This is the basic stack of packages we will be using in this workshop. Additionally we will use a few packages that add some functionality to the data science process. These include

- seaborn: Better statistical graphs
- plotly: Interactive graphics
- biopython: Python for bioinformatics

We may also introduce the package `rpy2` which allows one to run R from within Python. This can be useful since many bioinformatic pipelines are already implemented in R.

> The PyData stack also includes `sympy`, a symbolic mathematics package emulating Maple

1.2 Numpy (numerical and scientific computing)

We start by importing the Numpy package into Python using the alias `np`.

[1]:
```
import numpy as np
```

Numpy provides both arrays (vectors, matrices, higher dimensional arrays) and vectorized functions which are very fast. Let's see how this works.

[2]:
```
z = [1,2,3,4,5,6,7,8,9.3,10.6] # This is a list
z_array = np.array(z)
z_array
```

[2]:
```
array([ 1. ,  2. ,  3. ,  4. ,  5. ,  6. ,  7. ,  8. ,  9.3, 10.6])
```

Now, we have already seen functions in Python earlier. In Numpy, there are functions that are optimized for arrays, that can be accessed directly from the array objects. This is an example of *object-oriented programming* in Python, where functions are provided for particular *classes* of objects, and which can be directly accessed from the objects. We will use several such functions over the course of this workshop, but we won't actually talk about how to do this program development here.

> Numpy functions are often very fast, and are *vectorized*, i.e., they are written to work on vectors of numbers rather than single numbers. This is an advantage in data science since we often want to do the same operation to all elements of a column of data, which is essentially a vector

We apply the functions `sum`, `min` (minimum value) and `max` (maximum value) to `z_array`.

[3]:
```
z_array.sum()
```

[3]:
```
55.9
```

[4]:
```
z_array.min()
```

[4]:
```
1.0
```

[5]:
```
z_array.max()
```

[5]:
```
10.6
```

The versions of these functions in Numpy are optimized for arrays and are quite a bit faster than the corresponding functions available in base Python. When doing data work, these are the preferred functions.

These functions can also be used in the usual function manner:

[6]:
```
np.max(z_array)
```

[6]:
```
10.6
```

Calling `np.max` ensures that we are using the `max` function from numpy, and not the one in base Python.

1.2.1 Numpy data types

Numpy arrays are homogeneous in type.

[7]:
```
np.array(['a','b','c'])
```

[7]:
```
array(['a', 'b', 'c'], dtype='<U1')
```

[8]:
```
np.array([1,2,3,6,8,29])
```

[8]:
```
array([ 1,  2,  3,  6,  8, 29])
```

But, what if we provide a heterogeneous list?

[9]:
```
y = [1,3,'a']
np.array(y)
```

[9]:
```
array(['1', '3', 'a'], dtype='<U21')
```

So what's going on here? Upon conversion from a heterogeneous list, numpy converted the numbers into strings. This is necessary since, by definition, numpy arrays can hold data of a single type. When one of the elements is a string, numpy casts all the other entities into strings as well. Think about what would happen if the opposite rule was used. The string 'a' doesn't have a corresponding number, while both numbers 1 and 3 have corresponding string representations, so going from string to numeric would create all sorts of problems.

> The advantage of numpy arrays is that the data is stored in a contiguous section of memory, and you can be very efficient with homogeneous arrays in terms of manipulating them, applying functions, etc. However, `numpy` does provide a "catch-all" `dtype` called `object`, which can be any Python object. This `dtype` essentially is an array of pointers to actual data stored in different parts of the memory. You can get to the actual objects by extracting them. So one could do

[10]:
```
np.array([1,3,'a'], dtype='object')
```

[10]:
```
array([1, 3, 'a'], dtype=object)
```

> which would basically be a valid `numpy` array, but would go back to the actual objects when used, much like a list. We can see this later if we want to transform a heterogeneous `pandas DataFrame` into a `numpy` array. It's not particularly useful as is, but it prevents errors from popping up during transformations from `pandas` to `numpy`.

1.2.2 Generating data in numpy

We had seen earlier how we could generate a sequence of numbers in a list using `range`. In numpy, you can generate a sequence of numbers in an array using `arange` (which actually creates the array rather than provide an iterator like `range`).

[11]:
```
np.arange(10)
```

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

You can also generate regularly spaced sequences of numbers between particular values

[12]:
```
np.linspace(start=0, stop=1, num=11) # or np.linspace(0, 1, 11)
```

[12]:
```
array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

You can also do this with real numbers rather than integers.

[13]:
```
np.linspace(start = 0, stop = 2*np.pi, num = 10)
```

[13]:
```
array([0.        , 0.6981317 , 1.3962634 , 2.0943951 , 2.7925268 ,
       3.4906585 , 4.1887902 , 4.88692191, 5.58505361, 6.28318531])
```

More generally, you can transform lists into `numpy` arrays. We saw this above for vectors. For matrices, you can provide a list of lists. Note the double `[` in front and back.

[14]:
```
np.array([[1,3,5,6],[4,3,9,7]])
```

[14]:
```
array([[1, 3, 5, 6],
       [4, 3, 9, 7]])
```

You can generate an array of 0's

[15]:
```
np.zeros(10)
```

[15]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

This can easily be extended to a two-dimensional array (a matrix), by specifying the dimension of the matrix as a tuple.

[16]:
```
np.zeros((10,10))
```

[16]:
```
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., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0., 0., 0., 0.]])
```

You can also generate a matrix of 1s in a similar manner.

[17]:
```
np.ones((3,4))
```

[17]:
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In matrix algebra, the identity matrix is important. It is a square matrix with 1's on the diagonal and 0's everywhere else.

[18]:
```
np.eye(4)
```

[18]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

You can also create numpy vectors directly from lists, as long as lists are made up of atomic elements of the same type. This means a list of numbers or a list of strings. The elements can't be more composite structures, generally. One exception is a list of lists, where all the lists contain the same type of atomic data, which, as we will see, can be used to create a matrix or 2-dimensional array.

[19]:
```
a = [1,2,3,4,5,6,7,8]
b = ['a','b','c','d','3']

np.array(a)
```

[19]:
```
array([1, 2, 3, 4, 5, 6, 7, 8])
```

[20]:
```
np.array(b)
```

[20]:
```
array(['a', 'b', 'c', 'd', '3'], dtype='<U1')
```

Random numbers Generating random numbers is quite useful in many areas of data science. All computers don't produce truly random numbers but generate *pseudo-random* sequences. These are completely deterministic sequences defined algorithmically that emulate the properties of random numbers. Since these are deterministic, we can set a *seed* or starting value for the sequence, so that we can exactly reproduce this sequence to help debug our code. To actually see how things behave in simulations we will often run several sequences of random numbers starting at different seed values.

The seed is set by the `RandomState` function within the `random` submodule of numpy. Note that all Python names are case-sensitive.

[21]:
```
rng = np.random.RandomState(35) # set seed
rng.randint(0, 10, (3,4))
```

[21]:
```
array([[9, 7, 1, 0],
       [9, 8, 8, 8],
       [9, 7, 7, 8]])
```

We have created a 3x4 matrix of random integers between 0 and 10 (in line with slicing rules, this includes 0 but not 10).

We can also create a random sample of numbers between 0 and 1.

[22]:
```
rng.random_sample((5,2))
```

[22]:
```
array([[0.04580216, 0.91259827],
       [0.21381599, 0.3036373 ],
       [0.98906362, 0.1858815 ],
       [0.98872484, 0.75008423],
       [0.22238605, 0.14790391]])
```

We'll see later how to generate random numbers from particular probability distributions.

1.2.3 Vectors and matrices

Numpy generates arrays, which can be of arbitrary dimension. However the most useful are vectors (1-d arrays) and matrices (2-d arrays).

In these examples, we will generate samples from the Normal (Gaussian) distribution, with mean 0 and variance 1.

[23]:
```
A = rng.normal(0,1,(4,5))
```

We can compute some characteristics of this matrix's dimensions. The number of rows and columns are given by `shape`.

[24]:
```
A.shape
```

[24]:
```
(4, 5)
```

The total number of elements are given by `size`.

[25]:
```
A.size
```

[25]:
```
20
```

If we want to create a matrix of 0's with the same dimensions as `A`, we don't actually have to compute its dimensions. We can use the `zeros_like` function to figure it out.

[26]:
```
np.zeros_like(A)
```

[26]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

We can also create vectors by only providing the number of rows to the random sampling function. The number of columns will be assumed to be 1.

[27]:
```
B = rng.normal(0, 1, (4,))
B
```

[27]:
```
array([-0.45495378,  1.04307172,  0.70451207, -0.6171649 ])
```

Extracting elements from arrays The syntax for extracting elements from arrays is almost exactly the same as for lists, with the same rules for slices.

Exercise: State what elements of B are extracted by each of the following statements

```
B[:3]
B[:-1]
B[[0,2,4]]
B[[0,2,5]]
```

For matrices, we have two dimensions, so you can slice by rows, or columns or both.

[28]:
```
A
```

[28]:
```
array([[-2.45677354,  0.36686697, -0.20453263, -0.54380446,  0.09524207],
       [ 1.06236144,  1.03937554,  0.01247733, -0.35427727, -1.18997812],
       [ 0.95554288,  0.30781478,  0.7328766 , -1.28670314, -1.03870027],
       [-0.81398211, -1.02506031,  0.12407205,  1.21491023, -1.44645123]])
```

We can extract the first column by specifying `:` (meaning everything) for the rows, and the index for the column (reminder, Python starts counting at 0)

[29]:
```
A[:,0]
```

[29]:
```
array([-2.45677354,  1.06236144,  0.95554288, -0.81398211])
```

Similarly the 4th row can be extracted by putting the row index, and `:` for the column index.

[30]:
```
A[3,:]
```

[30]:
```
array([-0.81398211, -1.02506031,  0.12407205,  1.21491023, -1.44645123])
```

All slicing operations work for rows and columns

[31]:
```
A[:2,:2]
```

[31]:
```
array([[-2.45677354,  0.36686697],
       [ 1.06236144,  1.03937554]])
```

Array operations We can do a variety of vector and matrix operations in `numpy`.

First, all usual arithmetic operations work on arrays, like adding or multiplying an array with a scalar.

[32]:
```
A = rng.randn(3,5)
A
```

[32]:
```
array([[-0.15367796,  2.50215522,  0.19420725,  0.54928294, -1.1737166 ],
       [ 1.11456557,  0.07447758,  1.58518354,  1.61986225, -0.24616333],
       [-0.02682273,  0.2196577 ,  0.41680351, -0.86319929,  0.50355595]])
```

[33]:
```
A + 10
```

[33]:
```
array([[ 9.84632204, 12.50215522, 10.19420725, 10.54928294,  8.8262834 ],
       [11.11456557, 10.07447758, 11.58518354, 11.61986225,  9.75383667],
       [ 9.97317727, 10.2196577 , 10.41680351,  9.13680071, 10.50355595]])
```

We can also add and multiply arrays **element-wise** as long as they are the same shape.

[34]:
```
B = rng.randint(0,10, (3,5))
B
```

[34]:
```
array([[6, 2, 3, 9, 8],
       [5, 9, 3, 9, 7],
       [0, 4, 2, 5, 0]])
```

[35]:
```
A + B
```

[35]:
```
array([[ 5.84632204,  4.50215522,  3.19420725,  9.54928294,  6.8262834 ],
       [ 6.11456557,  9.07447758,  4.58518354, 10.61986225,  6.75383667],
       [-0.02682273,  4.2196577 ,  2.41680351,  4.13680071,  0.50355595]])
```

[36]:
```
A * B
```

[36]:
```
array([[-0.92206775,  5.00431043,  0.58262175,  4.94354649, -9.38973278],
       [ 5.57282784,  0.67029821,  4.75555061, 14.57876027, -1.72314331],
       [-0.        ,  0.8786308 ,  0.83360701, -4.31599644,  0.        ]])
```

You can also do **matrix multiplication**. Recall what this is.

If you have a matrix A_{mxn} and another matrix B_{nxp}, as long as the number of columns of A and rows of B are the same, you can multiply them ($C_{mxp} = A_{mxn}B_{nxp}$), with the (i,j)-th element of C being

$$c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}, i = 1, \ldots, m; j = 1, \ldots, p$$

In `numpy` the operant for matrix multiplication is `@`.

In the above examples, `A` and `B` cannot be multiplied since they have incompatible dimensions. However, we can take the *transpose* of `B`, i.e. flip the rows and columns, to make it compatible with `A` for matrix multiplication.

[37]:
```
A @ np.transpose(B)
```

[37]:
```
array([[ 0.21867814, 19.0611592 , 13.14345008],
       [24.20135281, 23.85429363, 11.56758865],
       [-2.21155643, -1.15068575, -2.60375863]])
```

[38]:
```
np.transpose(A) @ B
```

[38]:
```
array([[  4.65076009,   9.61644327,   2.82901737,   8.51387483,
          6.57253531],
       [ 15.38531919,   6.55323945,   8.16921379,  24.28798365,
         20.53858478],
       [  9.09116118,  16.32228035,   6.17177937,  18.0985346 ,
         12.64994275],
       [ 11.39500892,  12.22452901,   4.78103701,  15.20631032,
         15.73329932],
       [ -8.27311623,  -2.54867934,  -3.25252787, -10.26113957,
        -11.11287609]])
```

More generally, you can *reshape* a `numpy` array into a new shape, provided it is compatible with the number of elements in the original array.

[39]:
```
D = rng.randint(0,5, (4,4))
D
```

[39]:
```
array([[0, 2, 0, 0],
       [4, 0, 0, 4],
       [0, 3, 2, 0],
       [3, 0, 0, 3]])
```

[40]:
```
D.reshape(8,2)
```

[40]:
```
array([[0, 2],
       [0, 0],
       [4, 0],
       [0, 4],
       [0, 3],
       [2, 0],
       [3, 0],
       [0, 3]])
```

[41]:
```
D.reshape(1,16)
```

[41]:
```
array([[0, 2, 0, 0, 4, 0, 0, 4, 0, 3, 2, 0, 3, 0, 0, 3]])
```

This can also be used to cast a vector into a matrix.

[42]:
```
e = np.arange(20)
E = e.reshape(5,4)
E
```

[42]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
```

```
       [12, 13, 14, 15],
       [16, 17, 18, 19]])
```

One thing to note in all the reshaping operations above is that the new array takes elements of the old array **by row**. See the examples above to convince yourself of that.

Statistical operations on arrays You can sum all the elements of a matrix using `sum`. You can also sum along rows or along columns by adding an argument to the `sum` function.

[43]:
```
A = rng.normal(0, 1, (4,2))
A
```

[43]:
```
array([[-1.60798054, -0.05162306],
       [-0.49218049, -0.1262316 ],
       [ 0.56927597,  0.05438786],
       [ 0.33120322, -0.81820729]])
```

[44]:
```
A.sum()
```

[44]:
```
-2.141355938226197
```

You can sum along rows (i.e., down columns) with the option `axis = 0`

[45]:
```
A.sum(axis=0)
```

[45]:
```
array([-1.19968185, -0.94167409])
```

You can sum along columns (i.e., across rows) with `axis = 1`.

[46]:
```
A.sum(axis=1)
```

[46]:
```
array([-1.6596036 , -0.61841209,  0.62366383, -0.48700407])
```

Of course, you can use the usual function calls: `np.sum(A, axis = 1)`

We can also find the minimum and maximum values.

[47]:
```
A.min(axis = 0)
```

[47]:
```
array([-1.60798054, -0.81820729])
```

[48]:
```
A.max(axis = 0)
```

[48]:
```
array([0.56927597, 0.05438786])
```

We can also find the **position** where the minimum and maximum values occur.

[49]:
```
A.argmin(axis=0)
```

[49]:
```
array([0, 3])
```

[50]:
```
A.argmax(axis=0)
```

[50]:
```
array([2, 2])
```

We can sort arrays and also find the indices which will result in the sorted array. I'll demonstrate this for a vector, where it is more relevant

[51]:
```
a = rng.randint(0,10, 8)
a
```

[51]:
```
array([9, 2, 6, 6, 4, 4, 3, 4])
```

[52]:
```
np.sort(a)
```

[52]:
```
array([2, 3, 4, 4, 4, 6, 6, 9])
```

[53]:
```
np.argsort(a)
```

[53]:
```
array([1, 6, 4, 5, 7, 2, 3, 0])
```

[54]:
```
a[np.argsort(a)]
```

[54]:
```
array([2, 3, 4, 4, 4, 6, 6, 9])
```

`np.argsort` can also help you find the 2nd smallest or 3rd largest value in an array, too.

[55]:
```
ind_2nd_smallest = np.argsort(a)[1]
a[ind_2nd_smallest]
```

[55]:
```
3
```

[56]:
```
ind_3rd_largest = np.argsort(a)[-3]
a[ind_3rd_largest]
```

[56]:
```
6
```

You can also sort strings in this way.

[57]:
```
m = np.array(['Aram','Raymond','Elizabeth','Donald','Harold'])
np.sort(m)
```

[57]:
```
array(['Aram', 'Donald', 'Elizabeth', 'Harold', 'Raymond'], dtype='<U9')
```

If you want to sort arrays **in place**, you can use the `sort` function in a different way.

[58]:
```
m.sort()
m
```

[58]:
```
array(['Aram', 'Donald', 'Elizabeth', 'Harold', 'Raymond'], dtype='<U9')
```

Putting arrays together We can put arrays together by row or column, provided the corresponding axes have compatible lengths.

[59]:
```
A = rng.randint(0,5, (3,5))
B = rng.randint(0,5, (3,5))
print('A = ', A)
print('B = ', B)
```

```
A =  [[3 4 2 1 3]
 [0 3 1 1 1]
 [4 0 2 0 4]]
B =  [[1 4 2 1 3]
 [2 0 3 2 0]
 [4 0 2 3 3]]
```

[60]:
```
np.hstack((A,B))
```

[60]:
```
array([[3, 4, 2, 1, 3, 1, 4, 2, 1, 3],
       [0, 3, 1, 1, 1, 2, 0, 3, 2, 0],
       [4, 0, 2, 0, 4, 4, 0, 2, 3, 3]])
```

[61]:
```
np.vstack((A,B))
```

[61]:
```
array([[3, 4, 2, 1, 3],
       [0, 3, 1, 1, 1],
       [4, 0, 2, 0, 4],
       [1, 4, 2, 1, 3],
       [2, 0, 3, 2, 0],
       [4, 0, 2, 3, 3]])
```

Note that both `hstack` and `vstack` take a **tuple** of arrays as input.

Logical/Boolean operations You can query a matrix to see which elements meet some criterion. In this example, we'll see which elements are negative.

[62]:
```
A < 0
```

[62]:
```
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [False, False, False, False, False]])
```

This is called **masking**, and is useful in many contexts.

We can extract all the negative elements of A using

[63]:
```
A[A<0]
```

[63]:
```
array([], dtype=int64)
```

This forms a 1-d array. You can also count the number of elements that meet the criterion

[64]:
```
np.sum(A<0)
```

[64]:
```
0
```

Since the entity `A<0` is a matrix as well, we can do row-wise and column-wise operations as well.

1.2.4 Beware of copies

One has to be a bit careful with copying objects in Python. By default, if you just assign one object to a new name, it does a *shallow copy*, which means that both names point to the same memory. So if you change something in the original, it also changes in the new copy.

[65]:
```
A[0,:]
```

[65]:
```
array([3, 4, 2, 1, 3])
```

[66]:
```
A1 = A
A1[0,0] = 4
A[0,0]
```

[66]:
```
4
```

To actually create a copy that is not linked back to the original, you have to make a *deep copy*, which creates a new space in memory and a new pointer, and copies the original object to the new memory location

[67]:
```
A1 = A.copy()
A1[0,0] = 6
A[0,0]
```

[67]:
```
4
```

You can also replace sub-matrices of a matrix with new data, provided that the dimensions are compatible. (Make sure that the sub-matrix we are replacing below truly has 2 rows and 2 columns, which is what `np.eye(2)` will produce)

[68]:
```
A[:2,:2] = np.eye(2)
A
```

[68]:
```
array([[1, 0, 2, 1, 3],
       [0, 1, 1, 1, 1],
       [4, 0, 2, 0, 4]])
```

Reducing matrix dimensions Sometimes the output of some operation ends up being a matrix of one column or one row. We can reduce it to become a vector. There are two functions that can do that, `flatten` and `ravel`.

[69]:
```
A = rng.randint(0,5, (5,1))
A
```

[69]:
```
array([[2],
       [1],
       [4],
       [2],
       [4]])
```

[70]:
```
A.flatten()
```

[70]:
```
array([2, 1, 4, 2, 4])
```

[71]:
```
A.ravel()
```

[71]:
```
array([2, 1, 4, 2, 4])
```

So why two functions? I'm not sure, but they do different things behind the scenes. `flatten` creates a **copy**, i.e. a new array disconnected from `A`. `ravel` creates a **view**, so a representation of the original array. If you then changed a value after a `ravel` operation, you would also change it in the original array; if you did this after a `flatten` operation, you would not.

1.2.5 Broadcasting in Python

Python deals with arrays in an interesting way, in terms of matching up dimensions of arrays for arithmetic operations. There are 3 rules:

1. If two arrays differ in the number of dimensions, the shape of the smaller array is padded with 1s on its *left* side
2. If the shape doesn't match in any dimension, the array with shape = 1 in that dimension is stretched to match the others' shape
3. If in any dimension the sizes disagree and none of the sizes are 1, then an error is generated

[72]:
```
A = rng.normal(0,1,(4,5))
B = rng.normal(0,1,5)
```

[73]:
```
A.shape
```

[73]:
```
(4, 5)
```

[74]:
```
B.shape
```

[74]:
```
(5,)
```

[75]:
```
A - B
```

[75]:
```
array([[ 0.25410957,  1.89009891, -0.87300221, -0.17852271, -0.64735645],
       [ 0.72991872,  4.58821268, -0.03379553, -1.67352398,  1.43345867],
       [-1.51421293,  1.69993003,  1.81140727, -1.71622014,  0.52276992],
       [-1.30611819,  3.29767231,  0.91060221,  0.29490453,  1.24919619]])
```

B is 1-d, A is 2-d, so B's shape is made into (1,5) (added to the left). Then it is repeated into 4 rows to make it's shape (4,5), then the operation is performed. This means that we subtract the first element of B from the first column of A, the second element of B from the second column of A, and so on.

You can be explicit about adding dimensions for broadcasting by using `np.newaxis`.

[76]:
```
B[np.newaxis,:].shape
```

[76]:
```
(1, 5)
```

[77]:
```
B[:,np.newaxis].shape
```

[77]:
```
(5, 1)
```

An example (optional, intermediate/advanced)) This can be very useful, since these operations are faster than for loops. For example:

[78]:
```
d = rng.random_sample((10,2))
d
```

[78]:
```
array([[0.9497432 , 0.22672332],
       [0.96319737, 0.61011348],
       [0.2542308 , 0.60550727],
       [0.64054935, 0.85273037],
       [0.19747218, 0.45957414],
       [0.41571736, 0.9902779 ],
       [0.33720945, 0.30637872],
       [0.15139082, 0.30126537],
       [0.72158605, 0.3560211 ],
       [0.86288412, 0.66608767]])
```

We want to find the Euclidean distance (the sum of squared differences) between the points defined by the rows. This should result in a 10x10 distance matrix

[79]:
```
d.shape
```

[79]:
```
(10, 2)
```

[80]:
```
d[np.newaxis,:,:]
```

[80]:
```
array([[[0.9497432 , 0.22672332],
        [0.96319737, 0.61011348],
```

```
       [0.2542308 , 0.60550727],
       [0.64054935, 0.85273037],
       [0.19747218, 0.45957414],
       [0.41571736, 0.9902779 ],
       [0.33720945, 0.30637872],
       [0.15139082, 0.30126537],
       [0.72158605, 0.3560211 ],
       [0.86288412, 0.66608767]]])
```

creates a 3-d array with the first dimension being of length 1

[81]:
```
d[np.newaxis,:,:].shape
```

[81]:
```
(1, 10, 2)
```

[82]:
```
d[:, np.newaxis,:]
```

[82]:
```
array([[[0.9497432 , 0.22672332]],

       [[0.96319737, 0.61011348]],

       [[0.2542308 , 0.60550727]],

       [[0.64054935, 0.85273037]],

       [[0.19747218, 0.45957414]],

       [[0.41571736, 0.9902779 ]],

       [[0.33720945, 0.30637872]],

       [[0.15139082, 0.30126537]],

       [[0.72158605, 0.3560211 ]],

       [[0.86288412, 0.66608767]]])
```

creates a 3-d array with the 2nd dimension being of length 1

[83]:
```
d[:,np.newaxis,:].shape
```

[83]:
```
(10, 1, 2)
```

Now for the trick, using broadcasting of arrays. These two arrays are incompatible without broadcasting, but with broadcasting, the right things get repeated to make things compatible

[84]:
```
dist_sq = np.sum((d[:,np.newaxis,:] - d[np.newaxis,:,:]) ** 2)
```

[85]:
```
dist_sq.shape
```

[85]:
```
()
```

[86]:
```
dist_sq
```

[86]:
```
29.512804540441067
```

Whoops! we wanted a 10x10 matrix, not a scalar.

[87]:
```
(d[:,np.newaxis,:] - d[np.newaxis,:,:]).shape
```

[87]:
```
(10, 10, 2)
```

What we really want is the 10x10 distance matrix.

[88]:
```
dist_sq = np.sum((d[:,np.newaxis,:] - d[np.newaxis,:,:]) ** 2, axis=2)
```

You can verify what is happening by creating `D = d[:,np.newaxis,:]-d[np.newaxis,:,:]` and then looking at `D[:,:,0]` and `D[:,:,1]`. These are the difference between each combination in the first and second columns of d, respectively. Squaring and summing along the 3rd axis then gives the sum of squared differences.

[89]:
```
dist_sq
```

[89]:
```
array([[0.        , 0.14716903, 0.62721478, 0.48748566, 0.6201312 ,
        0.8681992 , 0.38154258, 0.64292305, 0.0687736 , 0.20058553],
       [0.14716903, 0.        , 0.5026548 , 0.16296469, 0.60899716,
        0.44425934, 0.48411568, 0.75441703, 0.12293897, 0.01319586],
       [0.62721478, 0.5026548 , 0.        , 0.21036128, 0.02451802,
        0.17412634, 0.09636334, 0.1031392 , 0.28066428, 0.37412884],
       [0.48748566, 0.16296469, 0.21036128, 0.        , 0.35088921,
        0.06946875, 0.39051522, 0.54338972, 0.25328705, 0.08426824],
       [0.6201312 , 0.60899716, 0.02451802, 0.35088921, 0.        ,
        0.32927744, 0.04299534, 0.02718516, 0.28541859, 0.48542089],
       [0.8681992 , 0.44425934, 0.17412634, 0.06946875, 0.32927744,
        0.        , 0.47388157, 0.54460678, 0.49583735, 0.30505741],
       [0.38154258, 0.48411568, 0.09636334, 0.39051522, 0.04299534,
        0.47388157, 0.        , 0.03455471, 0.15020974, 0.40572438],
       [0.64292305, 0.75441703, 0.1031392 , 0.54338972, 0.02718516,
        0.54460678, 0.03455471, 0.        , 0.3281208 , 0.63931803],
       [0.0687736 , 0.12293897, 0.28066428, 0.25328705, 0.28541859,
        0.49583735, 0.15020974, 0.3281208 , 0.        , 0.11610642],
       [0.20058553, 0.01319586, 0.37412884, 0.08426824, 0.48542089,
        0.30505741, 0.40572438, 0.63931803, 0.11610642, 0.        ]])
```

[90]:
```
dist_sq.shape
```

[90]:
```
(10, 10)
```

[91]:
```
dist_sq.diagonal()
```

[91]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

1.2.6 Conclusions moving forward

It's important to understand numpy and arrays, since most data sets we encounter are rectangular. The notations and operations we saw in numpy will translate to data, except for the fact that data is typically heterogeneous, i.e., of different types. The problem with using numpy for modern data analysis is that if you have mixed data types, it will all be coerced to strings, and then you can't actually do any data analysis.

The solution to this issue (which is also present in Matlab) came about with the `pandas` package, which is the main workhorse of data science in Python

[ ]:

[ ]:

[ ]: