Data Analysis using Python

Chapter 4 page no. 80-88

// try all function with 1D and 2D array.

array (): - Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.

arange() :- Like the built-in range but returns an ndarray instead of a list.
np.arange(15).

ones() :- Produce an array of all 1's with the given shape and dtype. ones_like takes another array and produces a ones array of the same shape and dtype.

zeros() :- Like ones and ones_like but producing arrays of 0's instead
np.zeros(10)

empty() :- Create new arrays by allocating new memory, but do not populate with any values like ones and zeros
np.empty((2, 3, 2))

data1 = [6, 7.5, 8, 0, 1]

arr1 = np.array(data1)

data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

arr2 = np.array(data2)

arr2.ndim //will return dimension

arr2.shape // size of array

arr2.dtype // data type of array

Operations between Arrays and Scalars

Array enable you to express batch operations on data without writing any for loops. **It is called vectorization**

```
data1 * 10
data1+ data2
data1-data2
```
//try and see output

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
arr * arr
arr – arr
1 / arr
```

//try and see output

arr ** 0.5

Note: Operations between differently sized arrays is called *broadcasting will discuss later*

explicit type casting

```
float a = 1.2;
//int b  = a; //Compiler will throw an error for this
int b = (int)a + 1
```

implicit type casting

```
int a=10; //initializing variable of short data type
float b; //declaring int variable
b=a; //implicit type casting
```

Chapter 4 page no 84 for NumPy data types

arr1 = np.array([1, 2, 3], dtype=np.float64) //try and see output

arr2 = np.array([1, 2, 3], dtype=np.int32) //try and see output

- You can explicitly convert or *cast* an array from one dtype to another using ndarray's astype method:

arr = np.array([1, 2, 3, 4, 5])

arr.dtype

float_arr = arr.astype(np.float64)

float_arr.dtype //try and see output

- If I cast some floating point numbers to be of integer dtype, the decimal part will be truncated:

arr = np.array([3.7, -1.2, -2.6, 0.5, 12.9, 10.1])

arr.astype(np.int32) //try and see output

- You have an array of strings representing numbers, you can use astype to convert them to numeric form:

numeric_strings = np.array(['1.25', '-9.6', '42'], dtype=np.string_)

numeric_strings.astype(float) //try and see output

- NumPy is smart enough to alias the Python types to the equivalent dtypes.

int_array = np.arange(10)

calibers = np.array([.22, .270, .357, .380, .44, .50], dtype=np.float64)

int_array.astype(calibers.dtype) //try and see output

Note: Calling astype *always* creates a new array (a copy of the data), even if the new dtype is the same as the old dtype.

Basic Indexing and Slicing: **Chapter 4 page no.86**

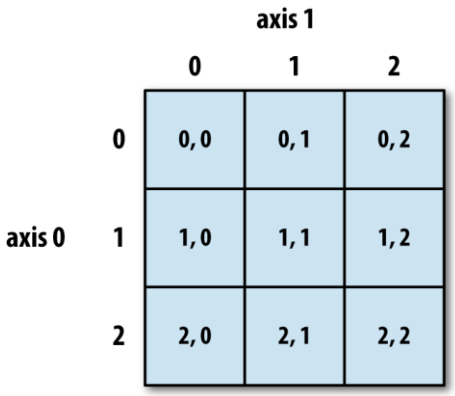
arr = np.arange(10)
arr[5]
arr[5:8]
arr[5:8] = 12 // print(arr)

arr_slice = arr[5:8]
arr_slice[1] = 12345
arr_slice[:] = 64 // print(arr)

Note: If you want a copy of a slice of an ndarray instead of a view, you will need to explicitly copy the array; for example arr[5:8].copy(). //try this

Slicing With 2D array

arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d[2]
arr2d[0][2]
arr2d[0, 2]



3D array or multidimensional arrays,
Example : 2 × 2 × 3 array

```
                                0   1  2
arr3d = np.array( [  0=  [  0=   [1, 2, 3],
                         0=   1=   [4, 5, 6]     ],

                         1= [ 0 =   [7, 8, 9],
                         1=  1 = [10, 11, 12]   ]     ]    ) // print(arr3d)
```

arr3d[0] is a 2 × 3 array: Indexing of 2 × 2 × 3 array

```
[0][0][0]  [0][0][1]  [0][0][2]
[0][1][0]  [0][1][1]  [0][1][2]

[1][0][0]  [1][0][1]  [1][0][2]
[1][1][0]  [1][1][1]  [1][1][2]
```

arr3d[0]

Both scalar values and arrays can be assigned to arr3d[0]:

```
old_values = arr3d[0].copy()
arr3d[0] = 42                // Print arr3d
arr3d[0] = old_values     // Print arr3d
arr3d[1, 0]
```

Indexing with slices: Chapter 4 page no.88

Like one-dimensional objects such as Python lists, ndarrays can be sliced using the familiar syntax:

```
arr2d = ([[1, 2, 3], [4, 5, 6],[7, 8, 9]])
arr2d[:2]
arr2d[:2, 1:]
arr2d[1, :2]
arr2d[2, :1]
arr2d[:, :1]
arr2d[:2, 1:] = 0
```

Chapter 4: NumPy Basics: Arrays and Vectorized Computation

Indexing with slices: Chapter 4 page no.88
Like one-dimensional objects such as Python lists, ndarrays can be sliced using the familiar syntax:

```
arr2d = ([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
arr2d[:2]
arr2d[:2, 1:]
arr2d[1, :2]
arr2d[2, :1]
arr2d[:, :1]
arr2d[:2, 1:] = 0
```

Boolean Indexing: Chapter 4 page no. 99

Let's consider an example where we have some data in an array and an array of names with duplicates. I'm going to use here the randn function in numpy.random to generate some random normally distributed data:

```
In []: names = np.array(['Bob', 'Joe', 'Will', 'Bob', 'Will', 'Joe', 'Joe'])
In []: names

In []: data = np.random.randn(7, 4)
In []: data
Out[]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
       [ 1.0072, -1.2962, 0.275 , 0.2289],
       [ 1.3529, 0.8864, -2.0016, -0.3718],
       [ 1.669 , -0.4386, -0.5397, 0.477 ],
       [ 3.2489, -1.0212, -0.5771, 0.1241],
       [ 0.3026, 0.5238, 0.0009, 1.3438],
       [-0.7135, -0.8312, -2.3702, -1.8608]])
```

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'.

```
In []: names = = 'Will'
Out[]: array([ False, False, True, False, True, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In []: data[names == 'Will']
Out[]:
array([[ 1.3529, 0.8864, -2.0016, -0.3718],
[ 3.2489, -1.0212, -0.5771, 0.1241]])
```

The boolean array must be of the same length as the array axis it's indexing. You can even mix and match boolean arrays with slices or integers .

Note: Boolean selection will not fail if the boolean array is not the correct length, so I recommend care when using this feature.

In these examples, I select from the rows where names == 'Bob' and index the columns, too:

```
In []: data[names == 'Bob', 2:]
Out[]:
array([[ 0.769 , 1.2464],
[-0.5397, 0.477 ]])

In []: data[names == 'Bob', 3]
Out[105]: array([ 1.2464, 0.477 ])
```

To select everything but 'Bob', you can either use!= or negate the condition using ~:

```
In []: names != 'Bob'
Out[]: array([False, True, True, False, True, True, True], dtype=bool)

In []: data[~(names == 'Bob')]
Out[]:
array([[ 1.0072, -1.2962, 0.275 , 0.2289],
```

```
[ 1.3529, 0.8864, -2.0016, -0.3718],
[ 3.2489, -1.0212, -0.5771, 0.1241],
[ 0.3026, 0.5238, 0.0009, 1.3438],
[-0.7135, -0.8312, -2.3702, -1.8608]]
```

The ~ operator can be useful when you want to invert a general condition:

```
In []: cond = names == 'Bob'
In []: data[~cond]
Out[]:
array([[ 1.0072, -1.2962, 0.275 , 0.2289],
[ 1.3529, 0.8864, -2.0016, -0.3718],
[ 3.2489, -1.0212, -0.5771, 0.1241],
[ 0.3026, 0.5238, 0.0009, 1.3438],
[-0.7135, -0.8312, -2.3702, -1.8608]])
```

Selecting two of the three names to combine multiple boolean conditions, use boolean arithmetic operators like & (and) and | (or):

```
In []: mask = (names == 'Bob') | (names == 'Will')
In []: mask         ///[0][2][3][4]
Out[]: array([ True, False, True, True, True, False, False], dtype=bool)
In []: data[mask]
Out[]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
[ 1.3529, 0.8864, -2.0016, -0.3718],
[ 1.669 , -0.4386, -0.5397, 0.477 ],
[ 3.2489, -1.0212, -0.5771, 0.1241]])
```

Note :The Python keywords and and or do not work with boolean arrays. Use & (and) and | (or) instead.

Setting values with boolean arrays works in a common-sense way. To set all of the negative values in data to 0 we need only do:

```
In []: data[data < 0] = 0
In []: data
Out[]:
array([[ 0.0929, 0.2817, 0.769 , 1.2464],
[ 1.0072, 0. , 0.275 , 0.2289],
[ 1.3529, 0.8864, 0. , 0. ],
[ 1.669 , 0. , 0. , 0.477 ],
[ 3.2489, 0. , 0. , 0.1241],
[ 0.3026, 0.5238, 0.0009, 1.3438],
[ 0. , 0. , 0. , 0. ]])
```

Setting whole rows or columns using a one-dimensional boolean array is also easy:

```
In []: data[names != 'Joe'] = 7
In []: data
Out[]:
array([[ 7. , 7. , 7. , 7. ],
[ 1.0072, 0. , 0.275 , 0.2289],
[ 7. , 7. , 7. , 7. ],
```

[7. , 7. , 7. , 7.],
[7. , 7. , 7. , 7.],
[0.3026, 0.5238, 0.0009, 1.3438],
[0. , 0. , 0. , 0.]])

Note: As we will see later, these types of operations on two-dimensional data are convenient to do with pandas.

From:-
Ritu Meena
Assistant Professor
Shivaji College
Delhi University