# 01-03 The Power of NumPy

*Compiled by Shipra De, Fall 2016*

## What is NumPy?



- In this lesson, we're going to learn about the NumPy numerical library.
- NumPy is a Python library that acts as a wrapper around underlying C and Fortran code. Because of that, it's very, very fast.
- NumPy focuses on matrices which are called in the arrays. The syntax is very similar to MATLAB, so if you've used MATLAB before It'll look familiar to you.
- NumPy is one of the important reasons people use Python for financial research.

# Lesson Outline

*If you're familiar with NumPy (esp. the following operations), feel free to skim through this lesson.*

- Create a NumPy array:
    - from a pandas dataframe: pandas.DataFrame.values
    - from a Python sequence: numpy.array
    - with constant initial values: numpy.ones, numpy.zeros
    - with random values: numpy.random
- Access array attributes: shape, ndim, size, dtype
- Compute statistics: sum, min, max, mean
- Carry out arithmetic operations: add, subtract, multiply, divide
- Measure execution time: time.time, profile
- Manipulate array elements: Using simple indices and slices, integer arrays, boolean arrays

## Relationship to Pandas
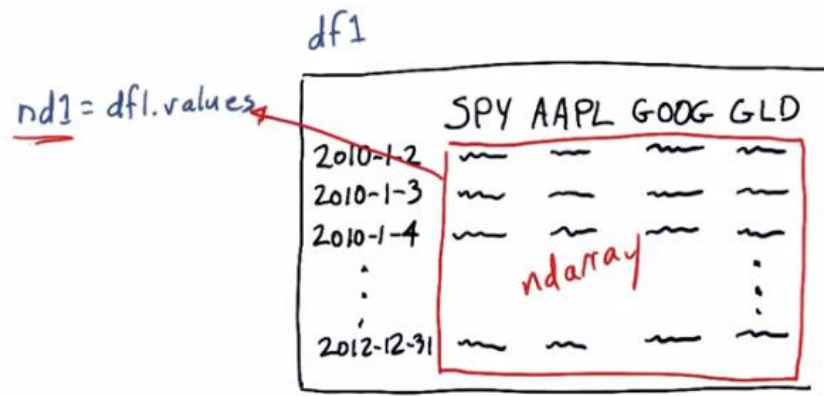


- Now, how does NumPy relate to Pandas?
- Well, I said just a moment ago that NumPy is a wrapper for numerical  libraries, well it turns out that Pandas is a kind of wrapper for NumPy.
- So remember our traditional data frame here, with our columns being symbols and  our rows being dates.  This data frame is just a wrapper around this ndarray,  access the columns with symbols and the rows by dates.
- But you can, in fact, just treat this inside part as an ndarray directly.  If you use this syntax in Python, that pulls these values out and  lets you access it directly and then ndarray.
- You don't really need to do that though, you can,  if you like, treat a data frame just like a NumPy ndarray.
- And so we're going to assume in the rest of this lesson  that we're just working with an ndarray.
- And like I said, you can use all of these mechanisms that we're going to  show you with ndarrays and with data frames directly.
- What you get if you create something as a data frame, as we'll see in  a lesson a little bit later, you get many, many, many more routines.  And you can treat it, like I said, just like an ndarray but  you get a vast new number of statistical functions and so on.

# Notes on Notation



Notes on notation

nd1 [row, col]
nd1 [0,0]
nd1 [3,2]
nd1 [0:3, 1:3]
nd1 [:, 3]
nd1 [-1, 1:3]

nd1

-2
-1
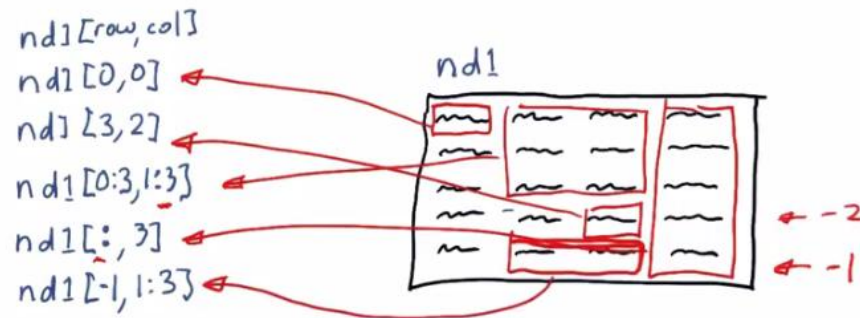
- Consider an nd array, nd1.  I'm going to show you now how to access cells within that.
- Now, the notation, at first, might seem sort of familiar, but  there's some new and different things that you probably haven't seen before.
- So the usual syntax is the name of your nd array,  bracket, the row and the column.
- So again, these are our rows.  So row indicates which row we're using.  Column, which column.
- It's important to know that in NumPy, our columns and rows begin at 0.  So this element is nd1[0,0].  It then continues of course, 1, 2, 3, 4 in the rows, and  in the columns, 0, 1, 2, 3.
- Before I tell you, see if you can guess how to address this cell.  The answer is that this cell is nd1 [3,2],
- Now, this is probably the kind of stuff you've seen before.  It turns out, though, that the NumPy is much more powerful and  can do interesting and different kinds of slicing.
- What if, for instance,  you wanted to address this sub portion of the nd array?  How could you indicate that?  NumPy uses a special symbol, the colon, to let you indicate ranges.  So we can indicate this range in rows with 0:3,  which indicates the zeroth to the, just before the third row.
- And in the columns, we've got 01:3.  So this syntax indicates starting at the zeroth row to just before the third and the first column to just before the third.  And in fact, captures this region.
- The key thing to remember here that's a little bit tricky is that  this last value is one past the one that you actually want to include.  So, for instance, this is column 3, but it's not included.
- Now, if we just use the colon by itself that indicates, for instance,  if we place it in the rows position, that we want all of the rows.  So you don't have to use the colon just to indicate a range.  You can use it by itself for all of them.
- Now, look at this statement, see if you can figure out  which part of this nd array it refers to before I show you.  It is this region right here.  So it's all the rows and column 3, 0, 1, 2, 3.  So it's this section right here.
- NumPy includes some special syntax that lets  you refer to the last row or column.  So, for instance, the last row here, you can indicate with  negative 1, second to last row would be negative 2.  S

- o if we wanted to refer to these 2 cells here,  we would take advantage of this negative 1 syntax. So a negative 1 indicates that last row.  And then to get these 2 columns, we would use 1:3. then we don't include the last 1 there.
- There is a bunch of new syntax.  I hope that you find it exciting.  This is really one of the most powerful aspects of Python and NumPy.  And it really enables you to do some interesting things.
- Now, we've got a quiz to see if you can figure out how to use this new  syntax yourself.

# Quiz: Replace a Slice

## Quiz: Which statement does the job?

nd1

☐ nd1 = nd2
☐ nd1[1:2, 1:2] = nd2[4:5, 3:4]
☐ nd1[0:2, 0:2] = nd2[-2:, 2:4]
☐ nd2[3:5, 2:4] = nd1[1:3, 1:3]

nd2
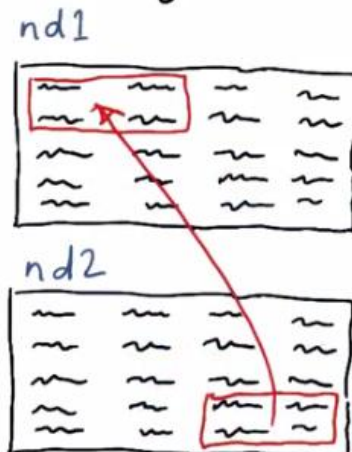
- Now we've shown you how to address slices of ND arrays. We're going to give you a little quiz to see if you can figure something out.
- Suppose we have these two ND arrays, nd1 and nd2. And we want to replace some of the values in nd1, with these values from nd2.
- Here are four alternatives, see which one you think makes the most sense.

## Quiz: Which statement does the job?

nd1

☐ nd1 = nd2
☐ nd1[1:2, 1:2] = nd2[4:5, 3:4]
☑ nd1[0:2, 0:2] = nd2[-2:, 2:4]
☐ nd2[3:5, 2:4] = nd1[1:3, 1:3]

nd2

-2
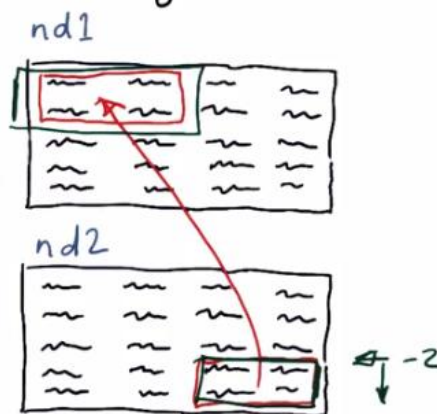
- Of these four, this one is the right answer. Here's why.
- This is the only left-hand side that singles out these correct rows, starting at zero, and ending at one. And these correct columns, again starting at zero and ending at one.
- If you look at the right hand side, here is a little bit of new syntax that you hadn't seen before. We indicate the rows second from last by a -2. So that's what that -2 means. And a colon with

nothing after it means go all the way to the end.  So we've singled out these rows and this indicates these two columns.  So that's why this one is the correct answer.

- Now, I'm going to hand it over to Dev, and  she is going to show you how to do all these things directly in Python syntax.  Here's to you Dev.

# Creating NumPy Arrays

```
1   """Creating NumPy arrays."""
2
3   import numpy as np
4
5   def test_run():
6       # List to 1D array
7       print np.array([2, 3, 4])
8
9   if __name__ == "__main__":
10      test_run()
11
```

- You can access the underlining NumPy array within a Pandas data frame  using the values property.  But you can also create NumPy arrays from scratch.  There are many ways to create an array.
- Let's start with creating a one dimensional array from known values.  NumPy has an array function which can convert most  array-like objects into an n d array.  What do we mean by nd array is n-dimensional array.
- Let's see how this works for Python lists.  To start with, we need to import the library numpy.  And we rename it as np for the ease of use.
- Next, we simply call a function np.array and  pass a list which has value [2,3,4].  Note that this function can take as input a list,  a template, or other sequence.
- Check out the documentation for the array function and nd array type for  more information.  Now let's see the output.

```
[2 3 4]
```

- The output you see here is not a list but it is an array.  Let's go ahead and create a 2D array.

```
1   """Creating NumPy arrays."""
2
3   import numpy as np
4
5   def test_run():
6       # List of tuples to 2D array
7       print np.array([(2, 3, 4), (5, 6, 7)])
8
9   if __name__ == "__main__":
10      test_run()
11
```

- Now if you want to create a 2D array,  we simply pass in a sequence of sequences to this function.  Each tuple enclosed in round parenthesis  serves as one row in the resulting array.

- We could also have passed a list of lists.  This is called sequence of sequences.  Let's go ahead and print it.

```
[[2 3 4]
 [5 6 7]]
```

- Here is the output and as expected there are two rows and three columns.
- This function is mainly useful when you have a list of sequence of values, and  you want to convert them into NumPy arrays.

# Arrays With Initial Values

```
1  """Creating NumPy arrays."""
2
3  import numpy as np
4
5  def test_run():
6      # Empty array
7      print np.empty(5)
8      print np.empty((5,4,3))
9
10 if __name__ == "__main__":
11     test_run()
12
```

- NumPy also offers several function to create empty arrays with initial values. For certain computations these help avoid growing arrays incrementally which can be an expensive operation.
- Let's start with creating an empty array. The empty function takes the shape of the array as input. The shape can be defined as a single integer, as we did over here, for creating a one dimensional array, or a sequence of integers denoting the size in each dimension.
- For a two dimensional array, a sequence of two integers is needed. That is the number of rows and the number of columns.
- For this example, we will create an empty array with five rows and four columns. Passing in a tuple with values 5 and 4. So here I pass a tuple with values 5 and 4.
- In case you need a three dimensional array, or any greater number of dimensions, you can just add another number to the sequence. This will give you a 3 dimensional array with a depth of 3, and each depth having 5 rows and 4 columns.
- For this lesson we will only work with two dimensional arrays. Now let's check the output.

```
[ 0.    0.25  0.5   0.75  1.  ]
[[ 6.92788349e-310  2.07616661e-316  2.61369225e-316  2.61746493e-316]
 [ 0.00000000e+000  0.00000000e+000  0.00000000e+000  5.58294180e-322]
 [ 2.59185099e-316  2.64070894e-316  2.66353201e-316  2.64128917e-316]
 [ 0.00000000e+000  0.00000000e+000  0.00000000e+000  2.42092166e-322]
 [ 2.35741407e-316  2.15252307e-316  0.00000000e+000  0.00000000e+000]]
```

- Hm, strange. The empty array is not actually empty.
- What happens is that when we call numpy.empty to create an array, the elements of the array read in whatever values were present in the corresponding memory location.
- These are effectively random values that depend on the state of the computer's memory. Also observe that by default the elements are the floating points.

```
1   """Creating NumPy arrays."""
2
3   import numpy as np
4
5   def test_run():
6       # Array of 1s
7       print np.ones((5, 4))
8
9   if __name__ == "__main__":
10      test_run()
11
```

- Next we create an array full of ones.  Like the empty function, we pass in the number of rows and  columns as a sequence.
- To create an array full of ones, you call the one function and  pass the sequence, which has number of rows and number of columns.
- You can expect this time to have an array of 5 rows and  4 columns with all the values equal to 1. Let's go ahead and check this.

```
[[ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]
 [ 1.  1.  1.  1.]]
```

- Here it is.  An array with 5 rows, 4 columns, and all the values of the array equal to 1.

# Specify the Data Type

```
1  """Creating NumPy arrays."""
2
3  import numpy as np
4
5  def test_run():
6
7      # Array of 1s
8      print np.ones((5, 4))
9
10
11 if __name__ == "__main__":
12     test_run()
13
```



parameter = value

- We notice that the default data type of all the values in the array is.  Fortunately, you can change this when creating the array.
- What parameter do you need to add to this function to create  an array of integers instead? Type the name of the parameter and  the correct value in the corresponding boxes. Documentation for the array.ones function might be helpful.

```
1  """Creating NumPy arrays."""
2
3  import numpy as np
4
5  def test_run():
6
7      # Specifying the datatype
8      print np.ones((5, 4), dtype=np.int_)
9
10
11 if __name__ == "__main__":
12     test_run()
13
```

- dtype is the parameter we passed to the function  to specify the type of the value we want in each array location.
- Here we defined the values to be integers using NumPy data type np.int.  Just as a matter of fact,  NumPy supports a much greater variety of numerical types than Python.  Let's run this.

```
[[1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]
 [1 1 1 1]]
```

- Here it is.  The array now has integer values.  And since we defined the array as ones, it has all the values as 1.  Just like function np and  ones, you can create an array full of zeroes using the zeroes function.  All these functions accept the dtype parameter.
- Before moving forward, I would like to mention that we can also create  n-dimensional array using the low-level NumPy function ndarray.  But ones, zeroes and empty provide a more friendly interface for  creating arrays.  And are hence generally preferred.
- Refer to the documentation links and instructor notes for more information.

# Generating Random Numbers

```
1  """Generating random numbers."""
2
3  import numpy as np
4
5
6  def test_run():
7      # Generate an array full of random numbers, uniformly sampled from [0.0, 1.0)
8      print np.random.random((5, 4))  # pass in a size tuple
9
10 if __name__ == "__main__":
11     test_run()
12
```

- Numpy also comes with bunch of handy functions to generate arrays  filled with random values. These functions are defined in numpy's random module.
- The random function generates uniformly sampled  floating point values between 0 and 1, with 0 inclusive and 1 exclusive.  More formally,  we can say that it generates values in the half open interval 0.0 and 1.0.  Let's go ahead and print this.

```
[[ 0.07225625  0.97905632  0.89758156  0.29849324]
 [ 0.61697108  0.22355585  0.52496941  0.46650802]
 [ 0.59349213  0.81397425  0.80168121  0.35397074]
 [ 0.54816879  0.41716406  0.10875819  0.88556204]
 [ 0.31047183  0.2483835   0.58044803  0.93047677]]
```

- Here is the generated array with five rows and four columns.  Note that we pass the array shape as a tuple.

```
1  """Generating random numbers."""
2
3  import numpy as np
4
5
6  def test_run():
7      # Generate an array full of random numbers, uniformly sampled from [0.0, 1.0)
8      print np.random.rand(5, 4)  # function arguments (not a tuple)
9
10 if __name__ == "__main__":
11     test_run()
12
```

- A slightly radiant of this function is rand which randomly accepts  a sequence of numbers as arguments and straight of the tuple.  It is otherwise equally valid.
- Observe that, we directly pass the values of the rows and  columns through the function and did not define a tuple.

```
[[ 0.97936087  0.35199054  0.75700872  0.21773279]
 [ 0.51469436  0.13792218  0.60113068  0.49844396]
 [ 0.03884726  0.68547752  0.92831869  0.70716921]
 [ 0.78714968  0.56014382  0.06057555  0.36082925]
 [ 0.7883221   0.26769747  0.87395281  0.99826105]]
```

- Here it is the array with same shape as before five rows and four columns.
- Numpy provides this to achieve a greater compatibility with the more established MATLAB syntax.
-  We highly recommend using a more consistent NumPy function that explicitly accepts a shape tuple.
- Now both the function, rand and random, sample, uniformly, from the range 0 and 1.
- What if you wanted a sample from a different distribution? To sample, for normal distribution, we can use the normal function.

```
1   """Generating random numbers."""
2
3   import numpy as np
4
5
6   def test_run():
7
8       # Sample numbers from a Gaussian (normal) distribution
9       print np.random.normal(size=(2, 3))  # "standard normal" (mean = 0, s.d. = 1)
10
11
12
13  if __name__ == "__main__":
14      test_run()
15
```

- Recall the normal function from numpy dot random and pass the shape of the array required. Let's run this.

```
[[-0.4293319  -0.29616839 -1.10015845]
 [-1.75087356  0.23994969 -1.02156598]]
```

- The code produced a 2 into 3 array of random numbers with a standard normal distribution. That is 0 mean, and unit standard deviation. You can change the mean and the standard deviation as well.

```python
1 """Generating random numbers."""
2
3 import numpy as np
4
5
6 def test_run():
7
8     # Sample numbers from a Gaussian (normal) distribution
9     print np.random.normal(50, 10. size=(2. 3))  # change mean to 50 and s.d. to 10
10
11
12
13 if __name__ == "__main__":
14     test_run()
15
```

- Let's see how to do that.  We change the mean to 50 and standard deviation to 10.  Now, let's see the output.

$$[[ 43.40214498 \quad 43.16943426 \quad 39.65748171]$$
$$[ 48.6551351 \quad\quad 46.37764162 \quad 67.73721141]]$$

- Notice that the values are centered around 50.

```python
1 """Generating random numbers."""
2
3 import numpy as np
4
5
6 def test_run():
7
8     # Random integers
9     print np.random.randint(10)  # a single integer in [0, 10)
10     print np.random.randint(0, 10)  # same as above, specifying [low, high) explicit
11     print np.random.randint(0, 10, size=5)  # 5 random integers as a 1D array
12     print np.random.randint(0, 10, size=(2, 3))  # 2x3 array of random integers
13
14
15
16 if __name__ == "__main__":
17     test_run()
18
```
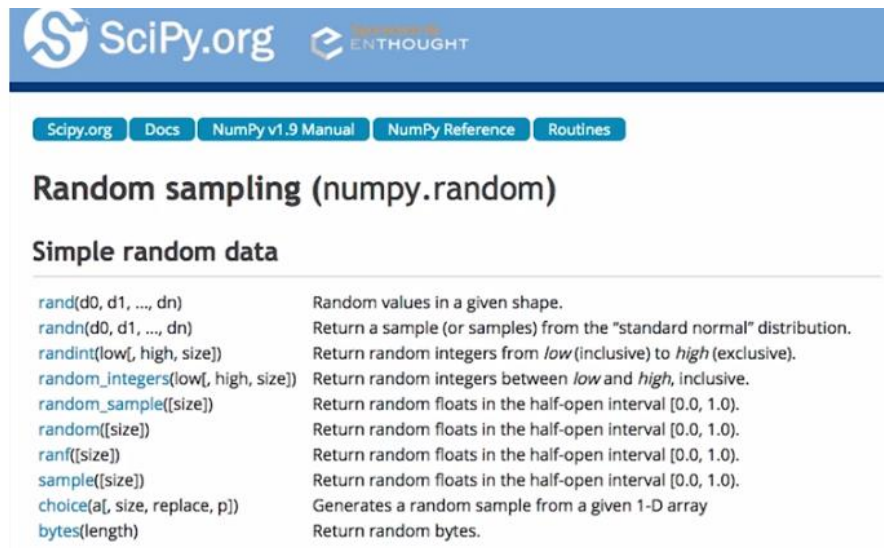
- To generate integers,  we can use the randint function in one of the several ways.  Passing two values 0 and 10 will notify randint to generate  a single integer between the range 0 and 10.
- We can also specify randint how many integers we want between between 0 and  10 by specifying the size attribute and giving it a value.
- So, this statement will give us a 1d array  of 5 integers between the range 0 and 10.
- Going forward with that, we can pass a tuple value to the attribute size,  which will create a 2d array with all the values between the range 0 and 10.
- Now, let's see the output.

```
a single integer
1
a single integer
7
1d-array
[3 2 2 3 6]
2d array
[[7 4 2]
 [8 0 3]]
```

- These are the single random integers between the range 0 and 10.  Next, we created a 1d array with five values.
- Note that, we mentioned the number of values needed in the one dimensional array with the parameter size.  Passing a tuple to the size parameter gave us the 2d values.
- And also note that all the values of the array are between 0 and 10.



**Random sampling (numpy.random)**

**Simple random data**

| | |
|---|---|
| rand(d0, d1, ..., dn) | Random values in a given shape. |
| randn(d0, d1, ..., dn) | Return a sample (or samples) from the "standard normal" distribution. |
| randint(low[, high, size]) | Return random integers from *low* (inclusive) to *high* (exclusive). |
| random_integers(low[, high, size]) | Return random integers between *low* and *high*, inclusive. |
| random_sample([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| random([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| ranf([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| sample([size]) | Return random floats in the half-open interval [0.0, 1.0). |
| choice(a[, size, replace, p]) | Generates a random sample from a given 1-D array |
| bytes(length) | Return random bytes. |

- Check out the random sampling routines on the numpy website for  more distribution and usage radiations.  Find the link in the instructor's notes.

## Array Attributes

```
1  """Array attributes."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      a = np.random.random((5, 4))  # 5x4 array of random numbers
9      print a
10
11
12 if __name__ == "__main__":
13     test_run()
```

- Any numpy array has a number of attributes that describes it in addition to the elements it contains.
- One of the most useful one is shape. Essentially a tuple containing the number of rows and columns are height and width of the array.
- We have already seen how to specify this when creating arrays. The shape of the array A would be five rows and four columns.

```
[[ 0.96136986  0.4167991   0.55704591  0.41034615]
 [ 0.16532119  0.83201378  0.71535281  0.57202699]
 [ 0.85014899  0.12720707  0.29955882  0.97506376]
 [ 0.52166033  0.74687469  0.72815694  0.5955816 ]
 [ 0.10273617  0.6176485   0.77731634  0.63172813]]
```

- So this is your array. Now let's see how to access the shape of the given array by using the shape attribute.

```
1  """Array attributes."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      a = np.random.random((5, 4))  # 5x4 array of random numbers
9      print a
10     print a.shape
11
12
13 if __name__ == "__main__":
14     test_run()
```

- a.shape will give you the shape of the array.  Let's run it.

```
[[ 0.3450274   0.13742288  0.6344561   0.76183558]
 [ 0.84588047  0.40003584  0.72715559  0.27408832]
 [ 0.99468627  0.6548842   0.95990234  0.87094856]
 [ 0.45663215  0.86427198  0.63980195  0.76983459]
 [ 0.99435966  0.50241508  0.01248806  0.7294946 ]]
(5, 4)
```

- Array.shape will return you a tuple with the first value specifying  the number of rows and the
  second value specifying the number of columns.
- Next we will learn how to individually access number of rows or  number of columns.

```
1  """Array attributes."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      a = np.random.random((5, 4))  # 5x4 array of random numbers
9      print a.shape[0] #number of rows
10     print a.shape[1] #number of columns
11
12
13 if __name__ == "__main__":
14     test_run()
```

- a.shape[0] would return the number of rows and  a.shape[1] would return the number of
  columns.  Let's check the output.

```
5
4
```

- Here you will see that the number of rows are correctly extracted as five.  And number of
  columns as 4.  If you have more dimensions,  you will have additional elements in the shape
  tuple.

```
1  """Array attributes."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      a = np.random.random((5, 4))   # 5x4 array of random numbers
9      print len(a.shape)
10
11
12 if __name__ == "__main__":
13     test_run()
```

- The number of dimensions in an array can be found by simply asking for the length of this tuple. a.shape will return a tuple and the length of that tuple would inform us what is the dimension of the array.

2

- It rightly tells us that the dimension of the defined array A is 2.

```
1  """Array attributes."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      a = np.random.random((5, 4))   # 5x4 array of random numbers
9      print a.size
10
11
12 if __name__ == "__main__":
13     test_run()
```

- Okay, how about total number of elements in an array? Yes for a 2D array, it will be the product of number of rows and columns.
- But if you had more dimension, this calculation could be a little complicated. Fortunately we can retrieve the number of elements directly using size attributes.
- a.size will give us the number of elements present in the array A.
- We can expect the output to be the product of the rows and the columns which is 5 into 4, which is 20. Let's check it.

20

- As we expected, the output is 20 which means there are 20 values in the array.
- Attributes like size and shape are very useful when you have to over array elements to perform some computation.

```python
1  """Array attributes."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      a = np.random.random((5, 4))  # 5x4 array of random numbers
9      print a
10     print a.dtype
11
12
13 if __name__ == "__main__":
14     test_run()
```

- You can also access the data type of each element using the D type attribute of the array.
- Let's check the data type of the values present in array A.

```
[[ 0.20432042  0.26642814  0.34949183  0.0011192 ]
 [ 0.04043668  0.3925168   0.94642959  0.06172868]
 [ 0.2271386   0.01020622  0.4692534   0.24644589]
 [ 0.86112155  0.74380659  0.8010506   0.16760849]
 [ 0.80423467  0.17408713  0.16258331  0.93353746]]
float64
```

- In this case, our array elements are of the type float64. That is 64-bit floating point numbers.

# Operations on ndarrays

```
1  """Operations on arrays."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      np.random.seed(693)  # seed the random number generator
9      a = np.random.randint(0, 10, size=(5, 4))  # 5x4 random integers in [0, 10)
10     print "Array:\n", a
11
12
13
14
15 if __name__ == "__main__":
16     test_run()
17
```

- Next you will see how to perform various mathematical operations on np arrays. Let's use a random heading of integers. Let's create an array with shape five rows and four columns. Let's see the output.

```
Array:
[[2 0 5 1]
 [1 3 4 4]
 [9 2 9 1]
 [9 3 7 5]
 [4 7 0 3]]
```

- So here's an array with five rows, four columns, and all the values between the range 0 and 10.

```
1  """Operations on arrays."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      np.random.seed(693)  # seed the random number generator
9      a = np.random.randint(0, 10, size=(5, 4))  # 5x4 random integers in [0, 10)
10     print "Array:\n", a
11
12
13
14
15 if __name__ == "__main__":
16     test_run()
17
```

- Note how we used seed, the random number generator with the constant, to get the same sequence of numbers every time. Let's run again and see if the output remains the same.

```
Array:
[[2 0 5 1]
 [1 3 4 4]
 [9 2 9 1]
 [9 3 7 5]
 [4 7 0 3]]
```

- You can see that we have the same values for the array.

```
 1  """Operations on arrays."""
 2
 3  import numpy as np
 4
 5
 6  def test_run():
 7
 8      np.random.seed(693)  # seed the random number generator
 9      a = np.random.randint(0, 10, size=(5, 4))  # 5x4 random integers in [0, 10)
10      print "Array:\n", a
11
12      # Sum of all elements
13      print "Sum of all elements:", a.sum()
14
15
16
17
18  if __name__ == "__main__":
19      test_run()
20
```

- Summing all the elements in an array is as simple as calling the function sum on the array.
- Here is our array a, and we call the sum function on it. Let's check the output.

```
Array:
[[2 0 5 1]
 [1 3 4 4]
 [9 2 9 1]
 [9 3 7 5]
 [4 7 0 3]]
Sum of all elements: 79
```

- This is our array and this is sum of all the elements present in the array, which comes out to be 79.

```
1  """Operations on arrays."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      np.random.seed(693)  # seed the random number generator
9      a = np.random.randint(0, 10, size=(5, 4))  # 5x4 random integers in [0, 10)
10     print "Array:\n", a
11
12     # Iterate over rows, to compute sum of each column
13     print "Sum of each column:\n", a.sum(axis=0)
14
15     # Iterate over columns to compute sum of each row
16     print "Sum of each row:\n", a.sum(axis=1)
17
```

- We can also sum in a specific direction of the array.
- What I mean by direction is along rows or columns.  NumPy gives this direction a special name. It is called access.
    - Access is equal to zero, signifies rows, and  access is equal to one indicates columns. Remember this terminology as you will use it frequently.
-  Let's code to make things clear.  Passing the parameter, axis,  along with a specific value will give you the sum along that axis.  To understand this, let's first see the output.

```
Array:
[[2 0 5 1]
 [1 3 4 4]
 [9 2 9 1]
 [9 3 7 5]
 [4 7 0 3]]
Sum of each column:
[25 15 25 14]
Sum of each row:
[ 8 12 21 24 14]
```

- To get the sum of each column, we pass the value to the axis attribute as zero.  And to get the sum of the rows, we pass the value as one.
- To understand this imagine if you wanted to sum the values of each column,  what would you iterate on?
    - You would say something like,  For each column, sum all the values of each row of that column.
    - So you would essentially iterate over the rows.  Hence we pass axis=0 to compute column sums and similarly axis=1 for row sums.
- Observe the output when we pass axis=0, we get four values.  These are basically the sum of each columns.

- And when we passed axis=1.  We get five values which are the sum of each row.

```python
1  """Operations on arrays."""
2
3  import numpy as np
4
5
6  def test_run():
7
8      np.random.seed(693)  # seed the random number generator
9      a = np.random.randint(0, 10, size=(5, 4))  # 5x4 random integers in [0, 10)
10     print "Array:\n", a
11
12     # Statistics: min, max, mean (across rows, cols, and overall)
13     print "Minimum of each column:\n", a.min(axis=0)
14     print "Maximum of each row:\n", a.max(axis=1)
15     print "Mean of all elements:", a.mean()  # leave out axis arg.
16
17
18
19
20 if __name__ == "__main__":
21     test_run()
```

- Let's go ahead and try some basic operation like finding minimum,  maximum, and mean of an array.
- So, if I want minimum along columns, I have to go through each row  of each column, so axis equal to zero to get the minimum of each column.
- To get the maximum of each row, similarly we call a max function and  pass access equal to one.
- Just calling a.mean,  that is array dot mean, will give us the mean of the entire array.
-  Of course we can get mean along each axis as we did for max and min.

```
Array:
[[2 0 5 1]
 [1 3 4 4]
 [9 2 9 1]
 [9 3 7 5]
 [4 7 0 3]]
Minimum of each column:
[1 0 0 1]
Maximum of each row:
[5 4 9 9 7]
Mean of all elements: 3.95
```

- Observe the output.  Minimum of the first column is one, which is shown over here.  This value is essentially, minimum of the first column.  This is of second, this is of third, and this is of fourth.

- Similarly, for maximum of the each row, you can observe that for the first row, the maximum is five, and it is shown here. The mean of all the elements is 3.95 which is calculated using the mean function.
- There are many more functions which you can experiment with. Check the documentation link in the instructor's notes.

# Locate Maximum Value

```python
1  """Locate maximum value."""
2
3  import numpy as np
4
5
6  def get_max_index(a):
7      """Return the index of the maximum value in given 1D array."""
8      # TODO: Your code here
9
10
11 def test_run():
12     a = np.array([9, 6, 2, 3, 12, 14, 7, 10], dtype=np.int32)  # 32-bit integer array
13     print "Array:", a
14
15     # Find the maximum and its index in array
16     print "Maximum value:", a.max()
17     print "Index of max.:", get_max_index(a)
18
19
20 if __name__ == "__main__":
21     test_run()
```

- So far we have seen how to compute certain measures.  How about finding the position of some element in an array?
- Can you implement this simple function to find the index of the maximum value  in the one-dimensional array.  Remember NumPy is your friend.

```python
1  """Locate maximum value."""
2
3  import numpy as np
4
5
6  def get_max_index(a):
7      """Return the index of the maximum value in given 1D array."""
8      return a.argmax()
9
10
11 def test_run():
12     a = np.array([9, 6, 2, 3, 12, 14, 7, 10], dtype=np.int32)  # 32-bit integer array
13     print "Array:", a
14
15     # Find the maximum and its index in array
16     print "Maximum value:", a.max()
17     print "Index of max.:", get_max_index(a)
18
19
20 if __name__ == "__main__":
21     test_run()
```

- To get the maximum value in a given 1D array, you could loop through the array,  finding the maximum and keeping track of an index.
- But numpy can help you do this in a single call.  You must have seen argmax and  argmin used to described optimization equations.  This is the same idea.  Let's check the output.

```
Array: [ 9  6  2  3 12 14  7 10]
Maximum value: 14
Index of max.: 5
```

- So the function returned us the maximum value, along with the index.
- Now for multidimensional arrays, finding and representing indices is a little tricky. But numpy provides some utility functions like underscore index to help you out.

# Timing Python Operations

```
1  """Using time function."""
2
3
4  import time
5
6  def test_run():
7      t1 = time.time()
8      print "ML4T"
9      t2 = time.time()
10     print "The time taken by print statement is ",t2 - t1," seconds"
11
12 if __name__ == "__main__":
13     test_run()
14
```

- We claim that run.py is fast, very fast. So let's confirm this.
- But before that, we need to learn how to time a particular operation. We need to import a library for that. We import the time library to help us know how fast our operation is.
- We use the time function from the imported library. The idea is to capture the time snapshot before the operation, and then again capture the time snapshot after the operation is performed. We then subtract the two times. Simple, right?
- Let's check the code. Here we will check how much time a Python print statement takes. So we capture the time before the print statement and then record the time after the print operation is performed. Then, finally, subtract t2- t1. Let's run the code, now.

```
ML4T
The time taken by print statement is  3.09944152832e-05  seconds
```

- Here, we get the time taken by a print statement. Oh, the number is really very small.

## How Fast is NumPy

```
1  """How fast is NumPy?"""
2
3  import numpy as np
4  from time import time
5
6
7  def how_long(func, *args):
8      """Execute function with given arguments, and measure execution time."""
9      t0 = time()
10     result = func(*args)  # all arguments are passed in as-is
11     t1 = time()
12     return result, t1 - t0
13
14
15 def manual_mean(arr):
16     """Compute mean (average) of all elements in the given 2D array."""
17     sum = 0
18     for i in xrange(0, arr.shape[0]):
19         for j in xrange (0, arr.shape[1]):
20             sum = sum + arr[i, j]
21     return sum / arr.size
```

- Now when we know how to time an operation an operation in Python let's check how fast NumPy is.
- Let's define a really large array so that the time taken for the operation will be significant to compare. So here is our large array of 1,000 rows and 10,000 columns.
- Before I go ahead, I would like to mention that this is just a demo code to show the speed of NumPy. So, I will be giving a high level explanation of this code.

```
14
15 def manual_mean(arr):
16     """Compute mean (average) of all elements in the given 2D array."""
17     sum = 0
18     for i in xrange(0, arr.shape[0]):
19         for j in xrange (0, arr.shape[1]):
20             sum = sum + arr[i, j]
21     return sum / arr.size
22
23
24 def numpy_mean(arr):
25     """Compute mean (average) using NumPy."""
26     return arr.mean()
27
28
29 def test_run():
30     """Function called by Test Run."""
31     nd1 = np.random.random((1000, 10000))  # use a sufficiently large array
32
33     # Time the two functions, retrieving results and execution times
34     res_manual, t_manual = how_long(manual_mean, nd1)
```

- Moving ahead, we will be comparing how to compute mean of the array using NumPy and using standard iteration.
- Here is the manual mean function which computes the mean of the values in the defined array. We iterate over each row, and for each row iterate over each column. We then sum present all the values throughout the array. Finally, we divide by the size of the array and hence we get the mean.
- In case of using Num-Py for calculating the mean, we just write array.mean to get the mean of the entire array.

```
7  def how_long(func, *args):
8      """Execute function with given arguments, and measure execution time."""
9      t0 = time()
10     result = func(*args)  # all arguments are passed in as-is
11     t1 = time()
12     return result, t1 - t0
13
14
15 def manual_mean(arr):
16     """Compute mean (average) of all elements in the given 2D array."""
17     sum = 0
18     for i in xrange(0, arr.shape[0]):
19         for j in xrange (0, arr.shape[1]):
20             sum = sum + arr[i, j]
21     return sum / arr.size
22
23
24 def numpy_mean(arr):
25     """Compute mean (average) using NumPy."""
26     return arr.mean()
27
```

- How long function will just compute the time each matter takes. Now, let's check what's the time difference.

```
20             sum = sum + arr[i, j]
21     return sum / arr.size
22
23
24 def numpy_mean(arr):
25     """Compute mean (average) using NumPy."""
26     return arr.mean()
27
```

[ Reset ]  [ Test Run ]

```
Manual: 0.499846 (4.874 secs.) vs. NumPy: 0.499846 (0.017 secs.)
NumPy mean is 290.270481329 times faster than manual for loops.
```

- Do you see the difference? This is the time taken by the numpy.mean function to calculate the mean of the entire array.
- And remember the size of the array are in thousands.
- On the other hand, the time taken by the manual method is about 5 seconds. Hence proved, NumPy is super fast.

```
28
29  def test_run():
30      """Function called by Test Run."""
31      nd1 = np.random.random((1000, 10000))  # use a sufficiently large array
32
33      # Time the two functions, retrieving results and execution times
34      res_manual, t_manual = how_long(manual_mean, nd1)
35      res_numpy, t_numpy = how_long(numpy_mean, nd1)
36      print "Manual: {:.6f} ({:.3f} secs.) vs. NumPy: {:.6f} ({:.3f} secs.)".format(re:
37
38      # Make sure both give us the same answer (upto some precision)
39      assert abs(res_manual - res_numpy) <= 10e-6, "Results aren't equal!"
40
41      # Compute speedup
42      speedup = t_manual / t_numpy
43      print "NumPy mean is", speedup, "times faster than manual for loops."
44
45
46  if __name__ == "__main__":
47      test_run()
48
```

- We also compute the rate of how fast the NumPy is and the numbers are crazy.

```
46  if __name__ == "__main__":
47      test_run()
48
```

Reset    Test Run

```
Manual: 0.499846 (4.874 secs.) vs. NumPy: 0.499846 (0.017 secs.)
NumPy mean is 290.270481329 times faster than manual for loops.
```

- It's about 290 times faster than the manual for loops.

```
15 def manual_mean(arr):
16     """Compute mean (average) of all elements in the given 2D array."""
17     sum = 0
18     for i in xrange(0, arr.shape[0]):
19         for j in xrange (0, arr.shape[1]):
20             sum = sum + arr[i, j]
21     return sum / arr.size
22
23
24 def numpy_mean(arr):
25     """Compute mean (average) using NumPy."""
26     return arr.mean()
27
28
29 def test_run():
30     """Function called by Test Run."""
31     nd1 = np.random.random((1000, 10000))  # use a sufficiently large array
32
33     # Time the two functions, retrieving results and execution times
34     res_manual, t_manual = how_long(manual_mean, nd1)
35     res_numpy, t_numpy = how_long(numpy_mean, nd1)
```

- Observe that Num-Py not only makes the code more cleaner as compared to the manual method, but there's about 290 times faster than other method.
- Don't you think it's just awesome?

# Accessing Array Elements

```
run.py   util.py
1   """Accessing array elements."""
2
3   import numpy as np
4
5
6   def test_run():
7       a = np.random.rand(5, 4)
8       print "Array:\n", a
9
10      # Accessing element at position (3, 2)
11      element = a[3, 2]
12      print element
13
14
15
16
17  if __name__ == "__main__":
18      test_run()
```

- Accessing array elements is straightforward.
- You can access a particular element by referring to its row and  column number inside the square brackets.
-  The first integer over here denotes to the row number and  the second integer denotes to the column number.
- Let's see which element do we get at position 3,2.

```
Array:
[[ 0.42456764  0.00308835  0.51016443  0.50257536]
 [ 0.29642501  0.96737159  0.89500169  0.93181217]
 [ 0.40034064  0.64298823  0.92292136  0.71617354]
 [ 0.52293447  0.1247283   0.42998973  0.22859047]
 [ 0.32607692  0.9652539   0.89649885  0.36870615]]
0.42998973222
```

- Observe that the element we get actually belongs to the fourth row and  the third column, but note that the row and the column indexing start from zero.
- Hence if you want an element of the fourth row and tall column,  you pass the parameter as we did, that is 3,2.
- Now let's do some interesting stuff, accessing elements and ranges.

```
Array:
[[ 0.15399079  0.30851642  0.42457825  0.66934838]
 [ 0.07024643  0.25714868  0.82320407  0.5142289 ]
 [ 0.69908733  0.02825044  0.87399002  0.59096267]
 [ 0.01352434  0.14606483  0.42761314  0.00098667]
 [ 0.23445617  0.71847992  0.00585441  0.25825431]]
```

- If I would want to access elements from first through third column in the zeroeth row, here is how I would do it.

```
run.py   util.py

1   """Accessing array elements."""
2
3   import numpy as np
4
5
6   def test_run():
7       a = np.random.rand(5, 4)
8       print "Array:\n", a
9
10      # Accessing element at position (3, 2)
11      element = a[3, 2]
12
13      # Elements in defined range
14      print a[0, 1:3]
15
```

- This operation is called slicing, as explained before using data frame.
-  Let's read out this slicing operation.  For the 0 through,  get values from first through third column excluding the third column.
- Now let's run this.  So here's the output.

```
Array:
[[ 0.67458856  0.55414156  0.72609102  0.83560772]
 [ 0.49188453  0.47962849  0.10215371  0.89388678]
 [ 0.82429163  0.41124161  0.03806624  0.66483233]
 [ 0.13846567  0.4360862   0.22763106  0.98618785]
 [ 0.44040658  0.07401505  0.35928538  0.99661418]]
 [ 0.55414156  0.72609102]
```

- For the 0 through, first through the third column excluding the third column.  This was just column slicing.  We can combine row and column slicing and get a subset of the array.

```
3  import numpy as np
4
5
6  def test_run():
7      a = np.random.rand(5, 4)
8      print "Array:\n", a
9
10     # Top-left corner
11     print a[0:2, 0:2]
12
13
14
15
16  if __name__ == "__main__":
17      test_run()
18
```

- If I would like to access the top left corner of the array, I would do this as follows. We can combine row and column slicing and get the subset of the array.

```
Array:
[[ 0.33734051  0.38865194  0.51797224  0.56832433]
 [ 0.61326692  0.63539064  0.11585872  0.6545827 ]
 [ 0.51575178  0.84259784  0.64580402  0.71337171]
 [ 0.20828728  0.51970248  0.22949974  0.83624597]
 [ 0.24174786  0.64931855  0.9207263   0.03564511]]
[[ 0.33734051  0.38865194]
 [ 0.61326692  0.63539064]]
```

- Here is the top left corner, which has elements at position 0-0, 0-1, 1-0, and 1-1.

run.py   util.py

```
1   """Accessing array elements."""
2
3   import numpy as np
4
5
6   def test_run():
7       a = np.random.rand(5, 4)
8       print "Array:\n", a
9
10      # Slicing
11      # Note: Slice n:m:t specifies a range that starts at n, and stops before m, in
12      print a[:, 0:3:2]   # will select columns 0, 2 for every row
13
14
15
16
17  if __name__ == "__main__":
18      test_run()
19
```

- One last interesting thing in slicing, which I would like to bring in front of you.

- You see a lot of numbers over here, so let's break it down and read it.
- The three number separated by the colon, this is not accessing the third access, but a slicing of the form, n is to m is to t, will give you values in the range n before m, but in steps of size t.
- Hence this statement will give you values of the column 0, skip the values of the column one, and then give the values of the column 2. Let's run this.

```
Array:
[[  5.97332254e-01   2.95243106e-02   7.96007236e-01   2.18685405e-01]
 [  2.45797443e-01   2.22397065e-01   8.66750108e-01   8.08763276e-01]
 [  7.21944658e-01   8.95130783e-01   1.45609511e-01   6.78955909e-01]
 [  6.92727020e-04   3.91098996e-01   8.80794543e-01   9.03043808e-01]
 [  8.51591922e-01   8.47337152e-01   8.27649217e-01   6.06688882e-01]]
[[  5.97332254e-01   7.96007236e-01]
 [  2.45797443e-01   8.66750108e-01]
 [  7.21944658e-01   1.45609511e-01]
 [  6.92727020e-04   8.80794543e-01]
 [  8.51591922e-01   8.27649217e-01]]
```

- As explained, you get the 0, and the second column with all the rows. Seems like magic, right?

# Modifying Array Elements

```
run.py    util.py

1   """Accessing array elements."""
2
3   import numpy as np
4
5
6   def test_run():
7       a = np.random.rand(5, 4)
8       print "Array:\n", a
9
10      # Assigning a value to a particular location
11      a[0, 0] = 1
12      print "\nModified (replaced one element):\n", a
13
14
15
16
17  if __name__ == "__main__":
18      test_run()
```

- Moving forward. It is good that we can access elements in a, but another important operation is assigning values to specific location in a.
- This will give us access to the element at the position 0, 0 in the a.
- Using the assignment operator, we can assign a value one to it. Let's see the output.

```
Array:
[[ 0.22458503  0.58638835  0.10985301  0.20615276]
 [ 0.33131889  0.63563917  0.9642139   0.76662268]
 [ 0.84790102  0.62288425  0.79558522  0.64312203]
 [ 0.76259293  0.62325317  0.62318946  0.92514925]
 [ 0.60299002  0.36039706  0.61649769  0.4701966 ]]

Modified (replaced one element):
[[ 1.          0.58638835  0.10985301  0.20615276]
 [ 0.33131889  0.63563917  0.9642139   0.76662268]
 [ 0.84790102  0.62288425  0.79558522  0.64312203]
 [ 0.76259293  0.62325317  0.62318946  0.92514925]
 [ 0.60299002  0.36039706  0.61649769  0.4701966 ]]
```

- Here you go. This is the original array where we replaced the element at 00 with 1.
- But that's not all, with a minor change you can assign a value to the entire row or column. Let's do it.

```
run.py   util.py

1  """Accessing array elements."""
2
3  import numpy as np
4
5
6  def test_run():
7      a = np.random.rand(5, 4)
8      print "Array:\n", a
9
10     # Assigning a single value to an entire row
11     a[0,:] = 2
12     print "\nModified (replaced a row with a single value):\n", a
13
14
15
16
17  if __name__ == "__main__":
18      test_run()
19
```

- This will give us access to the 0 true and we assign the entire row a value of 2. Let's run this.

```
Array:
[[ 0.13099195  0.01263959  0.47709207  0.20407478]
 [ 0.08299993  0.36021041  0.01930437  0.80149915]
 [ 0.72799634  0.18662185  0.66226836  0.44974558]
 [ 0.47251976  0.9138894   0.20183727  0.68636864]
 [ 0.08619194  0.94594588  0.21750898  0.61772761]]

Modified (replaced a row with a single value):
[[ 2.          2.          2.          2.         ]
 [ 0.08299993  0.36021041  0.01930437  0.80149915]
 [ 0.72799634  0.18662185  0.66226836  0.44974558]
 [ 0.47251976  0.9138894   0.20183727  0.68636864]
 [ 0.08619194  0.94594588  0.21750898  0.61772761]]
```

- Using similar operation and column slicing, we can also assign an entire column a single values.
- Now what if we need each column or row to have different value and not the same as we did over here?
- Let's see how we can achieve this.

```
run.py   util.py

1   """Accessing array elements."""
2
3   import numpy as np
4
5
6   def test_run():
7       a = np.random.rand(5, 4)
8       print "Array:\n", a
9
10      # Assigning a list to a column in an array
11      a[:, 3] = [1, 2, 3, 4, 5]
12      print "\nModified (replaced a column with a list):\n", a
13
14
15
16
17  if __name__ == "__main__":
18      test_run()
19
```

- Yes, this is a list of values.  You can assign a list of values to a row or a column.
- Here we assign this list of values to column number three, but  make sure you keep an eye on the dimension.  That is, for this example,  if you have five rows in an area the list should have five elements.

```
Array:
[[ 0.36608297  0.75550753  0.91662172  0.49050069]
 [ 0.63073565  0.82895227  0.33608152  0.11176744]
 [ 0.49860528  0.68474105  0.75710413  0.07301984]
 [ 0.94742807  0.22220841  0.0955131   0.41071771]
 [ 0.37570482  0.90943729  0.11516962  0.18618816]]

Modified (replaced a column with a list):
[[ 0.36608297  0.75550753  0.91662172  1.        ]
 [ 0.63073565  0.82895227  0.33608152  2.        ]
 [ 0.49860528  0.68474105  0.75710413  3.        ]
 [ 0.94742807  0.22220841  0.0955131   4.        ]
 [ 0.37570482  0.90943729  0.11516962  5.        ]]
```

- As you can see, all the list values have been assigned to the third column.  So now it's your turn. Go ahead and try more row and column slicing.

# Indexing an Array with Another Array

```
1  ''' Accessing elements '''
2
3  import numpy as np
4
5
6  def test_run():
7
8      a = np.random.rand(5)
9
10     #accessing using list of indices
11     indices = np.array([1,1,2,3])
12     print a[indices]
13
14
15 if __name__ == "__main__":
16     test_run()
```

- There are various options of indexing.  And that gives NumPy indexing great power.
-  NumPy array can be indexed with other arrays.  It is just one other tool which can be used to make  process of accessing values in array easy.
- To start with, we create a one dimensional array of five random values.  Next we create a variable indices which is also a one dimensional array.  But the elements of this array, which is 1, 1, 2, 3,  are actually the index we need to access.  That is, we want the value at index 1, again at 1, 2, followed by 3.
- Next step we learn how to use this indices along with the edit  to access the desired values.  Yes, you saw that right.  Just passing the area of indices to another area will give us the values.  Let's check the output.

```
11
12     #accessing using list of indices
13     indices = np.array([1,1,2,3])
14
15     print a[indices]
16
17
18 if __name__ == "__main__":
19     test_run()
```

| Reset | Test Run |
|-------|----------|

```
[ 0.77411457  0.69220443  0.79423104  0.584985    0.86189932]
[ 0.69220443  0.69220443  0.79423104  0.584985   ]
```

- So here is the output.  Observe that the length of the indices array and  the returned array is the same.  Also it return value from array a at index 1,1,2,3.

- It is a bit difficult to understand the application of this now, but this is a tool you would like to use once you get hold of it. We can do such indexing using multidimensional array as well. But things get complicated with creating multi-dimensional index array.
- These are just a few interesting ways of indexing. There are a lot more out there for you to experiment with. Check the link in the instructor notes.

# Boolean or "Mask" Index Arrays

```
1  ''' Acessing elements '''
2
3  import numpy as np
4
5
6  def test_run():
7      a = np.array([(20,25,10,23,26,32,10,5,0),(0,2,50,20,0,1,28,5,0)])
8      print a
9
10     #calculating mean
11     mean = a.mean()
12     print mean
13
14 if __name__ == "__main__":
15     test_run()
```

- Next we will be working with boolean arrays.  In simple terms arrays with values true and false. This can also be used for indexing.
- Indexing using boolean arrays is very different as compared to index arrays,  we learned previously.
- Imagine a situation where we want to get all the values from the array,  which is less than mean of the entire array.
- The first step to solve this problem would be to calculate the mean.  Consider a two dimensional array.  As we learned before,  we will calculate the mean using the mean attribute of the array a. Let's check what the mean is.

```
[[20 25 10 23 26 32 10  5  0]
 [ 0  2 50 20  0  1 28  5  0]]
14.2777777778
```

- According to our problem,  we want all the values from the area which is less than mean.  Mean is 14.2.  You can imagine that the solution would contain values 10, 10 again, 5, 0.
- If you need all these values, one way is to run the for  loop over the array, and get them.  But using masking, we do this in one single line.

```
1  ''' Acessing elements '''
2
3  import numpy as np
4
5
6  def test_run():
7      a = np.array([(20,25,10,23,26,32,10,5,0),(0,2,50,20,0,1,28,5,0)])
8      print a
9
10     #calculating mean
11     mean = a.mean()
12     print mean
13
14     #masking
15     print a[a<mean]
16
17 if __name__ == "__main__":
18     test_run()
```

- To read this operation, it would be for each value in array A, compare it with the mean. If it is less, we retain the value. Let's check the output.

```
14     #masking
15     print a[a<mean]
16
17 if __name__ == "__main__":
18     test_run()
```

Reset    Test Run

```
[[20 25 10 23 26 32 10  5  0]
 [ 0  2 50 20  0  1 28  5  0]]
14.2777777778
[10 10  5  0  0  2  0  1  5  0]
```

- Here is the values which we expected in the form of list. Now to go ahead with this concept, we can also replace these values with the mean value.

```
1  ''' Acessing elements '''
2
3  import numpy as np
4
5
6  def test_run():
7      a = np.array([(20,25,10,23,26,32,10,5,0),(0,2,50,20,0,1,28,5,0)])
8      print a
9
10     #calculating mean
11     mean = a.mean()
12     print mean
13
14     #masking
15     a[a<mean]= mean
16     print a
17
18 if __name__ == "__main__":
19     test_run()
```

- We just assign the mean value to masking operation we performed before.  Let's see the output.

```
15     a[a<mean]= mean
16     print a
17
18 if __name__ == "__main__":
19     test_run()
```

Reset    Test Run

```
[[20 25 10 23 26 32 10  5  0]
 [ 0  2 50 20  0  1 28  5  0]]
14.2777777778
[[20 25 14 23 26 32 14 14 14]
 [14 14 50 20 14 14 28 14 14]]
```

- Observe that all the values previously less than mean  have been replaced by the mean.  This is one of the important operation that shows the power of the and  justifies its extensive use throughout.

# Arithmetic Operations

```
run.py   util.py

1  """Arithmetic operations."""
2
3  import numpy as np
4
5  def test_run():
6      a = np.array([(1, 2, 3, 4, 5), (10, 20, 30, 40, 50)])
7      print "Original array a:\n", a
8
9      # Multiply a by 2
10     print "\nMultiply a by 2:\n", 2 * a
11
12
13
14
15 if __name__ == "__main__":
16     test_run()
```

- Arithmetic operations on arrays are always applied element wise.  Let's start with simple multiplication operation.
- Here we define a simple array so that we can easily track the changes.  This will multiply each element by 2.  Let's check the output.

```
Original array a:
[[ 1  2  3  4  5]
 [10 20 30 40 50]]

Multiply a by 2:
[[  2   4   6   8  10]
 [ 20  40  60  80 100]]
```

- When using arithmetic operation, a new array is created and  the values are stored in that array.  So our original array a still holds the same values.
- And this is our new array which we get after multiplying array a by two.  Observe that, it is element wise multiplication.

```
run.py   util.py

1  """Arithmetic operations."""
2
3  import numpy as np
4
5  def test_run():
6      a = np.array([(1, 2, 3, 4, 5), (10, 20, 30, 40, 50)])
7      print "Original array a:\n", a
8
9      # Divide a by 2
10     print "\nDivide a by 2:\n", a / 2
11
12
13
14
15 if __name__ == "__main__":
16     test_run()
17
```

- Let's try division.  Here we use the division operation to divide each element of array a by two. Let's check the output.

```
Original array a:
[[ 1  2  3  4  5]
 [10 20 30 40 50]]

Divide a by 2:
[[ 0  1  1  2  2]
 [ 5 10 15 20 25]]
```

- Observe that, when you divide 1 by 2,  you get a value 0 over here instead of 0.5.  This is because both the array and the divisor are integers.

```
run.py   util.py

1  """Arithmetic operations."""
2
3  import numpy as np
4
5  def test_run():
6      a = np.array([(1, 2, 3, 4, 5), (10, 20, 30, 40, 50)])
7      print "Original array a:\n", a
8
9      # Divide a by 2
10     print "\nDivide a by 2:\n", a / 2.0
11
12
13
14
15 if __name__ == "__main__":
16     test_run()
17
```

- If we were to do 2.0 instead of 2, you will get float values.

- Keep this point in mind before performing division in general. That is int divided by int will give you an integer output.
- To get float values, you need at least, the numerator or the denominator to be a float value. Let's check the output.

```
Original array a:
[[ 1  2  3  4  5]
 [10 20 30 40 50]]

Divide a by 2:
[[ 0.5   1.    1.5   2.    2.5]
 [ 5.   10.   15.   20.   25. ]]
```

- Observe that, we could successfully get a floating point value instead of 0.

```
run.py   util.py

1   """Arithmetic operations."""
2
3   import numpy as np
4
5   def test_run():
6       a = np.array([(1, 2, 3, 4, 5), (10, 20, 30, 40, 50)])
7       print "Original array a:\n", a
8
9       b = np.array([(100, 200, 300, 400, 500), (1, 2, 3, 4, 5)])
10      print "\nOriginal array b:\n", b
11
12      # Add the two arrays
13      print "\nAdd a + b:\n", a + b
14
15
16
17
18  if __name__ == "__main__":
19      test run()
```

- How about arithmetic operation using two arrays? We will start with addition. We create another array b with these values.
- Now, let's just add a and b using the plus operator. As mentioned, this is element wise.

```
Original array a:
[[ 1  2  3  4  5]
 [10 20 30 40 50]]

Original array b:
[[100 200 300 400 500]
 [  1   2   3   4   5]]

Add a + b:
[[101 202 303 404 505]
 [ 11  22  33  44  55]]
```

- This is our new array a plus b.  One important thing to note here is that the shape of a and  b should be similar before the operation a plus b, else it will throw error.
- Similar to the addition, you can perform subtraction.

```
Original array a:
[[ 1  2  3  4  5]
 [10 20 30 40 50]]

Original array b:
[[100 200 300 400 500]
 [  1   2   3   4   5]]
```

- Now, let's move ahead with multiplying two arrays.  This is interesting, because unlike other many metrics languages  multiplication operator when used with two array  will not give you matrix product, but will do element wise multiplication.  That is, element at position 0,0 in array a is multiplied only with the element at position 0,0 in b.

```
run.py   util.py
1   """Arithmetic operations."""
2
3   import numpy as np
4
5   def test_run():
6       a = np.array([(1, 2, 3, 4, 5), (10, 20, 30, 40, 50)])
7       print "Original array a:\n", a
8
9       b = np.array([(100, 200, 300, 400, 500), (1, 2, 3, 4, 5)])
10      print "\nOriginal array b:\n", b
11
12      # Multiply a and b
13      print "\nMultiply a and b:\n", a * b
14
15
16
17
18  if __name__ == "__main__":
19      test_run()
```

- Let's print the multiplication of matrix a and b. Can you observe element-wise multiplication?
- But the next question would be, what about matrix multiplication? How do you achieve that? Like, for everything, NumPy has a function. It has function called dot, which performs matrix multiplication.

```
run.py   util.py
1   """Arithmetic operations."""
2
3   import numpy as np
4
5   def test_run():
6       a = np.array([(1, 2, 3, 4, 5), (10, 20, 30, 40, 50)])
7       print "Original array a:\n", a
8
9       b = np.array([(100, 200, 300, 400, 500), (1, 2, 3, 4, 5)])
10      print "\nOriginal array b:\n", b
11
12      # Divide a by b
13      print "\nDivide a by b:\n", a / b
14
15
16
17
18  if __name__ == "__main__":
19      test_run()
```

- Similar to multiplication, division of two arrays can be performed. Just include division of operators between the two arrays. Let's check the output.

```
Original array a:
[[ 1  2  3  4  5]
 [10 20 30 40 50]]

Original array b:
[[100 200 300 400 500]
 [  1   2   3   4   5]]

Divide a by b:
[[ 0  0  0  0  0]
 [10 10 10 10 10]]
```

- As seen before, since array a and b are integers, we get the final array in the form of integers as well. If you want to see floating values, convert one of the arrays to float. Well, that's all for now. Keep practicing.

# Learning More NumPy

- All the operations and functions explained in this lesson are those which will help you perform computation.
- But there is a lot more to learn. Check out the link in the instructor notes. I will meet you in the next lesson with some more coding.
- Happy coding with Python and bye until then.