CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computer Science

http://www.cit.ie

Programming for Data Analytics

Introduction to Numpy

Dr. Mohammed Hasanuzzaman
Room J102, Melbourne Building (Office)
e-mail:mohammed.hasanuzzaman@cit.ie
Website: https://mohammedhasanuzzaman.github.io

Introduction to Numpy

- **NumPy** is an open-source add-on module to Python that provides routines for **manipulating large arrays** and matrices of numeric data in **pre-compiled**, fast functions.

- At the core of the NumPy package, is the **Ndarray** object.
  - This encapsulates n-dimensional arrays of **homogeneous** data types, with many operations being performed in compiled code for performance.

- There are several important differences between NumPy arrays and the standard Python lists
  - NumPy arrays have a **fixed size** at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will **create a new array** and delete the original.
  - The elements in a NumPy array are all required to be of the **same data type**.

Introduction

- NumPy arrays facilitate a range of mathematical and other types of operations on large amounts of data.
  - Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences.
  - Further, NumPy implements an array language, so that it attempts to **minimise the need for loops**.

- NumPy arrays make operations with large amounts of numeric data very fast and are generally <u>much</u> more efficient than lists.

Click here to view NumPy documentation

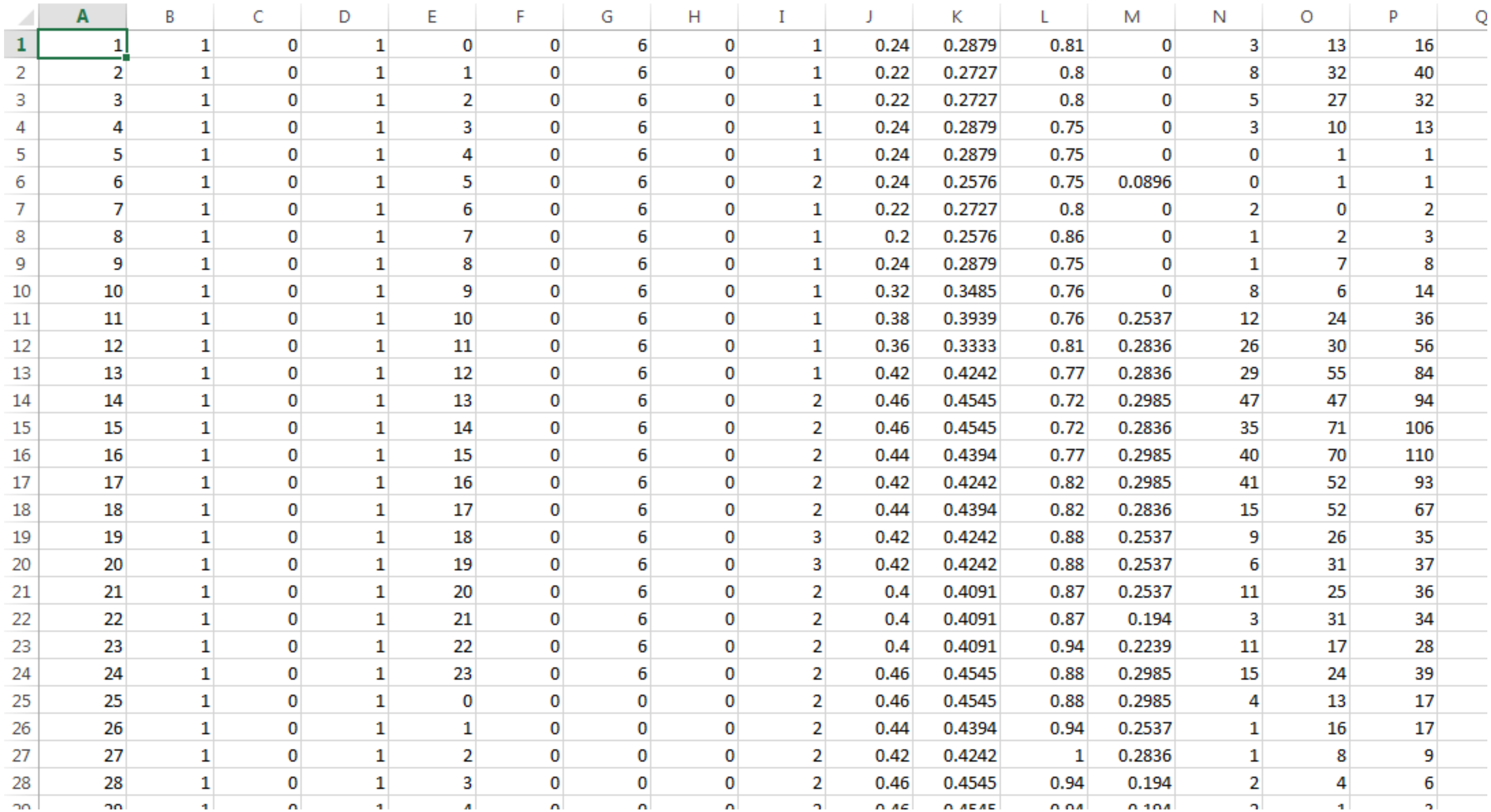
```
import numpy as np
import time

startPython = int(round(time.time() * 1000))
# Creates a list c by adding the elements of a and b
a = list(range(10000000))
b = list(range(10000000))
c = []
for i in range(len(a)):
  c.append(a[i] + b[i])
stopPython = int(round(time.time() * 1000))

# Below code does exactly the same thing but with Numpy Arrays
a = np.arange(10000000, dtype=int)
b = np.arange(10000000, dtype=int)
c = a + b

stopNumPy = int(round(time.time() * 1000))

# Here we express the time take by NumPy as a fraction of the time taken
# by Python to perform the same operation
print ((stopNumPy- stopPython)/(stopPython-startPython))
```

This code can take up to 5 seconds to run depending on the type of machine your use

The Numpy code by comparison is shorter, doesn’t need a loop and it is almost instantaneous by comparison

Bike Sharing Dataset

1. instant: record index
2. season : season (1:springer, 2:summer, 3:fall, 4:winter)
3. yr : year (0: 2011, 1:2012)
4. mnth : month (1 to 12)
5. hr : hour (0 to 23)
6. holiday : weather day is holiday or not (extracted from [Web Link])
7. weekday : day of the week
8. workingday : if day is neither weekend nor holiday is 1, otherwise is 0.
9. + weathersit :
10. temp : Normalized temperature in Celsius. The values are divided to 41 (max)
11. atemp: Normalized feeling temperature in Celsius. The values are divided to 50 (max)
12. hum: Normalized humidity. The values are divided to 100 (max)
13. windspeed: Normalized wind speed. The values are divided to 67 (max)
14. casual: count of casual users
15. registered: count of registered users
16. cnt: count of total rental bikes including both casual and registered



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
1	1	1	0	1	0	0	6	0	1	0.24	0.2879	0.81	0	3	13	16	
2	2	1	0	1	1	0	6	0	1	0.22	0.2727	0.8	0	8	32	40	
3	3	1	0	1	2	0	6	0	1	0.22	0.2727	0.8	0	5	27	32	
4	4	1	0	1	3	0	6	0	1	0.24	0.2879	0.75	0	3	10	13	
5	5	1	0	1	4	0	6	0	1	0.24	0.2879	0.75	0	0	1	1	
6	6	1	0	1	5	0	6	0	2	0.24	0.2576	0.75	0.0896	0	1	1	
7	7	1	0	1	6	0	6	0	1	0.22	0.2727	0.8	0	2	0	2	
8	8	1	0	1	7	0	6	0	1	0.2	0.2576	0.86	0	1	2	3	
9	9	1	0	1	8	0	6	0	1	0.24	0.2879	0.75	0	1	7	8	
10	10	1	0	1	9	0	6	0	1	0.32	0.3485	0.76	0	8	6	14	
11	11	1	0	1	10	0	6	0	1	0.38	0.3939	0.76	0.2537	12	24	36	
12	12	1	0	1	11	0	6	0	1	0.36	0.3333	0.81	0.2836	26	30	56	
13	13	1	0	1	12	0	6	0	1	0.42	0.4242	0.77	0.2836	29	55	84	
14	14	1	0	1	13	0	6	0	2	0.46	0.4545	0.72	0.2985	47	47	94	
15	15	1	0	1	14	0	6	0	2	0.46	0.4545	0.72	0.2836	35	71	106	
16	16	1	0	1	15	0	6	0	2	0.44	0.4394	0.77	0.2985	40	70	110	
17	17	1	0	1	16	0	6	0	2	0.42	0.4242	0.82	0.2985	41	52	93	
18	18	1	0	1	17	0	6	0	2	0.44	0.4394	0.82	0.2836	15	52	67	
19	19	1	0	1	18	0	6	0	3	0.42	0.4242	0.88	0.2537	9	26	35	
20	20	1	0	1	19	0	6	0	3	0.42	0.4242	0.88	0.2537	6	31	37	
21	21	1	0	1	20	0	6	0	2	0.4	0.4091	0.87	0.2537	11	25	36	
22	22	1	0	1	21	0	6	0	2	0.4	0.4091	0.87	0.194	3	31	34	
23	23	1	0	1	22	0	6	0	2	0.4	0.4091	0.94	0.2239	11	17	28	
24	24	1	0	1	23	0	6	0	2	0.46	0.4545	0.88	0.2985	15	24	39	
25	25	1	0	1	0	0	0	0	2	0.46	0.4545	0.88	0.2985	4	13	17	
26	26	1	0	1	1	0	0	0	2	0.44	0.4394	0.94	0.2537	1	16	17	
27	27	1	0	1	2	0	0	0	2	0.42	0.4242	1	0.2836	1	8	9	
28	28	1	0	1	3	0	0	0	2	0.46	0.4545	0.94	0.194	2	4	6	
29	29	1	0	1	4	0	0	0	2	0.46	0.4545	0.94	0.194	2	1	3	

- Take a basic problem.
- If I want to calculate the average wind speed with normal file processing I would have to read through every line the in file and add the wind speed to a counter variable.

```
fileObj = open("bikeSharing.csv")
sumWind = 0
numOfRows = 0

for line in fileObj:
    lineContents = line.split(',')
    sumWind += float(lineContents[12])
    numOfRows += 1

print ("Average is ", (sumWind/numOfRows)*67)
```

Basic of NumPy - Arrays

- Arrays can be created from lists

```
import numpy as np

arr = np.array([5.5, 45.6, 3.2], float)
print (arr)
```

Above Code will Output => [5.5 45.6   3.2]

Adding Elements NumPy Arrays

- Can add individual elements to an array using **append**

```
import numpy as np


arr1 = np.array([5.5], float)
arr2 = np.append(arr1, 4.3)
print (arr2)
```

Above Code will Output => [5.5 4.3]

Adding Elements NumPy Arrays

- Can add a list to an array using append

```
import numpy as np


arr1 = np.array([5.5], float)
arr2 = np.append(arr1, [4.3, 6.4, 6])
print (arr2)
```

Above Code will Output => [5.5 4.3 6.4 6.]

Word of caution about using append or insert in large arrays!

NumPy - Arrays

- We can use numpy.concatenate to add **one NumPy array to another** Numpy Array
- Note: The concatenate will return a new independent copy of the data.

```
import numpy as np

arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = np.array([3.5, 4.1, 8.4], float)
arr3 = np.concatenate((arr1, arr2))
print (arr3)
```

Above Code will Output => [5.5 45.6   3.2   3.5   4.1   8.4]

NumPy - Arrays

- We can *insert* a value at a specific location
  - ***insert****(array, insertIndex, value)*

```
import numpy as np

arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = np.insert(arr1, 1, 99)
print (arr2)
```

Above Code will Output => [5.5 99. 45.6 3.2]

NumPy - Arrays

- We can *delete* a value at a specific location
  - ***delete****(array, insertIndex)*

```
import numpy as np

arr1 = np.array([5.5, 45.6, 3.2], float)
arr2 = np.delete(arr1, 1)
print (arr2)
```

Above Code will Output => [5.5 3.2]

NumPy - Arrays

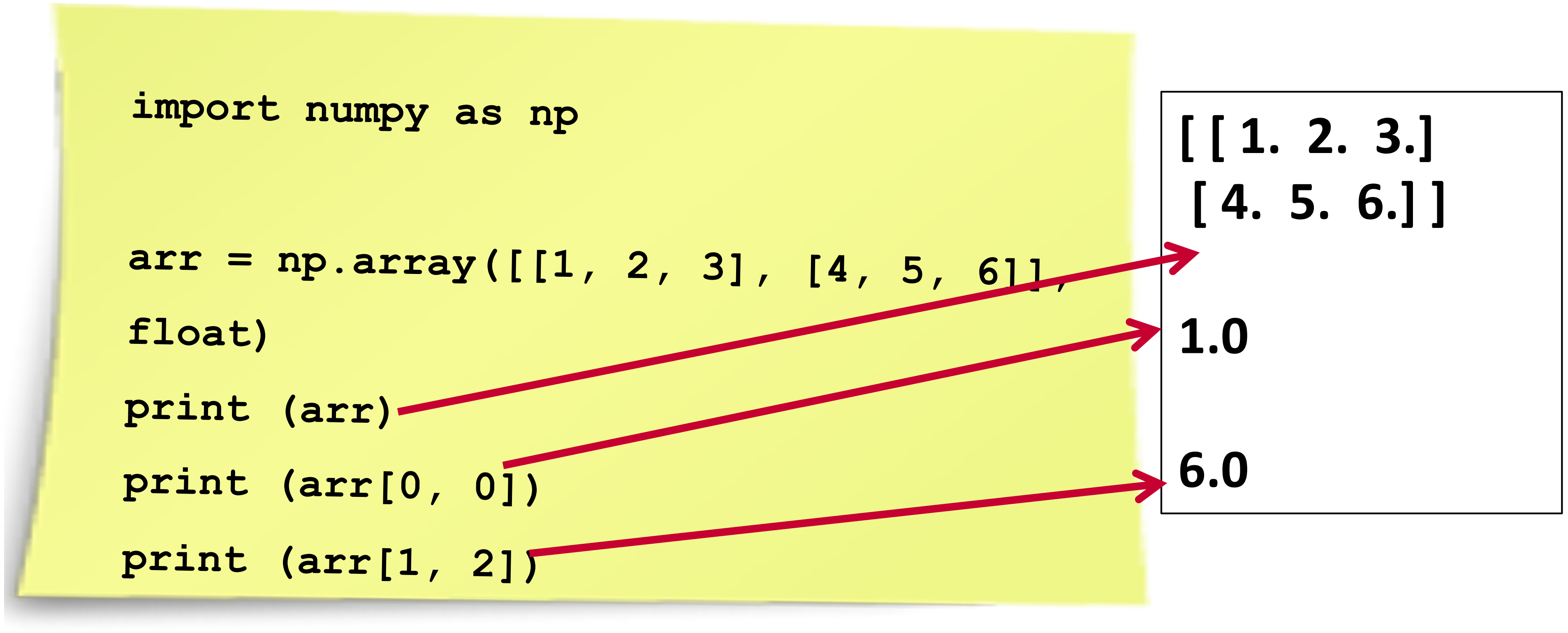
- Array can be accessed using the **square bracket notation** just as with a list

```
import numpy as np

arr = np.array([5.5, 45.6, 3.2], float)
print (arr[0])
arr[0] = 5
print (arr)
```

```
5.5

[ 5.  45.6  3.2]
```

NumPy – Multi-Dimensional Arrays

- Arrays can be multidimensional. Elements are accessed using **[row, column]** format inside bracket notation



```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)
print (arr[0, 0])
print (arr[1, 2])
```

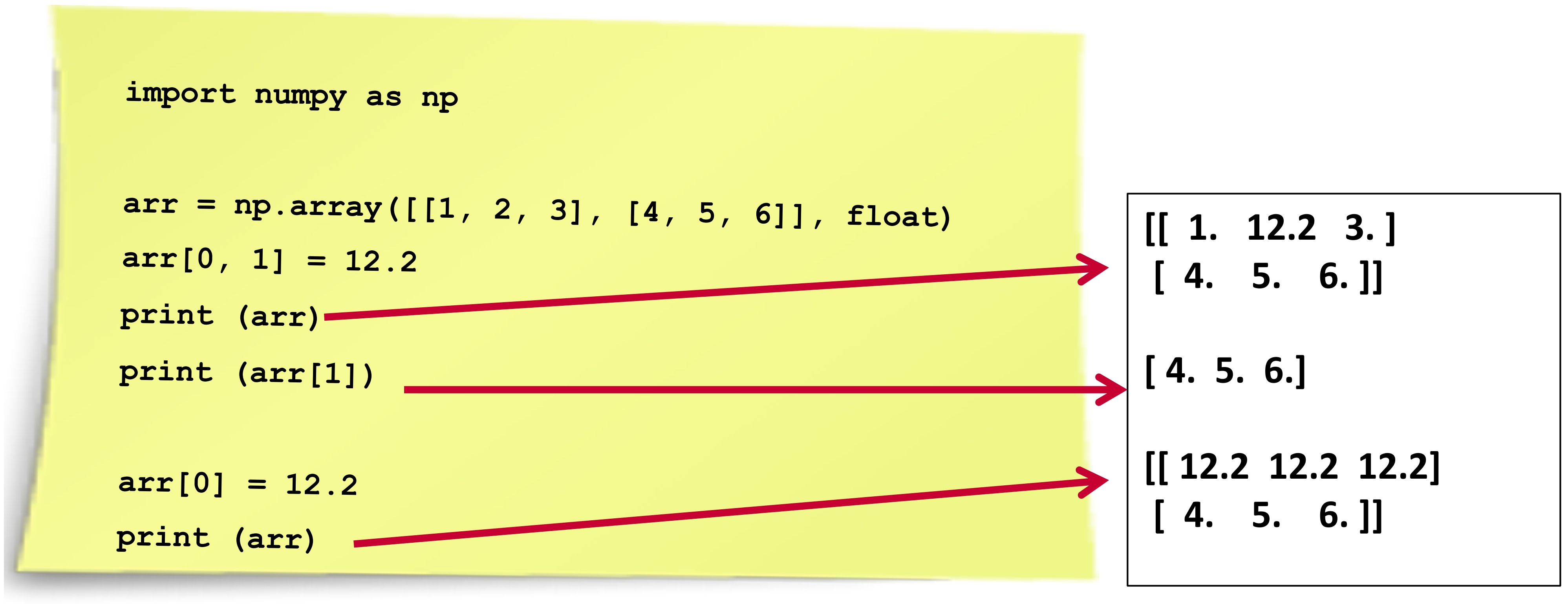
```
[[1.  2.  3.]
 [4.  5.  6.]]

1.0

6.0
```

`arr`

	0	1	2
0	1	2	3
1	4	5	6

NumPy – Single Index to 2D Array

- A single index value provided to a multi-dimensional array will refer to an entire row.



```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)
arr[0, 1] = 12.2
print (arr)
print (arr[1])

arr[0] = 12.2
print (arr)
```

```
[[  1.   12.2   3. ]
 [  4.    5.    6. ]]

[ 4.  5.  6.]

[[ 12.2  12.2  12.2]
 [  4.    5.    6. ]]
```

NumPy – Slicing Operations

- We can use slicing effectively to extract a subset of items from a NumPy array. We can slice along each axis.

NumPy – Slicing Operations

- We can also extract a single column using slicing.
- Use of a single ":" in a dimension indicates the use of everything along that dimension

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr2 = arr[:,0]
print (arr2)

arr[:,0] = 56

print (arr)
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

[ 1.  4.]

[[ 56.   2.   3.]
 [ 56.   5.   6.]]
```

Slicing with MD Arrays

- The following code will retrieve the elements in row 0 and 1 and the elements in column 0 and 1

```
import numpy as np

arr = np.array([[14.4, 2.4, 3.5], [54.3, 34.4,
98.22], [100, 200, 300]], float)
print (arr)

print (arr[0:2, 0:2])
```

```
[[  14.4     2.4     3.5 ]
 [  54.3    34.4    98.22]
 [ 100.    200.    300.  ]]

[[ 14.4   2.4]
 [ 54.3  34.4]]
```

NumPy – Multi-Dimensional Arrays

- Notice that we can select a number of columns from a matrix.
- When we select a single column, NumPy flattens the array to again be one dimensional

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [4, 5, 6, 7], [7, 8,
9, 10], [10, 11, 12, 13]], float)
print (arr)

arr2 = arr[:, 1:3]
print (arr2)

arr2 = arr[:,3]
print (arr2)
```

```
[[ 1.  2.  3.  4.]
 [ 4.  5.  6.  7.]
 [ 7.  8.  9. 10.]
 [ 10. 11. 12. 13.]]

[[ 2.  3.]
 [ 5.  6.]
 [ 8.  9.]
 [ 11. 12.]]

[ 4.  7. 10. 13.]
```

NumPy – Multi-Dimensional Arrays

We can use the same notation as an alternative way to access a specific row

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]], float)
print (arr)

arr2 = arr[1,:]
print (arr2)
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]


[ 4.  5.  6.]
```

Use of len Function in M-D Arrays

- len function can be used to obtain the number of rows or the number of columns
  - len of 2D array will return the number of rows
  - len of 2D row will return the number of columns within that row

```
import numpy as np

arr = np.array([[14.4, 2.4, 56.4], [54.3, 34.4,
98.22]], float)

print (len(arr))
print (len(arr[0]))
```

```
2

3
```

Important consideration when slicing!!

- When we use slicing on **lists**, it returns a new list.
- However, when performing slicing on a NumPy array it will return **a view of the original array**. In other words while it is a subset of the array it is still pointing at the same data in memory as the original array.
- Technically, that means that the data of both objects is *shared*

```
import numpy as np

data = np.array([[1, 2, 3], [2, 4, 5], [4, 5, 7],
[6, 2, 3]], float)

resultA = data[:,0]
resultA[0] = 200
print (data)
```

```
[[ 200.    2.    3.]
 [   2.    4.    5.]
 [   4.    5.    7.]
 [   6.    2.    3.]]
```

Copying a NumPy Object

- To copy a NumPy array use **NumPy.copy** function, which takes in as an argument the array you want to copy

```
data = np.array([[1, 2, 3], [2, 4,
5], [4, 5, 7], [6, 2, 3]], float)

dataCopyA = data
dataCopyA[0,0] = 200

print (data)
```

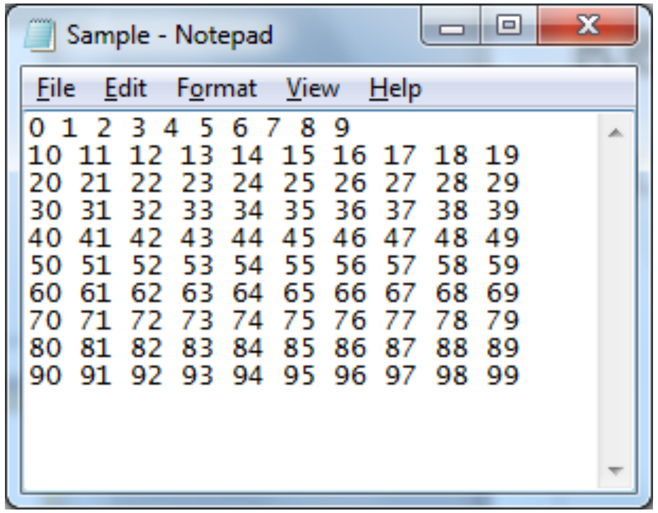
```
[[ 200.   2.   3.]
 [   2.   4.   5.]
 [   4.   5.   7.]
 [   6.   2.   3.]]
```

```
data = np.array([[1, 2, 3], [2, 4,
5], [4, 5, 7], [6, 2, 3]], float)

dataCopyA = np.copy(data)
dataCopyA[0,0] = 200

print (data)
```

```
[[ 1.  2.  3.]
 [ 2.  4.  5.]
 [ 4.  5.  7.]
 [ 6.  2.  3.]]
```

Reading Data From a File

- The code below shows how to read that data depicted in the file Sample.txt, into a two-dimension array.
- *np.genfromtxt* uses *dtype=float* by default
- The function had a broad range of parameters. One import parameter is called the delimiter and allows you to specify the delimiter used in the file you are reading.



```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

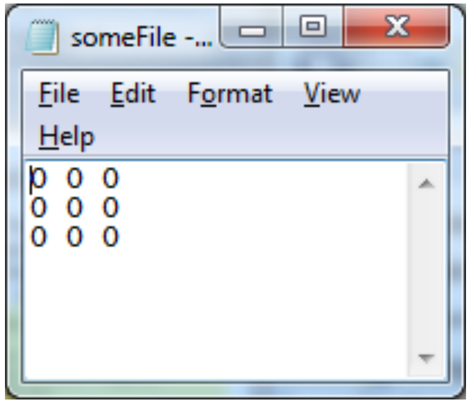
```python
import numpy as np


data = np.genfromtxt('Sample.txt', dtype=int)
print (data[0,0])
print (data[1,0])
print (data[2,0])
print (data[3,0])
```

```
0
10
20
30
```

Writing Data To a File

- Use savetxt method to save data from an array to a file. The code below saves the 2D array data to the file called someFile.txt.

```
import numpy as np

data = np.zeros((3,3))
np.savetxt("someFile.txt", data)
```



```
0 0 0
0 0 0
0 0 0
```

Use of in statement in M-D Arrays

- The *in* statement can be used to test if values are present in an array:

```
import numpy as np

arr = np.array([[14.4, 2.4, 56.4], [54.3, 34.4, 98.22]], float)

if 2.4 in arr:
    print ("Found")
else:
    print ("Not Found")
```

Other Ways to Create Arrays

- The *arange* function is similar to the range function but returns an numpy array. However, unlike the range function, the arrange function immediately create the NumPy array.
- Only possible to create 1D array with arange
    - But we can reshape arrays (see next slide)

```
import numpy as np

arr1 = np.arange(5, dtype=float)
arr2 = np.arange(1, 6, 2, dtype=int)

print (arr1)
print (arr2)
```

```
[ 0.  1.  2.  3.  4.]

[1, 3, 5]
```

Reshaping Arrays

- Arrays can be reshaped by specifying new dimensions.
  - In the example, we turn a ten-element one-dimensional array into a two-dimensional one whose first axis has 10 elements and whose second axis has 10 elements:
  - The capacity of the new array must match exactly the capacity of the original array.

```
import numpy as np


arr1 = np.arange(0,100, dtype=float)
arr2 = arr1.reshape((10, 10))
print (arr2)
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.  8.  9.]
 [ 10. 11. 12. 13. 14. 15. 16. 17. 18. 19.]
 [ 20. 21. 22. 23. 24. 25. 26. 27. 28. 29.]
 [ 30. 31. 32. 33. 34. 35. 36. 37. 38. 39.]
 [ 40. 41. 42. 43. 44. 45. 46. 47. 48. 49.]
 [ 50. 51. 52. 53. 54. 55. 56. 57. 58. 59.]
 [ 60. 61. 62. 63. 64. 65. 66. 67. 68. 69.]
 [ 70. 71. 72. 73. 74. 75. 76. 77. 78. 79.]
 [ 80. 81. 82. 83. 84. 85. 86. 87. 88. 89.]
 [ 90. 91. 92. 93. 94. 95. 96. 97. 98. 99.]]
```

Other Ways to Create Arrays

- The functions zeros and ones create new arrays of specified dimensions filled with these values
  - Commonly used functions to create new arrays

```
import numpy as np


arr1 = np.ones((2,3), dtype=float)
arr2 = np.zeros(7, dtype=int)
print (arr1)
print (arr2)
```

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

[0 0 0 0 0 0 0]
```

Exercise

- Use np.zeros to create an array with 12 rows and three columns. Print out the array.

- Next use np.reshape to convert it to an array with 6 rows and 6 columns

```
import numpy as np

arr = np.zeros((12, 3), float)

print (arr)

arr2 = np.reshape(arr, (6,6))

print (arr2)
```

Lab Exercises – Bike Sharing Scheme

Lets go back to our bike sharing exercise that we looked at earlier in the lecture.

If I want to calculate the average wind speed with normal file processing I would have to read through every line the in file and add the wind speed to a counter variable.

```
import numpy as np

data = np.genfromtxt('bikeSharing.csv', dtype=float, delimiter =
',')

count = data[:, 12]

total = 0.0
for value in count:
    total += value

print ("Average wind is ", total/len(count)*67)
```

NumPy – Appending to MD Arrays

- We can add elements using append to MD arrays in NumPy

  - numpy.**append**(*arr*, *values*, *axis=None*)
    - arr - Values are appended to a copy of this array.
    - values - These values are appended to a copy of arr. It must be of the correct shape
    - axis = The axis along which values are appended. **If axis is not given, both arr and values are flattened before use.**
      - In Numpy dimensions are called axes.

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]])

print (arr1)
```

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

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

Notice the output array has been flattened. This is because no axis was specified

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis = ?)

print (arr1)
```

axis = 0 refers to the vertical axis

axis = 1 refers to the horizontal axis

Dimension of values being added must be same as the specific axis we are adding to

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis = 0)

print (arr1)
```

Add a row containing the values [7, 8, 9] to axis = 0

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

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

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)

arr1 = np.append(arr, [[7, 8, 9]], axis = 1)

print (arr1)
```

Add a column containing the values [7, 8, 9] to axis = 1

Generates an error specifying array dimensions don't match because each column only contains two values (not three)

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]
```

NumPy – Multi-Dimensional Arrays

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]],
float)
print (arr)

arr1 = np.append(arr, [[7], [8]], axis = 1)

print (arr1)
```

Notice we use two [] brackets, that is because we are adding a single column element to each row

```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

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

Obtain Max or Min in an Array

- numpy.amin(array, axis)
  - Return the minimum of an array or minimum along an axis.
- numpy.amax(array, axis)
  - Return the maximum of an array or maximum along an axis.

```
import numpy as np

arr1 = np.array([10,20,30], float)
arr2 = np.array([1,2,3], float)

print (np.amax(arr1))
print (np.amax(arr2))
```

```
30.0
3.0
```

Obtain Max or Min in an Array

- For multi-dimensional arrays we can specify the axis.
  - If we don’t specify the axis it will determine the maximum for the entire array
- The way to understand it is whichever axis you are using will be 'collapsed' into the shape of the array. If axis is 0 the collapse is down to rows.

```
import numpy as np

arr1 = np.array([[10,20,30],[50, 60, 10]], float)
print (arr1)

print (np.amax(arr1))

print (np.amax(arr1, axis=0))

print (np.amax(arr1, axis=1))
```

```
[[ 10.  20.  30.]
 [ 50.  60.  10.]]

60.0

[ 50.  60.  30.]

[ 30.  60.]
```

Basic Array Operations

- Many functions exist for extracting whole-array properties.
- The items in an array can be summed or multiplied:
- These functions can be performed on multi-dimensional arrays
  - We can also provide an additional element of the axis we wish to access

```
import numpy as np

arr1 = np.array([[1, 2, 4],[3, 4, 2]], float)
print (arr1)
print (np.sum(arr1))
print (np.product(arr1))
print (np.sum(arr1, axis = 0))
print (np.mean(arr1, axis = 1))
```

```
[[ 1.  2.  4.]
 [ 3.  4.  2.]]

16.0

192.0

[ 4.  6.  6.]

[ 2.33333333  3.  ]
```

Basic Array Operations

- A number of routines enable computation of statistical quantities in array datasets, such as the mean (average), variance, and standard deviation.
- You can specify an axis on these operations as well
- Large number of mathematical functional available.
- http://docs.scipy.org/doc/numpy/reference/routines.math.html
- http://docs.scipy.org/doc/numpy/reference/routines.statistics.html

```
import numpy as np

arr = np.array([2, 1, 9], float)
print (np.mean(arr))
print (np.var(arr))
print (np.std(arr))
```

```
4.0

12.6666666667

3.55902608401
```

Note you can also specify the axis that we wish to perform these operations on

Lab Exercises – Bike Sharing Scheme

Lets go back to our bike sharing exercise that we looked at earlier in the lecture.

If I want to calculate the average wind speed with normal file processing I would have to read through every line the in file and add the wind speed to a counter variable.

```
import numpy as np

data = np.genfromtxt('bikeSharing.csv', dtype=float, delimiter = ',')

print (np.mean(data[:, 12])*67)
```

Exercises

- Using NumPy solve the problem below.

- We want to see the max, min and mean number of causal (index 13) and registered users (index 14) from the bike sharing scheme.

```
Maximum Results
            Casual Users 367.0
            Registered Users 886.0
Minimum Results
            Casual Users 0.0
            Registered Users 0.0
Mean Results
            Casual Users 35.6762184245
            Registered Users 153.78686921
```

```
def getMeanData(data):
    resultMaxArr = np.amax(data, axis=0)
    resultMinArr = np.amin(data, axis=0)
    resultMeanArr = np.mean(data, axis=0)

    print ("Maximum Results")
    print ("\t Casual Users", resultMaxArr[13])
    print ("\t Registered Users", resultMaxArr[14])

    print ("Minimum Results")
    print ("\t Casual Users", resultMinArr[13])
    print ("\t Registered Users", resultMinArr[14])

    print ("Mean Results")
    print ("\t Casual Users", resultMeanArr[13])
    print ("\t Registered Users", resultMeanArr[14])
```

Data refers to a 2D array into which we have read all data from the file

CIT

Array Mathematical Operations

- When standard mathematical operations are used with two arrays, they are applied on an **element by-element** basis.
  - This means that the arrays should be the same size during addition, subtraction,
  - NumPy arrays support the typical range of operators +,-, *, /, %, **
  - An error is thrown if arrays don't match in size

Array Mathematical Operations

- Example

```
import numpy as np

arr1 = np.array([10,20,30], float)
arr2 = np.array([1,2,3], float)

print (arr1*arr2)
print (arr1/arr2)
print (arr1**arr2)
```

```
[ 10.  40.  90.]

[  10.  10.  10.]

[   10.    400.  27000.]
```

Array Mathematical Operations

- For two-dimensional arrays, multiplication remains element-wise and does not correspond to matrix multiplication.

```
import numpy as np

arr1 = np.array([[10,20], [30, 40]], float)
arr2 = np.array([[1,2], [3,4]], float)

print (arr1+arr2)
```

```
[[ 11.  22.]
 [ 33.  44.]]
```

Array Mathematical Operations

- In addition to the standard operators, NumPy and SciPy offer a large library of common mathematical functions that can be applied elementwise to arrays.
- abs, sign, sqrt, log2, log10, exp, sin, cos, tan, arcsin, arccos, arctan, floor, ceil, and rint, cumsum, degrees, etc

http://docs.scipy.org/doc/numpy/reference/routines.math.html

```
import numpy as np


arr1 = np.array([[10,20], [30, 40]], float)
arr2 = np.sqrt(arr1)
arr3 = np.log10(arr1)
arr4 = np.rint(arr2)


print (arr2)
print (arr3)
print (arr4)
```

```
[ [ 3.16227766
4.47213595]
 [ 5.47722558
6.32455532] ]

[ [ 1.         1.30103   ]
 [ 1.47712125
1.60205999] ]

[ [ 3.  4.]
 [ 5.  6.] ]
```

Array Selectors

- We have already seen that, like lists, individual elements and slices of arrays can be selected using bracket notation. Unlike lists, however, **arrays also permit selection using other arrays**.

- In NumPy this is referred to as fancy indexing

- That is, we can use an array to filter for specific subsets of elements of other arrays.
  - Before we look at this we must look at the result of using relational operators on NumPy arrays.

Comparison operators

- Boolean comparisons can be used to compare members element-wise on arrays of equal size.
- These operators (<,>, >=, <=, ==) return a boolean array as a result

```
import numpy as np

arr1 = np.array([1, 3, 0], float)
arr2 = np.array([1, 2, 3], float)

resultArr = arr1>arr2
print (resultArr)
print (arr1== arr2)
```

```
[False  True False]

[ True False False]
```

Array Selectors

- We can use a Boolean array to **filter** the contents of another array.
- Below we use a Boolean array to select a subset of element from the NumPy array

```
import numpy as np

arr1 = np.array([45, 3, 2, 5, 67], float)
boolArr1 = np.array([True, False, True, False, True], bool)
print (arr1[boolArr1])
```

```
[ 45.   2.  67.]
```

Notice the program only returns the elements in arr1, where the corresponding element in the Boolean array is true

```
import numpy as np

arr2D = np.array([[45, 3, 67, 34],[12, 43, 73, 36]], float)
boolArr3 = np.array([True, False], bool)
print (arr2D[boolArr3])


arr2D = np.array([[45, 3, 67, 34],[12, 43, 73, 36]], float)
boolArr3 = np.array([[True, False, True, False],[True, True,
False, True]], bool)
print (arr2D[boolArr3])
```

If we provide a 1D Boolean array as an index to a 2D array the boolean values refer to rows

If we provide a matching Boolean array it will select individual values and return a flat array

```
[[ 45.   3.  67.  34.]]

[ 45.  67.  12.  43.  36.]
```

The example illustrates the impact of using Boolean arrays to filter 2D arrays.

Selecting Columns from 2D Array

```
import numpy as np

arr2D = np.array([[45, 3, 67],[12, 43, 73]], float)

boolArr4 = np.array([True, False, True], bool)
print (arr2D[:,boolArr4])
```

```
[[ 45.  67.]
 [ 12.  73.]]
```

Here we use booleans to select particular columns from a 2D array. We specify all rows using : and we select the first and last column for selection

Comparison operators

```
import numpy as np

arr1 = np.array([1, 3, 20, 5, 6, 78], float)
arr2 = np.array([1, 2, 3, 67, 56, 32], float)

resultArr = arr1>arr2
print (arr1[resultArr])
```

```
[ 3.  20.  78.]
```

Notice here we combine comparison operators and boolean selection. This will print out all those values in arr1 that are greater than the corresponding value in arr2 (very useful)

Lab Exercises – Bike Sharing Exercise

- From the previous results we saw that the mean number of registered users is much larger than the mean number of casual users in our bike sharing dataset.

- Use comparison operators and boolean selection to determine the percentage of time where the number of casual users (column index 13) is greater than the number of registered users (column index 14).

```
def compareCasualRegUsers(data):

    result = data[:, 13]>data[:, 14]

    print ("Percentage of time where causal users > registered ",
                (len(data[result])*100.0)/len(data))
```

Result is an array of booleans, True if an element of the 13 column is greater then the corresponding value in the 14 column, False otherwise

We use the boolean array to obtain the rows in the data array where the value in column 13 is greater than column 14

Comparison Operators

- An Array can be compared to a single value.
  - The operation below compares the Array arr with the int 2. The result1 variable is a boolean array indicating if each element in arr is greater than 2.

```
import numpy as np

arr = np.array([1, 3, 0, 2, 4, 5], float)

result1 = arr > 2
result2 = arr == 2
print (result1)
print (result2)
print (arr[result1])
print (arr[result2])
```

```
[False  True False False  True  True]

[False False False  True False False]
```

```
[ 3.  4.  5.]
[ 2.]
```

Comparison Operators

- The following code applies a conditional operator the column with index 1 and returns the rows the satisfy this condition.

```
[[ 1.  2.  3.]
 [ 2.  4.  5.]
 [ 4.  5.  7.]
 [ 6.  2.  3.]]

[[ 1.  2.  3.]
 [ 6.  2.  3.]]
```

```
import numpy as np

data = np.array([[1, 2, 3], [2, 4, 5], [4, 5, 7], [6, 2, 3]], float)
print (data)

# return all rows in array where the element at index 1 in a row equals 2
newdata = data[data[:,1] == 2.]
print (newdata)
```

Returns all rows in the 2D array such that the value of the column with index 1 in that row contains the value 1

Comparison Operators

- When we perform array based indexing it produces a **new copy** of the array (this is unlike slicing which produces a view of the original data)

```
import numpy as np

arr2D = np.array([[45, 3, 67],[12, 43, 73],[22, 62, 17]], float)
boolArr2 = np.array([True, False, True], bool)

newArr = arr2D[boolArr2]
print (newArr)

newArr[0, 1] = 50000

print (arr2D)
```

```
[[ 45.   3.  67.]
 [ 22.  62.  17.]]

[[ 45.   3.  67.]
 [ 12.  43.  73.]
 [ 22.  62.  17.]]
```

Notice the change made to the newArr array is not relfected in the original array

Logical Operators

- You can combine multiple conditions using logical operators.
- Unlike standard Python the logical operators used are **&** and **|**

```
import numpy as np

data = np.array([[1, 2, 3], [2, 4, 5],
[4, 5, 7], [6, 2, 3]], float)

resultA = data[:,0]>3
resultB = data[:,2]>6

print (data[resultA & resultB])
```

Notice in the code we combine two conditions using & (we could chain as many conditions as we wish)

```
[[ 4.  5.  7.]]
```

Back to Array Selectors

- In addition to Boolean selection, it is possible to select using integer arrays.
- In this example the new array *c* is composed by selecting the elements from *a* using the index specified by the elements of *b*.

```
import numpy as np

a = np.array([2, 4, 6, 8], float)
b = np.array([0, 0, 1, 3, 2, 1], int)
c = a[b]
print (c)
```

```
[ 2.  2.  4.  8.  6.  4.]
```

Notice the array c is composed of index 0,0, 1, 3, 2, 1 of the array a

Discussion

CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Computer Science

http://www.cit.ie

Thank you