CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Practical Machine Learning

Practical Machine Learning

Lecture: NumPy / Pandas Overview

Ted Scully

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 (parallelization).

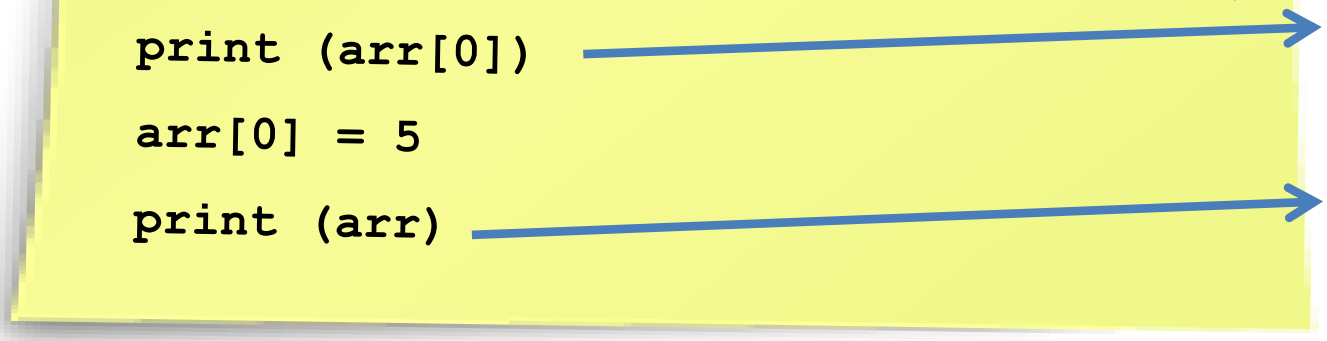
- NumPy arrays facilitate a wide range of mathematical and other types of operations on large amounts of data.
  - Typically, such operations are executed much more efficiently and with less code than is possible using Python's built-in sequences.
  - Further, NumPy implements an array language, so that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

- 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**.

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)
```

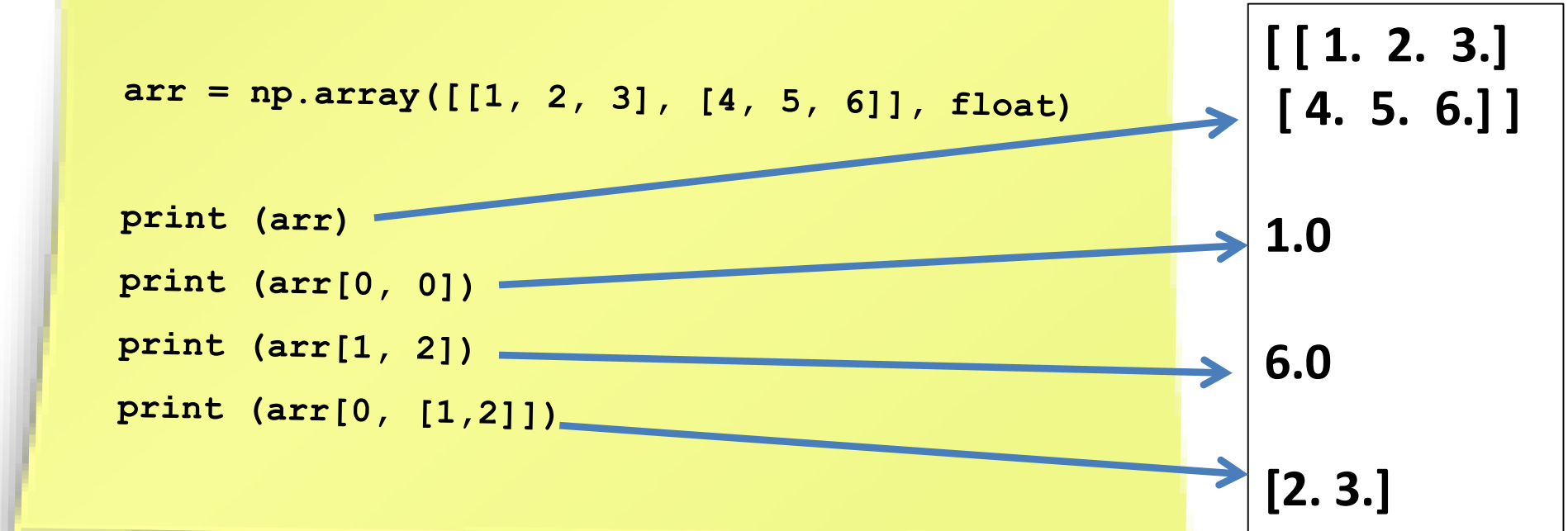


```
5.5

[ 5.  45.6  3.2]
```

NumPy – Multi-Dimensional Arrays

- Arrays can be multidimensional. Elements are accessed using **[row, column]** format inside bracket notation
- Notice we can also provide a list of indices to access along a specific axis.

```
import numpy as np

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

print (arr)
print (arr[0, 0])
print (arr[1, 2])
print (arr[0, [1,2]])
```



NumPy – Single Index to 2D Array

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

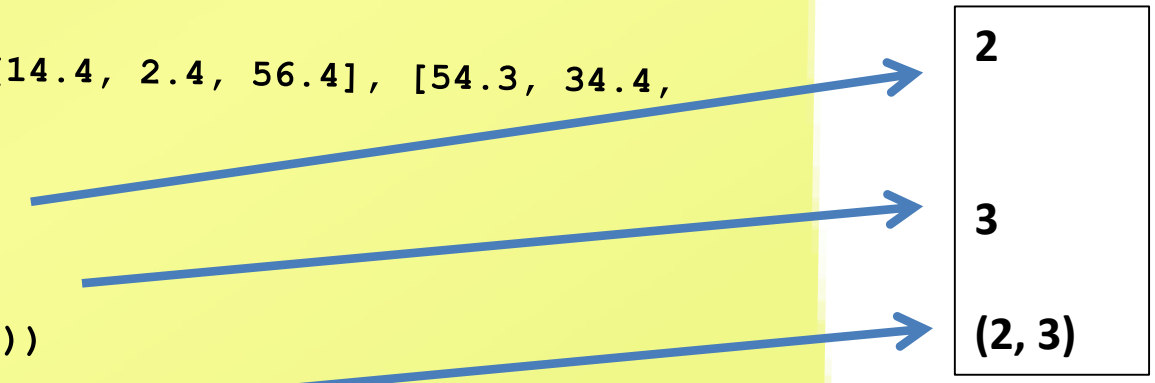
```
import numpy as np

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

```
[[ 1.  12.2  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]))
print (arr.shape)
```



NumPy – Slicing Operations

- Slice: a span of items that are taken from a sequence
  - Slicing format: array[start : end]
  - Span is a list containing copies of elements from ***start*** up to, but not including, ***end***.
  - Slicing expressions can include a step value in the format
    - *array[start : stop : step]*

- It is also really important to understand that we can leave either start stop or step blank. If we leave:
  - ***Start*** **blank** it's going to default to index 0
  - ***Stop*** **blank** then it will default to len of the array
  - ***Step*** **blank** it defaults to 1

Slicing Example

```
import numpy as np

arr = np.array([2, 4, 6, 8, 10], float)

print (arr[0:2])

print (arr[0:5:2])

print (arr[3:])

print (arr[:4])

print (arr[:])
```

```
[ 2.  4.]

[ 2.  6.  10.]

[ 8.  10.]

[ 2.  4.  6.  8.]

[ 2.  4.  6.  8.  10.]
```

Notice in the last example we don't specify start or stop so Python set start to 0 and stop to 5 (last index + 1)

NumPy – Slicing Operations in 2D Arrays

- We can just as easily use slicing operations on 2D arrays as well.
- We can specify a slice on a row or column axis.

- *array[start1:stop1, start2:stop2]*

```
import numpy as np

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

print (arr[0:2, 0:3] )

print (arr[0:2, 0:4:2] )

print (arr[:, 0:2] )

print (arr[:, [0,3]] )
```

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

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

[[ 1.  3.]
 [ 5.  7.]]

[[ 1.  2.]
 [ 5.  6.]
 [ 9. 10.]]

[[ 1.  4.]
 [ 5.  8.]
 [ 9. 12.]]
```

Important consideration when slicing!!

- 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.
- NumPy documentation defines a view as "**An array that does not own its data, but refers to another array's data instead**." This is one area in which NumPy array slicing differs from Python list slicing: in lists, slices will be copies.
- **Using integers and slices is what is called Basic Indexing in Python. This rule does not hold for what we refer to as Advanced Indexing.**

```
import numpy as np

data = np.array([[1, 2, 3, 4], [2, 4, 5, 6], [4,
5, 7, 8], [6, 2, 3, 9]], float)
print (data)

resultA = data[:, 0:2]
resultA[0] = 200
print (resultA)
print (data)
```

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

[[200. 200.]
 [ 2.  4.]
 [ 4.  5.]
 [ 6.  2.]]

[[200. 200.  3.  4.]
 [ 2.  4.  5.  6.]
 [ 4.  5.  7.  8.]
 [ 6.  2.  3.  9.]]
```

Writing and Reading Data From a File

- Data can be read from and written to files of various formats by using :
  - *np.genfromtxt allows you to read from files*
  - *np.savetxt allows you to save data to a file.*

```
import numpy as np


arr = np.arange(50)
arr2 = np.resize(arr, (10,5))
```

```
[[ 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]]
```

Resize is a really useful function that allows you to resize an existing 2D array. The second argument specifies the new dimensional of arr.

np.arange allow you to create a flat numpy array rising in value from 0 to the argument specified.

Writing and Reading Data From a File

- Data can be read from and written to files of various formats by using :
  - *np.genfromtxt allows you to read from files*
  - *np.savetxt allows you to save data to a file.*

```
import numpy as np


arr = np.arange(50)
arr2 = np.resize(arr, (10,5))


np.savetxt("numbers.txt", arr2, fmt="%d", delimiter=",")
```

Writing and Reading Data From a File

- Data can be read from and written to files of various formats by using :
  - *np.genfromtxt allows you to read from files*
  - *np.savetxt allows you to save data to a file.*

```
import numpy as np

arr = np.arange(50)
arr2 = np.resize(arr, (10,5))
np.savetxt("numbers.txt", arr2, fmt="%d", delimiter=",")

data = np.genfromtxt("numbers.txt", delimiter=",")
print (data)
```

```
[[ 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.]]
```

Performing Operations on a Specific Axis

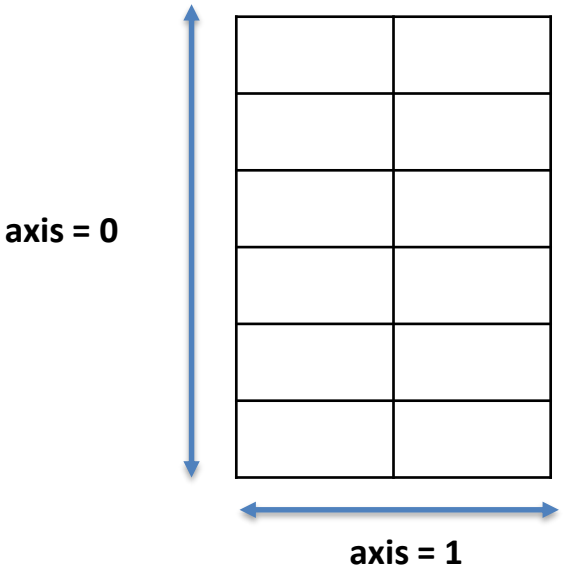
- NumPy provides a wide range of operations that we can perform on data.

- One very important element of this NumPy functionality is that it allows you to specify the **axis (dimension)** on which you want to perform the operation.

NumPy – Appending to MD Arrays

- Consider the following NumPy function to add elements in an array

  - numpy.**sum**(*arr*, *axis=None*)
    - arr – The array on which you want the sum operation performed.
    - **axis = <u>The axis along which the sum operation is performed</u>. It can also be useful to think of the axis as specifying the direction in which the operation is performed (or viewing it as the axis that will be collapsed after the operation is performed).**.



The axes of an array refer to the order with which you index an array.

Remember we index the row axis first. Therefore, you can think of axis =0 as the axis that **runs down along the rows**.

We index the column index second. Therefore, you can think of axis =1 as **running across the columns**.

NumPy – Multi-Dimensional Arrays

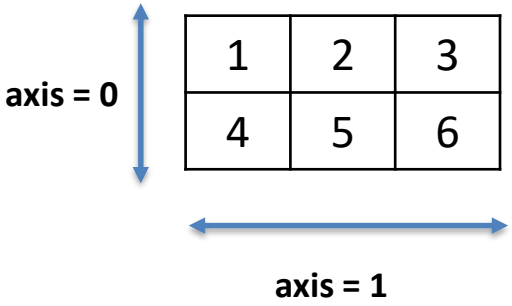
```
import numpy as np


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

print (np.sum(arr))
```

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

21
```

Notice it performs the sum operation through the entire array (that is sums up all elements in the array)

NumPy – Multi-Dimensional Arrays

```
import numpy as np


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

print (np.sum(arr, axis = 0))
```

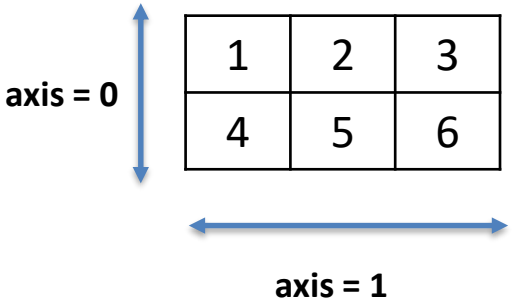



```
[[ 1.  2.  3.]
 [ 4.  5.  6.]]

[5. 7. 9.]
```

Notice in this case the sum is performed along axis = 0.
Axis = 0 runs vertically. Therefore, the operation is performed for each vertical.

NumPy – Multi-Dimensional Arrays

```
import numpy as np


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

print (np.sum(arr, axis = 1))
```




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

[ 6. 15.]
```

Notice in this case the sum is performed along axis = 1. Axis = 1 runs horizontally. Therefore, the operation is performed on each horizontal

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.**

NumPy – Multi-Dimensional Arrays

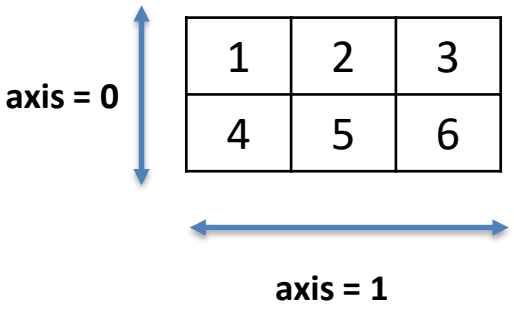
```
import numpy as np

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

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

print (arr1)
```

Notice the output array has been flattened. This is because no axis was not specified

```
[[ 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 = ?)

print (arr1)
```



Adding an extra row to the row (vertical) axis

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.

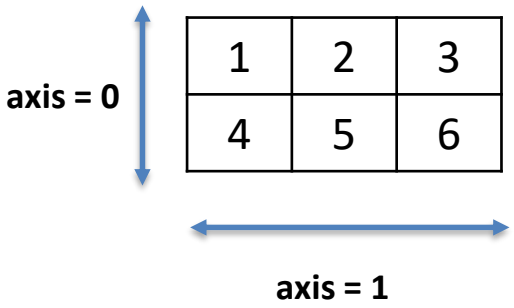
NumPy – Multi-Dimensional Arrays

```
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 to axis = 0 means we are adding to this vertical axis.

In this case we are adding the values [7, 8, 9] to axis = 0

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

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

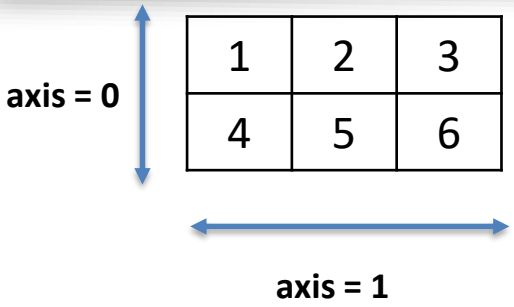


NumPy – Multi-Dimensional Arrays

```
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)
```

What do you think would happen if we ran the following code?

Generates an error specifying array dimensions don't match. To add to the vertical axis we need to add one column contain two values. Here we are trying to add a row that contains three values to the vertical axis.

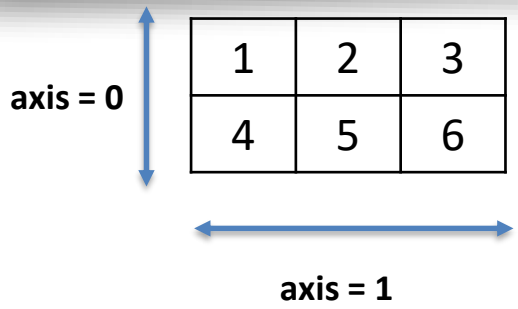


NumPy – Multi-Dimensional Arrays

```
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 containing two elements to the horizontal axis.

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

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



Basic Array Operations

- Many functions exist for extracting whole-array properties.
- Large number of mathematical functional available.
  - If axis is 0 then think of it as collapsing down the rows.
  - http://docs.scipy.org/doc/numpy/reference/routines.math.html
  - http://docs.scipy.org/doc/numpy/reference/routines.statistics.html

```
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.  ]
```

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 when performing addition, subtraction, etc.
    - We will later mention an exception to this rule.
  - NumPy arrays support the typical range of operators +,-, *, /, %, **
  - NumPY also allows us to use the above operators with a NumPy array and a single operand value.

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)
print (arr1*2)
print (arr1/2)
```

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

[[ 20.  40.]
 [ 60.  80.]]

[[  5.  10.]
 [ 15.  20.]]
```

Example

Assume we have a basic csv file called testData.csv, containing 20 columns of numerical data. The tenth column (index 9) contains total student attendance for each day in the academic year. I want to quickly find out the average student attendance.

Example

Assume we have a basic csv file called testData.csv, containing 20 columns of numerical data. The tenth column (index 9) contains total student attendance for each day in the academic year. I want to quickly find out the average student attendance.

```
import numpy as np

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

print ( np.mean( data[:, 9] ) )
```

Advanced Indexing

- In NumPy advanced indexing occurs when we pass an array containing **booleans or integers** as an index.
- Advanced indexing always returns a copy of the data (in contrast with basic slicing that returns a view).
- 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 advanced indexing (sometimes called Fancy Indexing)
- This allows us to build up expressive **filters** for the data we are using. Before we illustrate this idea we will first look at the use of comparison operators in NumPy.

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.
- Note the arrays need to be of the same size.

```
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. This is our first example of **advanced indexing**.

```
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

Array Selectors

- Can you remember what happens when we provide a single integer value as an index to a 2D array?

```
import numpy as np

arr2D = np.array([[45, 3, 67, 34],[12, 43, 73, 36]], float)

print (arr2D[1])
```

```
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])
```

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

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

Building and Running a Query

Now we have seen that a **comparison operator** used in NumPy returns a **Boolean array**. We have also seen that we can provide a **Boolean array** as a **filter** for an existing array so that it will only extract the elements where the corresponding index is True in the Boolean array.

In the example below we want to identify all elements in arr1 that are greater than the corresponding elements in arr2

```
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)

Example

Lets go back to our example where we have a basic csv file called testData.csv, containing 20 columns of numerical data. The tenth column (index 9) contains total student attendance for each day in the academic year. I want to extract all rows from the dataset where the student attendance was lower than the average

Example

Lets go back to our example where we have a basic csv called testData.csv, containing 20 columns of numerical data. The tenth column contains total student attendance for each day in the academic year. I want to extract all rows from the dataset where the student attendance was lower than the average

```
import numpy as np

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

# calculate average student attendence
avgAtt = np.mean( data[:, 9] )

# comparison operator will return True if column value is less than average
dataFilter = data[:, 9] < avgAtt

# extract all rows where attendance is less than average
subsetData = data[dataFilter]
```

Example

Lets go back to our example where we have a basic csv called testData.csv, containing 20 columns of num al student attendance for each c ll rows from the dataset where the st age

Result is an array of booleans, True if an element of the 10 column is less than the average and False otherwise.

```
import numpy as np

data = np.genfromtxt('testData.c pe=float, delimiter = ',')

# calculate average student att ce
avgAtt = np.mean( data[:, 9] )

# comparison operator will return True if column value is less than average
dataFilter = data[:, 9] < avgAtt

# extract all rows where attendance is less than average
subsetData = data[dataFilter]
```

Example

Lets go back to our example where we have a basic csv called testData.csv, containing 20 columns of numerical data. The tenth column contains total student attendance for each day in the academic year. I want to extract all rows from the dataset where the studer

```
import numpy as np

data = np.genfromtxt('testDat

# calculate average student attend
avgAtt = np.mean( data[:, 9] )

# comparison operator will r      n True if column value is less than average
dataFilter = data[:, 9] < a     Att

# extract all rows where attendance is less than average
subsetData = data[dataFilter]
```

We use the boolean array to obtain the rows in the data array where the value in 10th column is less than the average attendence

Comparison Operators

- You will often find that the Boolean comparison and the index appear in the same line (as shown below).

```
[[ 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

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

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])
```

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

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

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

Advanced Indexing with Integer Arrays

- 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