CORK INSTITUTE OF TECHNOLOGY
INSTITIÚID TEICNEOLAÍOCHTA CHORCAÍ

Practical Machine Learning

Practical Machine Learning

Lecture: NumPy / Pandas Overview

Ted Scully

Introduction to Numpy

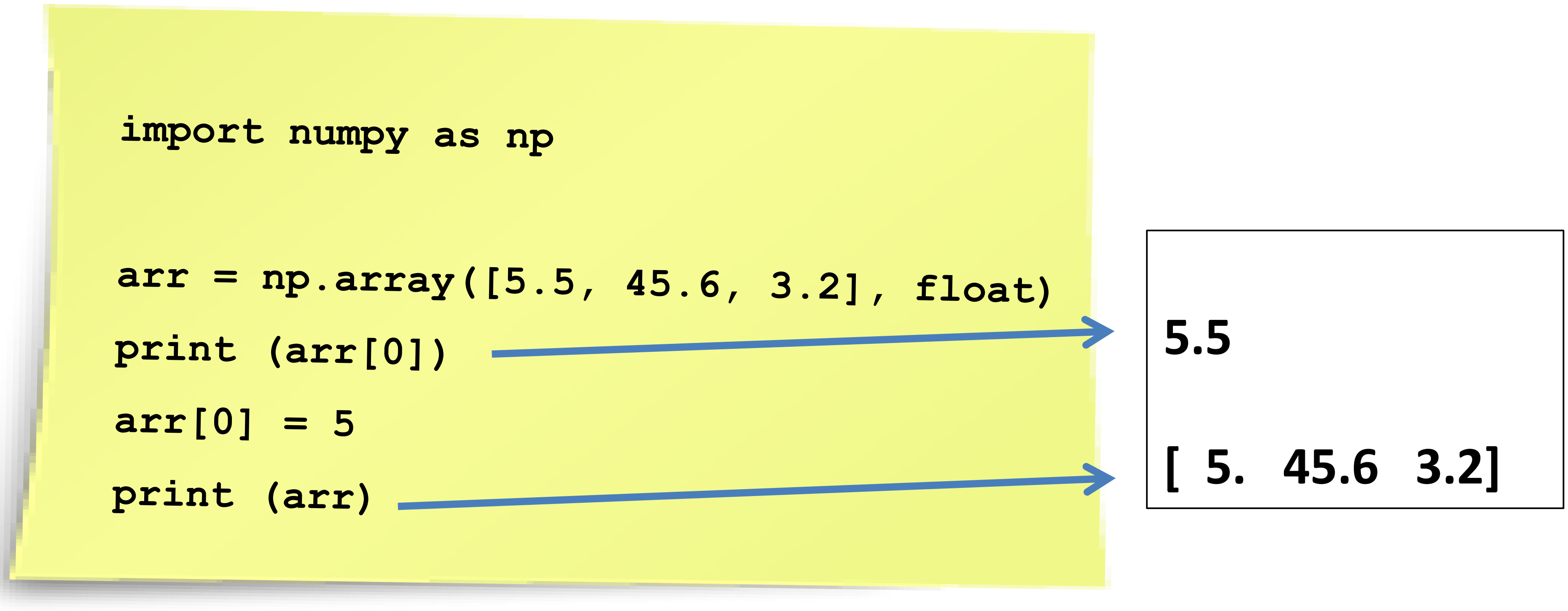
- NumPy is an open-source add-on module to Python that provides routines for **manipulating large arrays** and matrices of numeric data in **pre-compiled**, fast functions (parallelization).

- NumPy arrays facilitate a wide range of mathematical and other types of operations on large amounts of data.
  - Typically, such operations are executed 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

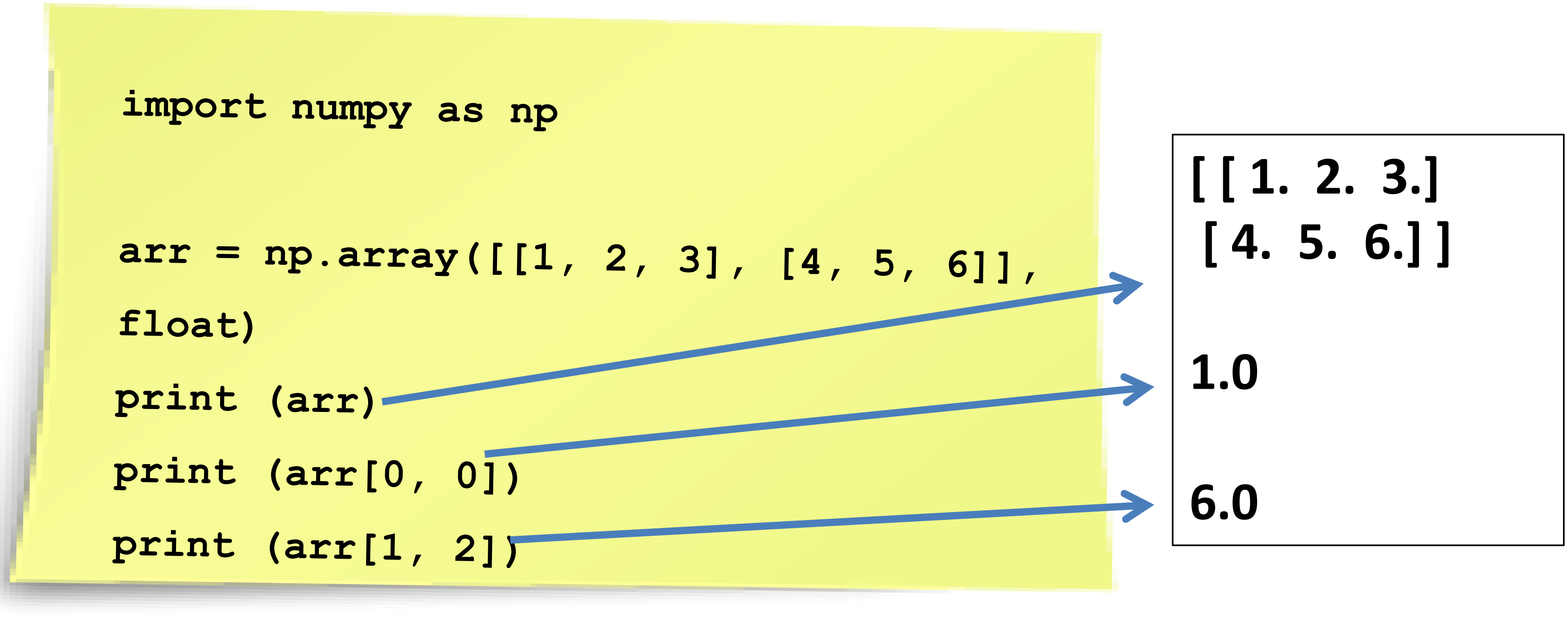


```
import numpy as np

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

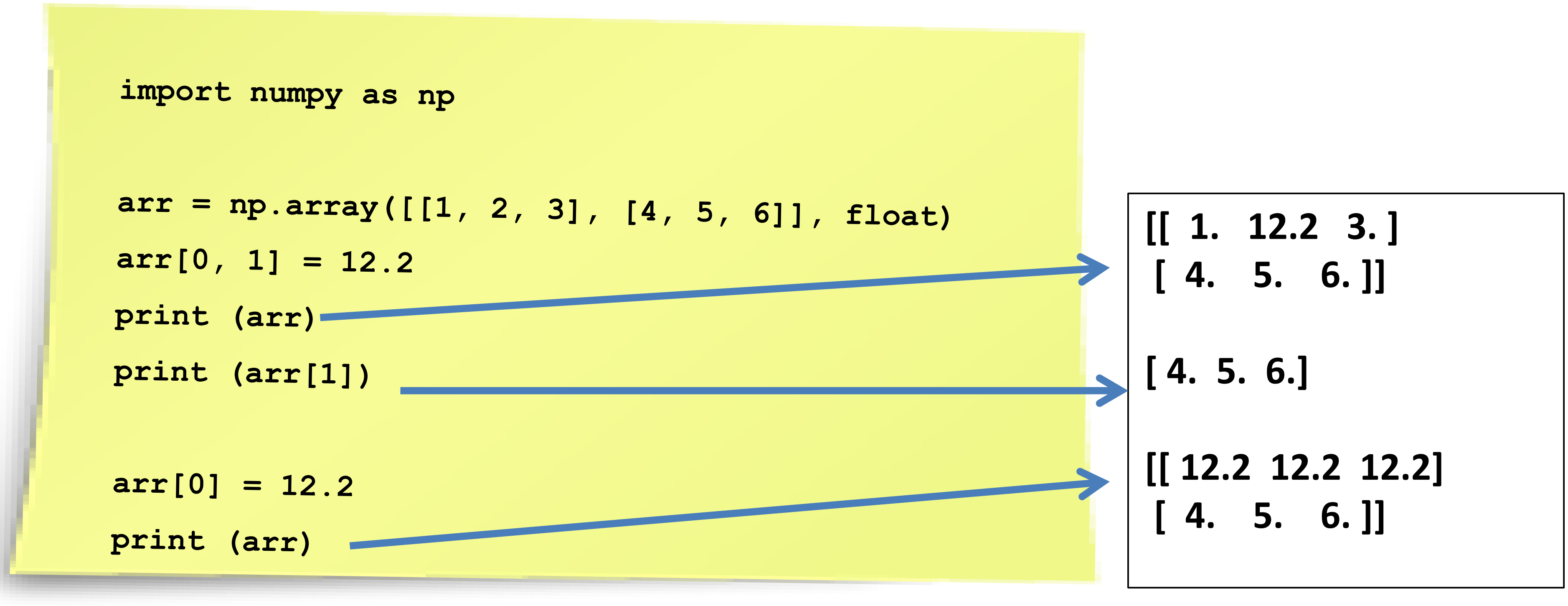


```
import numpy as np

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
```

NumPy – Single Index to 2D Array

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

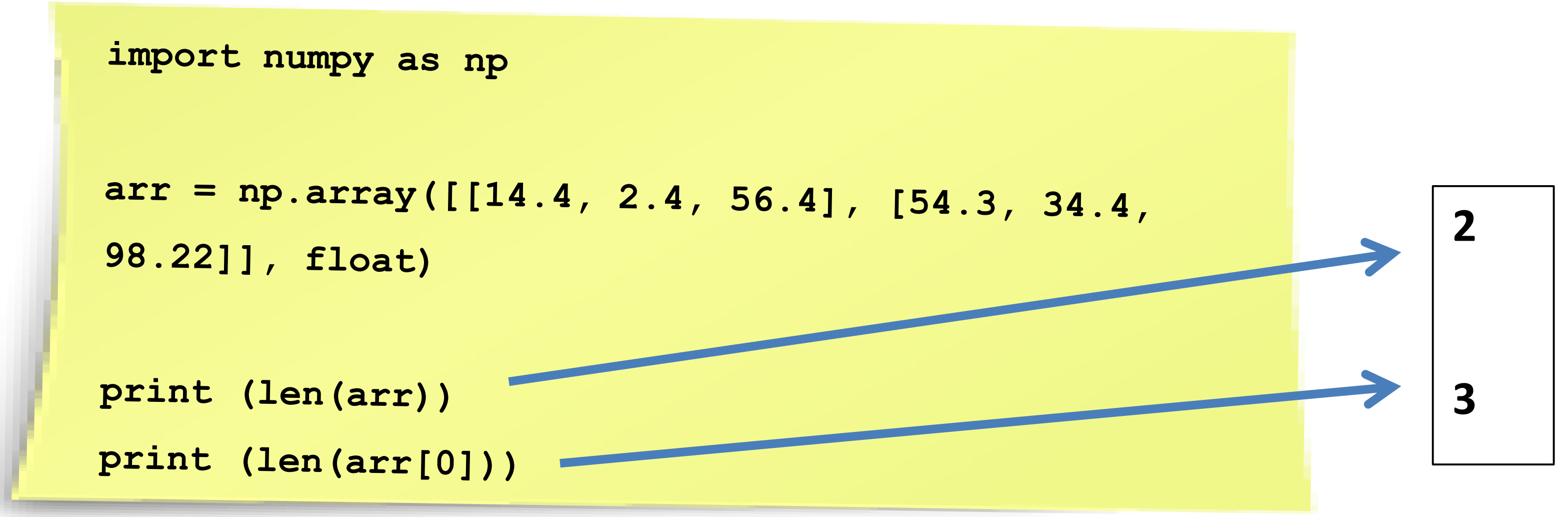


```
import numpy as np

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. ]]
```

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
```

NumPy – Slicing Operations

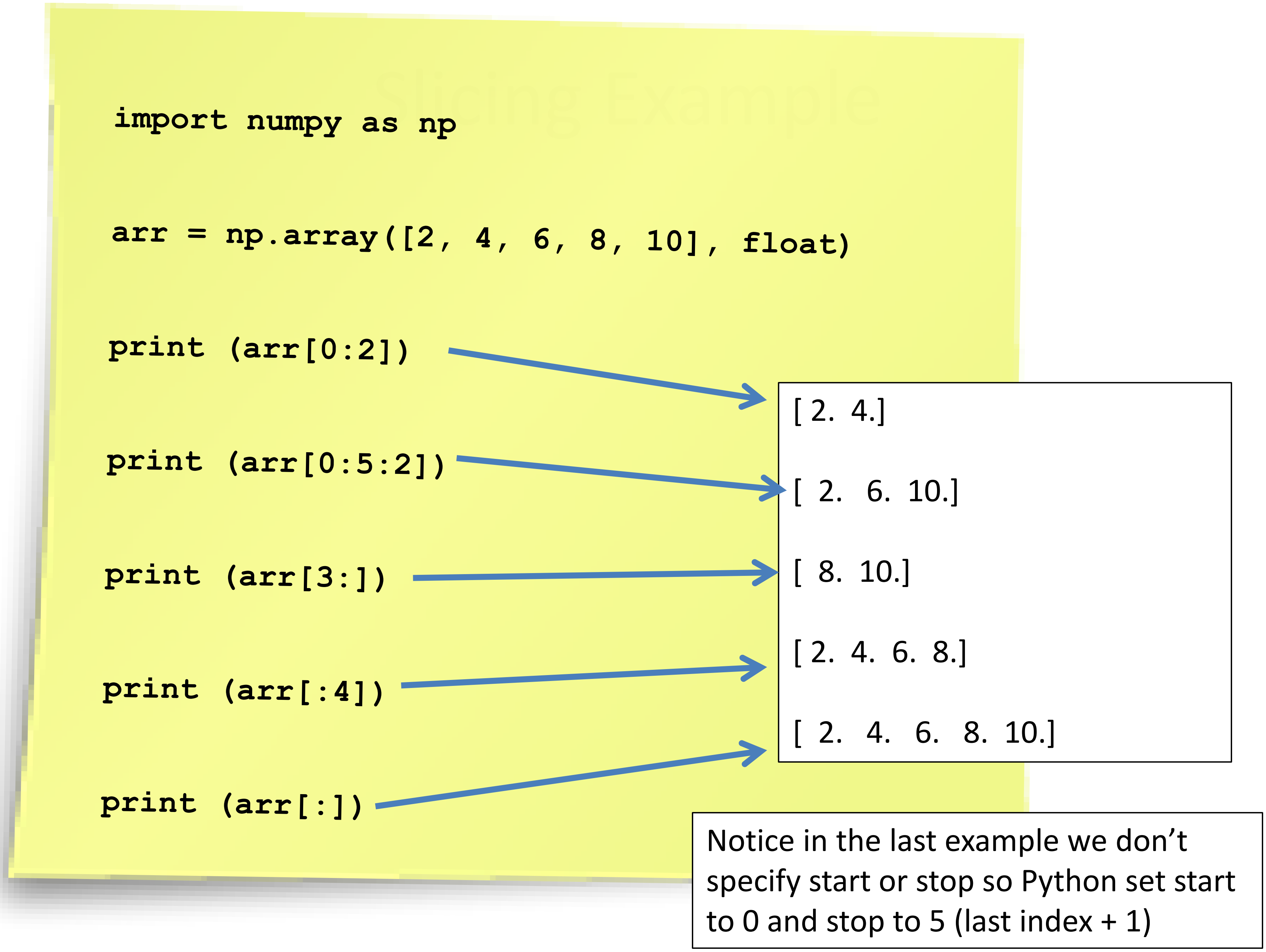
- Slice: a span of items that are taken from a sequence
  - Slicing format: array[start : end]

  - 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.
- Rather than specifying an integer index for a specific dimension we can specify the slice for that dimension.

- *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!!

- However, when performing slicing on a NumPy array it will return **<u>a view of the original array</u>**. 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 <u>view</u> 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 <u>Advanced Indexing</u>.**

```
import numpy as np


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


resultA = data[:, 0:3]
resultA[0] = 200
print (resultA)
print (data)
```

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

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

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))
```

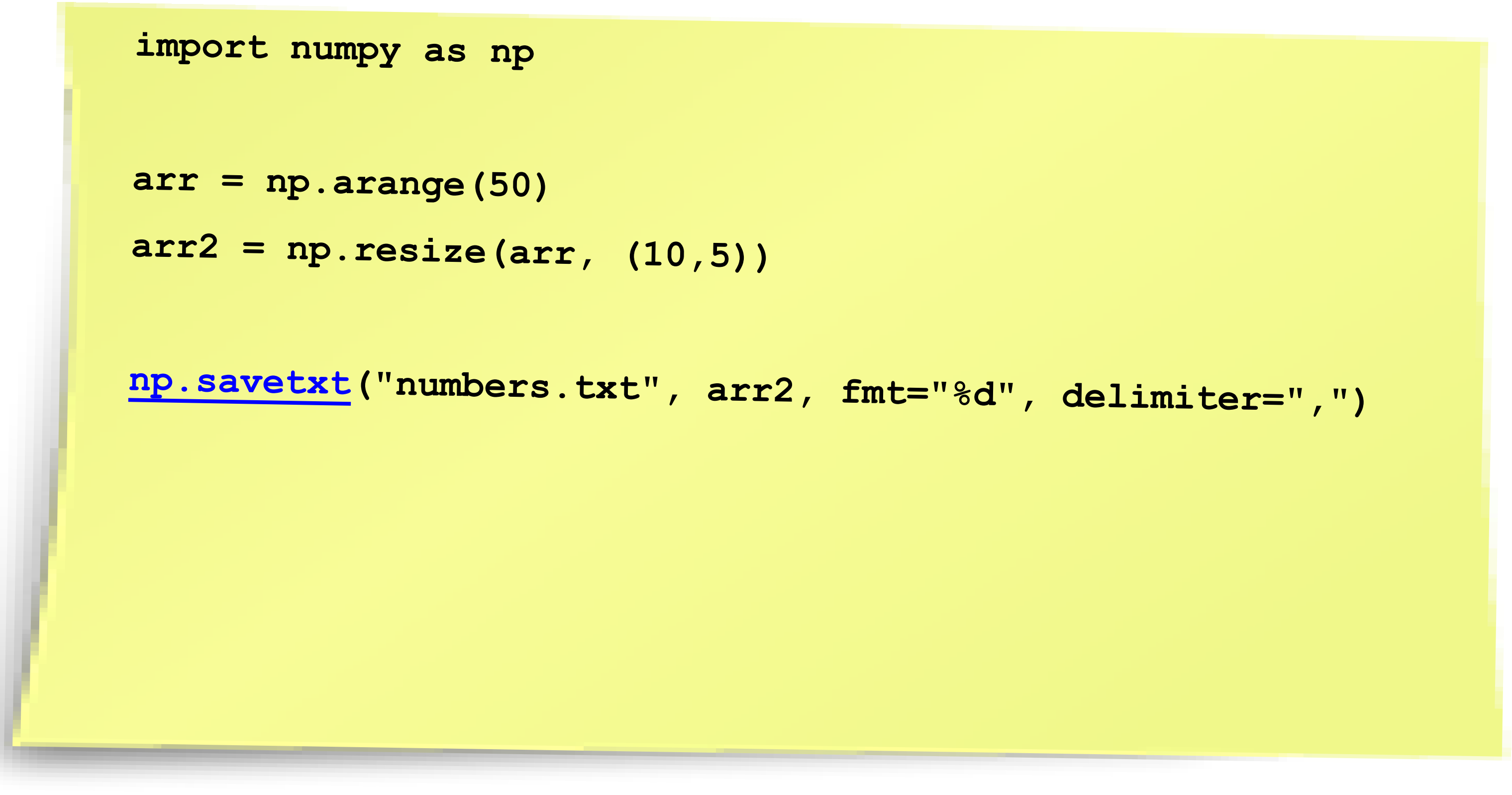
```
[[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]
 [25 26 27 28 29]
 [30 31 32 33 34]
 [35 36 37 38 39]
 [40 41 42 43 44]
 [45 46 47 48 49]]
```

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

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

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

- 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 row dimension

axis = 1 refers to the columns dimension

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.]]
```

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.
  - An error is thrown if arrays don’t match in size

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 Python.

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])



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

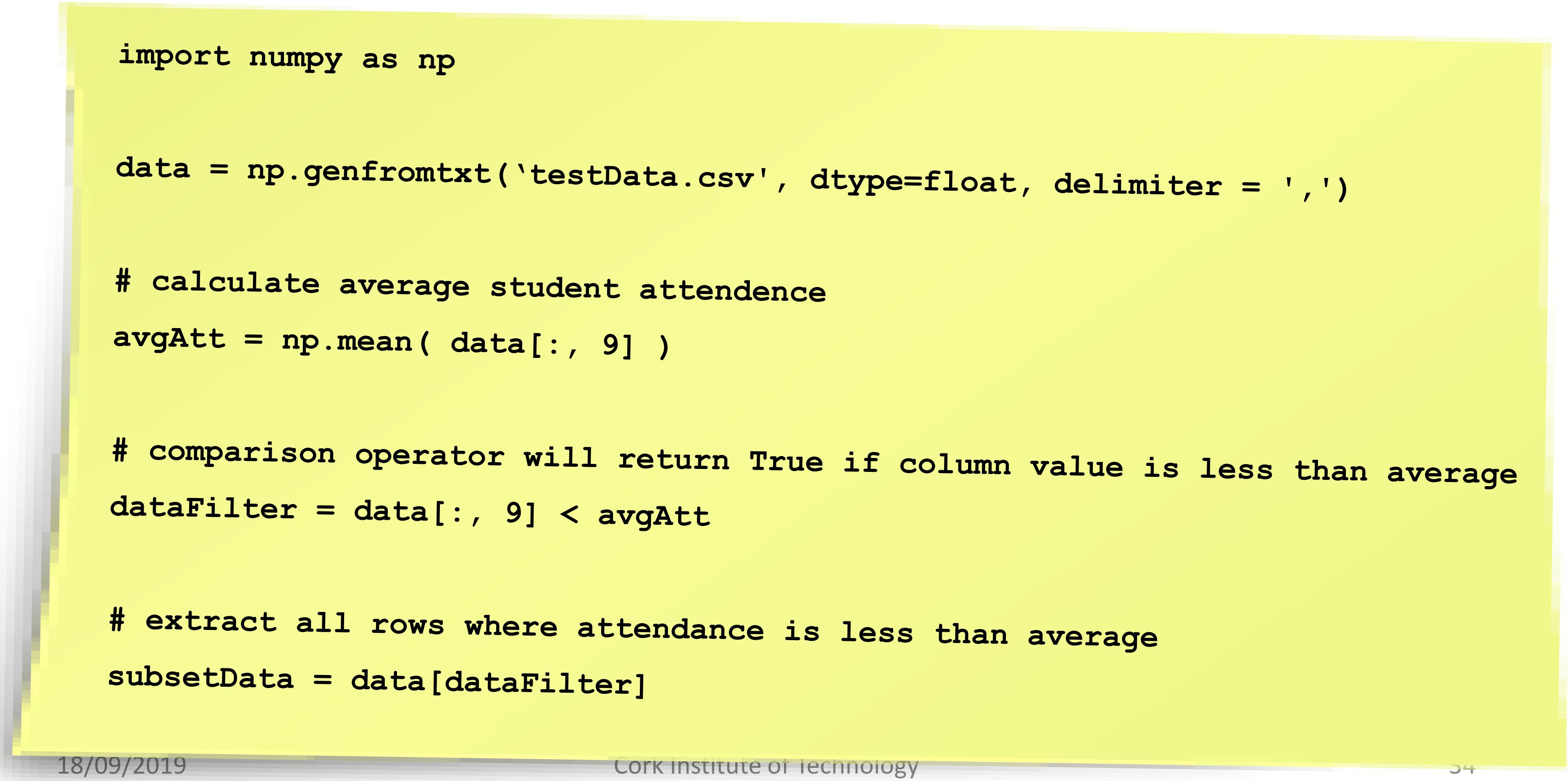
```
import numpy as np

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)

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



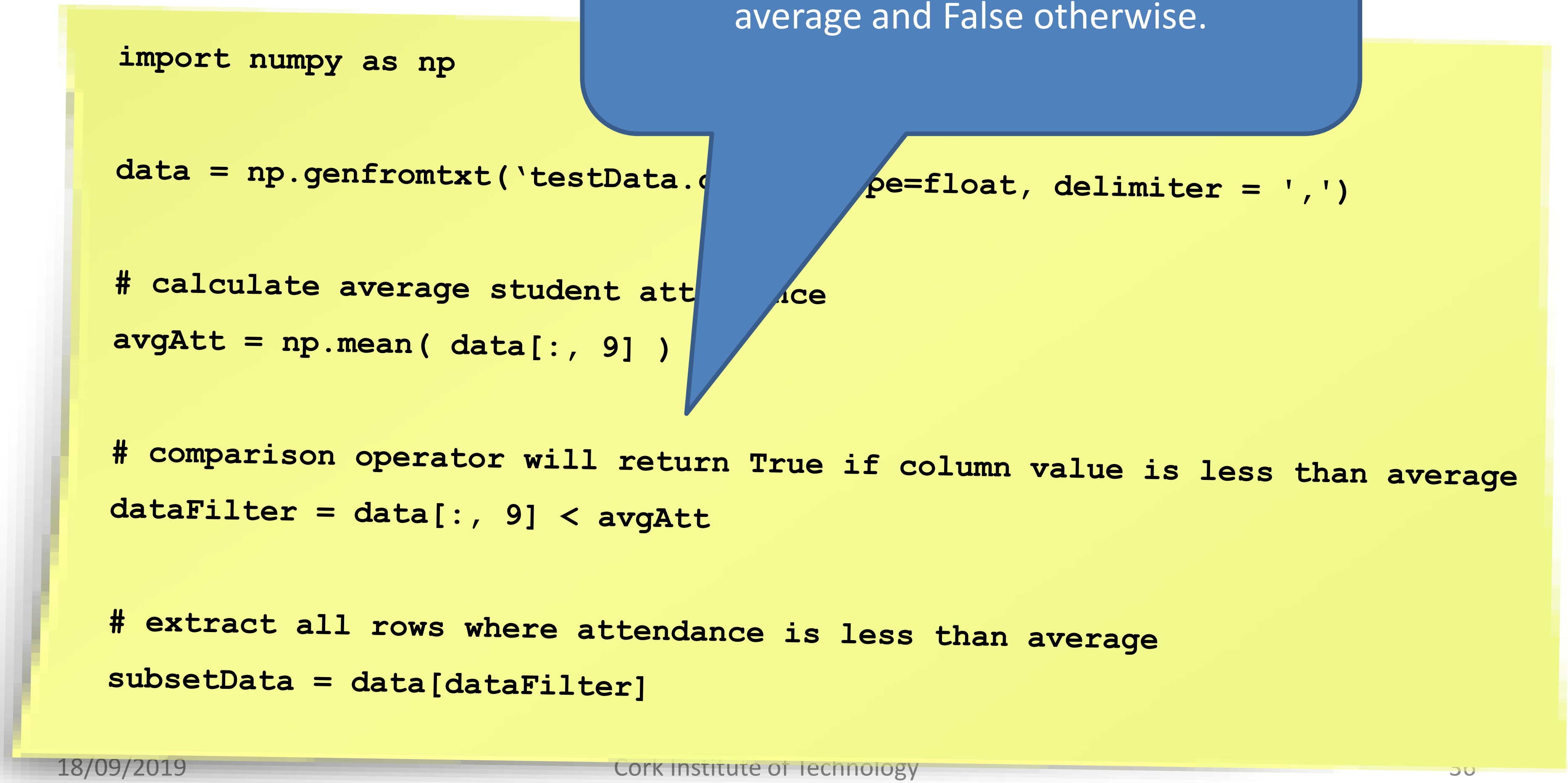
```
import numpy as np

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 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 num al student attendance for each c ll rows from the dataset where the st age

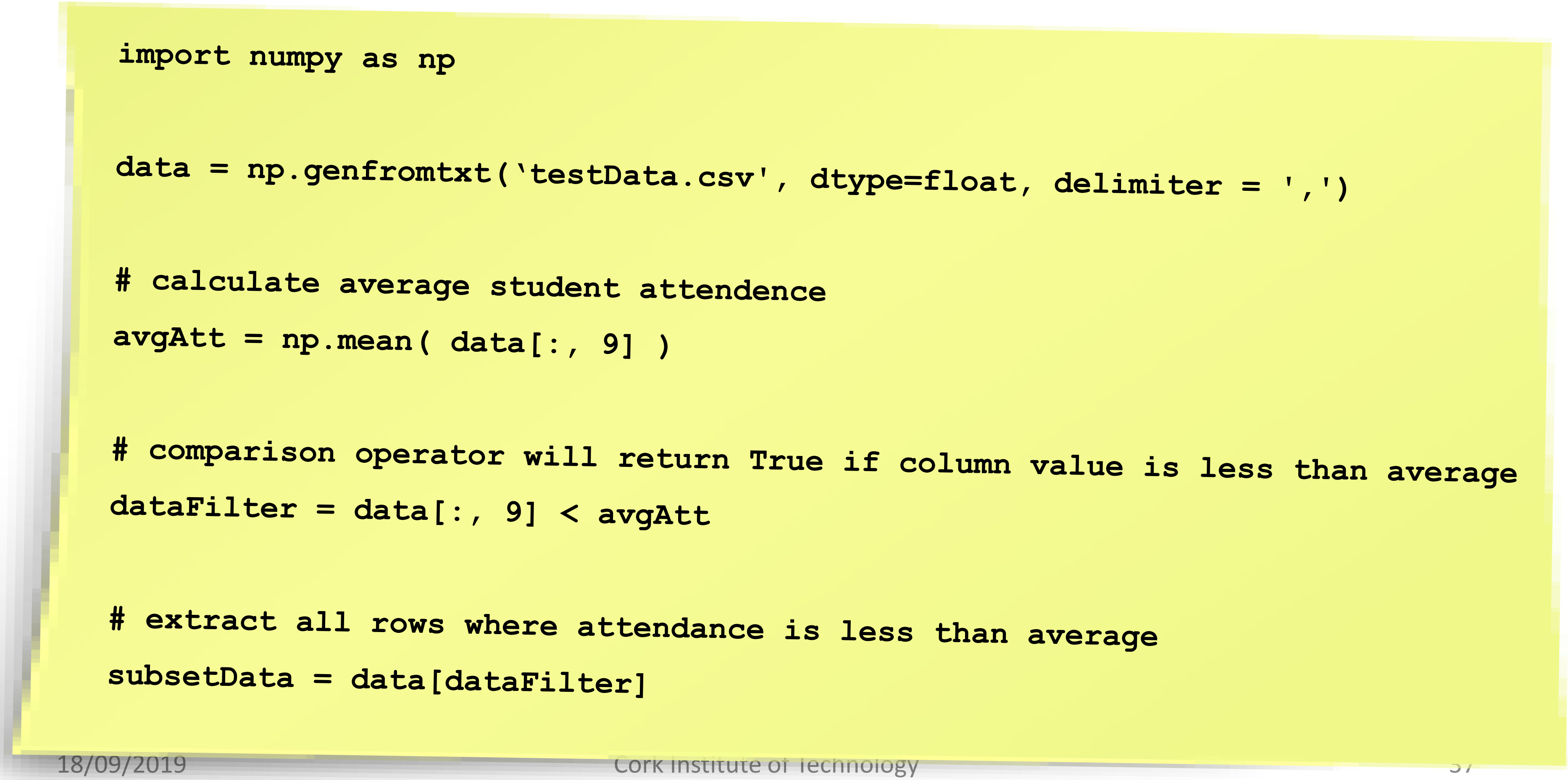


```
import numpy as np

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

# calculate average student att      nce
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]
```

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

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 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

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

```
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  gAtt

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

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

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.]]
```

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

Pandas

- NumPy is a great tool for dealing with **numeric matrices** and vectors in Python
  - For more complex data, it is limited.

- Fortunately, when dealing with complex data we can use the **powerful Python data analysis toolkit** (a.k.a. pandas).

- Pandas is an open source library providing high-performance, easy-to-use data structures for the Python programming language.
  - Used primarily for data manipulation and analysis.

- Resources
  - http://pandas.pydata.org/pandas-docs/version/0.13.1/pandas.pdf

Data Structures in Pandas

- Pandas introduces two new data structures to Python –
  - Series
  - DataFrame

- Both of which are built on top of NumPy (which means it's very fast).

- **A Series** is a one-dimensional object similar to an array, list, or column in a table.

- Pandas will assign a **labelled index** to each item in the Series.
  - By default, each item will receive an index label from 0 to N, where N is the length of the Series minus one.

  - **S = Series(data, index = index)**
    - The data can be many different things such as a NumPy arrays, list of scalar values, dictionary

Series - Examples

You will notice the syntax and functionality used in a Series object is quite similar to that of a NumPy array.

```
import numpy as np
import pandas as pd

s1 = pd.Series( np.random.randn(5), index=['a','b','c','d','e'])


s2 = pd.Series(np.random.randn(5))

print (s1)
print (s2)

print (s2[1])
print (s2[[1, 2]])

s3 = s2[[1, 2]]
s3[0] = 12
print (s3)

print (np.square(s3))
```

```
a    0.482188
b    1.730022
c    0.518800
d    0.039572
e   -0.946694
dtype: float64


0    0.489512
1    0.100944
2   -0.310478
3    1.554981
4   -0.599866
dtype: float64
```

Series - Examples

```
import numpy as np
import pandas as pd

s1 = pd.Series(np.random.randn(5), index=['a','b','c','d','e'])


s2 = pd.Series(np.random.randn(5))

print (s1)
print (s2)

print (s2[1])
print (s2[ [1, 2] ])

s3 = s2[[1, 2]]
s3[0] = 12
print (s3)

print (np.square(s3))
```

```
0.100944

1    0.100944
2   -0.310478
dtype: float64
```

```
1     0.100944
2    -0.310478
0    12.000000
dtype: float64
```

```
1      0.010190
2      0.096397
0    144.000000
dtype: float64
```

Series

- Another useful feature of a series is using **boolean conditions**
  - **irishCities <200** returns a Series of True/False values, which we then pass to our Series cities, returning the corresponding True items.

```
# Dictionary with annual car robberies in each Irish city
d = {'Dublin': 245, 'Cork': 150, 'Limerick': 125,' Galway': 360, 'Belfast': 300}

irishCities = pd.Series(d)

print (irishCities [ irishCities <200  ]  )

print ( type ( irishCities[irishCities <200] ) )
```

```
Belfast     300
Cork        150
Dublin      245
Galway      360
Limerick    125
dtype: int64
```

As with NumPy, relational operators return a **separate copy** of the data. The original series and the one returned by the relational operator don’t refer to the same copy of the same data.

```
Cork        150
Limerick    125
dtype: int64
<class 'pandas.core.series.Series'>
```

Data Frame

- A DataFrame is a data structure comprised of **rows and columns** of data.
  - It is similar to a spreadsheet or a database table.
  - You can also think of a DataFrame as a **collection of Series objects** that share an index

- The syntax for creating a data frame is as follows:
  - ***DataFrame(data, columns=listOfColumns)***

- Using the columns parameter allows us to tell the constructor how we'd like the columns ordered.

Creating a DataFrame

- Remember we mentioned you can view a dataset as a group of Series object. Here create a DataFrame by passing it a number of Series objects.

```
seriesA = pd.Series( np.random.rand(3), index=['a', 'b', 'c']  )
seriesB = pd.Series( np.random.rand(4), index=['a', 'b', 'c', 'd'] )
seriesC = pd.Series( np.random.rand(3), index=['b', 'c', 'd'] )

df = pd.DataFrame( { 'one' : seriesA,
                     'two' : seriesB,
                     'three' : seriesC } )
print df
```

```
   one         three       two
a  0.307010    NaN         0.396005
b  0.671142    0.263916    0.532836
c  0.116057    0.839463    0.826531
d       NaN    0.439335    0.984332
```

Creating a Dataframe from a NumPy Array

- In the example below we create a dataframe from a 2D NumPy array. The array is passed as an argument when the dataframe is created.

```
import pandas as pd
import numpy as np

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

df = pd.DataFrame( arr )

print (df)
```

```
     0    1    2
0  1.0  2.0  3.0
1  4.0  5.0  6.0
2  7.0  8.0  9.0
```

Revert from DataFrame to NumPy Array

- It is very easy to convert from a DataFrame object to a NumPy array using .values.
- We can also convert a **<u>Series object</u>** to a NumPy array in the same way!

```
import pandas as pd
import numpy as np

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

df = pd.DataFrame( arr )

dataArr = df.values

print (dataArr)
print (type(dataArr))
```

```
     0    1    2
0  1.0  2.0  3.0
1  4.0  5.0  6.0
2  7.0  8.0  9.0

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

<class 'numpy.ndarray'>
```

Dataframe

- The most common way of creating a dataframe is by reading existing data directly into a dataframe

- There are a number of ways of doing this
  - read_csv
  - read_excel
  - read_hdf
  - read_sql
  - read_json
  - read_sas …

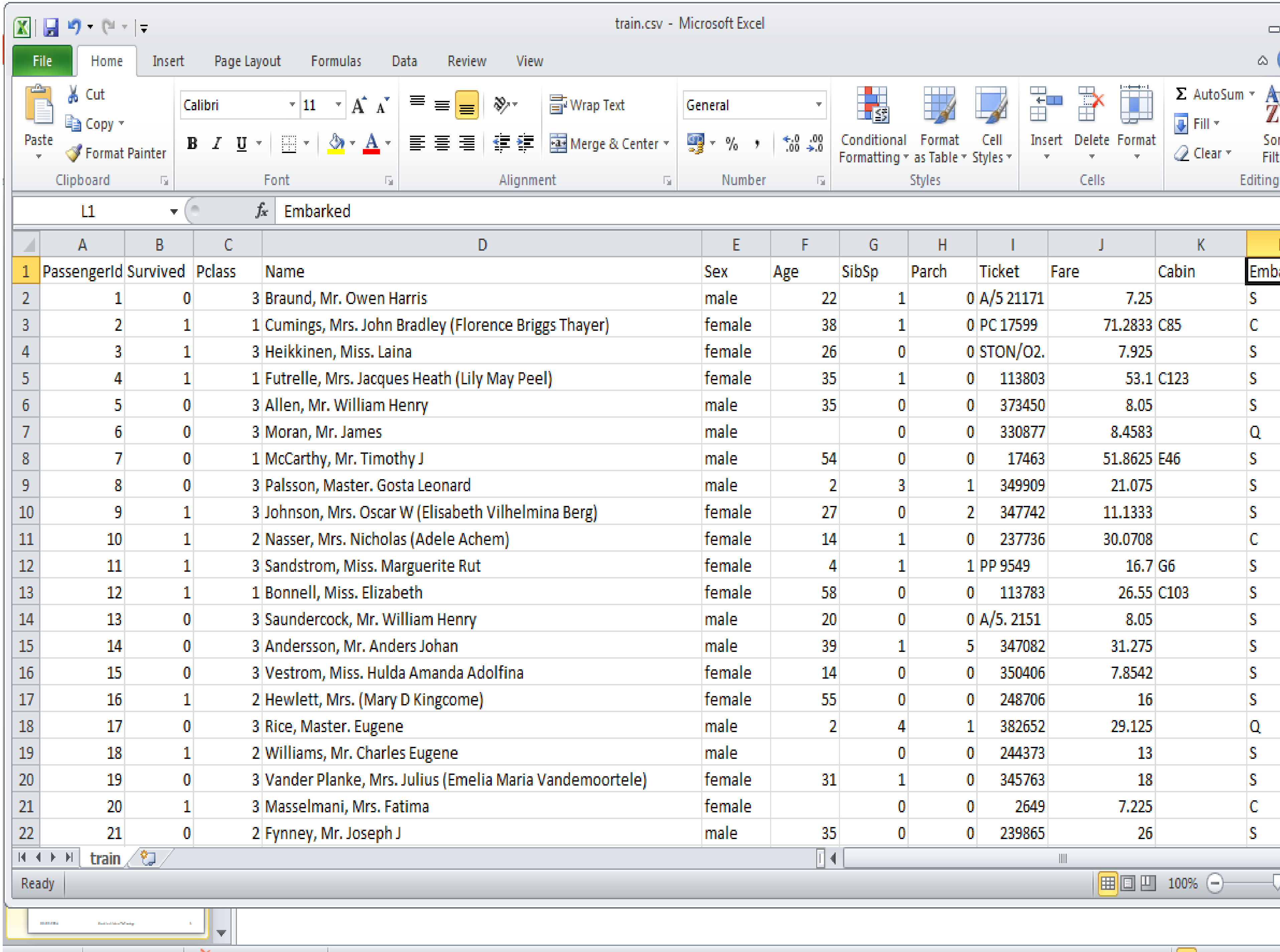
- We will look at how to read from a CSV file.

Titanic - Dataset

Available as .csv file on Blackboard.

VARIABLE DESCRIPTIONS:

survival	Survival (0 = No; 1 = Yes)
pclass	Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
name	Name
sex	Sex
age	Age
sibsp	Number of Siblings/Spouses Aboard
parch	Number of Parents/Children Aboard
ticket	Ticket Number
fare	Passenger Fare
cabin	Cabin
embarked	Port of Embarkation (C = Cherbourg; Q = Queenstown; S = Southampton)



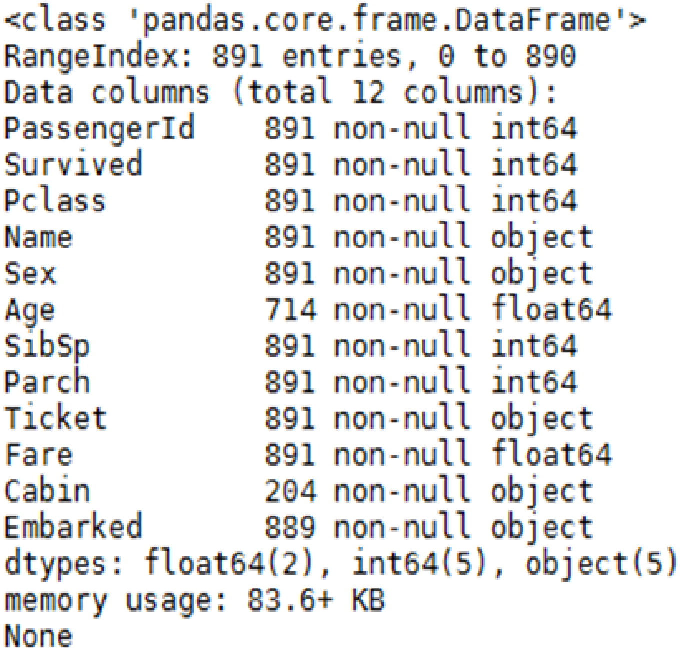
L1 | fx | Embarked

	A	B	C	D	E	F	G	H	I	J	K	
1	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Emba
2	1	0	3	Braund, Mr. Owen Harris	male	22	1	0	A/5 21171	7.25		S
3	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer)	female	38	1	0	PC 17599	71.2833	C85	C
4	3	1	3	Heikkinen, Miss. Laina	female	26	0	0	STON/O2.	7.925		S
5	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1	0	113803	53.1	C123	S
6	5	0	3	Allen, Mr. William Henry	male	35	0	0	373450	8.05		S
7	6	0	3	Moran, Mr. James	male		0	0	330877	8.4583		Q
8	7	0	1	McCarthy, Mr. Timothy J	male	54	0	0	17463	51.8625	E46	S
9	8	0	3	Palsson, Master. Gosta Leonard	male	2	3	1	349909	21.075		S
10	9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27	0	2	347742	11.1333		S
11	10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14	1	0	237736	30.0708		C
12	11	1	3	Sandstrom, Miss. Marguerite Rut	female	4	1	1	PP 9549	16.7	G6	S
13	12	1	1	Bonnell, Miss. Elizabeth	female	58	0	0	113783	26.55	C103	S
14	13	0	3	Saundercock, Mr. William Henry	male	20	0	0	A/5. 2151	8.05		S
15	14	0	3	Andersson, Mr. Anders Johan	male	39	1	5	347082	31.275		S
16	15	0	3	Vestrom, Miss. Hulda Amanda Adolfina	female	14	0	0	350406	7.8542		S
17	16	1	2	Hewlett, Mrs. (Mary D Kingcome)	female	55	0	0	248706	16		S
18	17	0	3	Rice, Master. Eugene	male	2	4	1	382652	29.125		Q
19	18	1	2	Williams, Mr. Charles Eugene	male		0	0	244373	13		S
20	19	0	3	Vander Planke, Mrs. Julius (Emelia Maria Vandemoortele)	female	31	1	0	345763	18		S
21	20	1	3	Masselmani, Mrs. Fatima	female		0	0	2649	7.225		C
22	21	0	2	Fynney, Mr. Joseph J	male	35	0	0	239865	26		S

Describing a DataFrame

- To pull in the text file, we will use the pandas function *read_csv* method.
- The read_csv has a very large number of parameters such as specifying the delimiter, included headers, etc
- Typically it's not very useful to print out an entire dataframe.
- However, there are some useful functions you can use to get summary data.

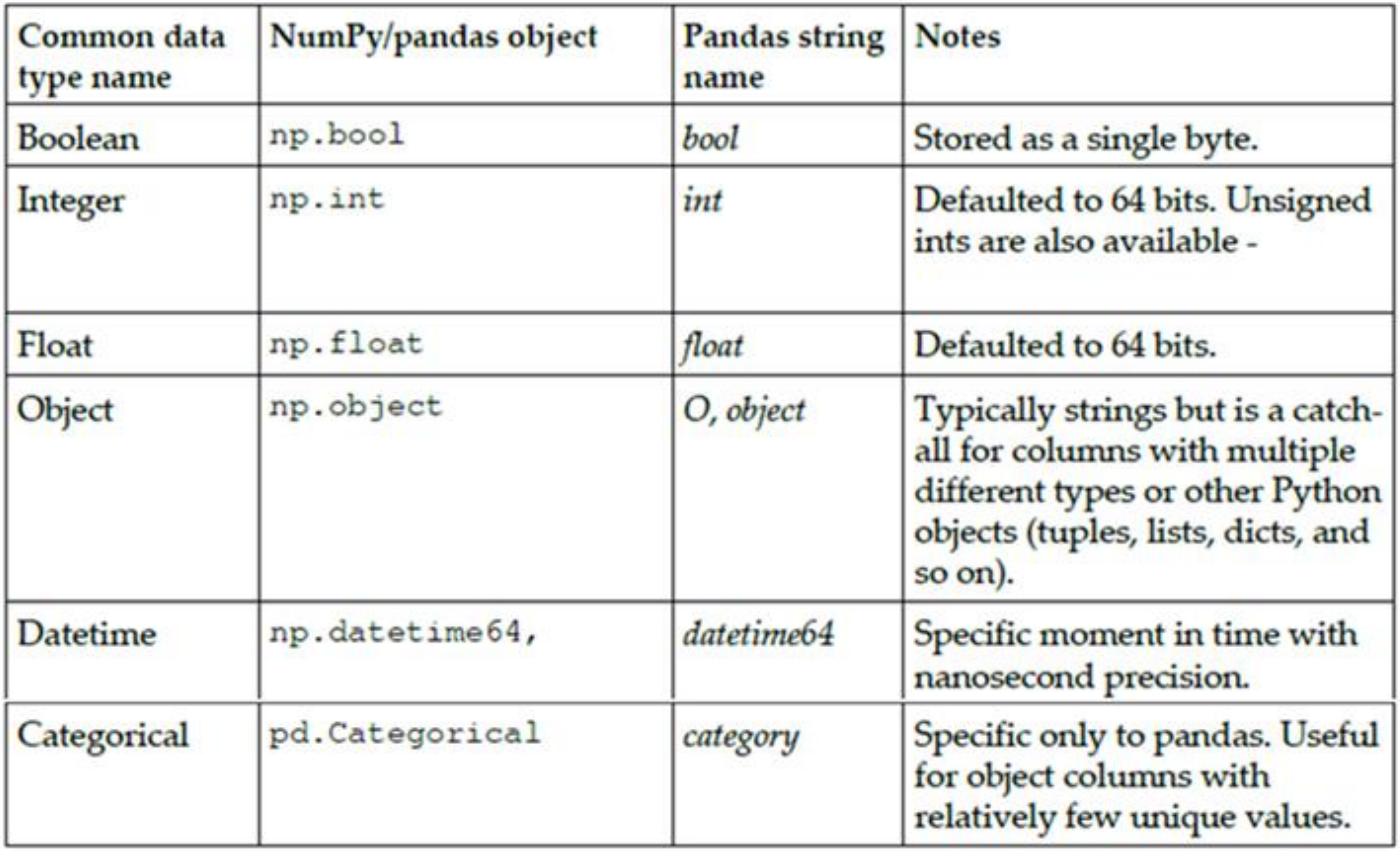
```
import pandas as pd

df = pd.read_csv("titanic.csv")

print (df.info())
```



```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
PassengerId    891 non-null int64
Survived       891 non-null int64
Pclass         891 non-null int64
Name           891 non-null object
Sex            891 non-null object
Age            714 non-null float64
SibSp          891 non-null int64
Parch          891 non-null int64
Ticket         891 non-null object
Fare           891 non-null float64
Cabin          204 non-null object
Embarked       889 non-null object
dtypes: float64(2), int64(5), object(5)
memory usage: 83.6+ KB
None
```

Pandas Data Types

- This table contains all pandas data types, with their string equivalents, and some notes on each type.

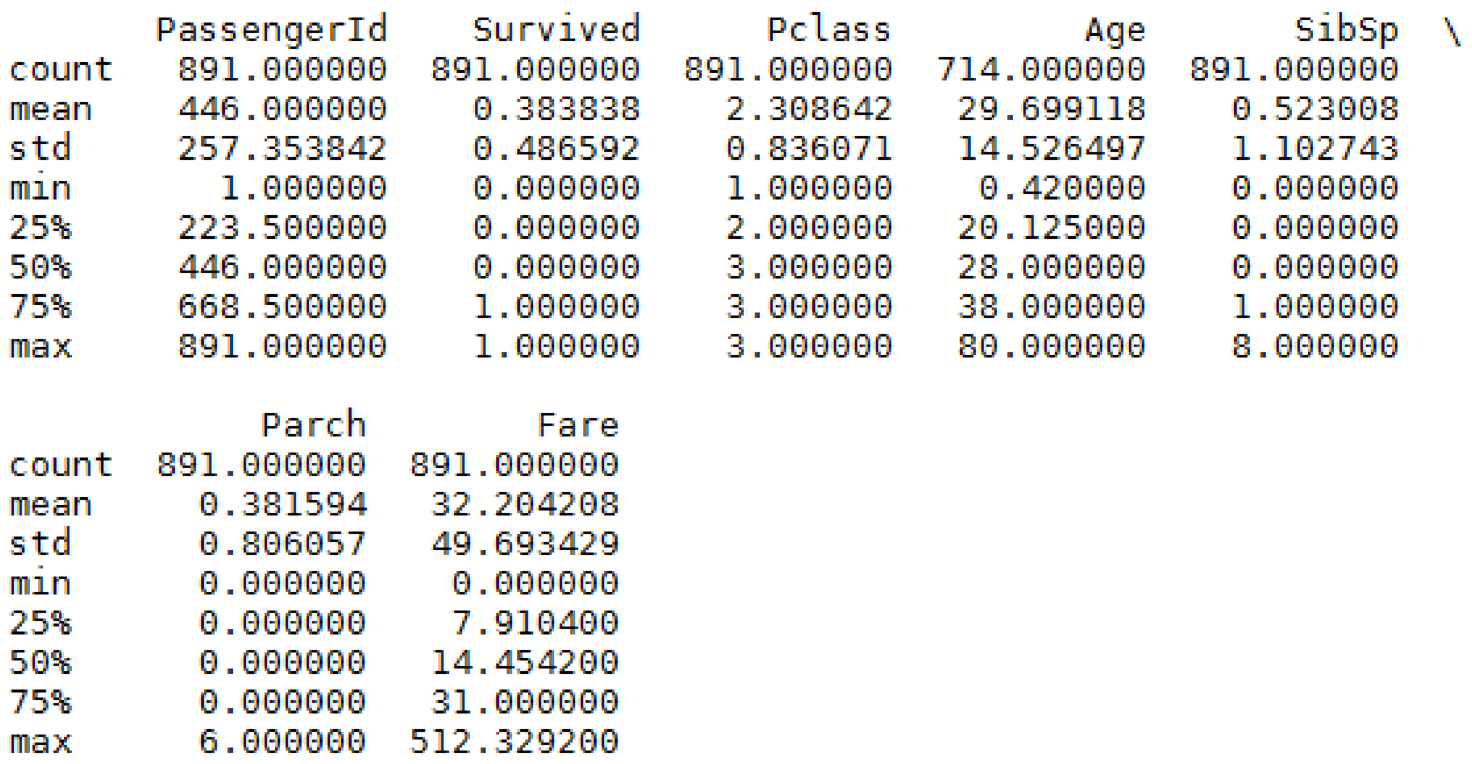


Common data type name	NumPy/pandas object	Pandas string name	Notes
Boolean	`np.bool`	*bool*	Stored as a single byte.
Integer	`np.int`	*int*	Defaulted to 64 bits. Unsigned ints are also available -
Float	`np.float`	*float*	Defaulted to 64 bits.
Object	`np.object`	*O, object*	Typically strings but is a catch-all for columns with multiple different types or other Python objects (tuples, lists, dicts, and so on).
Datetime	`np.datetime64,`	*datetime64*	Specific moment in time with nanosecond precision.
Categorical	`pd.Categorical`	*category*	Specific only to pandas. Useful for object columns with relatively few unique values.

Describing a DataFrame

- DataFrame's also have a useful **describe** method, which is used for viewing **basic statistics** about the dataset's numeric columns.
  - It will return information on all columns of a numeric datatype, therefore some of the data may not be of use .
  - The data type of what is returned is itself a dataframe

```
df = pd.read_csv("titanic.csv")

print (type(df))

print ( df.describe() )
```

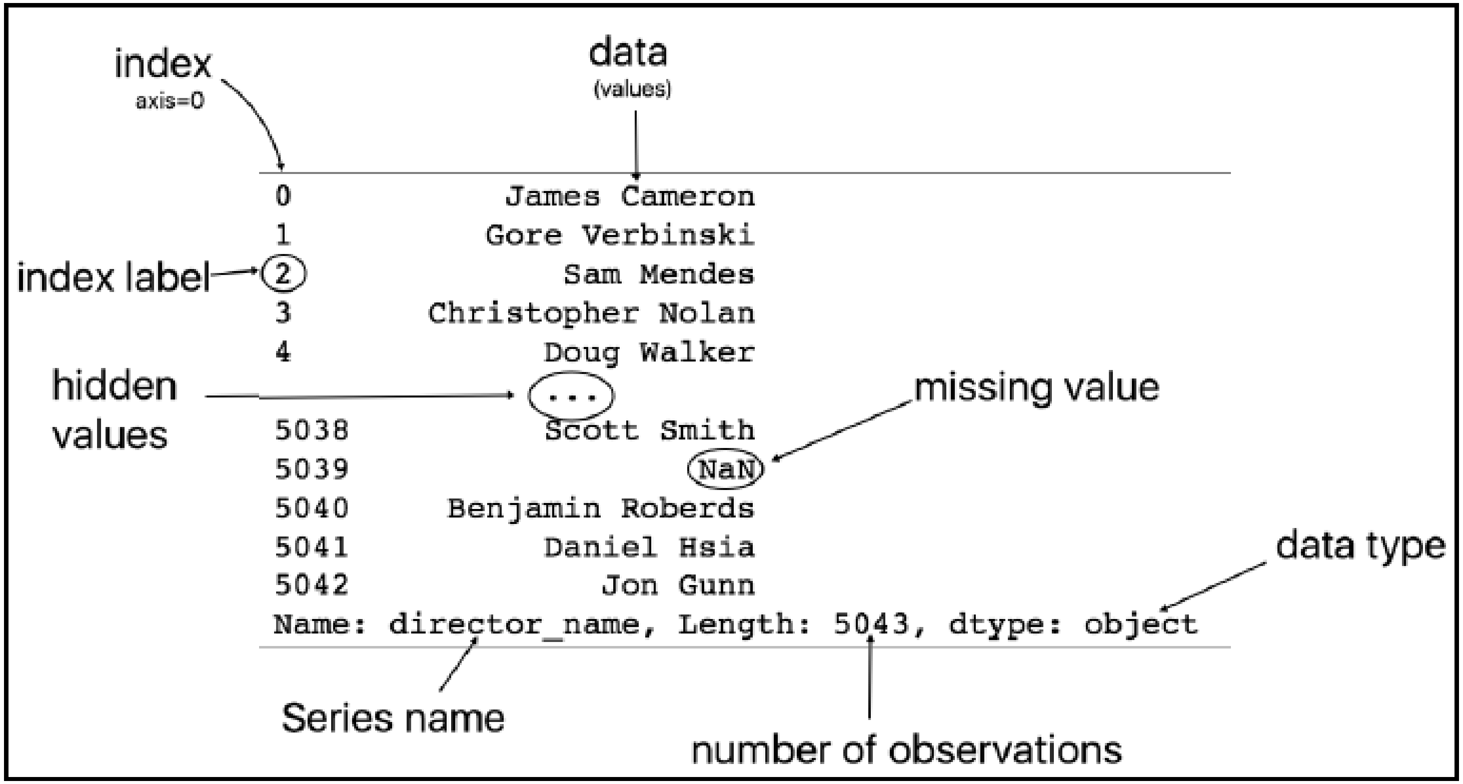


```
       PassengerId    Survived      Pclass         Age       SibSp  \
count   891.000000  891.000000  891.000000  714.000000  891.000000
mean    446.000000    0.383838    2.308642   29.699118    0.523008
std     257.353842    0.486592    0.836071   14.526497    1.102743
min       1.000000    0.000000    1.000000    0.420000    0.000000
25%     223.500000    0.000000    2.000000   20.125000    0.000000
50%     446.000000    0.000000    3.000000   28.000000    0.000000
75%     668.500000    1.000000    3.000000   38.000000    1.000000
max     891.000000    1.000000    3.000000   80.000000    8.000000

            Parch        Fare
count  891.000000  891.000000
mean     0.381594   32.204208
std      0.806057   49.693429
min      0.000000    0.000000
25%      0.000000    7.910400
50%      0.000000   14.454200
75%      0.000000   31.000000
max      6.000000  512.329200
```

We can easily see the average age of the passengers is 29.6 years old, with the youngest being 0.42 and the oldest being 80. The median age is 28, with the youngest quartile of users being 20 or younger, and the oldest quartile being at least 38

Accessing Column Data (A Series Object)

- **Selecting the data in specific columns** of a DataFrame can be easily performed by using the **[]** operator.
- This differs from a Series, where **[]** specified rows. The **[]** operator can be passed either a single object or a list of objects representing the columns to retrieve.
- As we have seen a Series is a single column of data from a DataFrame. It is a single dimension of data, composed of just an index and the data.



```
0           James Cameron
1          Gore Verbinski
2              Sam Mendes
3       Christopher Nolan
4             Doug Walker
              ...
5038          Scott Smith
5039                  NaN
5040      Benjamin Roberds
5041           Daniel Hsia
5042              Jon Gunn
Name: director_name, Length: 5043, dtype: object
```

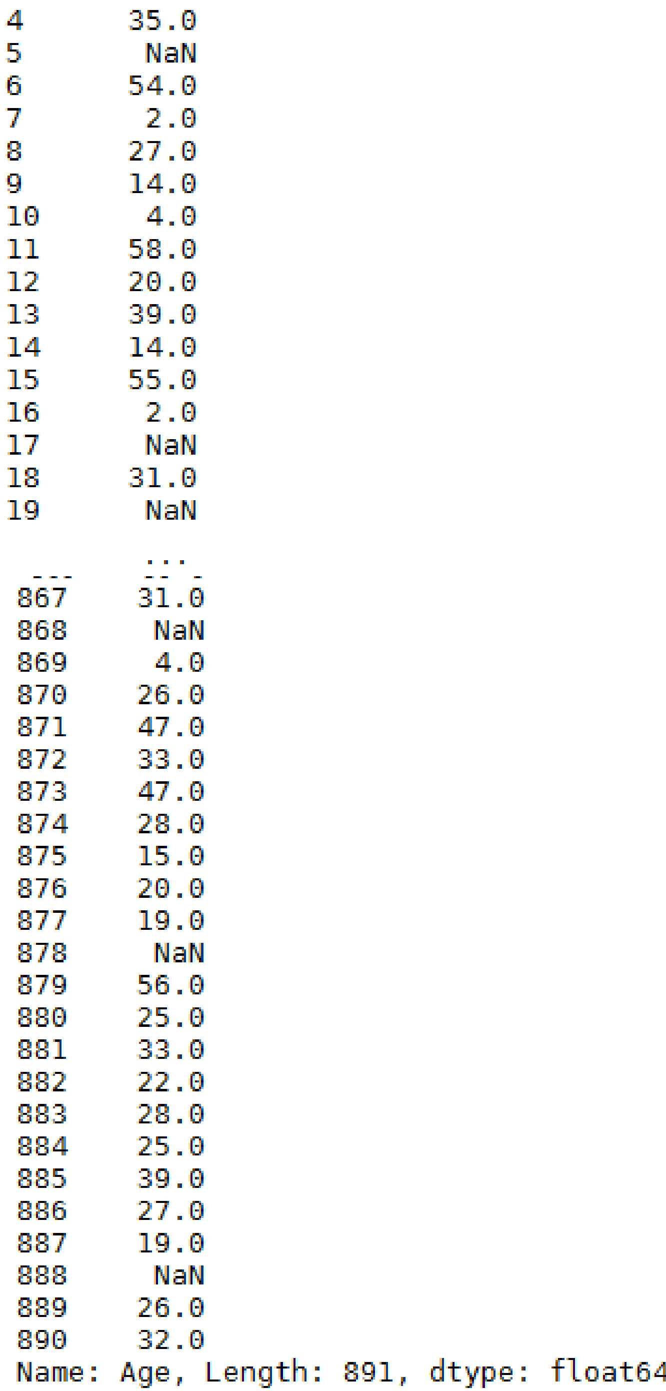
Accessing Column Data

- To select a column, we index with the name of the column:
- **dataframe[‘columnName’]**

```
df = pd.read_csv("titanic.csv")

print ( df['Age'] )
```

- Note this column is returned as a **Series object.**

Alternatively, a column of data may be accessed using the **dot notation** with the column name as an attribute (df.Age). Although it works with this particular example, it is not best practice and is prone to error and misuse. Column names with spaces or special characters cannot be accessed in this manner.



```
4       35.0
5        NaN
6       54.0
7        2.0
8       27.0
9       14.0
10       4.0
11      58.0
12      20.0
13      39.0
14      14.0
15      55.0
16       2.0
17       NaN
18      31.0
19       NaN
         ...
867     31.0
868      NaN
869      4.0
870     26.0
871     47.0
872     33.0
873     47.0
874     28.0
875     15.0
876     20.0
877     19.0
878      NaN
879     56.0
880     25.0
881     33.0
882     22.0
883     28.0
884     25.0
885     39.0
886     27.0
887     19.0
888      NaN
889     26.0
890     32.0
Name: Age, Length: 891, dtype: float64
```

Accessing Columns

- We mentioned in a previous slide that you can also think of a DataFrame as a **group of Series objects** that share an index. When you access an individual column from a dataframe the data type returned is a series.
- Note if you extract multiple columns the data type returned is still a DataFrame.

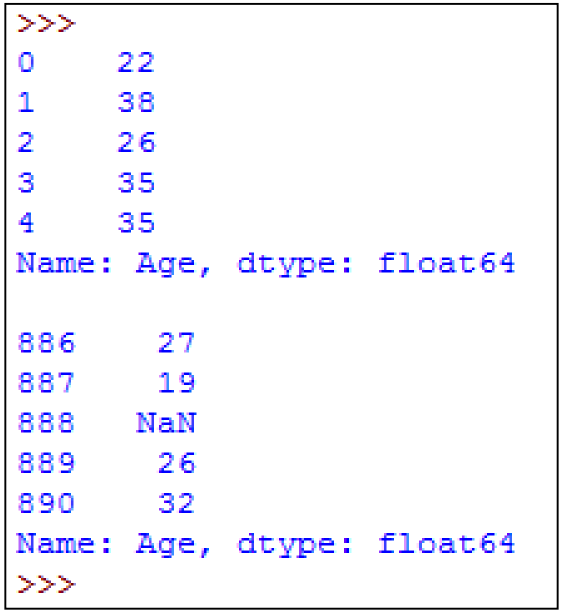
```
df = pd.read_csv("titanic.csv")

ages = df['Age']
print (type(ages))

moreInfo = df[['Age', 'Name']]
print (type(moreInfo))
```

```
<class 'pandas.core.series.Series'>
<class 'pandas.core.frame.DataFrame'>
```

Using Head and Tail

- To view a small sample of a **Series or DataFrame** object, use the head (start) and tail (end) methods. The default number of elements to display is five, but you can pass a number as an argument.

```
df = pd.read_csv("titanic.csv")
freqAges = df['Age']
print (freqAges.head())

print (freqAges.tail())
```



```
>>>
0     22
1     38
2     26
3     35
4     35
Name: Age, dtype: float64

886    27
887    19
888   NaN
889    26
890    32
Name: Age, dtype: float64
>>>
```

- If I want to capture the last 7 age values in the dataset

```
df = pd.read_csv("titanic.csv")
print (df["Age"].tail(7))
```

Using loc and iloc

- A DataFrame consists of both rows and columns, and as a result has constructs to select data from specific rows and columns.

- We have already seen the use of **[]** but Pandas also allows us to access data using .loc[], and .iloc[].

- The **.loc** function is primarily **label based indexing**, but may also be used with a Boolean indexing

- The **iloc** function is used for **integer-location based indexing** and is similar to what we used in NumPy (it cannot use Boolean indexing)

- Both .loc and .iloc work with Series and DataFrames.
- These indexers not only take scalar values, but also lists and slices.

Using loc on a Series

- **Rows** can be retrieved via an index label value using .loc[]
- To illustrate the operation on a Series we will use the cities Series containing city names and crime values.

```
Belfast     300
Cork        150
Dublin      245
Galway      360
Limerick    125
dtype: int64
```

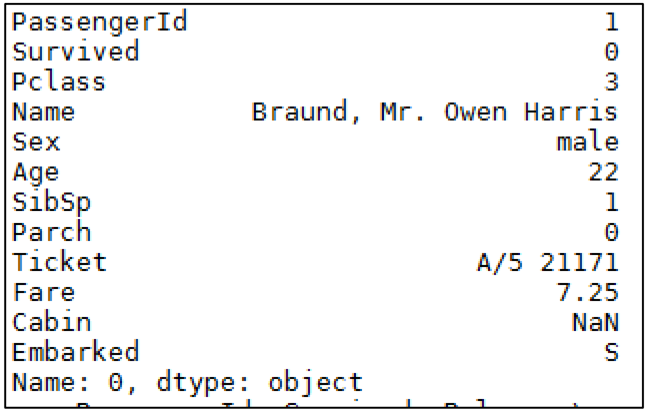
```
print (cities.loc[ 'Dublin‘ ])

print (cities.loc[ ['Dublin', 'Galway'] ])

print (cities.loc[ [True, False, True] ])

print (cities.loc[ cities>200 ])
```

```
245

Dublin    245
Galway    360
dtype: int64

Belfast    300
Dublin     245
dtype: int64

Belfast    300
Dublin     245
Galway     360
dtype: int64
```

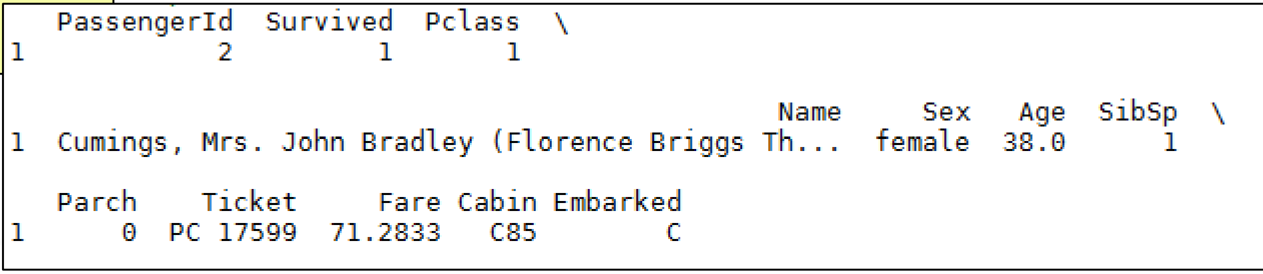
Using loc on a DataFrame

- Rows can be retrieved via an index label value using .loc[] on an entire dataframe
- Note, in the titanic dataset we use an **integer index** but the value passed to loc could also be a **String** index if applicable. We illustrate this in the next slide.

```
import pandas as pd

df = pd.read_csv("titanic.csv")


print ( df.loc[0] )

print (df.loc[  [1, 20] ])

print (df.loc[ [False, True] ] )
```



```
PassengerId                        1
Survived                           0
Pclass                             3
Name             Braund, Mr. Owen Harris
Sex                             male
Age                               22
SibSp                              1
Parch                              0
Ticket                     A/5 21171
Fare                            7.25
Cabin                            NaN
Embarked                           S
Name: 0, dtype: object
```

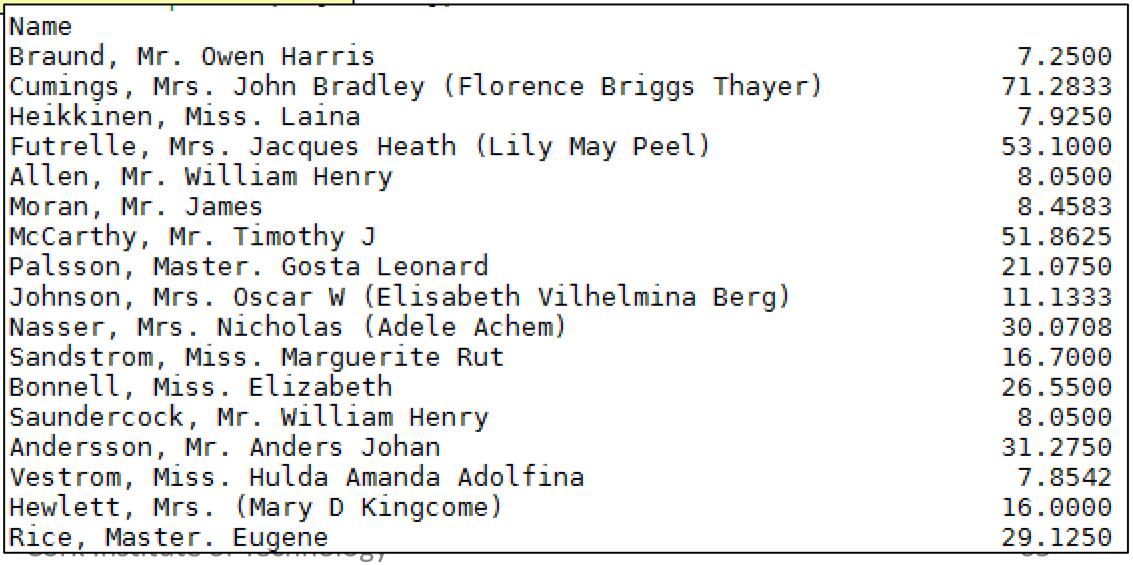


```
   PassengerId  Survived  Pclass  \
1            2         1       1

                                                Name     Sex   Age  SibSp  \
1  Cumings, Mrs. John Bradley (Florence Briggs Th...  female  38.0      1

   Parch    Ticket     Fare Cabin Embarked
1      0  PC 17599  71.2833   C85        C
```

Using loc on a DataFrame

- To properly illustrate the use of loc on a DataFrame when we read in the DataFrame we index it using the name column as shown below (previously it Pandas automatically generated an integer based index starting at 0)

```
import pandas as pd

df = pd.read_csv("titanic.csv", index_col='Name')

print (df["Fare"])
```



```
Name
Braund, Mr. Owen Harris                                     7.2500
Cumings, Mrs. John Bradley (Florence Briggs Thayer)        71.2833
Heikkinen, Miss. Laina                                      7.9250
Futrelle, Mrs. Jacques Heath (Lily May Peel)               53.1000
Allen, Mr. William Henry                                    8.0500
Moran, Mr. James                                            8.4583
McCarthy, Mr. Timothy J                                    51.8625
Palsson, Master. Gosta Leonard                             21.0750
Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)          11.1333
Nasser, Mrs. Nicholas (Adele Achem)                        30.0708
Sandstrom, Miss. Marguerite Rut                            16.7000
Bonnell, Miss. Elizabeth                                   26.5500
Saundercock, Mr. William Henry                              8.0500
Andersson, Mr. Anders Johan                                31.2750
Vestrom, Miss. Hulda Amanda Adolfina                        7.8542
Hewlett, Mrs. (Mary D Kingcome)                            16.0000
Rice, Master. Eugene                                       29.1250
```

Using loc on a DataFrame

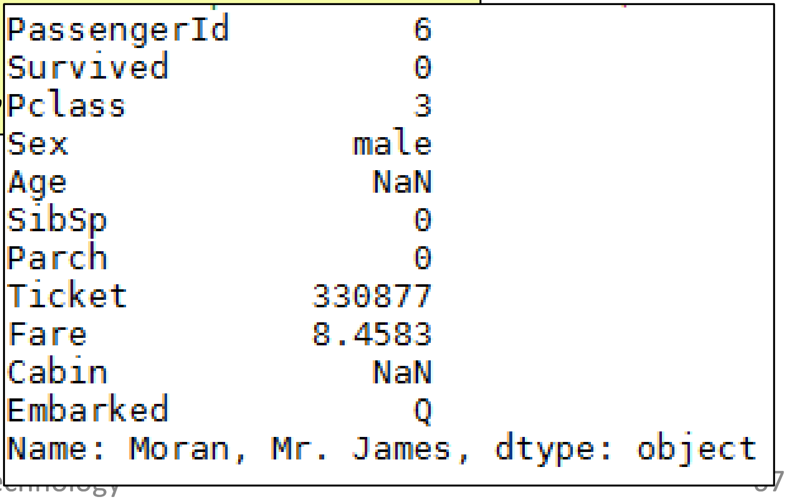
- To properly illustrate the use of loc on a DataFrame when I read in the DataFrame I index is using the name column as shown below

```
df = pd.read_csv("titanic.csv", index_col='Name')

print (df.loc[ "Moran, Mr. James“ ])

print (df.loc[  [True, False, True]  ])

print (df.loc[ "Moran, Mr. James": "Bonnell, Miss. Elizabeth“ ])
```

Using loc on a DataFrame

- To properly illustrate the use of loc on a DataFrame when I read in the DataFrame I index is using the name column as shown below

```
df = pd.read_csv("titanic.csv", index_col='Name')

print (df.loc[ "Moran, Mr. James“ ])

print (df.loc[  [True, False, True]  ])

print (df.loc[ "Moran, Mr. James": "Bonnell,
```

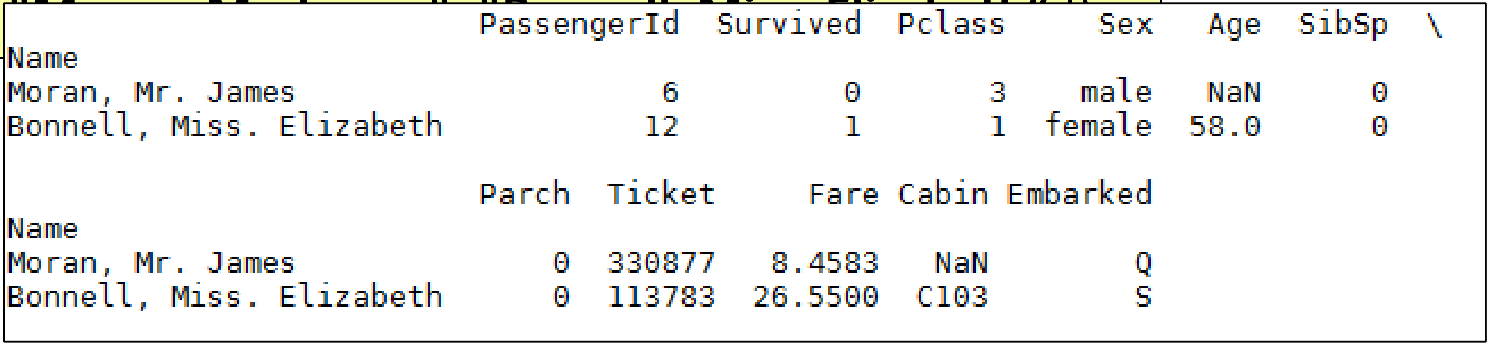


```
PassengerId         6
Survived            0
Pclass              3
Sex              male
Age               NaN
SibSp               0
Parch               0
Ticket         330877
Fare           8.4583
Cabin             NaN
Embarked            Q
Name: Moran, Mr. James, dtype: object
```

Using loc on a DataFrame

- To properly illustrate the use of loc on a DataFrame when I read in the DataFrame I index is using the name column as shown below

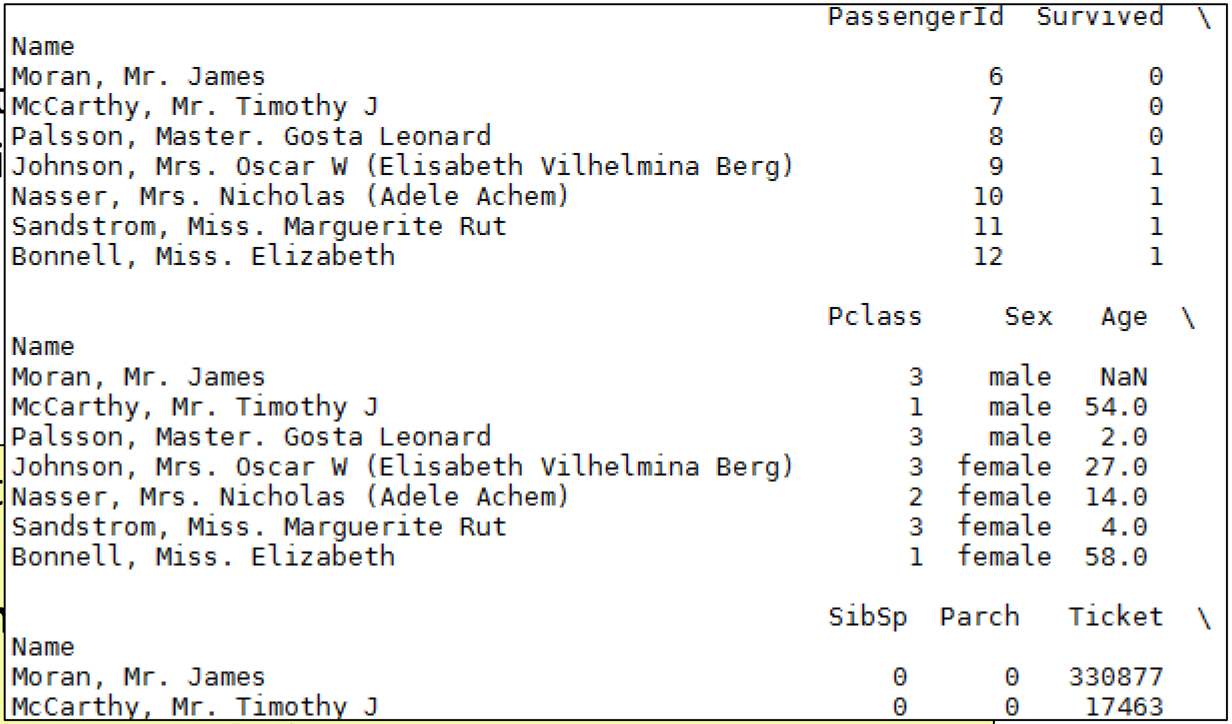
```
df = pd.read_csv("titanic.csv", index_col='Name')

print (df.loc[ "Moran, Mr. James“ ])

print (df.loc[  [True, False, True]  ])

print (df.loc[
```



```
                          PassengerId  Survived  Pclass     Sex   Age  SibSp  \
Name
Moran, Mr. James                    6         0       3    male   NaN      0
Bonnell, Miss. Elizabeth           12         1       1  female  58.0      0

                          Parch  Ticket     Fare Cabin Embarked
Name
Moran, Mr. James              0  330877   8.4583   NaN        Q
Bonnell, Miss. Elizabeth      0  113783  26.5500  C103        S
```

Using loc on a DataFrame

- To properly illustrat
  DataFrame I index i



```
                                                     PassengerId  Survived  \
Name
Moran, Mr. James                                               6         0
McCarthy, Mr. Timothy J                                        7         0
Palsson, Master. Gosta Leonard                                 8         0
Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)              9         1
Nasser, Mrs. Nicholas (Adele Achem)                           10         1
Sandstrom, Miss. Marguerite Rut                               11         1
Bonnell, Miss. Elizabeth                                      12         1

                                                     Pclass     Sex   Age  \
Name
Moran, Mr. James                                          3    male   NaN
McCarthy, Mr. Timothy J                                   1    male  54.0
Palsson, Master. Gosta Leonard                            3    male   2.0
Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)         3  female  27.0
Nasser, Mrs. Nicholas (Adele Achem)                       2  female  14.0
Sandstrom, Miss. Marguerite Rut                           3  female   4.0
Bonnell, Miss. Elizabeth                                  1  female  58.0

                                                     SibSp  Parch   Ticket  \
Name
Moran, Mr. James                                         0      0   330877
McCarthy, Mr. Timothy J                                  0      0    17463
```

```
df = pd.read_csv("tit

print (df.loc[ "Moran

print (df.loc[  [True, False, True]  ])

print (df.loc[ "Moran, Mr. James": "Bonnell, Miss. Elizabeth“ ])
```

Using iloc on a Series

```
Belfast     300
Cork        150
Dublin      245
Galway      360
Limerick    125
dtype: int64
```

- The iloc function allows us to access via an integer index. Again to illustrate the operation on a Series we will use the cities Series object.
- Note you **cannot use Boolean indexing** with iloc on a Series object.

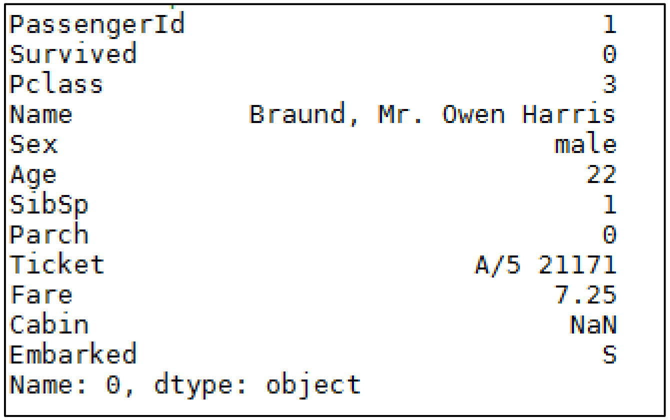
```
print (cities.iloc[0])

print (cities.iloc[[0,1]])

print (cities.iloc[0::2])
```

```
300

Belfast    300
Cork       150
dtype: int64


Belfast     300
Dublin      245
Limerick    125
dtype: int64
```

Using iloc on a DataFrame

- We can also access specific rows using the iloc function.
- Again we specify an integer index and it returns the corresponding row.
- As always index 0 refers to the first row.

```
import pandas as pd

df = pd.read_csv("titanic.csv")

# prints the entire first row
#(returned as a Series object)
print ( df.iloc[ 0 ] )

# selects the row with index 1
#and 3 (DataFrame obj)
print ( df.iloc[ [1, 3] ] )

# prints the index 4, 7 and 10
print ( df.iloc[ 4:11:3 ] )
```

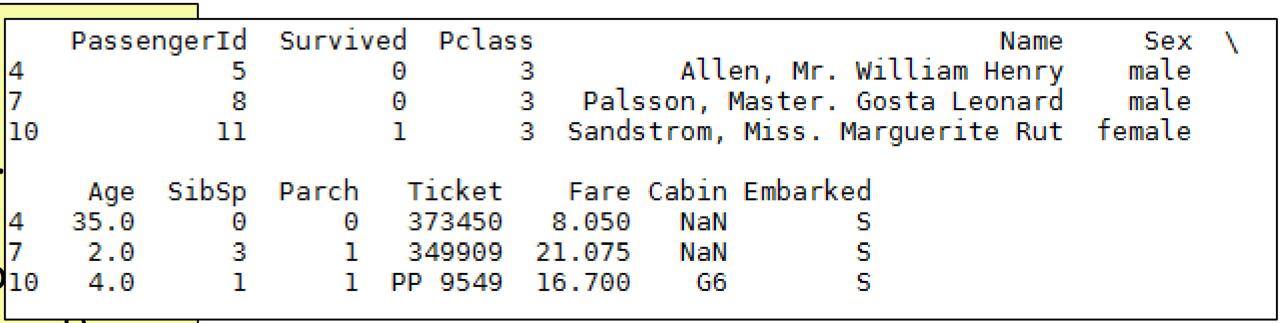


```
PassengerId                         1
Survived                            0
Pclass                              3
Name           Braund, Mr. Owen Harris
Sex                              male
Age                                22
SibSp                               1
Parch                               0
Ticket                      A/5 21171
Fare                             7.25
Cabin                             NaN
Embarked                            S
Name: 0, dtype: object
```

Using iloc on a DataFrame

- We can also access specific rows using the iloc function.
- Again we specify an integer index and it returns the corresponding row.

```
import pandas as pd

df = pd.read_csv("titanic.

# prints the entire first ro
#(returned as a Series object)
print (df.iloc[0])

# selects the row with index 1
#and 3 (DataFrame obj)
print (df.iloc[[1, 3]])

# prints the index 4, 7 and 10
print (df.iloc[4:11:3])
```



```
    PassengerId  Survived  Pclass                            Name     Sex  \
4             5         0       3        Allen, Mr. William Henry    male
7             8         0       3  Palsson, Master. Gosta Leonard    male
10           11         1       3 Sandstrom, Miss. Marguerite Rut  female

    Age  SibSp  Parch    Ticket    Fare Cabin Embarked
4  35.0      0      0    373450   8.050   NaN        S
7   2.0      3      1    349909  21.075   NaN        S
10  4.0      1      1   PP 9549  16.700    G6        S
```

Selecting rows and columns simultaneously

- Directly using the indexing operator [] is the correct method to select one or more **columns** from a DataFrame.

- However, it does not allow you to select both rows and columns simultaneously.
- To **select rows and columns simultaneously**, you will need to pass both valid row and column selections separated by a comma to either the .iloc or .loc indexers.

- The generic form to select rows and columns will look like the following code:
  - **df.iloc[rows, columns]**
  - **df.loc[rows, columns]**

- The rows and columns variables may be scalar values, lists, slice objects, or Boolean sequences.

Selecting DataFrame rows and columns simultaneously - iloc

- The method of using iloc is similar to how we selected data in NumPy
- The syntax is data.iloc[<row selection>, <column selection>]

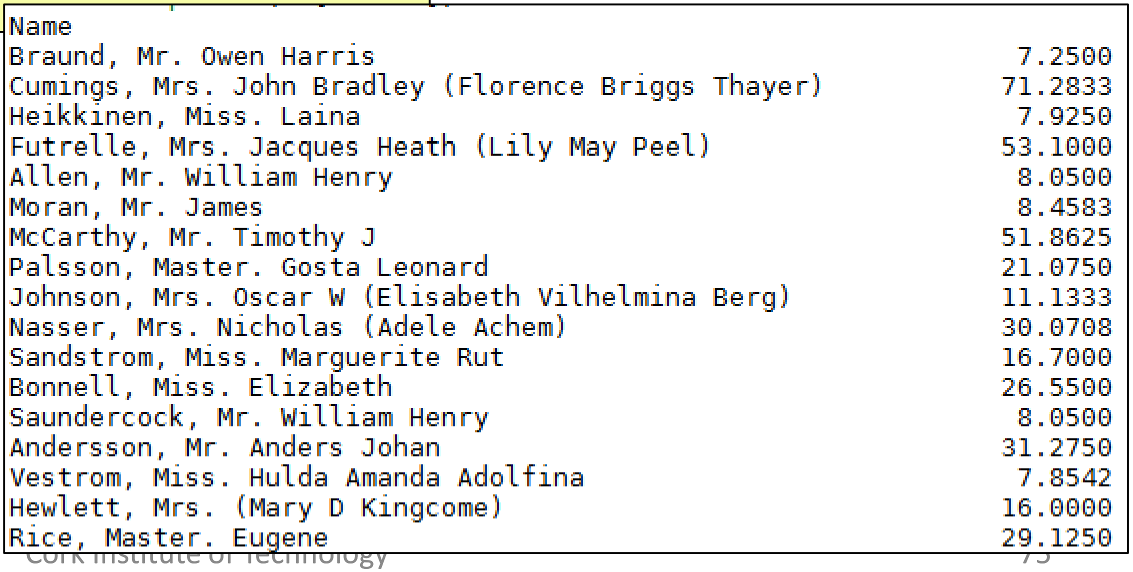
```
print (df.iloc[0])

print (df.iloc[:, 0])

print ( df.iloc[:, 0:3])

print ( df.iloc[:, [0,3]])
```

1. Print out the first row
2. Print out the first column (Series object)
3. Print out the first, second and third column (DataFrame Object)
4. Print out the first and fourth columns (DataFrame Object)

Selecting DataFrame rows and columns simultaneously - loc

- To illustrate the operation of the loc function, we previously we used a version of the Titanic dataset, where the passenger name as used as the index. We do the same here.

```
import pandas as pd

df = pd.read_csv("titanic.csv", index_col='Name')
print (df["Fare"])
```



```
Name
Braund, Mr. Owen Harris                                       7.2500
Cumings, Mrs. John Bradley (Florence Briggs Thayer)          71.2833
Heikkinen, Miss. Laina                                        7.9250
Futrelle, Mrs. Jacques Heath (Lily May Peel)                 53.1000
Allen, Mr. William Henry                                      8.0500
Moran, Mr. James                                              8.4583
McCarthy, Mr. Timothy J                                      51.8625
Palsson, Master. Gosta Leonard                               21.0750
Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)            11.1333
Nasser, Mrs. Nicholas (Adele Achem)                          30.0708
Sandstrom, Miss. Marguerite Rut                              16.7000
Bonnell, Miss. Elizabeth                                     26.5500
Saundercock, Mr. William Henry                                8.0500
Andersson, Mr. Anders Johan                                  31.2750
Vestrom, Miss. Hulda Amanda Adolfina                          7.8542
Hewlett, Mrs. (Mary D Kingcome)                              16.0000
Rice, Master. Eugene                                         29.1250
```

Selecting DataFrame rows and columns simultaneously - loc

```
import pandas as pd

df = pd.read_csv("titanic.csv", index_col='Name')

print (df.loc["Moran, Mr. James", "Fare"])

print (df.loc[:, ["Fare","Sex"]])

print (df.loc[[True, True, False], ["Fare","Sex"]])
```

Accesses the row indexed “Moran, Mr. James", and the column “Fare”, which returns the value 8.4583

Selecting DataFrame rows and columns simultaneously - loc

```
import pandas as pd

df = pd.read_csv("titanic.csv", index_col='Name')

print (df.loc["Moran, Mr. James", "Fare"])

print (df.loc[:, ["Fare","Sex"]])

print (df.loc[[True, True, False], ["Fare","Sex"]])
```

Accesses all rows but just the columns Fare and Sex

Selecting DataFrame rows and columns simultaneously - loc

```
import pandas as pd

df = pd.read_csv("titanic.csv", index_col='Name')

print (df.loc["Moran, Mr. James", "Fare"])

print (df.loc[:, ["Fare","Sex"]])

print (df.loc[[True, True, False], ["Fare","Sex"]])
```

We can use Boolean indexing to select specific rows. For example, here we select the first two rows but not the third and the Fare and Sex column

Counting – value_counts()

- A very useful method **value_counts()** can be used to count the **number of occurrences of each entry** in a column (it returns a Series object)

- It presents the results in **descending** order
- For examples, how many males and females are represented in dataset

```
df = pd.read_csv("titanic.csv")
print (df['Sex'].value_counts())
```

```
male      577
female    314
dtype: int64
```

Counting – value_counts()

- A very useful method **value_counts()** can be used to count the **number of occurrences of each entry** in a column (it returns a Series object)

- It presents the results in **descending** order
- For examples, how many males and females are represented in dataset

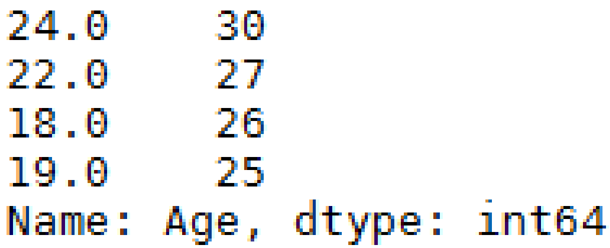
```
import pandas as pd

df = pd.read_csv("titanic.csv")

df = pd.read_csv("titanic.csv")
print (df['Sex'].value_counts(normalize=True))
```

```
male      0.647587
female    0.352413
Name: Sex, dtype: float64
```

Example 1

- Read data in from the titanic dataset and determine the four most common ages represented.

```
df = read_csv("titanic.csv")
freqAges = df['Age']
print (freqAges.value_counts().head(4))
```



```
24.0    30
22.0    27
18.0    26
19.0    25
Name: Age, dtype: int64
```

Performing Operations

- We can perform the same mathematical operations in Pandas as we could in NumPy

```
import pandas as pd

  df = pd.read_csv("titanic.csv")
  print ("Average age", np.mean(df["Age"]))

  print (df["Age"].head(5))
  df["Age"] += 5

  print (df["Age"].head(5))
```

```
Average age 29.6991176471
0    22
1    38
2    26
3    35
4    35
Name: Age, dtype: float64

0    27
1    43
2    31
3    40
4    40
Name: Age, dtype: float64
```

Summary

- NumPy 2D Arrays
  - [row, column] access
  - Slice operations [start:stop:step]
  - Performing operations of a specific axis (np.sum(arr1, axis = 0))
  - Comparison Operators
  - Advanced Index (Boolean index with comparison operation, interger list)
  - Logical Operators

- Pandas
  - Series and DataFrame
  - Accessing Columns
  - Using label based indexing (loc) and integer based indexing (iloc)