# Data Mining & Informatics
# Pandas & Numpy

Amin Noroozi
University of Wolverhampton
✉ a.noroozifakhabi@wlv.ac.uk
https://www.linkedin.com/in/amin-n-148350218/

# Pandas

# Python Pandas

- **Introduction**

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- Pandas allows us to analyze big data and make conclusions based on statistical theories.
- Pandas can clean messy data sets, and make them readable and relevant.
- The source code for Pandas is located at this github repository https://github.com/pandas-dev/pandas

# Python Pandas

- **Introduction**

- If you have Python and PIP already installed on a system, then installation of Pandas is very easy.
- You can install Pandas on your system by using pip in the windows command line as follows:

```
pip install pandas
```

- You can import Pandas as follows:

```
import pandas
```

- Pandas is usually imported under the pd alias.

```
import pandas as pd
```

- You can check Pandas version as follows:

```
import pandas as pd
print(pd.__version__)
```

# Python Pandas

- **DataFrames**

- A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.
- You can use the DataFrame() method to create a Pandas DataFrame from a dictionary, numpy array, or a list.

```
import pandas as pd
data = {"calories": [420, 380, 390], "duration": [50, 40, 45]}
#load data into a DataFrame object:
df = pd.DataFrame(data)
print(df)
---
data = [[420, 380, 390],[50, 40, 45]]
df = pd.DataFrame(data)
print(df)
```

# Python Pandas

- **DataFrames indexing**

- You can use loc() or iloc() to return values, rows, or columns inside a data frame. Using loc() you need to specify the column name, but using iloc() you only need to specify the column index.

```
import pandas as pd
data = {"calories": [420, 380, 390], "duration": [50, 40, 45]}
#load data into a DataFrame object:
df = pd.DataFrame(data)
print(df)
print("\n")
print(df.loc[0,'calories'])
print("\n")
print(df.loc[0,:])
print("\n")
print(df.loc[:,'calories'])
```

# Python Pandas

- **DataFrames indexing**

- You can use loc() or iloc() to return values, rows, or columns inside a data frame. Using loc() you need to specify the column name, but using iloc() you only need to specify the column index.

```
import pandas as pd
data = {"calories": [420, 380, 390], "duration": [50, 40, 45]}
#load data into a DataFrame object:
df = pd.DataFrame(data)
print(df)
print("\n")
print(df.loc[0, "duration"])
print("\n")
print(df.iloc[0,:])
print("\n")
print(df.iloc[:,0])
```

# Python Pandas

- **DataFrames indexing**

- You can get the indices and columns names using index and column attributes as follows

```
import pandas as pd
data = {"calories": [420, 380, 390], "duration": [50, 40, 45]}
#load data into a DataFrame object:
df  = pd.DataFrame(data)

index_names=list(df.index)
column_names=list(df.columns)

print(index_names,end='\n\n')
print(column_names)
```

# Python Pandas

- **DataFrames indexing**

- With the index and columns argument, you can name your own indexes and columns.

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

df = pd.DataFrame(data, index = ["day1", "day2", "day3"])

print(df)
```

# Python Pandas

- **DataFrames indexing**

- With the index argument, you can name your own indexes and columns.

```
import pandas as pd

data =[[420, 380, 390],[50, 40, 45]]

df = pd.DataFrame(data, columns = ["c1", "c2", "c3"])

print(df)
```

# Python Pandas

- **Reading CSV and JSON**

- You can import CSV and JSON files into a Pandas DataFrame using read_csv() and read_json() as follows:

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df)
```

---

```
import pandas as pd
df = pd.read_json('data.json')
print(df)
```

# Python Pandas

- **Writing CSV and JSON**

- You can export CSV and JSON files into a folder using to_csv() and to_json() as follows:

```
import pandas as pd
df = pd.DataFrame({'name': ['Raphael', 'Donatello'],
                   'mask': ['red', 'purple'],
                   'weapon': ['sai', 'bo staff']})

df.to_csv('C:/Users/Lenovo/Desktop/out.csv')
```

# Python Pandas

- **Extaracting columns**

- You can extract DataFrame columns using the columns' names or indices

```
# First method: using column names

import pandas as pd
df  = pd.DataFrame([['c',3,4,5], ['d',4,6,7]],columns=['c1', 'c2','c3','c4'])


df1=df[['c1','c2']] #extract column c1 and c2

print(df)
print('\n')

print(df1)
```

# Python Pandas

- **Extaracting columns**

- You can extract DataFrame columns using the columns' names or indices

```
# Second method: using columns indices

import pandas as pd
df  = pd.DataFrame([['c',3,4,5], ['d',4,6,7]],columns=['c1', 'c2','c3','c4'])

columns=df.columns # Getting columns as a list
df2=df[columns[2:4]] # Getting the third and fourth columns which are c3 an c4

print(df)
print('\n')

print(df2)
```

# Python Pandas

- **Deleting columns**

- Using the drop() method we can delete columns from a DataFrames. We can use columns' names or indices to delete the columns

```
# First method: Using columns names
import pandas as pd

df = pd.DataFrame([['c',3,4,5], ['d',4,6,7]],
                  columns=['c1', 'c2','c3','c4'])

df1=df.drop(['c1','c2'],axis=1)
print(df)
print('\n')
print(df1)
```

# Python Pandas

- **Deleting columns**

- Using the drop() method we can delete columns from a DataFrames. We can use columns' names or indices to delete the columns

```
# Second method: Using columns indices
import pandas as pd

df = pd.DataFrame([['c',3,4,5], ['d',4,6,7]],
                  columns=['c1', 'c2','c3','c4'])

columns=df.columns
df2=df.drop(columns[2:4],axis=1) # Dropping the third and fourth columns, i.e. c3 and c4
print(df)
print('\n')
print(df2)
```

# Python Pandas

- **Combining DataFrames**

- Using pd.concat() we can combine two or more DataFrames

```
import pandas as pd
df1 = pd.DataFrame([['a', 1], ['b', 2]],
                   columns=['letter', 'number'])

df2 = pd.DataFrame([['c', 3], ['d', 4]],
                   columns=['letter', 'number'])
df3=pd.concat([df1, df2])
df4=pd.concat([df1, df2], axis=1)
print(df1)
print(df2)
print(df3)
print(df4)
```

# Numpy

# NOTES

- **We may use the following online Python IDE for this lecture:**

https://www.w3schools.com/python/trypython.asp?filename=demo_compiler

# Python NumPy

- **Introduction**

- Numpy is a Python library used for working with arrays. Numpy is short for numercal Python.
- It also has functions for working in the domain of linear algebra and matrices.
- In Python we have lists that serve the purpose of arrays but are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are frequently used in data science, where speed and resources are important.
- The source code for NumPy is located at this github repository https://github.com/numpy/numpy

# Python NumPy

- **Introduction**

- If you have Python and PIP already installed on a system, then installation of NumPy is very easy.

- We can install NumPy using the following command in the Windows command line:

pip install nNumPy

- We can import NumPy as follows:

import numpy

- NumPy is usually imported under the np alias.

import numpy as np

- The version string is stored under __version__ attribute.

```
import numpy as np
print(np.__version__)
```

# Python NumPy

- **Creating Numpy arrays**

- NumPy is used to work with arrays. The array object in NumPy is called ndarray.
- We can create a NumPy ndarray object by using the array() function.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

- To create a ndarray, we can pass a list, tuple, or any array-like object into the array() method, and it will be converted into a ndarray

```
import numpy as np
arr = np.array((1, 2, 3, 4, 5))
print(arr)
```

# Python NumPy

- **Array dimension**

- A dimension in arrays is one level of array depth (nested arrays). Nested arrays are arrays that have arrays as their elements.

```
import numpy as np
arr = np.array(42)        # Zero dimensional array
print(arr)
------
arr = np.array([1, 2, 3, 4, 5]) # 1 dimensional array
print(arr)
-------
arr = np.array([[1, 2, 3], [4, 5, 6]])   # 2 dimensional array
print(arr)
```

**NOTE**: These are often used to represent matrix or 2nd-order tensors. NumPy has a whole sub-module dedicated towards matrix operations called NumPy.mat

```
-----
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])   # 3 dimensional array
print(arr)
```

# Python NumPy

- **Array dimension**

- NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

# Python NumPy

- **Array type and size**

- Using size and type we can get the total number of elements in an array and its type.

```
import numpy as np
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(type(d))
Print(d.size)
```

# Python NumPy

- **Array indexing**

- You can access an array element by referring to its index number. The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
----
arr = np.array([1, 2, 3, 4])
print(arr[2] + arr[3])
----
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])          # 2 dimensional
print('2nd element on 1st row: ', arr[0, 1])
----
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('5th element on 2nd row: ', arr[1, 4])
----
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])  # 3 dimensional
print(arr[0, 1, 2])
```

# Python NumPy

- **Array indexing**

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

print('Last element from 2nd dim: ', arr[1, -1])
```

# Python NumPy

- **Array slicing**

- We can slice and array using the following indexing: [start:end:step]. If we don't pass start its considered 0. The result includes the start index but excludes the end index.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5])
---
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[4:])
---
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[:4])
---
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1])
```

# Python NumPy

- **Array slicing**

```
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[1:5:2])
----
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
----
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 2])
----
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[0:2, 1:4])
```

# Python NumPy

- **Array slicing**

**Task**: Write a NumPy program to create an 8x8 matrix and fill it with a checkerboard pattern.

Checkerboard pattern:

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

```
import numpy as np
x = np.ones((3,3))
print("Checkerboard pattern:")
x = np.zeros((8,8),dtype=int)
x[1::2,::2] = 1
x[::2,1::2] = 1
print(x)
```

# Python NumPy

- **np.arrange()**

The advantage of numpy.arange() over the normal in-built range() function is that it allows us to generate sequences of numbers that are not integers.
**Example**:

```
import numpy as np

# Printing all numbers from 1 to
# 2 in steps of 0.1
print(np.arange(1, 2, 0.1))
```

# Python NumPy

- **np.zeros() and np.ones()**

These two functions could be used to create arrays with zeros or ones.
**Example**:

```
import numpy as np

print(np.zeros([3,3],dtype=int))

print(np.ones([3,3],dtype=int))
```

# Python NumPy

- **np.linspace()**

The numpy.linspace() function returns numbers evenly spread over an interval similar to np.arrange() function but instead of a step, it uses a sample number.

**Syntax**:
numpy.linspace(start, stop, num = 50, endpoint = True, retstep = False, dtype = None)

-> **start** : [optional] start of interval range. By default start = 0
-> **stop** : end of interval range
-> **restep** : If True, return (samples, step). By default restep = False
-> **num** : [int, optional] No. of samples to generate
-> **dtype** : type of output array

**Example**:

```
import numpy as np
print(np.linspace(2.0, 3.0, num=5, retstep=True, endpoint = True, dtype=float))
print(np.linspace(0, 2, 10))
```

# Python NumPy

- **Matrix Manipulation**

In python matrix can be implemented as 2D list or 2D Array.

1. **add()** :- This function is used to perform element wise matrix addition.
2. **subtract()** :- This function is used to perform element wise matrix subtraction.
3. **divide()** :- This function is used to perform element wise matrix division.
4. **multiply()** :- This function is used to perform element wise matrix multiplication.
5. **dot()** :- This function is used to compute the matrix multiplication, rather than element wise multiplication.
6. **sqrt()** :- This function is used to compute the square root of each element of matrix.
7. **sum(x,axis)** :- This function is used to add all the elements in matrix. Optional "axis" argument computes the column sum if axis is 0 and row sum if axis is 1.
8. **linalg.det()** : This function calculates the determinant of a square matrix
9. **linalg.inv()**: This function calculates the inverse of a square matrix
8. "**T**" :- This argument is used to transpose the specified matrix.

# Python NumPy

- **Matrix Manipulation**

**Example**

```
import numpy
# initializing matrices
x = numpy.array([[4, 2], [4, 5]])
y = numpy.array([[7, 8], [9, 10]])
# using add() to add matrices
print ("The element wise addition of matrix is : ")
print (numpy.add(x,y))
# using subtract() to subtract matrices
print ("The element wise subtraction of matrix is : ")
print (numpy.subtract(x,y))
# using divide() to divide matrices
print ("The element wise division of matrix is : ")
print (numpy.divide(x,y))
# using "T" to transpose the matrix
print ("The transpose of given matrix is : ")
print (x.T)
```

# Python NumPy

- **Matrix Manipulation**

**Example**

```
import numpy as np
x = np.array([[4, 2], [4, 5]])
det = np.linalg.det(x)
print("\nDeterminant of given 2X2 matrix:")
print(int(det))
```

# Python NumPy

- **np.eye()**

Creates a 2-D array with 1's on the diagonal.

**Example**:

```
print(np.eye(4))
print(np.eye(4,5))
```

# Python NumPy

- **Array copy and view**

- We can copy an array using cop()

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42

print(arr)
print(x)
```

**NOTE**: The copy IS NOT affected by the changes made to the original array.

# Python NumPy

- **Array copy and view**

- We can make a view of an array using view()

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42

print(arr)
print(x)
```

**NOTE**: The view IS affected by the changes made to the original array.

# Python NumPy

- **Array copy and view**

- Every NumPy array has the attribute base that returns None if the array owns the data (it is a copy).

```
import numpy as np

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

x = arr.copy()
y = arr.view()

print(x.base)
print(y.base)
```

# Python NumPy

- **Array shape**

- NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(arr.shape)
----
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8],[1,2]])
print(arr.shape)
----
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8],[1,2,1,1]])
print(arr.shape)
----
arr = np.array([[[1, 2, 3, 4], [5, 6, 7, 8],[1,2,1,1]]])
print(arr.shape)
```

# Python NumPy

- **Array reshape**

- Reshaping means changing the shape of an array. The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print(newarr)
```

---

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

---

**NOTE**: We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(3, 3)
print(newarr)
```

# Python NumPy

- **Array reshape**

**NOTE**: reshape() will return a view of the original array.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
print(arr.reshape(2, 4).base)
```

**NOTE**: We can specify one unknown dimension (not more) by passing -1, and NumPy will calculate this number for you.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, -1)
print(newarr)
```

# Python NumPy

- **Array reshape**

**NOTE**: We can use reshape(-1) to flatten an array, i.e. converting a multidimensional array into a 1D array.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
newarr = arr.reshape(-1)
print(newarr)
```

**NOTE**: We can also use ravel() to flatten an array.

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

# Python NumPy

- **Array reshape**

**Task:**

Write a NumPy program to create a 3x3 matrix with values ranging from 2 to 10.

Sample output:
[[ 2  3  4]
 [ 5  6  7]
 [ 8  9 10]]

```
import numpy as np
x =  np.arange(2, 11).reshape(3,3)
print(x)
```

# Python NumPy

- **Array reshape**

**Task:**

Write a NumPy program to reverse an array (the first element becomes the last) using indexing.
**Sample Output:**

Original array:
[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]
Reverse array:
[37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12]

```
import numpy as np
x = np.arange(12, 38)
print("Original array:")
print(x)
print("Reverse array:")
x = x[::-1]
print(x)
```

# Python NumPy

- **Array iterating**

- Iterating means going through elements one by one. As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python. If we iterate on a 1-D array it will go through each element one by one.

```
import numpy as np
arr = np.array([1, 2, 3])
for x in arr:
  print(x)
------
arr = np.array([[1, 2, 3], [4, 5, 6]])   # two dimensional
for x in arr:
  print(x)
------
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])  # 3 dimensional
for x in arr:
  print("x represents the 2-D array:")
  print(x)
```

# Python NumPy

- **Array join/stack**

- We can stack arrays along rows using hstack()

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.hstack((arr1, arr2))
print(arr)
```

- We can stack arrays along columns using vstack()

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.vstack((arr1, arr2))
print(arr)
```

# Python NumPy

- **Array join/stack**

```
arr1 = np.array([[1, 2, 3],[11,12,13]])
arr2 = np.array([[4, 5, 6],[4,8,9]])
arr = np.hstack((arr1, arr2))
print(arr1)
print(arr2)
print(arr)
---
arr1 = np.array([[1, 2, 3],[11,12,13]])
arr2 = np.array([[4, 5, 6],[4,8,9]])
arr = np.vstack((arr1, arr2))
print(arr1)
print(arr2)
print(arr)
```

# Python NumPy

- **Array join/stack**

**NOTE**: NumPy provides a helper function: dstack() to stack along the height, which is the same as depth.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.dstack((arr1, arr2))
print(arr)
```

---

```
arr1 = np.array([[1, 2, 3],[11,12,13]])
arr2 = np.array([[4, 5, 6],[4,8,9]])
arr = np.dstack((arr1, arr2))
print(arr1)
print(arr2)
print(arr)
```

# Python NumPy

- **Array split**

- We can use hsplit(), vsplit, and dsplit() to split arrays.

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])
newarr = np.hsplit(arr, 3)
print(newarr)
```

---

```
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])
newarr = np.vsplit(arr, 3)
print(newarr)
```

# Python NumPy

- **Array search**

- You can search an array for a certain value and return it index using the where() method.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)
```

# Python NumPy

- **Array sort**

- The NumPy ndarray object has a function called sort() that will sort a specified array.

```
import numpy as np
arr = np.array([3, 2, 0, 1])
x=np.sort(arr)     % in ascending order
print(x)
print(np.flip(x)) % in descending order

-----
arr = np.array(['banana', 'cherry', 'apple'])
print(np.sort(arr))
```

**NOTE**: If you use the sort() method on a 2-D array, both arrays will be sorted:

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

# Python NumPy

- **Array sort**

- The NumPy ndarray object has a function called argsort() that Returns the indices that would sort an array

```
import numpy as np
x = np.array([3, 1, 2])
np.argsort(x)

x = np.array([3, 1, 2])
index=np.argsort(x)
print(x[index])
```

# Python NumPy

- **Array sort**

- The NumPy ndarray object has a function called argsort() that Returns the indices that would sort an array

```
import numpy as np
x = np.array([3, 1, 2])
np.argsort(x)
```