SOCIETY OF PETROLEUM ENGINE
PORT HARCOURT SECTION

PYTHON FOR RESERVOIR ENGINEERING AND GEOSCIENCES

DESTINY OTTO & YOHANES NUWARA

SLIDE TWO

Courtesy: edureka

AGENDA

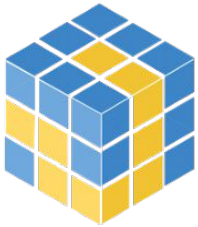
Python

Applications of Python in the oil and gas industry

Python Libraries

Getting Started with Python

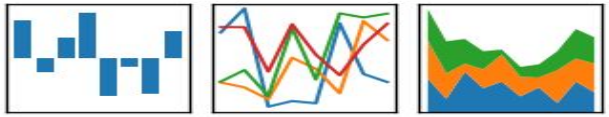
- Numpy, Pandas and Matplotlib



NumPy

pandas

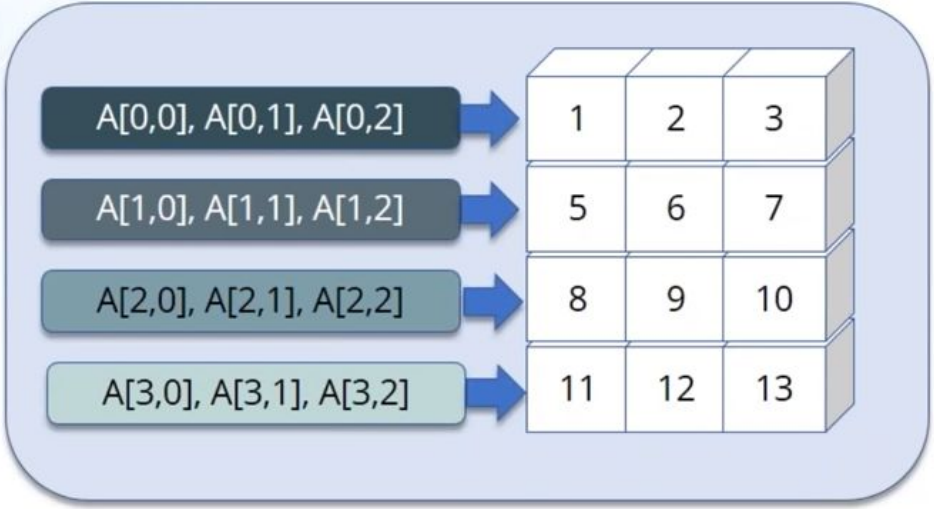
$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



INTRODUCTION TO NUMPY, PANDAS AND MATPLOTLIB

matplotlib

Numpy

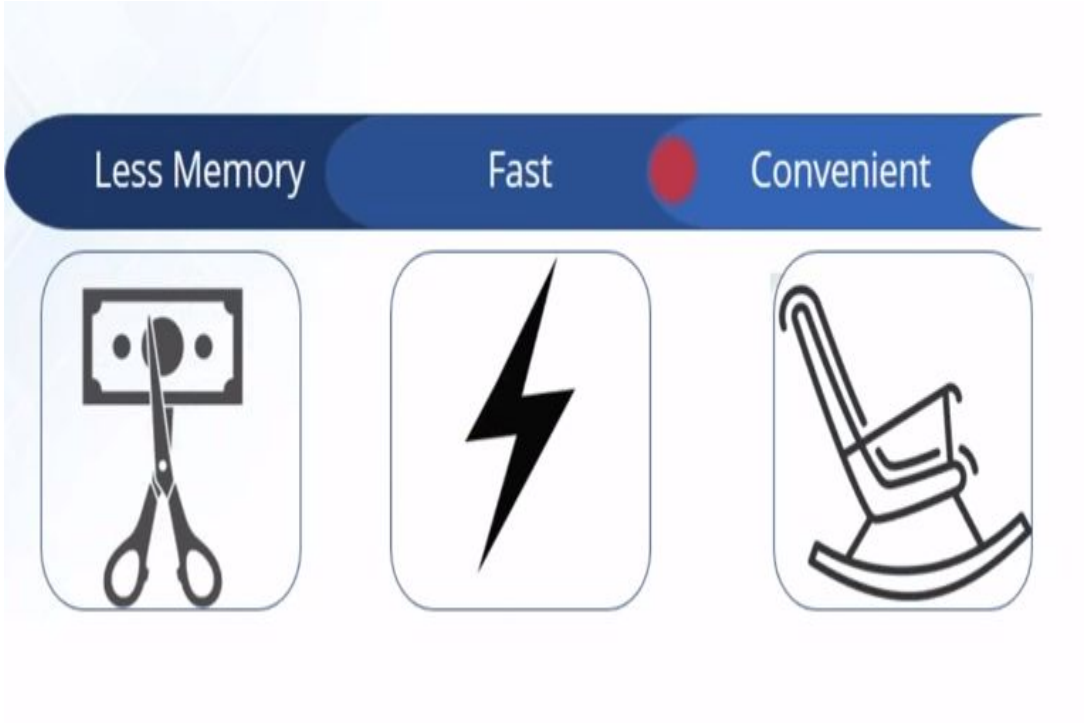


The core library for scientific computing in python

It provides a high performance multidimensional array object and tools for working with these array

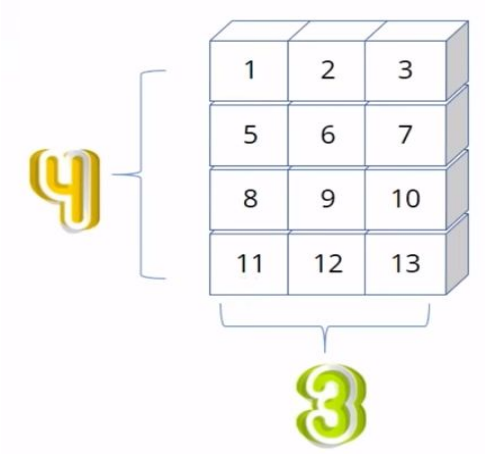
Numpy vs Lists

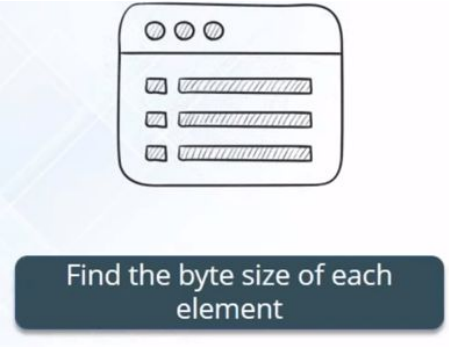
Why Numpy is better than lists

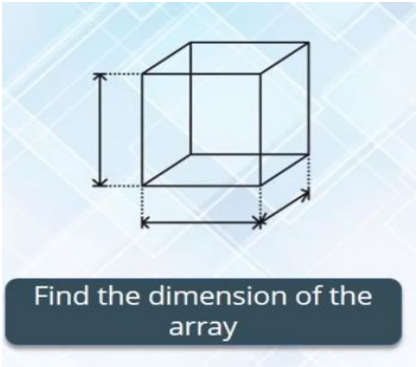


Numpy Operations

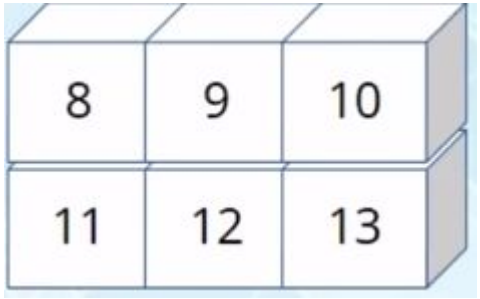
Tuples are immuta



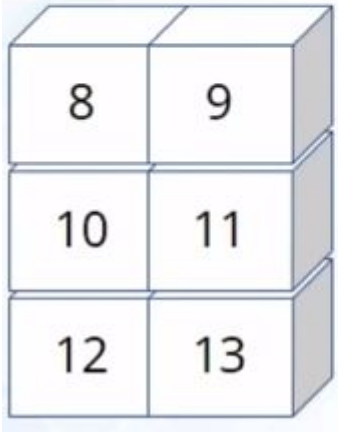


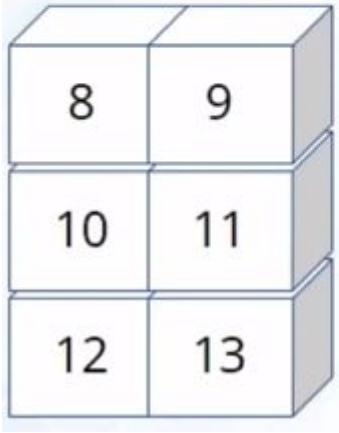


Numpy Operations

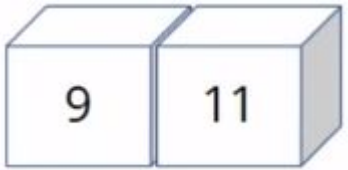


Reshape

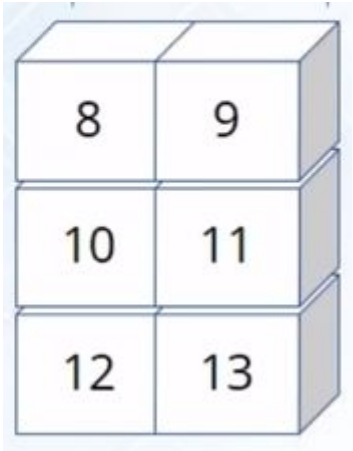




Slicing



Sum of axis



axis 1

axis 0

Sum of axis 0: [30, 33]

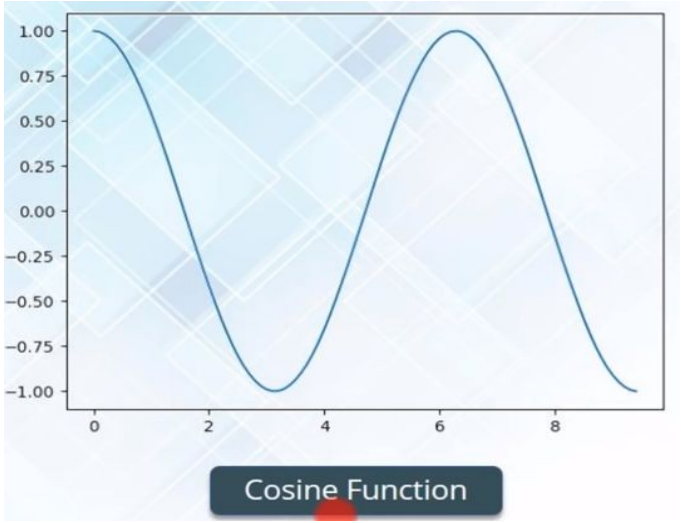
Sum of axis 1: [17, 21, 25]

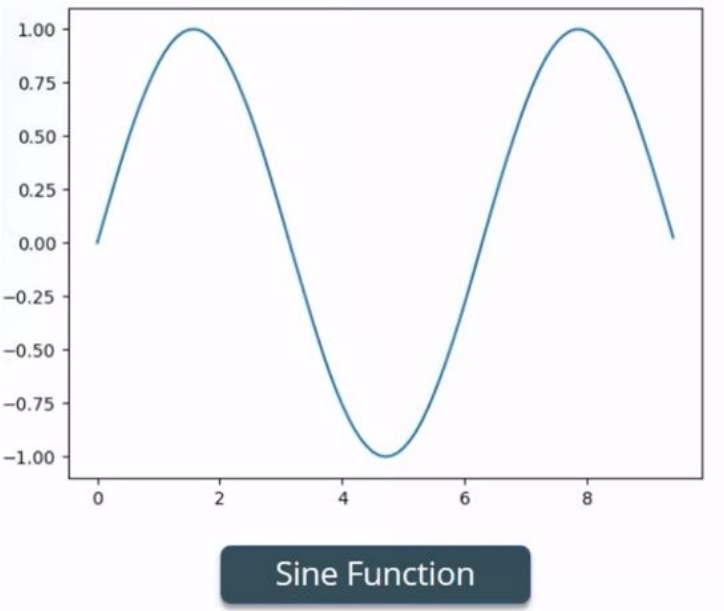
Minimum

Maximum

sum

Sine and cosine plots





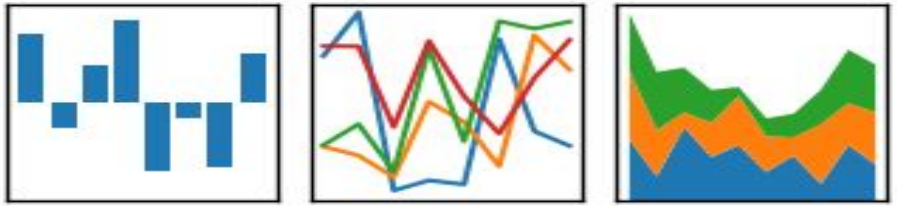
Numpy Applications

- **Manipulating multidimensional arrays (transposing, etc)** – *seismic data is a "4D" data consists of inlines, crosslines, time, and amplitude*
- **Data cleansing (search for NaNs, replace values)** – *data quality control (QC)*
- **Load and read files with text formats** – *parsing multiple files (e.g. ECLIPSE simulator files)*
- **Operations in Numpy (matrix linear algebra, FFT, etc)** – *signal processing in geophysics, inverse problems*
- **Generating random numbers** – *synthetic data generation when data is not available (!),*
  Monte Carlo simulation of a stochastic analysis in oil and gas
- **Write files** – *output results from program to a file*

Pandas

pandas

$$y_{it} = \beta' x_{it} + \mu_i + \epsilon_{it}$$



Loading Python Libraries

```
In [ ]:
#Import Python Libraries
import numpy as np
import scipy as sp
import pandas as pd
import matplotlib as mpl
import seaborn as sns
```

Press `Shift+Enter` to execute the *jupyter* cell

Reading data using pandas

```
In [ ]: #Read csv file
        df = pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/Salaries.csv")
```

Note: The above command has many optional arguments to fine-tune the data import process.

There is a number of pandas commands to read other data formats:

```
pd.read_excel('myfile.xlsx',sheet_name='Sheet1', index_col=None,
na_values=['NA'])
pd.read_stata('myfile.dta')
pd.read_sas('myfile.sas7bdat')
pd.read_hdf('myfile.h5','df')
```

Exploring data frames

In [3]:
```
#List first 5 records
df.head()
```

Out[3]:

Data Frame data types

Pandas Type	Native Python Type	Description
object	string	The most general dtype. Will be assigned to your column if column has mixed types (numbers and strings).
int64	int	Numeric characters. 64 refers to the memory allocated to hold this character.
float64	float	Numeric characters with decimals. If a column contains numbers and NaNs(see below), pandas will default to float64, in case your missing value has a decimal.
datetime64, timedelta[ns]	N/A (but see the datetime module in Python's standard library)	Values meant to hold time data. Look into these for time series experiments.

Data Frame data types

In [4]:
```
#Check a particular column type
df['salary'].dtype
```

Out[4]: dtype('int64')

In [5]:
```
#Check types for all the columns
df.dtypes
```

Out[4]:

rank	object
discipline	object
phd	int64
service	int64
sex	object
salary	int64

dtype: object

Data Frames attributes

Python objects have *attributes* and *methods*.

df.attribute	description
dtypes	list the types of the columns
columns	list the column names
axes	list the row labels and column names
ndim	number of dimensions
size	number of elements
shape	return a tuple representing the dimensionality
values	numpy representation of the data

Data Frames methods

Unlike attributes, python methods have *parenthesis.*
All attributes and methods can be listed with a *dir()* function:
dir(df)

df.method()	description
head([n]), tail([n])	first/last n rows
describe()	generate descriptive statistics (for numeric columns only)
max(), min()	return max/min values for all numeric columns
mean(), median()	return mean/median values for all numeric columns
std()	standard deviation
sample([n])	returns a random sample of the data frame
dropna()	drop all the records with missing values

Selecting a column in a Data Frame

Method 1: Subset the data frame using column name:

```
df['Rs']
```

Method 2: Use the column name as an attribute:

```
df.Rs
```

Note: there is an attribute *rank* for pandas data frames, so to select a column with a name "rank" we should use method 1.

Data Frames *groupby* method

Using "group by" method we can:

- Split the data into groups based on some criteria
- Calculate statistics (or apply a function) to each group
- Similar to dplyr() function in R

In [ ]:
```
#Group data using pressure
df_pressure = df.groupby( ['pressure'])
```

In [ ]:
```
#Calculate mean value for each numeric column per each group
df_pressure.mean()
```

Data Frames *groupby* method

Once groupby object is create we can calculate various statistics for each group:

In [ ]:
```
#Calculate mean salary for each pressure:
df.groupby('pressure')[['temperature']].mean()
```

Note: If single brackets are used to specify the column (e.g. salary), then the output is Pandas Series object. When double brackets are used the output is a Data Frame

Data Frames *groupby* method

groupby performance notes:

- no grouping/splitting occurs until it's needed. Creating the *groupby* object only verifies that you have passed a valid mapping
- by default the group keys are sorted during the *groupby* operation. You may want to pass sort=False for potential speedup:

In [ ]:
```
#Calculate mean Rs for each pressure:
df.groupby(['Rs'], sort=False)[['pressure']].mean()
```

Data Frame: filtering

To subset the data we can apply Boolean indexing. This indexing is commonly known as a filter. For example if we want to subset the rows in which the salary value is greater than $120K:

In [ ]:
```
#Calculate mean pressure for each Rs:
df_sub = df[ df['Rs'] > 120000 ]
```

Any Boolean operator can be used to subset the data:

> greater; >= greater or equal;

< less; <= less or equal;

== equal; != not equal;

In [ ]:
```
#Select only those rows that contain low pressures
df_f = df[ df['pressure'] == 'lowpressures' ]
```

Data Frames: Slicing

There are a number of ways to subset the Data Frame:

- one or more columns
- one or more rows
- a subset of rows and columns

Rows and columns can be selected by their position or label

Data Frames: Slicing

When selecting one column, it is possible to use single set of brackets, but the resulting object will be a Series (not a DataFrame):

In [ ]:
```
#Select column pressure:
df['pressure']
```

When we need to select more than one column and/or make the output to be a DataFrame, we should use double brackets:

In [ ]:
```
#Select column pressure:
df[['Rs','pressure']]
```

Data Frames: Selecting rows

If we need to select a range of rows, we can specify the range using ":"

```
In [ ]: #Select rows by their pressures:
        df[10:20]
```

Notice that the first row has a position 0, and the last value in the range is omitted:
So for 0:10 range the first 10 rows are returned with the positions starting with 0 and ending with 9

Data Frames: method loc

If we need to select a range of rows, using their labels we can use method loc:

In [ ]:
```
#Select rows by their labels:
df_sub.loc[10:20,['Pressure','FVF','temperature' ]]
```

Out[ ]:

Data Frames: method iloc

If we need to select a range of rows and/or columns, using their pressures we can use method iloc:

In [ ]:
```
#Select rows by their labels:
df_sub.iloc[10:20,[0, 3, 4, 5]]
```

Out[ ]:

Data Frames: method iloc (summary)

```
df.iloc[0]  # First row of a data frame
df.iloc[i]  #(i+1)th row
df.iloc[-1] # Last row
```

```
df.iloc[:, 0]  # First column
df.iloc[:, -1] # Last column
```

```
df.iloc[0:7]        #First 7 rows
df.iloc[:, 0:2]     #First 2 columns
df.iloc[1:3, 0:2]   #Second through third rows and first 2 columns
df.iloc[[0,5], [1,3]]  #1st and 6th rows and 2nd and 4th columns
```

Data Frames: Sorting

We can sort the data by a value in the column. By default the sorting will occur in ascending order and a new data frame is return.

In [ ]:
```
# Create a new data frame from the original sorted by the column
Salary
df_sorted = df.sort_values( by ='temperature')
df_sorted.head()
```

Out[ ]:

Data Frames: Sorting

We can sort the data using 2 or more columns:

In [ ]:
```
df_sorted = df.sort_values( by =['FVF', 'temperature'], ascending = [True, False])
df_sorted.head(10)
```

Out[ ]:

Missing Values

Missing values are marked as NaN

In [ ]:
```
# Read a dataset with missing values
flights = pd.read_csv("http://rcs.bu.edu/examples/python/data_analysis/PVTs.csv")
```

In [ ]:
```
# Select the rows that have at least one missing value
flights[flights.isnull().any(axis=1)].head()
```

Out[ ]:

Missing Values

There are a number of methods to deal with missing values in the data frame:

df.method()	description
dropna()	Drop missing observations
dropna(how='all')	Drop observations where all cells is NA
dropna(axis=1, how='all')	Drop column if all the values are missing
dropna(thresh = 5)	Drop rows that contain less than 5 non-missing values
fillna(0)	Replace missing values with zeros
isnull()	returns True if the value is missing
notnull()	Returns True for non-missing values

Missing Values

- When summing the data, missing values will be treated as zero
- If all values are missing, the sum will be equal to NaN
- cumsum() and cumprod() methods ignore missing values but preserve them in the resulting arrays
- Missing values in GroupBy method are excluded (just like in R)
- Many descriptive statistics methods have *skipna* option to control if missing data should be excluded . This value is set to *True* by default (unlike R)

Aggregation Functions in Pandas

Aggregation - computing a summary statistic about each group, i.e.

- compute group sums or means
- compute group sizes/counts

Common aggregation functions:

min, max
count, sum, prod
mean, median, mode, mad
std, var

Aggregation Functions in Pandas

agg() method are useful when multiple statistics are computed per column:

In [ ]:
```
flights[['dep_delay','arr_delay']].agg(['min','mean','max'])
```

Out[ ]:

Basic Descriptive Statistics

df.method()	description
describe	Basic statistics (count, mean, std, min, quantiles, max)
min, max	Minimum and maximum values
mean, median, mode	Arithmetic average, median and mode
var, std	Variance and standard deviation
sem	Standard error of mean
skew	Sample skewness
kurt	kurtosis

Pandas Applications

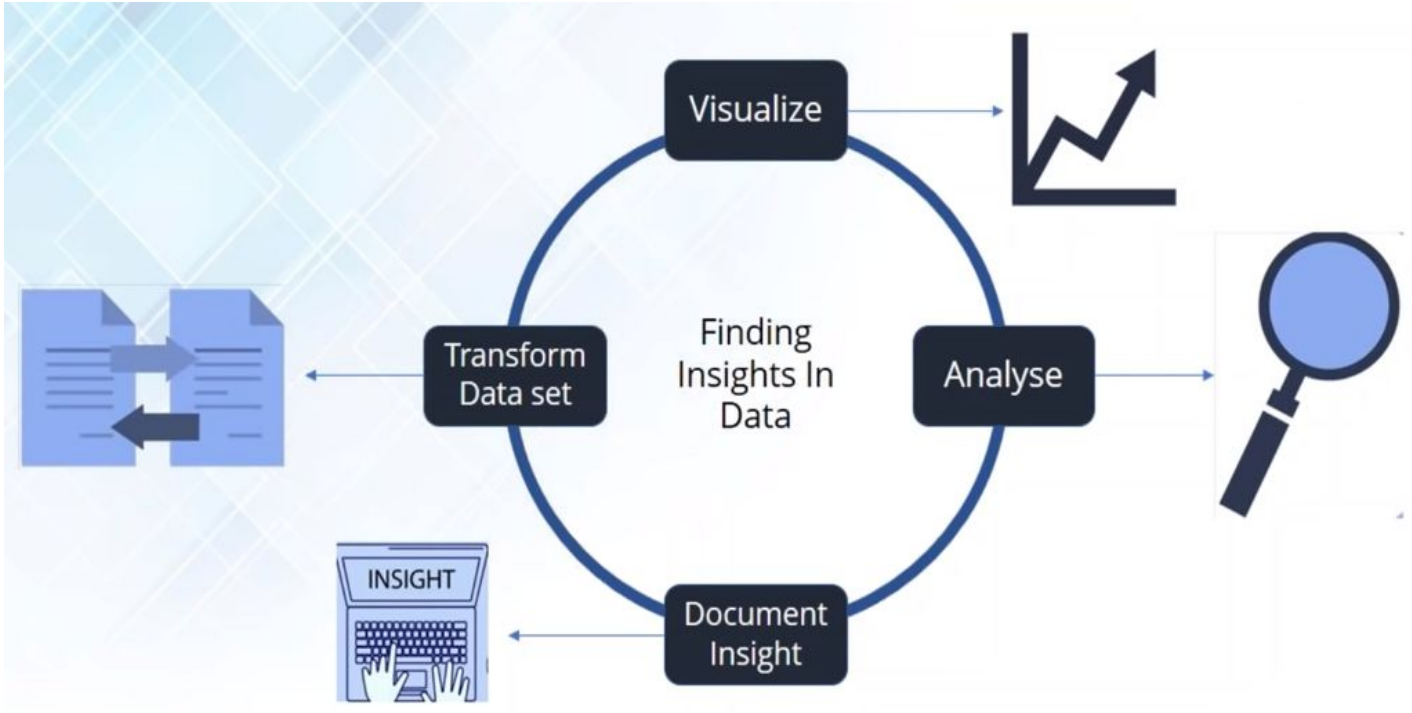
- **Read spreadsheets**
- **Dataframe manipulation** (slicing or creating subsets of dataframes, merging, etc)
- **Datetime conversion** (datetime format in Pandas is YYYY-MM-DD. If our input has
  different format, we need to convert it)
- **Summary statistics** (knowing the mean, interquartiles, standard deviation, min, max)
- **Data cleansing** (searching for NaNs, removing unwanted rows or columns)
- **Regex** (string contains)

Matplotlib

Data Visualization

Data Visualization

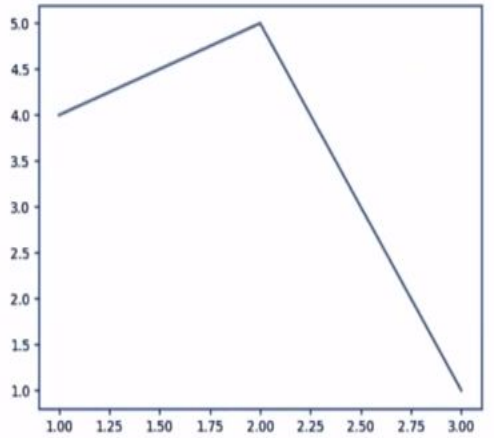
Data Visualization



Most common plot types in Matplotlib

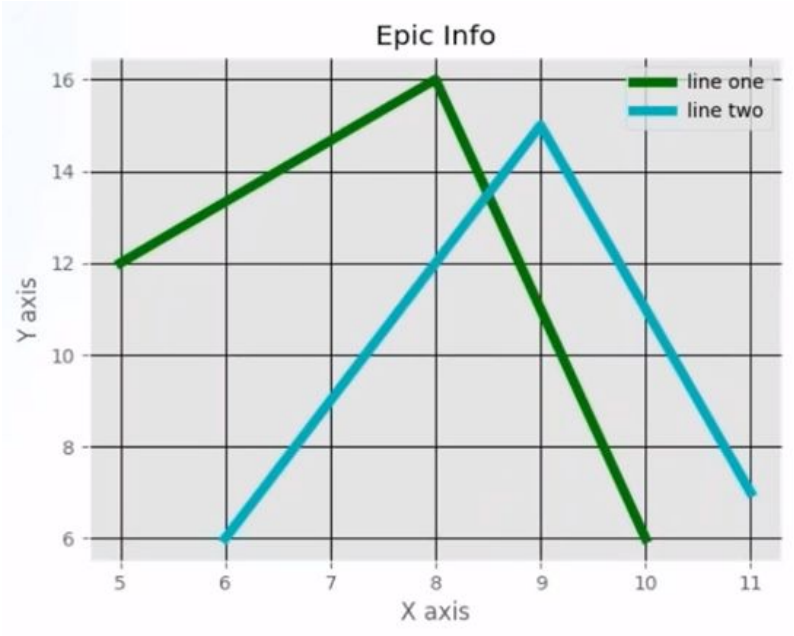
Matplotlib pyplot

```
from matplotlib import pyplot as plt

#Plotting to our canvas

plt.plot([1,2,3],[4,5,1])

#Showing what we plotted

plt.show()
```

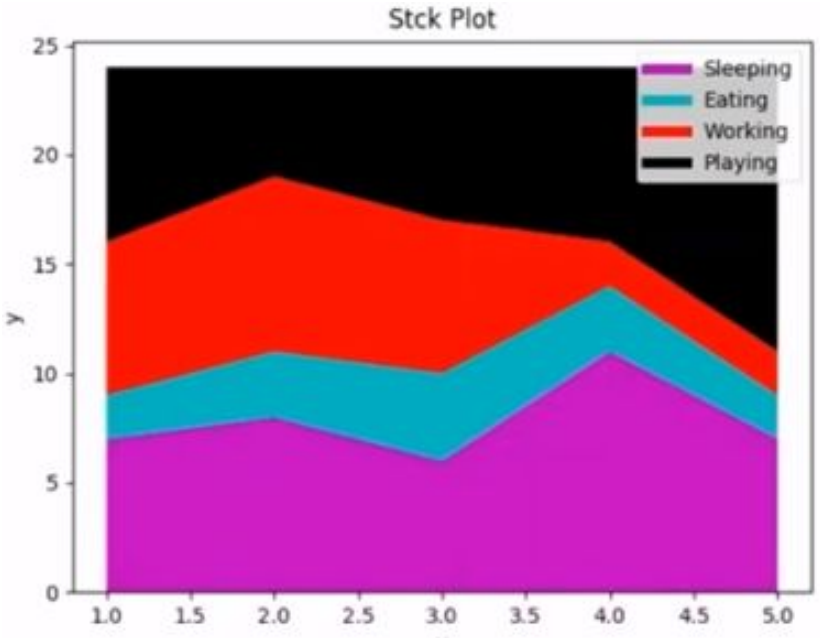


Plot styles

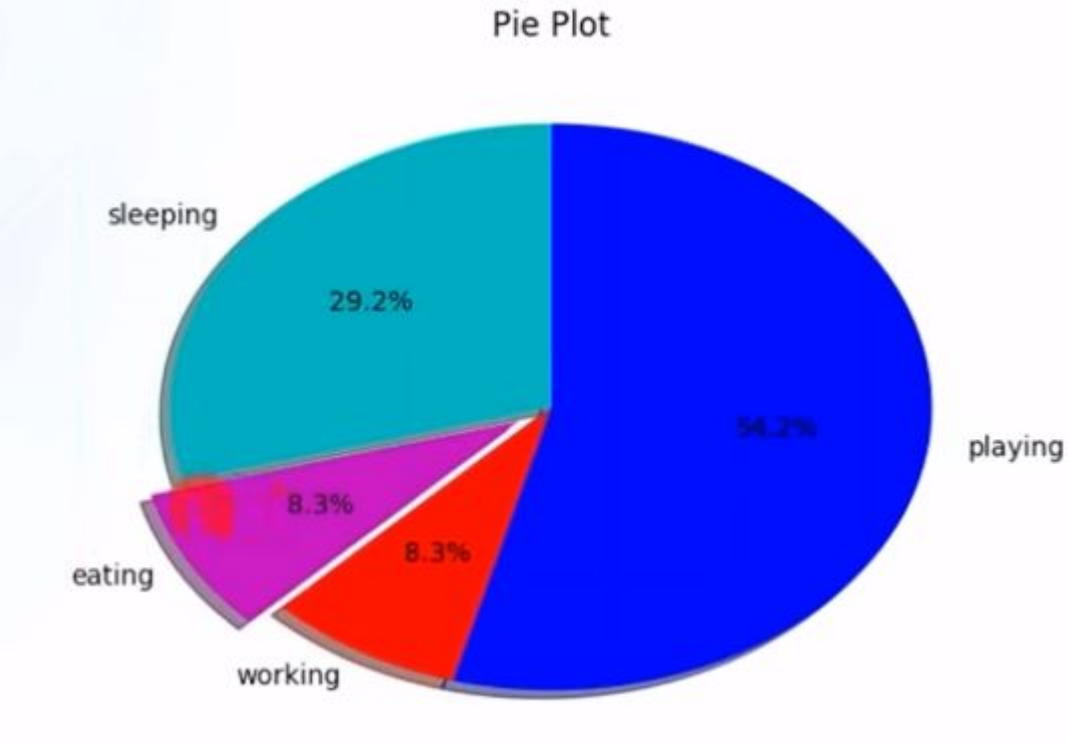
```
from matplotlib import pyplot as plt
from matplotlib import style

style.use('ggplot')

x = [5,8,10]
y = [12,16,6]

x2 = [6,9,11]
y2 = [6,15,7]

plt.plot(x,y,'g',label='line one', linewidth=5)
plt.plot(x2,y2,'c',label='line two',linewidth=5)

plt.title('Epic Info')
plt.ylabel('Y axis')
plt.xlabel('X axis')

plt.legend()

plt.grid(True,color='k')

plt.show()
```



Stack plots

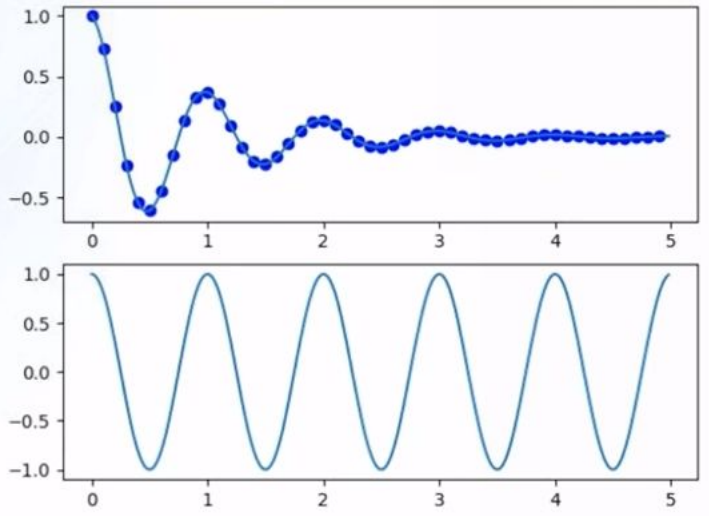

Stck Plot
Sleeping
Eating
Working
Playing
y
25
20
15
10
5
0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

Pie chart



Multiple plots (subplots)

```
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2))

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2))
plt.show()
```



Matplotlib Applications

- **Creating subplots through automation** – *displaying multiple well logs and composites (RHOB, NPHI, resistivity, GR, PEF, etc.) during formation eval*
- **Scatter plots** – *facies typing from petrophysical crossplots*
- **Logarithmic plots** – *semilog plot for well-test analysis*
- **Displaying histogram** – *evaluation of the statistical distribution of petrophysical variables*
- **Displaying boxplot** – *identifying outliers in well logs or production data*
- **Creating contour plots** – *spatial distribution of porosity, permeability, net pay, sand thickness, etc.*
- **Displaying 3D plot** – *wellbore trajectory visualization*

matplotlib

Scipy Applications

- **Gridding data** – *gridding from a reservoir model (porosity, permeability, net pay in 2D/3D)*
- **Interpolate** – *mapping porosity, permeability, net pay using different methods (spline,*
  cubic, or kriging)
- **Solve systems of equations and roots** – *solving PVT (e.g. Peng-Robinson cubic EOS)*
- **Using optimization methods** – *linear regression or curve fitting in well-test analysis, Arps' decline curve analysis*