

# MODULE 5

---

# OS and sys module in Python

## OS Module in Python

- The OS module in python provides functions for interacting with the operating system.
- OS, comes under Python's standard utility modules.
- This module provides a portable way of using operating system dependent functionality.

### Creating Directory:

We can create a new directory using the `mkdir()` function from the OS module.

```
import os  
os.mkdir("d:\\tempdir")
```

### Changing the Current Working Directory:

We must first change the current working directory to a newly created one before doing any operations in it. This is done using the `chdir()` function.

```
import os
```

```
os.chdir("d:\\tempdir")
```

- There is a `getcwd()` function in the OS module using which we can confirm if the current working directory has been changed or not.
- In order to set the current directory to the parent directory use `".."` as the argument in the `chdir()` function.

### Removing a Directory

- The `rmdir()` function in the OS module removes the specified directory either with an absolute or relative path.
- However, we can not remove the current working directory. Also, for a directory to be removed, it should be empty. For example, `tempdir` will not be removed if it is the current directory.

### List Files and Sub-directories:

- The `listdir()` function returns the list of all files and directories in the specified directory.
- If we don't specify any directory, then list of files and directories in the current working directory will be returned.

### Scan all the directories and subdirectories:

- Python method `walk()` generates the file names in a directory tree by walking the tree either top-down or bottom-up.

# sys Module in Python

The sys module provides functions and variables used to manipulate different parts of the Python runtime environment.

## sys.argv

returns a list of command line arguments passed to a Python script. The item at index 0 in this list is always the name of the script. The rest of the arguments are stored at the subsequent indices.

## sys.exit

This causes the script to exit back to either the Python console or the command prompt. This is generally used to safely exit from the program in case of generation of an exception.

### `sys.maxsize`

Returns the largest integer a variable can take.

### `sys.path`

This is an environment variable that is a search path for all Python modules.

### `sys.version`

This attribute displays a string containing the version number of the current Python interpreter.



# numpy

## NumPy (Numerical Python)

NumPy is a **library** consisting of **multidimensional array** objects and a collection of routines for processing those arrays. Using NumPy, mathematical and logical operations on arrays can be performed.

Using NumPy, a developer can perform the following operations –

- ✓ Mathematical and logical operations on arrays.
- ✓ Fourier transforms and routines for shape manipulation.
- ✓ Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

## What's the difference between a Python list and a NumPy array?

- NumPy gives you an enormous range of fast and efficient ways of creating arrays and manipulating numerical data inside them.
- While a Python list can contain different data types within a single list, all of the elements in a NumPy array should be homogeneous.
- The mathematical operations that are meant to be performed on arrays would be extremely inefficient if the arrays weren't homogeneous.



## ndarray Object

The most important object defined in NumPy is an **N-dimensional array type** called **ndarray**. It describes the collection of items of the same type. Items in the collection can be accessed using a **zero-based index**.

Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of **data-type** object (called dtype).

Any item extracted from ndarray object (by slicing) is represented by a **Python object** of one of **array scalar types**.

## Creating Arrays

```
import numpy as np
```

```
a = np.array([1,2,3,4])
```

```
print(a)
```

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

```
print(b)
```

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

```
print(c)
```

*Output:*

[1 2 3 4]

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

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

# Creating ndarrays

## ndarray Object - Parameters

### Some important attributes of ndarray object

#### (1) `ndarray.ndim`

`ndim` represents the number of dimensions (axes) of the ndarray.

#### (2) `ndarray.shape`

`shape` is a tuple of integers representing the size of the ndarray in each dimension.

#### (3) `ndarray.size`

`size` is the total number of elements in the ndarray. It is equal to the product of elements of the `shape`.

#### (4) `ndarray.dtype`

`dtype` tells the data type of the elements of a NumPy array. In NumPy array, all the elements have the same data type.

#### (5) `ndarray.itemsize`

`itemsize` returns the size (in bytes) of each element of a NumPy array.

```
import numpy as np
a = np.array([[[1,2,3],[4,3,5]],[[3,6,7],[2,1,0]]])
print("The dimension of array a is:", a.ndim)
print("The size of the array a is: ", a.shape)
print("The total no: of elements in array a is: ", a.size)
print("The datatype of elements in array a is: ", a.dtype)
print("The size of each element in array a is: ", a.itemsize)
```

### Output:

```
The dimension of array a is: 3
The size of the array a is: (2, 2, 3)
The total no: of elements in array a is: 12
The datatype of elements in array a is: int32
The size of each element in array a is: 4
```

# Creating ndarrays

## Different ways of creating an Array

There are 8 ways for creating arrays using NumPy Package

- (1) using `array()`
- (2) using `arange()`
- (3) using `linspace()`
- (4) using `logspace()`
- (5) using `zeros()`
- (6) using `ones()`
- (7) using `empty()`
- (8) using `eye()`



# Cntd:

## (1) Using array()

Numpy arrays can be created very easily by passing a list to the function `numpy.array()`.

Eg., `numpy_array = np.array([1,2,3,4])`  
`print(numpy_array)`

## (2) Using arange()

The `arange()` function is one of the Numpy's most used method for creating an array within a specified range.

*Syntax: `arange(start, end, step, dtype)`*

Among all of the arguments only end is mandatory and by default start=0 and step=1.

# Cntd:

```
import numpy as np  
a = np.arange(0,11,2)  
print(a)
```

```
b = np.arange(50,121,4)  
print(b)
```

```
c = np.arange(15)  
print(c)
```

Output:

```
[ 0  2  4  6  8 10]
```

```
[ 50  54  58  62  66  70  74  78  82  86  90  94  98 102 106 110 114 118]
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

### (3) Using linspace()

- In the `arange()` function, we had control over where to start the Numpy array from, where to stop and step points but with `linspace()` we can maintain a proper linear stepping or spacing between array elements value while generating the array.
- `linspace()` function takes arguments: `start index`, `end index` and the number of elements to be outputted.
- These number of elements would be linearly spaced in the range mentioned.

*Syntax: `linspace(start_index, end_index, num_of_elements)`*

```
import numpy as np
a = np.linspace(15,75,10)
print(a)
```

```
b = np.linspace(1,10,10)
print(b)
```

*Output:*

```
[15.    21.66666667 28.33333333 35.    41.66666667 48.33333333
 55.    61.66666667 68.33333333 75.    ]
```

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

### (5) Using zeros()

zeros() returns a new array with zeros.

*Syntax: numpy.zeros(shape, dtype=float, order='C')*

*shape: is the shape of the numpy zero array*

*dtype: is the datatype in numpy zeros. It is optional. The default value is float64*

*order: Default is C which is an essential row style for numpy.zeros() in Python.*

### (6) Using ones()

ones() function is used to create a matrix full of ones.

*Syntax: numpy.ones(shape, dtype=float, order='C')*



```
import numpy as np
a = np.zeros(shape = (2,3), order = 'C' )
print(a)
a = np.ones(shape = (3,3), dtype = int, order = 'C' )
print(a)
```

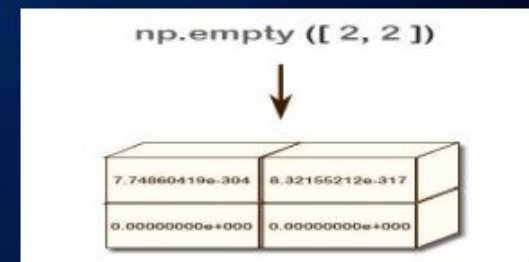
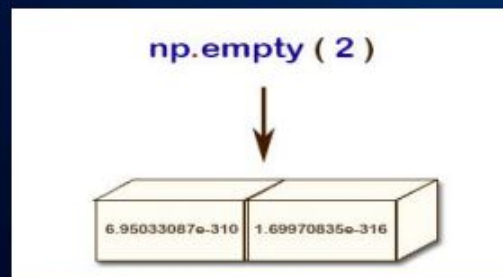
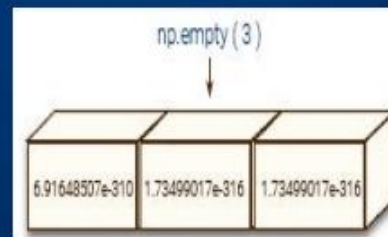
*Output:*

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

## (7) using empty()

The empty() function is used to create a new array of given shape and type, without initializing entries

*Syntax: numpy.empty(shape, dtype, order)*



## (8) Using eye()

The `eye()` function is used to create a 2-D array with ones on the diagonal and zeros elsewhere.

It returns an array where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one

Syntax: `numpy.eye(N, M=None, k=0, dtype=<class 'float'>, order='C')`

**N** Number of rows in the output.

**M** Number of columns in the output. If None, defaults to N.

**k** Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

**dtype** Data-type of the returned array.

**order** Whether the output should be stored in row-major or column-major order in memory

```
import numpy as np  
a = np.eye(3, 3, 0, dtype=int, order='C')  
print(a)
```

*Output:*

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

# Creating ndarrays

Table 4-1. Array creation functions

Function	Description
<code>array</code>	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
<code>asarray</code>	Convert input to ndarray, but do not copy if the input is already an ndarray
<code>arange</code>	Like the built-in <code>range</code> but returns an ndarray instead of a list.
<code>ones</code> , <code>ones_like</code>	Produce an array of all 1's with the given shape and dtype. <code>ones_like</code> takes another array and produces a ones array of the same shape and dtype.
<code>zeros</code> , <code>zeros_like</code>	Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0's instead

Function	Description
<code>empty</code> , <code>empty_like</code>	Create new arrays by allocating new memory, but do not populate with any values like <code>ones</code> and <code>zeros</code>
<code>eye</code> , <code>identity</code>	Create a square $N \times N$ identity matrix (1's on the diagonal and 0's elsewhere)



# Data types

- The data type or dtype is a special object containing the information the ndarray needs to interpret a chunk of memory as a particular type of data:

```
arr1 = np.array([1, 2, 3], dtype=np.float64)
```

```
arr2 = np.array([1, 2, 3], dtype=np.int32)
```

- The numerical dtypes are named the same way: a type name, like float or int, followed by a number indicating the number of bits per element.
- A standard double-precision floating point value takes up 8 bytes or 64 bits.
- Thus, this type is known in NumPy as float64.

# Numpy datatypes

Type	Type Code	Description
<code>int8, uint8</code>	<code>i1, u1</code>	Signed and unsigned 8-bit (1 byte) integer types
<code>int16, uint16</code>	<code>i2, u2</code>	Signed and unsigned 16-bit integer types
<code>int32, uint32</code>	<code>i4, u4</code>	Signed and unsigned 32-bit integer types
<code>int64, uint64</code>	<code>i8, u8</code>	Signed and unsigned 64-bit integer types
<code>float16</code>	<code>f2</code>	Half-precision floating point
<code>float32</code>	<code>f4</code> or <code>f</code>	Standard single-precision floating point. Compatible with C float
<code>float64, float128</code>	<code>f8</code> or <code>d</code>	Standard double-precision floating point. Compatible with C double and Python float object

Type	Type Code	Description
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing True and False values
object	O	Python object type
string_	S	Fixed-length string type (1 byte per character). For example, to create a string dtype with length 10, use 'S10'.
unicode_	U	Fixed-length unicode type (number of bytes platform specific). Same specification semantics as string_ (e.g. 'U10').

# Astype method: typecasting

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

```
arr.dtype
```

```
dtype('int64')
```

```
float_arr = arr.astype(np.float64)
```

```
float_arr.dtype
```

```
dtype('float64')
```

# Operations between arrays and scalars

- Arrays are important because they enable you to express batch operations on data without writing any for loops. This is usually called vectorization.
- Any arithmetic operations between equal-size arrays applies the operation elementwise:



# Basic indexing and slicing

- Slicing in python means taking elements from one given index to another given index.
- We pass slice instead of index like this: `[start:end]`.
- We can also define the step, like this: `[start:end:step]`.
- If we don't pass start its considered 0
- If we don't pass end its considered length of array in that dimension
- If we don't pass step its considered 1

# 2D array

- In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional array
- Axes are defined for arrays with more than one dimension. A 2-dimensional array has two corresponding axes: the first running vertically downwards across rows (axis 0), and the second running horizontally across columns (axis 1).

		axis 1		
		0	1	2
axis 0	0	0,0	0,1	0,2
	1	1,0	1,1	1,2
	2	2,0	2,1	2,2



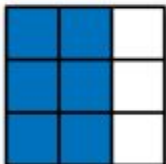

# 2D array

- you can pass a comma-separated list of indices to select individual elements. So these are equivalent:
- `: arr2d[0][2]`
- `arr2d[0, 2]`

# Slicing 2D array

- A slice, therefore, selects a range of elements along an axis.
- You can pass multiple slices just like you can pass multiple indexes

# Slicing

	Expression	Shape
	<code>arr[:2, 1:]</code>	<code>(2, 2)</code>
	<code>arr[2]</code> <code>arr[2, :]</code> <code>arr[2:, :]</code>	<code>(3,)</code> <code>(3,)</code> <code>(1, 3)</code>
	<code>arr[:, :2]</code>	<code>(3, 2)</code>
	<code>arr[1, :2]</code> <code>arr[1:2, :2]</code>	<code>(2,)</code> <code>(1, 2)</code>

# Shape of an array

- The shape of an array is the number of elements in each dimension.



# Arithmetic Operations with NumPy Array

- The arithmetic operations with **NumPy** arrays perform element-wise operations, this means the operators are applied only between corresponding elements.
- Arithmetic operations are possible only if the array has the **same structure and dimensions**.



## Basic operations : with scalars

```
import numpy as np
a = np.array([1,2,3,4,5])
b = a+1
print(b)
c = 2**a
print(c)
```

Output:

```
[2 3 4 5 6]
[ 2  4  8 16 32]
```

```
import numpy as np
a = np.array([1,2,3,4,5])
b = np.ones(5) + 1
c = a - b
print(c)
d = (a*b)
print(d)
e = np.arange(5)
print(e)
f = 2 ** (e+1) - e
print(f)
```

Output:

```
[-1.  0.  1.  2.  3.]
[ 2.  4.  6.  8. 10.]
[0 1 2 3 4]
[ 2  3  6 13 28]
```

# Matrix operations

- **Operation on Matrix :**
- **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**.

# example1

```
import numpy
```

```
# initializing matrices
```

```
x = numpy.array([[1, 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))
```

# Example 2

```
import numpy
```

```
# initializing matrices
```

```
x = numpy.array([[1, 2], [4, 5]])
```

```
y = numpy.array([[7, 8], [9, 10]])
```

```
# using multiply() to multiply matrices element wise
```

```
print ("The element wise multiplication of matrix is : ")
```

```
print (numpy.multiply(x,y))
```

```
# using dot() to multiply matrices
```

```
print ("The product of matrices is : ")
```

```
print (numpy.dot(x,y))
```

- **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. “T”** :- This argument is used to **transpose** the specified matrix.



# Example 3

```
import numpy
```

```
# initializing matrices
```

```
x = numpy.array([[1, 2], [4, 5]])
```

```
y = numpy.array([[7, 8], [9, 10]])
```

```
# using sqrt() to print the square root of matrix
```

```
print ("The element wise square root is : ")
```

```
print (numpy.sqrt(x))
```

```
# using sum() to print summation of all elements of matrix
```

```
print ("The summation of all matrix element is : ")
```

```
print (numpy.sum(y))
```

```
# using sum(axis=0) to print summation of all columns of matrix
print ("The column wise summation of all matrix is : ")
print (numpy.sum(y,axis=0))
```

```
# using sum(axis=1) to print summation of all columns of matrix
print ("The row wise summation of all matrix is : ")
print (numpy.sum(y,axis=1))
```

```
# using "T" to transpose the matrix
print ("The transpose of given matrix is : ")
print (x.T)
```

# Linear algebra functions

Table 4-7. Commonly-used `numpy.linalg` functions

Function	Description
<code>diag</code>	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
<code>dot</code>	Matrix multiplication
<code>trace</code>	Compute the sum of the diagonal elements
<code>det</code>	Compute the matrix determinant
<code>eig</code>	Compute the eigenvalues and eigenvectors of a square matrix
<code>inv</code>	Compute the inverse of a square matrix
<code>pinv</code>	Compute the Moore-Penrose pseudo-inverse inverse of a square matrix
<code>qr</code>	Compute the QR decomposition
<code>svd</code>	Compute the singular value decomposition (SVD)
<code>solve</code>	Solve the linear system $Ax = b$ for $x$ , where $A$ is a square matrix
<code>lstsq</code>	Compute the least-squares solution to $y = Xb$

# Random numbers

- Random number does NOT mean a different number every time.
- Random means something that can not be predicted logically.
- The `numpy.random` module supplements the built-in Python random with functions for efficiently generating whole arrays of sample values from many kinds of probability

*Table 4-8. Partial list of `numpy.random` functions*

Function	Description
<code>seed</code>	Seed the random number generator
<code>permutation</code>	Return a random permutation of a sequence, or return a permuted range
<code>shuffle</code>	Randomly permute a sequence in place
<code>rand</code>	Draw samples from a uniform distribution
<code>randint</code>	Draw random integers from a given low-to-high range
<code>randn</code>	Draw samples from a normal distribution with mean 0 and standard deviation 1 (MATLAB-like interface)
<code>binomial</code>	Draw samples a binomial distribution
<code>normal</code>	Draw samples from a normal (Gaussian) distribution
<code>beta</code>	Draw samples from a beta distribution
<code>chisquare</code>	Draw samples from a chi-square distribution
<code>gamma</code>	Draw samples from a gamma distribution
<code>uniform</code>	Draw samples from a uniform [0, 1) distribution

# Matplotlib

- Matplotlib is an amazing visualization library in Python for 2D plots of arrays.
- Matplotlib is a multi-platform data visualization library built on NumPy arrays and designed to work with the broader SciPy stack.
- One of the greatest benefits of visualization is that it allows us visual access to huge amounts of data in easily digestible visuals.
- Matplotlib consists of several plots like line, bar, scatter, histogram etc.



# Pyplot

- Pyplot is a Matplotlib module that provides a MATLAB-like interface.
- Pyplot provides functions that interact with the figure i.e. creates a figure, decorates the plot with labels, and creates a plotting area in a figure.
- The various plots we can utilize using Pyplot are **Line Plot, Histogram, Scatter, 3D Plot, Image, Contour, and Polar**.

# Plot()

- **Parameters:** This function accepts parameters that enables us to set axes scales and format the graphs. These parameters are mentioned below :-
- **plot(x, y):** plot x and y using default line style and color.x values are optional
- **plot.axis([xmin, xmax, ymin, ymax]):** scales the x-axis and y-axis from minimum to maximum values
- **plot(x, y, color='green', marker='o', linestyle='dashed', linewidth=2, markersize=12):** x and y co-ordinates are marked using circular markers of size 12 and green color line with — style of width 2
- **plot.xlabel('X-axis'):** names x-axis
- **plot.ylabel('Y-axis'):** names y-axis
- **plot(x, y, label = 'Sample line ')** plotted Sample Line will be displayed as a legend

# Legend()

- A legend is an area describing the elements of the graph. In the matplotlib library, there's a function called **legend()** which is used to Place a legend on the axes.
- Eg1: `ax.plot(x,y,'g: ',label=" one")`
- Once you've done this, you can either call `ax.legend()` or `plt.legend()` to automatically create a legend:
- Eg: `ax.legend(loc="best")` OR `ax.legend(loc="lowerright")`
- The `loc` tells matplotlib where to place the plot. If you aren't picky 'best' is a good option, as it will choose a location that is most out of the way.

# Figure and axes classes

- Matplotlib take care of the creation of inbuilt defaults like **Figure** and **Axes**.
- **Figure:** This class is the top-level container for all the plots means it is the overall window or page on which everything is drawn. A figure object can be considered as a box-like container that can hold one or more axes.
- **Axes:** This class is the most basic and flexible component for creating sub-plots. It is the region of image with the data space. A given figure may contain many axes but given axes can only be in one figure.

# Axes class

```
fig = plt.figure()  
#[left, bottom, width, height]  
ax = plt.axes([0.1, 0.1, 0.8, 0.8])
```

- Here in `axes([0.1, 0.1, 0.8, 0.8])`, the first `'0.1'` refers to the distance between the left side axis and border of the figure window is 10%, of the total width of the figure window.
- The second `'0.1'` refers to the distance between the bottom side axis and the border of the figure window is 10%, of the total height of the figure window.
- The first `'0.8'` means the axes width from left to right is 80% and the latter `'0.8'` means the axes height from the bottom to the top is 80%.

# Add\_axes(),plot()

- add the axes object to the figure by calling the **add\_axes()** method.
- It returns the axes object and adds axes at position [left, bottom, width, height] where all quantities are in fractions of figure width and height.
- plot() function of the axes class plots the values of one array versus another as line or marker.
- **Syntax** : *ax.plot(X, Y, 'CLM')*
- **Parameters:**
  - X is x-axis.*
  - Y is y-axis.*
  - 'CLM' stands for Color, Line and Marker.*



# Color, markers and linestyle

- Line can be of different styles such as dotted line (':'), dashed line ('—'), solid line ('-') and many more.
- Method 1. String abbreviations

Eg: `ax.plot(x, y, 'g--')`

- Method 2: keywords

Eg: `ax.plot(x, y, linestyle='--', color='g')`

# Markers

- Line plots can additionally have markers to highlight the actual data points
- The marker can be part of the style string, which must have color followed by marker type and line style
- Eg: `plot(x,y, color='k', linestyle='dashed', marker='o')`
- Eg: `plot(x,y, 'go--')`

# Limits and ticklabels

- Matplotlib automatically sets the values and the markers(points) of the x and y axis, however, it is possible to set the limit and markers manually.
- [set\\_xlim\(\)](#) and [set\\_ylim\(\)](#) functions are used to set the limits of the x-axis and y-axis respectively.
- Similarly, [set\\_xticklabels\(\)](#) and [set\\_yticklabels\(\)](#) functions are used to set tick labels.

# Multiple plots

## Method 1: Using the [add\\_axes\(\)](#) method

- The `add_axes()` method adds the plot in the same figure by creating another axes object

## Method 2: Using [subplot\(\)](#) method.

- This method adds another plot to the current figure at the specified grid position.

## Method 3: Using [subplots\(\)](#) method

- This function is used to create figure and multiple subplots at the same time. `plt.subplots()`, that creates a new figure and returns a NumPy array containing the created subplot objects

# subplots

*Table 8-1. pyplot.subplots options*

Argument	Description
nrows	Number of rows of subplots
ncols	Number of columns of subplots
sharex	All subplots should use the same X-axis ticks (adjusting the <code>xlim</code> will affect all subplots)
sharey	All subplots should use the same Y-axis ticks (adjusting the <code>ylim</code> will affect all subplots)

# Creating Different Types of Plots

1. Line graph
2. Bar chart
3. Histogram
4. Piechart
5. Scatter

# Bar chart

- A **bar plot** or bar chart is a graph that represents the category of data with rectangular bars with lengths and heights that is proportional to the values which they represent.
- The bar plots can be plotted horizontally or vertically. A bar chart describes the comparisons between the discrete categories.
- It can be created using the **bar()** method.



# Scatter

- Scatter plots are used to observe the relationship between variables and use dots to represent the relationship between them.
- The [scatter\(\)](#) method in the matplotlib library is used to draw a scatter plot.

# Pie chart

- A **Pie Chart** is a circular statistical plot that can display only one series of data.
- The area of the chart is the total percentage of the given data.
- The area of slices of the pie represents the percentage of the parts of the data.
- The slices of pie are called wedges.
- The area of the wedge is determined by the length of the arc of the wedge.
- It can be created using the `pie()` method.

# pie

**Syntax:** `matplotlib.pyplot.pie(data, explode=None, labels=None, colors=None, autopct=None, shadow=False)`

**Parameters:**

**data** represents the array of data values to be plotted,

**labels** is a list of sequence of strings which sets the label of each wedge.

**color** attribute is used to provide color to the wedges.

**autopct** is a string used to label the wedge with their numerical value.

**shadow** is used to create shadow of wedge.

# pandas

- Pandas is an open-source library that is made mainly for working with relational or labeled data both easily and intuitively.
- It provides various data structures and operations for manipulating numerical data and time series.
- This library is built on top of the NumPy library

# Pandas datastructures

1. Series
2. Dataframe

# Series

- A Series is a one-dimensional array-like object containing an array of data (of any NumPy data type) and an associated array of data labels, called its index.
- The simplest Series is formed from only an array of data:

# Dataframe

- A DataFrame represents a tabular, spreadsheet-like data structure containing an ordered collection of columns, each of which can be a different value type (numeric, string, boolean, etc.).
- The DataFrame has both a row and column index; it can be thought of as a dict of Series (one for all sharing the same index).

# Sorting and ranking

Sorting a data set by some criterion is another important built-in operation. To sort lexicographically by row or column index, use the `sort_index` method, which returns a new, sorted object:

```
In [169]: obj = Series(range(4), index=['d', 'a', 'b', 'c'])
```

```
In [170]: obj.sort_index()
```

```
Out[170]:
```

a	1
b	2
c	3
d	0



# Sorting

With a DataFrame, you can sort by index on either axis:

```
In [171]: frame = DataFrame(np.arange(8).reshape((2, 4)), index=['three', 'one'],  
.....:                      columns=['d', 'a', 'b', 'c'])
```

```
In [172]: frame.sort_index()
```

```
Out[172]:
```

	d	a	b	c
one	4	5	6	7
three	0	1	2	3

```
In [173]: frame.sort_index(axis=1)
```

```
Out[173]:
```

	a	b	c	d
three	1	2	3	0
one	5	6	7	4

# Sorting

To sort a Series by its values, use its `order` method:

```
In [175]: obj = Series([4, 7, -3, 2])
```

```
In [176]: obj.order()
```

```
Out[176]:
```

```
2   -3
3    2
0    4
1    7
```

Any missing values are sorted to the end of the Series by default:

```
In [177]: obj = Series([4, np.nan, 7, np.nan, -3, 2])
```

```
In [178]: obj.order()
```

```
Out[178]:
```

```
4   -3
5    2
0    4
2    7
1  NaN
3  NaN
```

# Summarizing and Computing Descriptive Statistics

```
In [198]: df = DataFrame([[1.4, np.nan], [7.1, -4.5],  
.....:                  [np.nan, np.nan], [0.75, -1.3]],  
.....:                  index=['a', 'b', 'c', 'd'],  
.....:                  columns=['one', 'two'])
```

```
In [199]: df
```

```
Out[199]:
```

	one	two
a	1.40	NaN
b	7.10	-4.5
c	NaN	NaN
d	0.75	-1.3

Calling DataFrame's `sum` method returns a Series containing column sums:

```
In [200]: df.sum()
```

```
Out[200]:
```

one	9.25
two	-5.80

# Summarizing and Computing Descriptive Statistics

Passing `axis=1` sums over the rows instead:

```
In [201]: df.sum(axis=1)
Out[201]:
a      1.40
b      2.60
c       NaN
d     -0.55
```

NA values are excluded unless the entire slice (row or column in this case) is NA. This can be disabled using the `skipna` option:

```
In [202]: df.mean(axis=1, skipna=False)
Out[202]:
a       NaN
b      1.300
c       NaN
d     -0.275
```

# Summarizing and Computing Descriptive Statistics

Other methods are *accumulations*:

```
In [204]: df.cumsum()
```

```
Out[204]:
```

	one	two
a	1.40	NaN
b	8.50	-4.5
c	NaN	NaN
d	9.25	-5.8

Another type of method is neither a reduction nor an accumulation. `describe` is one such example, producing multiple summary statistics in one shot:

```
In [205]: df.describe()
```

```
Out[205]:
```

	one	two
count	3.000000	2.000000
mean	3.083333	-2.900000
std	3.493685	2.262742
min	0.750000	-4.500000
25%	1.075000	-3.700000
50%	1.400000	-2.900000
75%	4.250000	-2.100000
max	7.100000	-1.300000

# Summarizing and Computing Descriptive Statistics

*Table 5-10. Descriptive and summary statistics*

Method	Description
<code>count</code>	Number of non-NA values
<code>describe</code>	Compute set of summary statistics for Series or each DataFrame column
<code>min, max</code>	Compute minimum and maximum values
<code>argmin, argmax</code>	Compute index locations (integers) at which minimum or maximum value obtained, respectively
<code>idxmin, idxmax</code>	Compute index values at which minimum or maximum value obtained, respectively
<code>quantile</code>	Compute sample quantile ranging from 0 to 1
<code>sum</code>	Sum of values
<code>mean</code>	Mean of values
<code>median</code>	Arithmetic median (50% quantile) of values
<code>mad</code>	Mean absolute deviation from mean value
<code>var</code>	Sample variance of values
<code>std</code>	Sample standard deviation of values



# Unique Values, Value Counts, and Membership

Another class of related methods extracts information about the values contained in a one-dimensional Series. To illustrate these, consider this example:

```
In [217]: obj = Series(['c', 'a', 'd', 'a', 'a', 'b', 'b', 'c', 'c'])
```

The first function is `unique`, which gives you an array of the unique values in a Series:

```
In [218]: uniques = obj.unique()
```

```
In [219]: uniques
```

```
Out[219]: array([c, a, d, b], dtype=object)
```

The unique values are not necessarily returned in sorted order, but could be sorted after the fact if needed (`uniques.sort()`). Relatedly, `value_counts` computes a Series containing value frequencies:

```
In [220]: obj.value_counts()
```

```
Out[220]:
```

c	3
a	3
b	2
d	1

# Isin()

Lastly, `isin` is responsible for vectorized set membership and can be very useful in filtering a data set down to a subset of values in a Series or column in a DataFrame:

```
In [222]: mask = obj.isin(['b', 'c'])
```

In [223]: mask	In [224]: obj[mask]
Out[223]:	Out[224]:
0    True	0    c
1   False	5    b
2   False	6    b
3   False	7    c
4   False	8    c
5    True	
6    True	
7    True	
8    True	

See [Table 5-11](#) for a reference on these methods.

*Table 5-11. Unique, value counts, and binning methods*

Method	Description
<code>isin</code>	Compute boolean array indicating whether each Series value is contained in the passed sequence of values.
<code>unique</code>	Compute array of unique values in a Series, returned in the order observed.
<code>value_counts</code>	Return a Series containing unique values as its index and frequencies as its values, ordered count in descending order.



# Missing Data

- pandas uses the floating point value NaN (Not a Number) to represent missing data in both floating as well as in non-floating point arrays.
- It is just used as a sentinel that can be easily detected

*Table 5-12. NA handling methods*

Argument	Description
<code>dropna</code>	Filter axis labels based on whether values for each label have missing data, with varying thresholds for how much missing data to tolerate.
<code>fillna</code>	Fill in missing data with some value or using an interpolation method such as 'ffill' or 'bfill'.
<code>isnull</code>	Return like-type object containing boolean values indicating which values are missing / NA.
<code>notnull</code>	Negation of <code>isnull</code> .

# IsNull()

```
In [229]: string_data = Series(['aardvark', 'artichoke', np.nan, 'avocado'])
```

```
In [230]: string_data
```

```
Out[230]:
```

```
0    aardvark
1    artichoke
2         NaN
3     avocado
```

```
In [231]: string_data.isnull()
```

```
Out[231]:
```

```
0    False
1    False
2     True
3    False
```

The built-in Python `None` value is also treated as NA in object arrays:

```
In [232]: string_data[0] = None
```

```
In [233]: string_data.isnull()
```

```
Out[233]:
```

```
0     True
1    False
2     True
3    False
```

# Filtering missing data-dropna()

```
In [234]: from numpy import nan as NA
```

```
In [235]: data = Series([1, NA, 3.5, NA, 7])
```

```
In [236]: data.dropna()
```

```
Out[236]:
```

```
0    1.0  
2    3.5  
4    7.0
```

Naturally, you could have computed this yourself by boolean indexing:

```
In [237]: data[data.notnull()]
```

```
Out[237]:
```

```
0    1.0  
2    3.5  
4    7.0
```

# Filtering missing data-dropna()

With DataFrame objects, these are a bit more complex. You may want to drop rows or columns which are all NA or just those containing any NAs. `dropna` by default drops any row containing a missing value:

```
In [238]: data = DataFrame([[1., 6.5, 3.], [1., NA, NA],  
.....:                    [NA, NA, NA], [NA, 6.5, 3.]])
```

```
In [239]: cleaned = data.dropna()
```

In [240]: data	In [241]: cleaned
Out[240]:	Out[241]:
0 1 6.5 3	0 1 6.5 3
1 1 NaN NaN	
2 NaN NaN NaN	
3 NaN 6.5 3	

Passing `how='all'` will only drop rows that are all NA:

```
In [242]: data.dropna(how='all')  
Out[242]:  
0 1 6.5 3  
1 1 NaN NaN  
3 NaN 6.5 3
```

# Filling in missing data

- Rather than filtering out missing data (and potentially discarding other data along with it), you may want to fill in the “holes” in any number of ways.
- For most purposes, the `fillna` method is the workhorse function to use. Calling `fillna` with a constant replaces missing values with that value:
  - `df.fillna(0)` OR
  - `df.fillna({1: 0.5, 3: -1})`

# Reading and writing data into csv file

- A CSV (Comma Separated Values) format is one of the most simple and common ways to store tabular data.
- To represent a CSV file, it must be saved with the **.csv** file extension

# Example 1: Read CSV Having Comma Delimiter

Name	Age	Profession
Jack	23	Doctor
Miller	22	Engineer

```
import csv
with open('people.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```



## Example 2: Read CSV file Having Tab Delimiter

```
import csv
with open('people.csv', 'r',) as file:
    reader = csv.reader(file, delimiter = '\t')
    for row in reader:
        print(row)
```

# Writing

- To write to a CSV file in Python, we can use the `csv.writer()` function.
- The `csv.writer()` function returns a writer object that converts the user's data into a delimited string.
- This string can later be used to write into CSV files using the `writerow()` function

# Example 1

```
import csv
with open('protagonist.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(["SN", "Movie", "Protagonist"])
    writer.writerow([1, "Lord of the Rings", "Frodo Baggins"])
    writer.writerow([2, "Harry Potter", "Harry Potter"])
```

## Example 2

```
import csv
csv_rowlist = [["SN", "Movie", "Protagonist"], [1, "Lord of the
Rings", "Frodo Baggins"], [2, "Harry Potter", "Harry Potter"]]
with open('protagonist.csv', 'w') as file:
    writer = csv.writer(file)
    writer.writerows(csv_rowlist)
```

# Using Pandas

- Pandas features a number of functions for reading tabular data as a DataFrame object.
- Table 6-1 has a summary of all of them, though `read_csv` and `read_table` are likely the ones you'll use the most

*Table 6-1. Parsing functions in pandas*

Function	Description
<code>read_csv</code>	Load delimited data from a file, URL, or file-like object. Use comma as default delimiter
<code>read_table</code>	Load delimited data from a file, URL, or file-like object. Use tab ( <code>'\t'</code> ) as default delimiter
<code>read_fwf</code>	Read data in fixed-width column format (that is, no delimiters)
<code>read_clipboard</code>	Version of <code>read_table</code> that reads data from the clipboard. Useful for converting tables from web pages

# Reading in pandas

- **Reading csv file**

```
df = pd.read_csv('ch06/ex1.csv')
```

- **Assigning default column names**

```
pd.read_csv('ch06/ex2.csv', header=None)
```

- **Assigning names**

```
pd.read_csv('ch06/ex2.csv', names=['a', 'b', 'c', 'd', 'message'])
```

- **Setting index column**

```
pd.read_csv('ch06/ex2.csv', names=names,  
index_col='message')
```

# Handling missing data

- Missing data is usually either not present (empty string) or marked by some sentinel value.
- By default, pandas uses a set of commonly occurring sentinels, such as NA, -1.#IND, and NULL:
- *pd.isnull(df)*
- The `na_values` option can take either a list or set of strings to consider missing values:
- *result = pd.read\_csv('ch06/ex5.csv', na\_values=['NULL'])*

# Skipping rows

- We can skip the first, third, and fourth rows of a file with skiprows:
- `pd.read_csv('ch06/ex4.csv', skiprows=[0, 2, 3])`



# Writing data in pandas

- Using DataFrame's `to_csv` method, we can write the data out to a comma-separated file:
- `data.to_csv('ch06/out.csv')`

# Exercise 1

- (a) Write Python program to write the data given below to a CSV file.

SN	Name	Country	Contribution	Year
1	Linus Torvalds	Finland	Linux Kernel	1991
2	Tim Berners-Lee	England	World Wide Web	1990
3	Guido van Rossum	Netherlands	Python	1991

# Solution

```
import pandas as pd
# creating a dataframe from a dictionary
df = pd.DataFrame([[1,' Linus Torvalds','Finland','Linux
Kernel ',1991], [2,'Tim Berners-Lee','England','World Wide
Web',1990],[3,'Guido van
Rossum','Netherlands','Python',1991]],columns=['SN','Name',
'Country','Contribution','Year'])
print("data frame with default index=\n",df)
df=df.set_index('SN')
print("data frame with SN as index=\n",df)
print(df)
df.to_csv('inventors.csv')
```

# Exercise 2

Given the sales information of a company as CSV file with the following fields *month\_number*, *facecream*, *facewash*, *toothpaste*, *bathingsoap*, *shampoo*, *moisturizer*, *total\_units*, *total\_profit*. Write Python codes to visualize the data as follows

- 1) Toothpaste sales data of each month and show it using a scatter plot
- 2) Face cream and face wash product sales data and show it using the bar chart

Calculate total sale data for last year for each product and show it using a Pie chart.

# 1. Read Total profit of all months and show it using a line plot

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
profitList = df ['total_profit'].tolist()
monthList = df ['month_number'].tolist()
plt.plot(monthList, profitList, label = 'Month-wise Profit data of last year')
plt.xlabel('Month number')
plt.ylabel('Profit in dollar')
plt.xticks(monthList)
plt.title('Company profit per month')
plt.yticks([100000, 200000, 300000, 400000, 500000])
plt.show()
```

## 2. Read toothpaste sales data of each month and show it using a scatter plot

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
monthList = df ['month_number'].tolist()
toothPasteSalesData = df ['toothpaste'].tolist()
plt.scatter(monthList, toothPasteSalesData, label = 'Tooth paste Sales data')
plt.xlabel('Month Number')
plt.ylabel('Number of units Sold')
plt.legend(loc='upper left')
plt.title(' Tooth paste Sales data')
plt.xticks(monthList)
#plt.grid(True, linewidth= 1, linestyle="--")
plt.show()
```

### 3. Read sales data of bathing soap of all months and show it using a bar chart

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
monthList = df ['month_number'].tolist()
bathingsoapSalesData = df ['bathingsoap'].tolist()
plt.bar(monthList, bathingsoapSalesData)
plt.xlabel('Month Number')
plt.ylabel('Sales units in number')

plt.xticks(monthList)
plt.grid(True, linewidth= 1, linestyle="--")
plt.title('bathingsoap sales data')
plt.show()
```

## 4. Read face cream and facewash product sales data and show it using the bar chart

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('D:\CEMP 2022\company_sales_data.csv')
monthList = df ['month_number'].tolist()
faceCremSalesData = df ['facecream'].tolist()
faceWashSalesData = df ['facewash'].tolist()

plt.bar(monthList, faceCremSalesData, width= 0.25, label = 'Face Cream sales data',
align='edge')
plt.bar(monthList, faceWashSalesData, width= -0.25, label = 'Face Wash sales data',
align='edge')
plt.xlabel('Month Number')
plt.ylabel('Sales units in number')
plt.legend(loc='upper left')
#plt.title(' Sales data')

plt.xticks(monthList)
#plt.grid(True, linewidth= 1, linestyle="--")
plt.title('Facewash and facecream sales data')
plt.show()
```



## 5. Calculate total sale data for last year for each product and show it using a Pie chart

```
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")
monthList = df['month_number'].tolist()

labels = ['FaceCream', 'FaseWash', 'ToothPaste', 'Bathing soap',
'Shampoo', 'Moisturizer']
salesData = [df['facecream'].sum(), df['facewash'].sum(), df
['toothpaste'].sum(), df['bathingsoap'].sum(), df['shampoo'].sum(), df
['moisturizer'].sum()]
plt.axis("equal")
plt.pie(salesData, labels=labels, autopct='%1.1f%%')
plt.legend(loc='lower right')
plt.title('Sales data')
plt.show()
```

## 6. Read Bathing soap facewash of all months and display it using the Subplot

- `import pandas as pd`
- `import matplotlib.pyplot as plt`
- `df = pd.read_csv("D:\\Python\\Articles\\matplotlib\\sales_data.csv")`
- `monthList = df ['month_number'].tolist()`
- `bathingsoap = df ['bathingsoap'].tolist()`
- `faceWashSalesData = df ['facewash'].tolist()`
- `f, axarr = plt.subplots(2, sharex=True)`
- `axarr[0].plot(monthList, bathingsoap, label = 'Bathingsoap Sales Data', color='k', marker='o', linewidth=3)`
- `axarr[0].set_title('Sales data of a Bathingsoap')`
- `axarr[1].plot(monthList, faceWashSalesData, label = 'Face Wash Sales Data', color='r', marker='o', linewidth=3)`
- `axarr[1].set_title('Sales data of a facewash')`
- `plt.xticks(monthList)`
- `plt.xlabel('Month Number')`
- `plt.ylabel('Sales units in number')`
- `plt.show()`

# Exercise 3

**Create a stud.csv file containing rollno,name,place and mark of students. Use this file and do the following**

- 1. Read and display the file contents**
- 2. Set rollno as index**
- 3. Display name and mark**
- 4. rollno,Name and mark in the order of name**
- 5. Display the rollno,name, mark in the descending order of mark**
- 6. Find the average mark,median and mode**
- 7. Find minimum and maximum marks**

# Exercise 4

Given a file “auto.csv” of automobile data with the fields *index*, *company*, *body-style*, *wheel-base*, *length*, *engine-type*, *num-of-cylinders*, *horsepower*, *average-mileage*, and *price*, write Python codes using Pandas to

- 1) Clean and Update the CSV file
- 2) Print total cars of all companies
- 3) Find the average mileage of all companies
- 4) Find the highest priced car of all companies.

# Solution- Link

<https://setscholars.net/wp-content/uploads/2019/07/Pandas-Python-Crash-Course.html>