Introduction to NumPy

Prof. Dr. Matangini Chattopadhyay
School of Education Technology
Jadavpur University

What is NumPy?

- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
- NumPy stands for ‘Numerical Python’.

Why Use NumPy?

- In Python, lists serve the purpose of arrays, but they 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.
- Arrays are very frequently used in data science, where speed and resources are very important.
- **Data Science** is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

Why is NumPy Faster Than Lists?

- NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
- This is the main reason why NumPy is faster than lists. Also, it is optimized to work with latest CPU architectures.
- NumPy is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Installation of NumPy

- Python and PIP already installed on a system, then install NumPy using the following command:

  C:\Users*Your Name*>pip install numpy

- If the above command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.
- Once NumPy is installed, import it in the applications by adding the import keyword:

  import numpy

Example

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

- NumPy is usually imported under the np alias.
- Create an alias with the as keyword while importing.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)   # [1 2 3 4 5]
print(np.__version__) # 1.16.3
```

Create a NumPy ndarray Object

- The array object in NumPy is called ndarray.
- Create a NumPy ndarray object by using the array() function.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)   # [1 2 3 4 5]
print(type(arr))   # <class 'numpy.ndarray'>
```

ndarray Object (Contd.)

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

Example: Use a tuple to create a NumPy array.

```
import numpy as np
arr = np.array((5, 6, 7, 8, 9))
print(arr)   # [5 6 7 8 9]
```

Dimensions in Arrays

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

0-D Arrays:

- 0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example: Create a 0-D array with value 100.

```
import numpy as np
arr = np.array(100)
print(arr)   # 100
```

1-D Arrays

- An array (the most common and basic arrays) that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

Example: Create a 1-D array containing the values 10, 20, 30, 40, 50

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print(arr)   # [10 20 30 40 50]
```

2-D Arrays

- An array that has 1-D arrays as its elements is called a 2-D array and are often used to represent matrix.
- NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

Example: Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6.

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

Output:

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

3-D Arrays

- An array that has 2-D arrays (matrices) as its elements is called 3-D array.

Example: Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6.

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

Output:

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

Number of Dimensions

- NumPy Array provides the ndim attribute that returns an integer that tells us how many dimensions the arrays 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)       # 0
print(b.ndim)       # 1
print(c.ndim)       # 2
print(d.ndim)       # 3
```

Higher Dimensional Arrays

- An array can have any number of dimensions.
- When the array is created, you can define the number of dimensions by using ndmin argument.

Example: Create an array with 5 dimensions and verify that it has 5 dimensions.

```
import numpy as np
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

Higher Dimensional Arrays (Contd.)

Output:

[[[[[1 2 3 4]]]]]

number of dimensions : 5

- The innermost dimension (5th dim) has 4 elements.
- The 4th dim has 1 element that is the vector.
- The 3rd dim has 1 element that is the matrix with the vector.
- The 2nd dim has 1 element that is 3D array.
- The 1st dim has 1 element that is a 4D array.

NumPy Array Indexing

- Array indexing is the same as accessing an array element.
- 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.

Example: Get the first and third element from the following array.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0], arr[2])     # 1 3
```

NumPy Array Indexing (Contd.)

Example: Get third and fourth elements from the array and add them.

```
import numpy as np

arr = np.array([10, 20, 30, 40])

print(arr[2] + arr[3])    # 70
```

Access 2-D Arrays

- To access elements from 2-D arrays, we can use comma separated integers representing the dimension and the index of the element.
- Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

Example :Access the element on the first row, second column and 2nd row , 5th column

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

Output:

2nd element on 1st dim: 2

5th element on 2nd dim: 10

Access 3-D Arrays

- To access elements from 3-D arrays, we can use comma separated integers representing the dimensions and the index of the element.

Example: Access the third element of the second array of the first array.

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])    #6
```

Example Explained

arr[0, 1, 2] prints the value 6.

First number represents the first dimension, which contains two arrays: [[1, 2, 3], [4, 5, 6]]
and: [[7, 8, 9], [10, 11, 12]]
Since we selected 0, we are left with the first array:
[[1, 2, 3], [4, 5, 6]]

Second number represents the second dimension, which also contains two arrays:
[1, 2, 3]
and: [4, 5, 6]
Since we selected 1, we are left with the second array:
[4, 5, 6]

Third number represents the third dimension, which contains three values:
4
5
6
Since we selected 2, we end up with the third value:
6

Negative Indexing

- Use negative indexing to access an array from the end.

Example: Print the last element from the 2nd dim.

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

Output: Last element from 2nd dim: 10

NumPy Array 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, it is considered 0.
- If we don't pass end, it is considered length of array in that dimension.
- If we don't pass step, it is considered 1.

Example

- Slice elements from index 1 to index 5 from the following array.

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

Note: The result *includes* the start index, but *excludes* the end index.

- Slice elements from index 4 to the end of the array.

```
print(arr[4:])        # [5 6 7]
```

- Slice elements from the beginning to index 4 (not included)

```
print(arr[:4])        # [1 2 3 4]
```

Negative Slicing

- Use the minus operator to refer to an index from the end.

Example: Slice from the index 3 from the end to index 1 from the end.

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

Step

- Use the step value to determine the step of the slicing.

Example: Return every other element from index 1 to index 5.

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

- Return every other element from the entire array.

```
print(arr[::2])     # [1 3 5 7]
```

Slicing 2-D Arrays

Example: From the second element, slice elements from index 1 to index 4 (not included)

```
import numpy as np

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

print(arr[1, 1:4]) # [7 8 9]
```

- From both elements, return index 2.

```
print(arr[0:2, 2])          # [3 8]
```

- From both elements, slice index 1 to index 4 (not included), this will return a 2-D array.

```
print(arr[0:2, 1:4])        # [[2 3 4]
                               [7 8 9]]
```

Array: Copy vs View

- Copy operation makes a new array, but the view is just a view of the original array.
- Copy of an array *owns* the data and any changes made to the copy will not affect the original array and vice versa.
- The view *does not own* the data and any changes made to the view will affect the original array and vice versa.

Copy

Example: Make a copy, change the original array, and display both arrays.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42
print(arr)   # [42 2 3 4 5]
print(x)     # [1 2 3 4 5]
```

View

Example: Make a view, change the original array, and display both arrays.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42
print(arr)   # [42 2 3 4 5]
print(x)     # [42 2 3 4 5]
x[4] = 50
print(arr)   # [ 1 2 3 4 50]
print(x)     # [ 1 2 3 4 50]
```

Check if Array owns its Data

- NumPy array has the attribute base that returns None if the array owns the data.
- Otherwise, the base attribute returns the original object.

Example: Print the value of the base attribute to check if an array owns its data or not.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
y = arr.view()
print(x.base)        # None
print(y.base)        # [1 2 3 4 5]
```

Shape of an Array

- The shape of an array is the number of elements in each dimension.
- An array has an attribute called shape that returns a tuple with each index having the number of corresponding elements.

Example: Print the shape of a 2-D array:

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

arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)        # ??
print('shape of array :', arr.shape)            # ??
```

Reshaping arrays

- Reshaping means changing the shape of an array.
- By reshaping ,we can add or remove dimensions or change number of elements in each dimension.
- Example: Convert a 1-D array with 12 elements into a 2-D array. The outermost dimension will have 4 arrays, each with 3 elements.

```
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)
```

Output:

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

Reshape From 1-D to 3-D

Example: Convert a 1-D array with 12 elements into a 3-D array. The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements.

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

Output:

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

Example

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

Output:

```
Traceback (most recent call last):
File "./prog.py", line 5, in <module>
ValueError: cannot reshape array of size 8 into shape (3,3)
```

Example: (Copy or View)

- Check if the returned array is a copy or a view.

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

- Returns the original array, so it is a view.

Unknown Dimension

- One "unknown" dimension is allowed.
- Do not specify an exact number for one of the dimensions in the reshape method.
- Pass -1 as the value, and NumPy will calculate this number.

Example: Convert 1D array with 8 elements to 3D array with 2x2 elements.

```
import numpy as np

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

newarr = arr.reshape(2, 2, -1)

print(newarr)
```

Output:

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

Flattening the arrays

- Flattening array means converting a multidimensional array into a 1D array.
- Use reshape(-1) to do this.

Example: Convert the array into a 1D array

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

Iterating Arrays

- Iterating means going through elements one by one.
- It is done using basic for loop of python.

Example: Iterate on the elements of 1-D array.

```
import numpy as np
arr = np.array([1, 2, 3])
for x in arr:
  print(x)
```

Output:

1

2

3

Iterating 2D Array

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

Output:

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

Question:

What changes are needed in the above program so that the output becomes:

```
1
2
3
4
5
6
```

Iterating 3D Array

- In a 3-D array, it will go through all the 2-D arrays.

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

Output:

```
x represents the 2-D array:
[[1 2 3]
  [4 5 6]]
x represents the 2-D array:
[[ 7 8 9]
  [10 11 12]]
```

Iterating n-D Array

- If we iterate on an *n*-D array, it will go through (n-1)th dimension one by one.
- To return the actual values (i.e., the scalars), we have to iterate the arrays in each dimension.

Example:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
for x in arr:
  for y in x:
    for z in y:
      print(z)
```

Output:

??

Using nditer()

- The function nditer() is a helping function that can be used from very basic to very advanced iterations.
- To iterate through each scalar of an array, we need to use *'n'* for loops for n-D array (arrays with high dimensionality).

Example:

```
import numpy as np
arr = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
for x in np.nditer(arr):
  print(x)
```

Output:

??

Iterating With Different Step Size

- We can use filtering and followed by iteration.

Example: Iterate through every scalar element of the 2D array skipping 1 element.

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for x in np.nditer(arr[:, ::2]):
  print(x)
```

Output:

1

3

5

7

Enumerated Iteration Using ndenumerate()

- Enumeration means mentioning sequence number of something one by one.
- The ndenumerate() method can be used when we need corresponding index of the element while iterating.

Example: Enumerate on following 1D arrays elements.

```
import numpy as np
arr = np.array([1, 2, 3])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Output:

```
(0,) 1
(1,) 2
(2,) 3
```

Enumerate on 2D array's elements

Example:

```
import numpy as np
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

Output:

```
(0, 0) 1
(0, 1) 2
(0, 2) 3
(0, 3) 4
(1, 0) 5
(1, 1) 6
(1, 2) 7
(1, 3) 8
```

Joining NumPy Arrays

- Joining means putting contents of two or more arrays in a single array.
- Pass a sequence of arrays that you want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Example: Join two arrays

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

Joining 2D Arrays

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
arr = np.concatenate((arr1, arr2), axis=1)
print(arr)
```

Output:

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

```
arr = np.concatenate((arr1, arr2), axis=0)
print(arr)
```

Output:

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

Joining Arrays Using Stack Functions

- Stacking is same as concatenation, the only difference is that stacking is done along a new axis.
- Concatenate two 1-D arrays along the second axis which would result in putting them one over the other, i.e., stacking.
- Pass a sequence of arrays to the stack() method along with the axis. If axis is not passed, it is taken as 0.

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

```
Output: [[1 4]
         [2 5]
         [3 6]]
```

Stacking along Rows/Columns/Height(Depth)

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.hstack((arr1, arr2))  # hstack() along rows
print(arr)      # [1 2 3 4 5 6]
arr = np.vstack((arr1, arr2))   # vstack() along columns
print(arr)
arr = np.dstack((arr1, arr2))  # dstack() along height (depth)
print(arr)
```

Output:

(Along columns)

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

Along height(or depth)

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

Splitting NumPy Arrays

- Joining merges multiple arrays into one and Splitting breaks one array into multiple.
- Use array_split() for splitting arrays; pass the array and the number of splits.

Example: Split the array in 3 parts.

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

Note: The return value is a list containing three arrays.

Splitting Arrays (Contd.)

- If the array has less elements than required, it will adjust from the end accordingly.

Example: Split the array in 4 parts.

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

Output:

```
[array([1, 2]), array([3, 4]), array([5]), array([6])]
```

Accessing splitted arrays

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr[0])    # [1 2]
print(newarr[1])    # [3 4]
print(newarr[2])    # [5 6]
```

Splitting 2-D Arrays

Example: Split the 2-D array into three 2-D arrays.

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

Output:

```
[array([[1, 2],
        [3, 4]]), array([[5, 6],
        [7, 8]]), array([[ 9, 10],
        [11, 12]])]
```

Splitting 2-D Arrays (Contd.)

Example: Split the 2-D array into three 2-D arrays along rows.

```
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.array_split(arr, 3, axis=1)

print(newarr)
```

Output: ??

```
# hsplit() method to split the 2-D array into three 2-D arrays
# along rows.
newarr = np.hsplit(arr, 3)
print(newarr)
```

Output

```
[array([[ 1],
        [ 4],
        [ 7],
        [10],
        [13],
        [16]]), array([[ 2],
        [ 5],
        [ 8],
        [11],
        [14],
        [17]]), array([[ 3],
        [ 6],
        [ 9],
        [12],
        [15],
        [18]])]
```

vsplit() and dsplit()

```
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.vsplit(arr, 3)
print(newarr)
```

Output:

```
[array([[1, 2, 3],
        [4, 5, 6]]), array([[ 7, 8, 9],
        [10, 11, 12]]), array([[13, 14, 15],
        [16, 17, 18]])]
```

Note: dsplit () only works on arrays of 3 or more dimensions.

Searching Arrays

- Search an array for a certain value, and return the indexes that get a match.
- Use the where() method.

Example: Find the indexes where the value is 4.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 4, 4])
x = np.where(arr == 4)
print(x)          # (array([3, 5, 6]),)
```

- Returns a tuple: (array([3, 5, 6],); it means that the value 4 is present at index 3, 5, and 6.

Searching Arrays: Example

- Find the indexes where the values are even.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 0)
print(x)        # (array([1, 3, 5, 7]),)
# For odd
x = np.where(arr%2 == 1)
print(x)        # (array([0, 2, 4, 6]),)
```

Sorting Arrays

- Sorting means putting elements in an *ordered sequence*.
- *Ordered sequence* is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.
- The NumPy ndarray object has a function called sort(), that will sort a specified array.

Example:

```
import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))  # [0 1 2 3]
```

Note: This method returns a copy of the array, leaving the original array unchanged.

Sorting Arrays (Contd.)

Example: Sort the array alphabetically.

```
import numpy as np
arr = np.array(['banana', 'cherry', 'apple'])
print(np.sort(arr))  # ['apple' 'banana' 'cherry']
arr1 = np.array([True, False, True])
print(np.sort(arr1))          # [False True True]
```

Sorting 2D Arrays

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

Output:

```
[[2 3 4]
 [0 1 5]]
```

Filtering Arrays

- Getting some elements from an existing array and creating a new array out of them is called *filtering*.
- Filtering an array is done using a *boolean index list*.
- A *boolean index list* is a list of booleans corresponding to indexes in the array.
- If the value at an index is True that element is contained in the filtered array
- If the value at that index is False that element is excluded from the filtered array.

Filtering Arrays: Example

- Create an array from the elements on index 0 and 2.

```
import numpy as np
arr = np.array([41, 42, 43, 44])
x = arr[[True, False, True, False]]
print(x)        # [41 43]
```

Filtering Arrays: More Example

- Create a filter array that will return only values higher than 42.

```
import numpy as np
arr = np.array([41, 42, 43, 44])
# Create an empty list
filter_arr = []
# go through each element in arr
for element in arr:
  # if the element > 42, set the value to True, otherwise False
  if element > 42:
    filter_arr.append(True)
  else:
    filter_arr.append(False)
newarr = arr[filter_arr]
print(filter_arr)          # [False, False, True, True]
print(newarr)              [43 44]
```

Assignment

1. Create a filter array that will return only even elements from the original array, [1, 2, 3, 4, 5, 6, 7]

Creating Filter Directly From Array

- Create a filter array that will return only values higher than 42.

```
import numpy as np
arr = np.array([41, 42, 43, 44])
filter_arr = arr > 42
newarr = arr[filter_arr]
print(filter_arr)      # [False False True True]
print(newarr)          # [43 44]
```

Creating Filter Directly From Array (Contd.)

- Create a filter array that will return only even elements from the original array.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
filter_arr = arr % 2 == 0
newarr = arr[filter_arr]
print(filter_arr) #[False True False True False True False True]
print(newarr)         # [2 4 6 8]
```

NumPy ufuncs

- ufuncs stands for "Universal Functions" and they are NumPy functions that operate on the ndarray object.
- ufuncs are used to implement *vectorization* in NumPy which is faster than iterating over elements.
- They also provide broadcasting and additional methods like reduce, accumulate etc. that are very helpful for computation.
- ufuncs also take additional arguments, like:
  - where: boolean array or condition defining where the operations should take place
  - dtype: defining the return type of elements
  - out: output array where the return value should be copied

Example

Add the Elements of Two Lists:

list 1: [1, 2, 3, 4]

list 2: [4, 5, 6, 7]

Solution1 : Iterate over both the lists and then sum each elements.

Solution2: Without ufunc, Python's built-in zip() method can be used.

```
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = []
for i, j in zip(x, y):
  z.append(i + j)
print(z)          # [5, 7, 9, 11]
```

Use of add() function

```
import numpy as np
x = [1, 2, 3, 4]
y = [4, 5, 6, 7]
z = np.add(x, y)
print(z)        # [ 5 7 9 11]
```

Simple Arithmetic

Addition: add() function sums the content of two arrays, and return the results in a new array.

```
import numpy as np
arr1 = np.array([10, 11, 12, 13, 14, 15])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.add(arr1, arr2)
print(newarr)          # [30 32 34 36 38 40]
```

Subtraction

Example: Subtract the values in arr2 from the values in arr1.

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.subtract(arr1, arr2)
print(newarr)          # [-10 -1 8 17 26 35]
```

Multiplication

- multiply() function multiplies the values from one array with the values from another array, and return the results in a new array.

Example: Multiply the values in arr1 with the values in arr2.

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.multiply(arr1, arr2)
print(newarr)         # [ 200 420 660 920 1200 1500]
```

Division

- divide() function divides the values from one array with the values from another array, and return the results in a new array.

Example: Divide the values in arr1 with the values in arr2.

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 5, 10, 8, 2, 33])
newarr = np.divide(arr1, arr2)
print(newarr)
```

Output:

```
[ 3.33333333 4. 3. 5. 25. 1.81818182]
```

Power

- power() function raises the values from the first array to the power of the values of the second array, and return the results in a new array.

Example: Raise the values in arr1 to the power of values in arr2.

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 5, 6, 8, 2, 3])
newarr = np.power(arr1, arr2)
print(newarr)
```

Output:

[1000 3200000 729000000 6553600000000 2500 216000]

Remainder

- Both the mod() and the remainder() functions return the remainder of the values in the first array corresponding to the values in the second array, and return the results in a new array.

```
import numpy as np
arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 7, 9, 8, 2, 33])
newarr = np.mod(arr1, arr2)
print(newarr)          # [ 1 6 3 0 0 27]
newarr = np.remainder(arr1, arr2)
print(newarr)          # [ 1 6 3 0 0 27]
```

Quotient and Mod

- divmod() function return both the quotient and the mod. The return value is two arrays, the first array contains the quotient and second array contains the mod.

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])

arr2 = np.array([3, 7, 9, 8, 2, 33])

newarr = np.divmod(arr1, arr2)

print(newarr) # (array([ 3, 2, 3, 5, 25, 1]), array([ 1, 6, 3, 0, 0, 27]))
print(newarr[0])    # [ 3 2 3 5 25 1]

print(newarr[1])    # [ 1 6 3 0 0 27]
```

Absolute Values

- Both the absolute() and the abs() functions do the same absolute operation element-wise but we should use absolute() to avoid confusion with python's inbuilt abs().

```
import numpy as np
arr = np.array([-1, -2, 1, 2, 3, -4])
newarr = np.absolute(arr)
print(newarr)          # [1 2 1 2 3 4]
newarr1 = np.abs(arr)
print(newarr1)         # [1 2 1 2 3 4]
```

Rounding Decimals

- Five ways of rounding off decimals in NumPy:
  - Truncation (Remove the decimals, and return the float number closest to zero. Use the trunc() and fix() functions.)
  - fix
  - rounding
  - floor
  - ceil

Truncation

```
import numpy as np
arr = np.trunc([-3.1666, 3.6667])
print(arr)      # [-3. 3.]
arr = np.fix([-3.1666, 3.6667])
print(arr)      # [-3. 3.]
```

Rounding

- around() function increments preceding digit or decimal by 1 if >=5 else do nothing.

Example: Round off 3.1666 to 2 decimal places.

```
import numpy as np
arr = np.around(3.1666, 2)
print(arr)      # ??
```

Floor and Ceil

- The floor() function rounds off decimal to nearest lower integer.

Example:floor of 3.166 is 3.

```
import numpy as np
arr = np.floor([-3.1666, 3.6667])
print(arr)     # [-4. 3.]
```

- The ceil() function rounds off decimal to nearest upper integer (ceil of 3.166 is 4).

```
import numpy as np
arr = np.ceil([-3.1666, 3.6667])
print(arr)     # [-3. 4.]
```

Summations

- Addition is done between two arguments whereas summation happens over n elements.

Example: Sum the values in arr1 and the values in arr2.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([1, 2, 3])
newarr = np.sum([arr1, arr2])
print(newarr)        # 12
```

Summation Over an Axis

- Specify axis=1, NumPy will sum the numbers in each array.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([1, 2, 3])
newarr = np.sum([arr1, arr2], axis=1)
print(newarr)         # [6 6]
newarr = np.sum([arr1, arr2], axis=0)
print(newarr)         # [2 4 6]
```

Cumulative Sum

- Cumulative sum means partially adding the elements in array.
- Partial sum of [1, 2, 3, 4] would be [1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10].
- Perform partial sum with the cumsum() function.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
newarr = np.cumsum(arr)
print(newarr)        # [ 1 3 6 10]
```

Mean

```
import numpy as np
# 1D array
arr = np.array([20, 2, 7, 1, 34])
print("arr : ", arr)    # arr : [20 2 7 1 34]
print("mean of arr : ", np.mean(arr))   # mean of arr : 12.8
```

Median

```
import numpy as np
# 1D array
arr = np.array([20, 2, 7, 1, 34])
print("arr : ", arr)                  # arr : [20 2 7 1 34]
print("median of arr : ", np.median(arr))        # 7.0
```

Mode

- The Mode value is the value that appears the most number of times.

```
# importing required packages
from scipy import stats as st
import numpy as np
abc = np.array([1, 1, 2, 2, 2, 3, 4, 5])
print(st.mode(abc))
```

Output:

```
ModeResult(mode=array([2]), count=array([3]))
```

Mode: Another way

```
# importing required packages
import statistics as st
import numpy as np
abc = np.array([1, 1, 2, 2, 2, 3, 4, 5])
print(st.mode(abc))          # 2
```

Products

- To find the product of the elements in an array, use the prod() function.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
x = np.prod(arr)
print(x)        # 24
```

Products (Contd.)

Example: Find the product of the elements of two arrays.

```
import numpy as np
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])
x = np.prod([arr1, arr2])
print(x)        # 40320
```

Product Over an Axis

- If you specify axis=1, NumPy will return the product of each array.

```
import numpy as np
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])
newarr = np.prod([arr1, arr2], axis=1)
print(newarr)        # [ 24 1680]
newarr = np.prod([arr1, arr2], axis=0)
print(newarr)        # [ 5 12 21 32]
```

Cumulative Product

- Cummulative product means taking the product partially.
- Partial product of [1, 2, 3, 4] is [1, 1*2, 1*2*3, 1*2*3*4] = [1, 2, 6, 24]
- Perform partial sum with the cumprod() function.

```
import numpy as np
arr = np.array([5, 6, 7, 8])
newarr = np.cumprod(arr)
print(newarr)         # [ 5 30 210 1680]
```

Differences

- A discrete difference means subtracting two successive elements.
- For [1, 2, 3, 4], the discrete difference would be [2-1, 3-2, 4-3] = [1, 1, 1]
- To find the discrete difference, use the diff() function.

Example:

```
import numpy as np
arr = np.array([10, 15, 25, 5])
newarr = np.diff(arr)
print(newarr)          # [ 5 10 -20]
```

Differences (Contd.)

- Repeated difference operations can be performed by giving parameter n.

Example - for [1, 2, 3, 4], the discrete difference with n = 2

[2-1, 3-2, 4-3] = [1, 1, 1]

since n=2, next iteration gives new result: [1-1, 1-1] = [0, 0]

```
import numpy as np
arr = np.array([10, 15, 25, 5])
newarr = np.diff(arr, n=2)
print(newarr)
```

Output:

1^{st} : # [5 10 -20]

2^{nd} : # [5 -30]

LCM (Lowest Common Multiple)

- The Lowest Common Multiple is the smallest number that is a common multiple of two numbers.

Example:

```
import numpy as np
num1 = 4
num2 = 6
x = np.lcm(num1, num2)
print(x)        # 12
```

LCM in Arrays

- Use reduce() method to calculate LCM of all values in an array.
- The reduce() method will use the ufunc, in this case the lcm() function, on each element, and reduce the array by one dimension.

```
import numpy as np
arr = np.array([3, 6, 9])
x = np.lcm.reduce(arr)
print(x)        # 18
```

Note: "reduce()" function uses the lambda function cumulatively to the elements of the input list. The lambda function calculates the LCM using the formula: LCM(a, b) = (a * b) / GCD(a, b).

LCM in Arrays (Contd.)

```
import numpy as np
from functools import reduce
arr = np.array([3, 6, 9])
x = reduce(lambda a, b: np.lcm(a, b), arr)
print(x)            # 18
```

Lambda function

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression. The expression is executed and the result is returned.

Syntax: lambda *arguments* : *expression*

Example: Add 10 to argument a, and return the result.

```
x = lambda a : a + 10
print(x(5))        # 15
```

```
# Summarize argument a, b, and c and return the result.
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))          # 13
```

- Note: Use lambda functions when an anonymous function is required for a short period of time.

LCM in Arrays: Example

- Find the LCM of all values of an array where the array contains all integers from 1 to 10.

```
import numpy as np
arr = np.arange(1, 11)
print(arr)        # [ 1 2 3 4 5 6 7 8 9 10]
x = np.lcm.reduce(arr)
print(x)          # 2520
```

GCD (Greatest Common Denominator)

The GCD (Greatest Common Denominator), also known as HCF (Highest Common Factor) is the biggest number that is a common factor of both of the numbers.

Example:

```
import numpy as np
num1 = 6
num2 = 9
x = np.gcd(num1, num2)
print(x)          # 3
#Finding GCD in Arrays
arr = np.array([20, 8, 32, 36, 16])
x = np.gcd.reduce(arr)
print(x)          # 4
```

Trigonometric Functions

- NumPy provides the functions sin(), cos() and tan() that take values in radians and produce the corresponding sin, cos and tan values.

Example: Find sine value of PI/2.

```
import numpy as np
x = np.sin(np.pi/2)
print(x)            #1.0
# Find sine values for all of the values in an array.
arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])
x = np.sin(arr)
print(x)            # [1. 0.8660254 0.70710678 0.58778525]
```

Convert Degree to Radian and vice versa

- By default, all of the trigonometric functions take radians as parameters .

  radians values = pi/180 * degree_values

- # Convert all the values in arr to radians.

```
import numpy as np
arr = np.array([90, 180, 270, 360])
x = np.deg2rad(arr)
print(x)            # [1.57079633 3.14159265 4.71238898 6.28318531]
# Convert all the values in arr to degrees.
arr = np.array([np.pi/2, np.pi, 1.5*np.pi, 2*np.pi])
x = np.rad2deg(arr)
print(x)
```

Finding Angles

- Finding angles from values of sine, cos, tan i.e., sin, cos and tan inverse (arcsin, arccos, arctan).
- NumPy provides ufuncs arcsin(), arccos() and arctan() that produce radian values for corresponding sin, cos and tan values.

```
import numpy as np
x = np.arcsin(1.0)
print(x)                # 1.5707963267948966
print(np.rad2deg(x))    # 90.0
# Find the angle for all of the sine values in an array.
arr = np.array([1, -1, 0.1])
x = np.arcsin(arr)      # [ 1.57079633 -1.57079633 0.10016742]
print(np.rad2deg(x))    # [ 90. -90. 5.73917048]
```

Hypotenues

- NumPy provides the hypot() function that takes the base and perpendicular values and produces hypotenues based on pythagoras theorem.

Example:

```
import numpy as np
base = 3
perp = 4
x = np.hypot(base, perp)
print(x)            # 5.0
```

Create Sets in NumPy

- NumPy's unique() method to find unique elements from any array.

Example: Convert the array with repeated elements to a set.

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

Finding Union, Intersection

- To find the unique values of two arrays, use the union1d() method.

```
import numpy as np
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([3, 4, 5, 6])
newarr = np.union1d(arr1, arr2)
print(newarr)            # [1 2 3 4 5 6]

# Find intersection of two set arrays.
newarr = np.intersect1d(arr1, arr2, assume_unique=True)
print(newarr)            # [3 4]
```

Note: assume_unique is an optional argument. Setting to True can speed up computation. It should always be set to True when dealing with sets.

Difference & Symmetric Difference

- setdiff1d() method is used to find only the values in the first set that is NOT present in the second set.

```
import numpy as np
set1 = np.array([1, 2, 3, 4])
set2 = np.array([3, 4, 5, 6])
newarr = np.setdiff1d(set1, set2, assume_unique=True)
print(newarr)               # [1 2]
# To find only the values that are NOT present in BOTH sets, use the setxor1d() method.
newarr = np.setxor1d(set1, set2, assume_unique=True)
print(newarr)               # [1 2 5 6]
```

Any question?

Data Analysis using Pandas

Prof. Dr. Matangini Chattopadhyay
School of Education Technology
Jadavpur University

What is Pandas?

- Pandas is a Python library used for working with data sets.
- It has functions for analyzing, cleaning, exploring, and manipulating data.
- The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.
- Adds data structures and tools designed to work with table-like data
- Provides tools for data manipulation: reshaping, merging, sorting, slicing, aggregation etc.
- Allows handling missing data

Installation of Pandas

- Python and PIP already installed on a system, then install Pandas using the following command:

  C:\Users*Your Name*>pip install pandas

- If the above command fails, then use a python distribution that already has Pandas installed like, Anaconda, Spyder etc.
- Once Pandas is installed, import it in the applications by adding the import keyword:

  import pandas

  import pandas as pd   # create an alias

Example

```
import pandas as pd
mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}
myvar = pd.DataFrame(mydataset)
print(myvar)
print(pd.__version__)   # 1.0.3
```

Output:

```
     cars   passings
0  BMW            3
1  Volvo          7
2  Ford           2
```

Pandas Series

- A Pandas Series is like a column in a table.
- It is a one-dimensional array holding data of any type.

Example: Create a simple Pandas Series from a list.

```
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a)
print(myvar)
```

Output: (values are labeled with their index number starting 0)

```
0    1
1    7
2    2
dtype: int64
```

Labels

- Label can be used to access a specified value.

print(myvar[0], myvar[1], myvar[2]) # 1 7 2

- Name your own labels with the index argument.

Example:

```
import pandas as pd
a = [1, 7, 2]
myvar = pd.Series(a, index = ["x", "y", "z"])
print(myvar)
```

Output:

```
x     1
y     7
z     2
dtype: int64
```

```
print(myvar["y"])            # 7
```

Key/Value Objects as Series

Example: Create a simple Pandas Series from a dictionary.

```
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)
```

Output:

```
day1    420
day2    380
day3    390
dtype: int64
```

Note: The keys of the dictionary become the labels.

Example

- To select only some of the items in the dictionary, use the index argument and specify only the items you want to include in the Series.

```
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories, index = ["day1", "day2"])
print(myvar)
```

Output:

```
day1   420
day2   380
dtype: int64
```

DataFrames

- Data sets in Pandas are usually multi-dimensional tables, called DataFrames.
- Series is like a column, a DataFrame is the whole table.

```
import pandas as pd
data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}
myvar = pd.DataFrame(data)
print(myvar)
```

	calories	duration
0	420	50
1	380	40
2	390	45

Locate Row

- Use the loc attribute to return one or more specified row(s).

Example: Return row 0.

```
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.loc[0])
```

Output:

```
calories    420
duration     50
Name: 0, dtype: int64
```

Locate Row (Contd.)

Example: Return row 0 and 1

#use a list of indexes

print(df.loc[[0, 1]])

Output:

	calories	duration
0	420	50
1	380	40

Named Indexes

- Name your own indexes using the index argument

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

Output:

	calories	duration
day1	420	50
day2	380	40
day3	390	45

```
# Locate the named index, “day2”.
print(df.loc["day2"])
```

Output:

```
calories    380
duration     40
Name: day2, dtype: int64
```

Named Indexes

- Name your own indexes using the index argument

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

Output:

```
          calories  duration
day1           420        50
day2           380        40
day3           390        45
```

```
# Locate the named index, "day2".
print(df.loc["day2"])
```

Output:

```
calories    380
duration     40
Name: day2, dtype: int64
```

Read CSV Files

- A simple way to store big data sets is to use CSV files (comma separated files).
- CSV files (e.g., 'data.csv‘)contains plain text and is a well known format that can be read by everyone including Pandas.

Example: Load the CSV into a DataFrame.

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.to_string()) # to_string() to print the entire
                      # DataFrame.
```

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
...	...	...	...	...

Read CSV Files (Contd.)

- Large DataFrame with many rows, Pandas will only return the first 5 rows, and the last 5 rows.

```
print(df)
     Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2
3
4
...       ...    ...       ...       ...

164
165
166
167
168
[169 rows x 4 columns]
```

max_rows

- Check your system's maximum rows with the pd.options.display.max_rows statement.

print(pd.options.display.max_rows) # 60

- Change the maximum rows number with the same statement.

Example: Increase the maximum number of rows to display the entire DataFrame.

```
import pandas as pd
pd.options.display.max_rows = 9999
df = pd.read_csv('data.csv')
print(df)       # All rows  along with the header will be
                # displayed
```

JSON

- Big data sets are often stored or extracted as JSON.
- JSON stands for JavaScript Object Notation.
- A data interchange format to store and transfer data.
- JSON is human and machine-readable, and is independent of any programming language.
- JSON is plain text, but has the format of an object.
- JSON represents data in two ways: objects and arrays.
  - Objects are collections of name-value pairs, defined within {}.
  - Arrays are ordered collections of values, defined within [].
- A JSON file called, 'data.json‘ has been used in our examples.

Read JSON

- Load the JSON file into a DataFrame:

Example:

```
import pandas as pd
df = pd.read_json('data.json')
print(df.to_string())          # To print the entire DataFrame
```

Dictionary as JSON

- JSON objects have the same format as Python dictionaries.
- If data is not in a JSON file, but in a Python Dictionary, it can be loaded into a DataFrame directly.

Example

```
import pandas as pd
data = {
  "Duration":{
    "0":60,
    "1":60,
    "2":60,
    "3":45,
    "4":45,
    "5":60
  },
  "Pulse":{
    "0":110,
    "1":117,
    "2":103,
    "3":109,
    "4":117,
    "5":102
  },
  "Maxpulse":{
    "0":130,
    "1":145,
    "2":135,
    "3":175,
    "4":148,
    "5":127
  },
  "Calories":{
    "0":409.1,
    "1":479.0,
    "2":340.0,
    "3":282.4,
    "4":406.0,
    "5":300.5
  }
}
df = pd.DataFrame(data)
print(df)
```

Analyzing DataFrames

- Viewing the Data
  - head() method returns the headers and a specified number of rows, starting from the top.

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df.head(10))  # print the first 10 rows of the DataFrame
                    # along with the header
print(df.head())    # Print the first 5 rows of the DataFrame
print(df.tail())    # Print the last 5 rows of the DataFrame
print(df.tail(10))  # print the last 10 rows of the DataFrame
                    # along with the header
```

Info about the Data

Example: Print information about the data.

print(df.info())

Output:

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 169 entries, 0 to 168 # 169 rows & 4 columns

Data columns (total 4 columns):

#	Column	Non-Null Count	Dtype	# Each column name & data type
---	------	--------------	-----	
0	Duration	169 non-null	int64	
1	Pulse	169 non-null	int64	
2	Maxpulse	169 non-null	int64	
3	Calories	164 non-null	float64	

dtypes: float64(1), int64(3) memory usage: 5.4 KB

None

Null Values

- info() method also tells how many Non-Null values present in each column.
- There are 164 of 169 Non-Null values in the "Calories" column (5 rows with no value at all).
- Empty values, or Null values, can be bad when analyzing data.
- Remove rows with empty values. This is a step towards what is called *cleaning data*

Data Cleaning

- Data cleaning means fixing bad data in the data set.
- Bad data could be:
  - Empty cells
  - Data in wrong format
  - Wrong data
  - Duplicates

Data Set

- The data set contains some empty cells ("Date" in row 22, and "Calories" in row 18 and 28).
- The data set contains wrong format ("Date" in row 26).
- The data set contains wrong data ("Duration" in row 7).
- The data set contains duplicates (row 11 and 12).

Cleaning Empty Cells

- Cleaning empty cells means removing rows that contain empty cells.
- Data sets can be very big and removing a few rows will not have a big impact on the result.

Example: Return a new Data Frame with no empty cells

```
import pandas as pd
df = pd.read_csv('data.csv')
new_df = df.dropna()
print(new_df.to_string())
# In the result, some rows have been removed (row 18, 22 and 28).
# These rows had cells with empty values.
```

Note: By default, the dropna() method returns a *new* DataFrame, and will not change the original.

Cleaning Empty Cells (Contd.)

- To change the original DataFrame, use the inplace = True argument.

Example: Remove all rows with NULL values.

```
import pandas as pd
df = pd.read_csv('data.csv')
df.dropna(inplace = True)
print(df.to_string())
```

Note: dropna(inplace = True) will NOT return a new DataFrame, but it will remove all rows containing NULL values from the original DataFrame.

Replace Empty Values

- Insert a *new* value in the empty cells.
- fillna() method allows us to replace empty cells with a value.

Example: Replace NULL values with the number 130.

```
import pandas as pd
df = pd.read_csv('data.csv')
df.fillna(130, inplace = True)
print(df.to_string())
# In the result: empty cells got the value 130 (in row 18, 22 and 28).
```

Note: Replaces all empty cells in the whole Data Frame.

Replace only for specified columns

- To replace empty values for one column, specify the *column name* for the DataFrame:

Example: Replace NULL values in the "Calories" columns with the number 130.

```
import pandas as pd

df = pd.read_csv('data.csv')

df["Calories"].fillna(130, inplace = True)

print(df.to_string())

#This operation inserts 130 in empty cells in the "Calories" column (row 18 and 28).
```

Replace Using Mean, Median, or Mode

- A common way to replace empty cells, is to calculate the mean, median or mode value of the column.
- Pandas uses the mean(), median() and mode() methods to calculate the respective values for a specified column

Example: Calculate the MEAN, and replace any empty values with it.

```
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].mean()
df["Calories"].fillna(x, inplace = True)
print(df.to_string())
```

In row 18 and 28, the empty values from "Calories" are replaced with the mean 304.68.

Note: **Mean** = the average value (the sum of all values divided by number of values)

Replace using Median

```
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].median()
df["Calories"].fillna(x, inplace = True)
print(df.to_string())
# In row 18 and 28, the empty values from "Calories" are replaced with the median 291.2.
```

Note: **Median** = the value in the middle, after all values are sorted in ascending order.

Replace using Mode

```
import pandas as pd
df = pd.read_csv('data.csv')
x = df["Calories"].mode()[0]
df["Calories"].fillna(x, inplace = True)
print(df.to_string())
# In row 18 and 28, the empty value from "Calories" are replaced with the mode 300.0
```

Note: **Mode** = the value that appears most frequently.

Cleaning Data of Wrong Format

- Two options: remove the rows, or convert all cells in the columns into the same format.

Example: Convert into a correct format (all cells in the 'Date' column into dates). Use to_datetime() method

```
import pandas as pd
df = pd.read_csv('data.csv')
df['Date'] = pd.to_datetime(df['Date'])
print(df.to_string())
```

- Date in row 26 has been fixed, but the empty date in row 22 got a NaT (Not a Time) value (an empty value).
- One way to deal with empty values is simply removing the entire row.

Removing Rows

Example: Remove rows with a NULL value in the "Date" column:

```
import pandas as pd
df = pd.read_csv('data.csv')
df['Date'] = pd.to_datetime(df['Date'])
df.dropna(subset=['Date'], inplace = True)
print(df.to_string())
```

Fixing wrong data

- "Wrong data" does not have to be "empty cells" or "wrong format“
- it can just be wrong, if someone registered “450" instead of “60".

Replacing Values: Set "Duration" = 45 in row 7.

```
import pandas as pd
df = pd.read_csv('data.csv')
df.loc[7,'Duration'] = 45
print(df.to_string())
```

Note: For small data sets, replace the wrong data one by one, but not for big data sets.

Fixing wrong data (Contd.)

- To replace wrong data for larger data sets, create some rules.
- Set some boundaries for legal values, and replace any values that are outside of the boundaries.

Example: Loop through all values in the "Duration" column.

If the value is higher than 120, set it to 120.

```
import pandas as pd
df = pd.read_csv('data.csv')
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.loc[x, "Duration"] = 120
print(df.to_string())
```

Removing Rows

- Another way of handling wrong data is to remove the rows that contains wrong data.

Example: Delete rows where "Duration" is higher than 120.

```
import pandas as pd

df = pd.read_csv('data.csv')

for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.drop(x, inplace = True)

#remember to include the 'inplace = True' argument to make
#the changes in the original DataFrame object instead of
#returning a copy

print(df.to_string())
```

Removing Duplicates

- Duplicate rows are rows that have been registered more than one time.
- To discover duplicates, use the duplicated() method.
- The duplicated() method returns a Boolean values for each row.

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

Output:

11 False

12 True

Removing Duplicates (Contd.)

- To remove duplicates, use the drop_duplicates() method.

Example: Remove all duplicates.

```
import pandas as pd
df = pd.read_csv('data.csv')
df.drop_duplicates(inplace = True)
print(df.to_string())
# row 12 has been removed from the result
```

Data Visualization with Matplotlib

Prof. Dr. Matangini Chattopadhyay
School of Education Technology
Jadavpur University

What is Matplotlib?

- Matplotlib is a low level graph plotting library in python that serves as a visualization utility.
- Matplotlib was created by John D. Hunter.
- Matplotlib is open source and can be used freely.
- Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Installation of Matplotlib

- Python and PIP already installed on a system, then install Matplotlib using the following command:

  C:\Users*Your Name*>pip install matplotlib

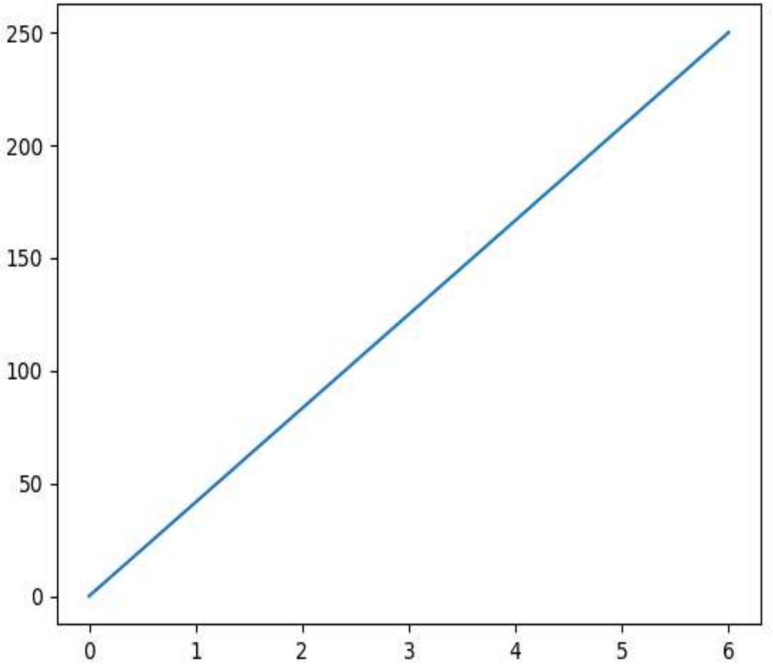
- If the above command fails, then use a python distribution that already has Matplotlib installed like, Anaconda, Spyder etc.
- Once Matplotlib is installed, import it in the applications by adding the import keyword:

```
import  matplotlib
print(matplotlib.__version__) # 2.0.0
```

Line Plot

- Most of the Matplotlib utilities lie under the pyplot submodule; imported under the plt alias

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

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.show()
```

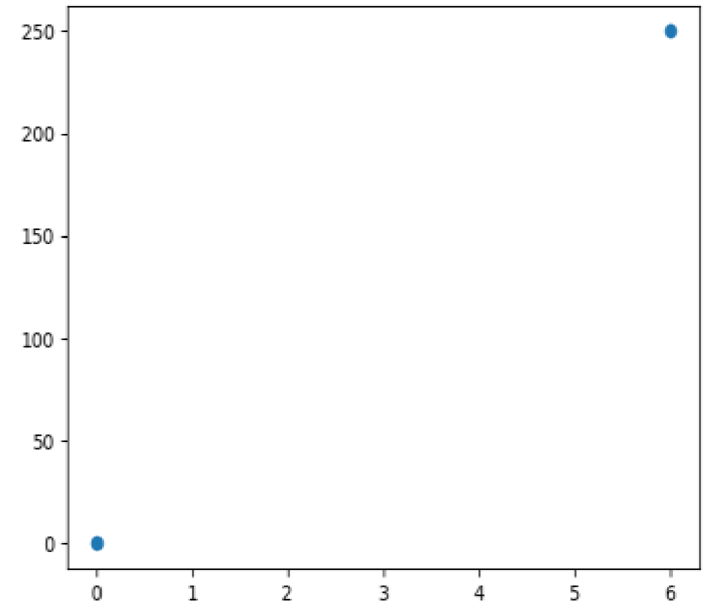


x-axis is the horizontal axis.
y-axis is the vertical axis.

Plotting Without Line

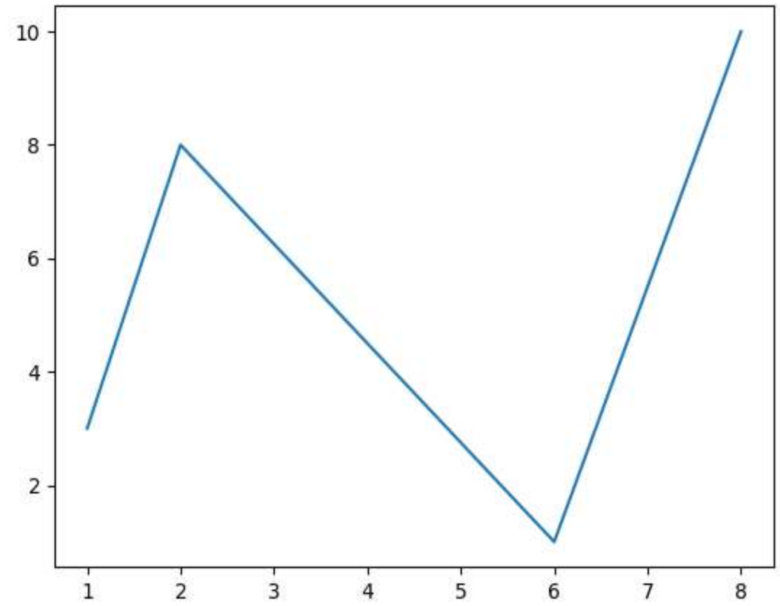
- To plot only the markers, you can use *shortcut string notation* parameter 'o', which means 'rings'.

```
import matplotlib.pyplot as plt
import numpy as np
xpoints = np.array([0, 6])
ypoints = np.array([0, 250])
plt.plot(xpoints, ypoints, ‘o’)
plt.show()
```



Line Plot with Multiple Points

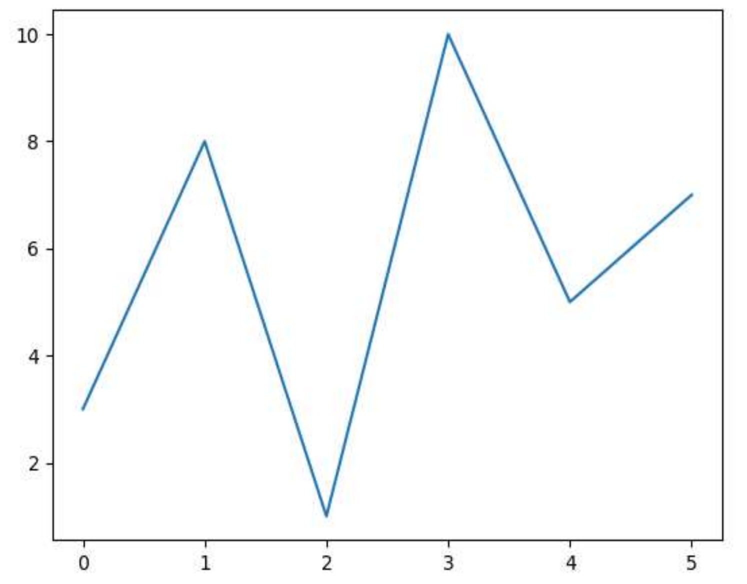
```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)
plt.show()
```



Line Plot with Default X-Points

- If points on the x-axis are not specified, they will get the default values 0, 1, 2, 3 etc., depending on the length of the y-points.

```
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10, 5, 7])
plt.plot(ypoints)
plt.show()
```



Markers

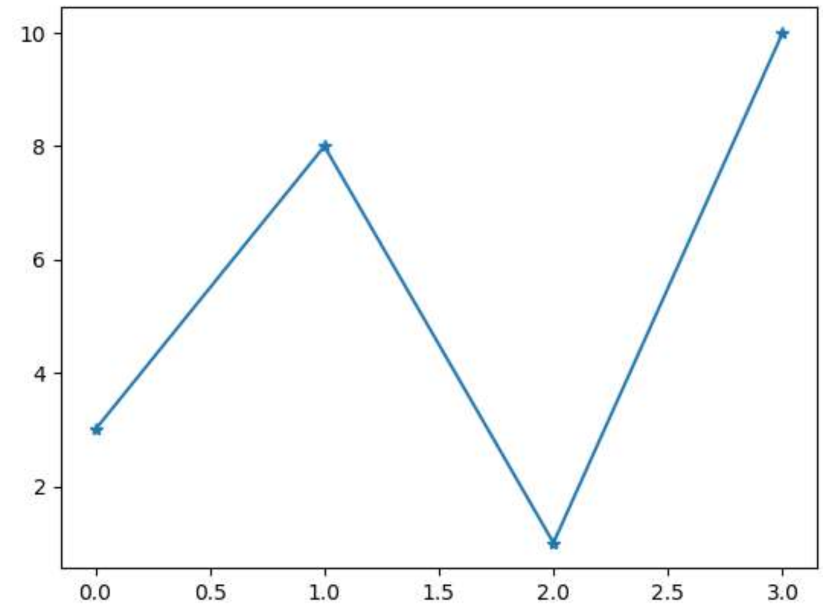
Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond

More Markers: https://www.w3schools.com/python/matplotlib_markers.asp

Markers: Example

Mark each point with a star (‘*’).

plt.plot(ypoints, marker = '*')

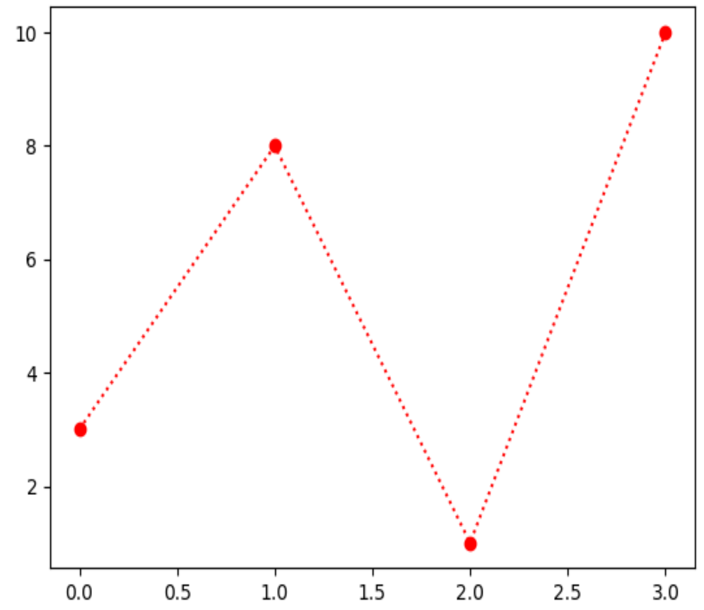


Markers: Format Strings (fmt)

- Use the *shortcut string notation* parameter (fmt) to specify the marker.

marker|line|color

```
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10])
plt.plot(ypoints, 'o:r')
plt.show()
```



Line Style

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

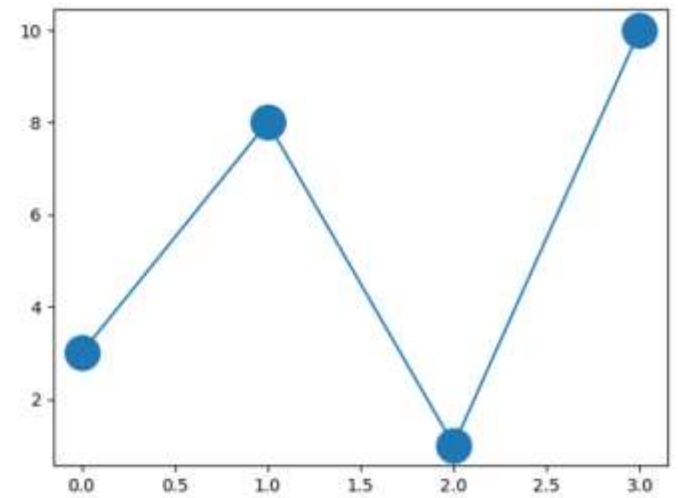
Color Reference

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

Marker Size

- To set the size of the markers, use markersize or the shorter version, ms as an argument.

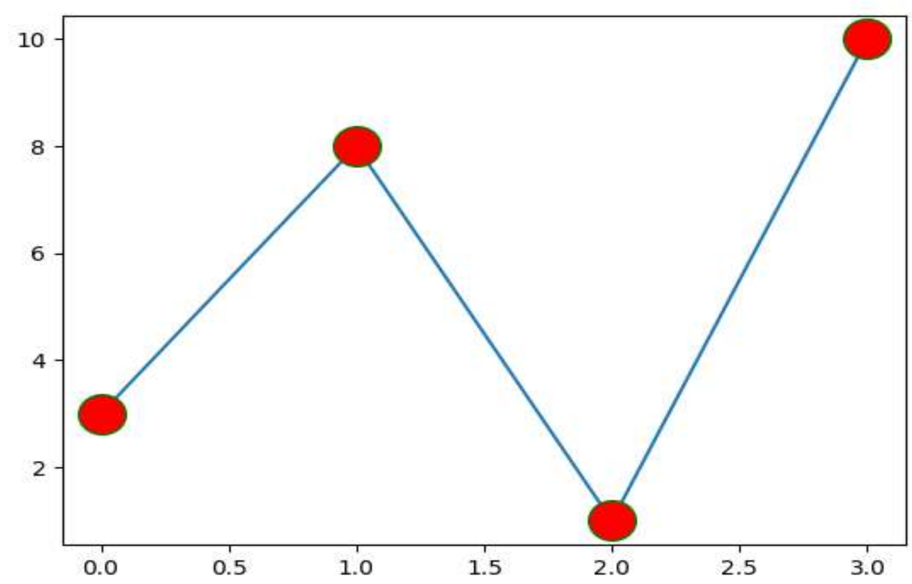
```
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10])
plt.plot(ypoints, marker = 'o', ms = 20)
plt.show()
```



Marker Color

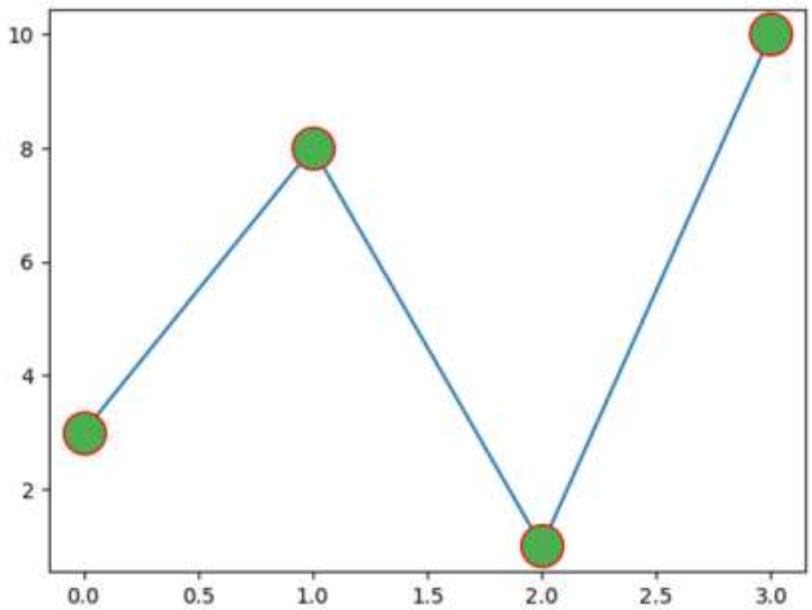
- Use the argument markeredgecolor or the shorter mec to set the color of the *edge* of the markers.
- Use markerfacecolor or the shorter mfc to set the color inside the edge of the markers.

```
plt.plot(ypoints, marker = 'o', ms = 20, mec = ‘g', mfc = 'r')
plt.show()
```



Marker Color (Contd.)

```
plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r', mfc = ‘#4CAF50')
plt.show()
```

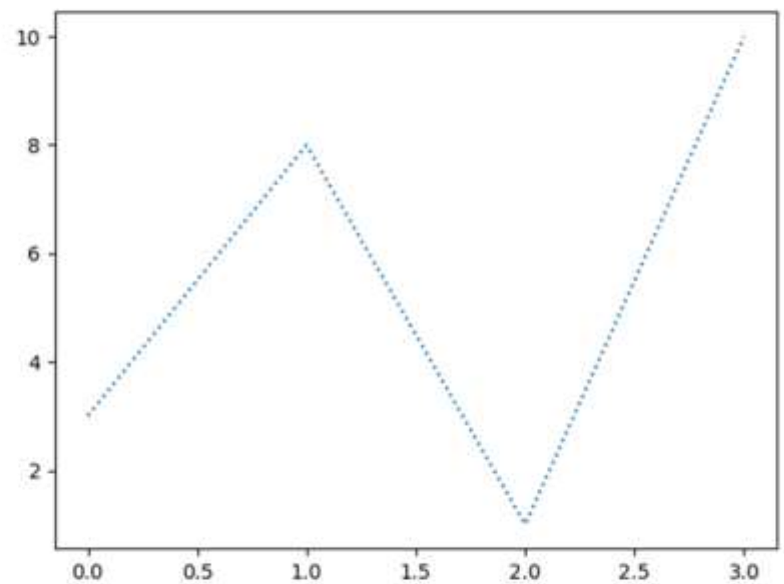


Color Names Supported: https://www.w3schools.com/colors/colors_names.asp

Line Style

- To change the style of the plotted line, argument linestyle, or ls is used.

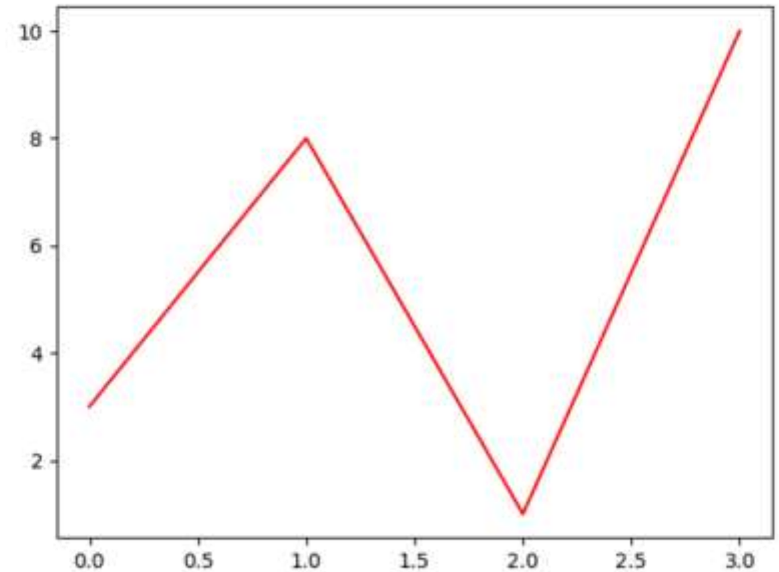
```
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10])
plt.plot(ypoints, linestyle = 'dotted')


# plt.plot(ypoints, ls = 'dotted')
# plt.plot(ypoints, ls = ':')
plt.show()
```



Line Color

- Use color or c to set the color of the line

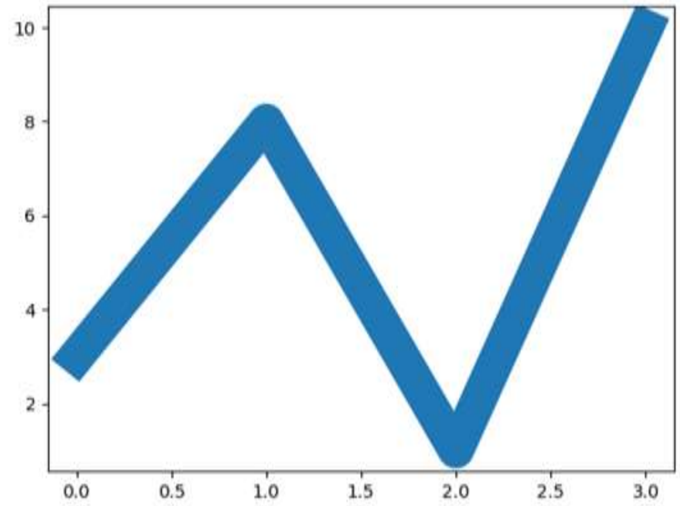
```
import matplotlib.pyplot as plt
import numpy as np
ypoints = np.array([3, 8, 1, 10])
plt.plot(ypoints, color = 'r')
#plt.plot(ypoints, c = 'r')
# plt.plot(ypoints, c = '#4CAF50') for green line
plt.show()
```



Line Width

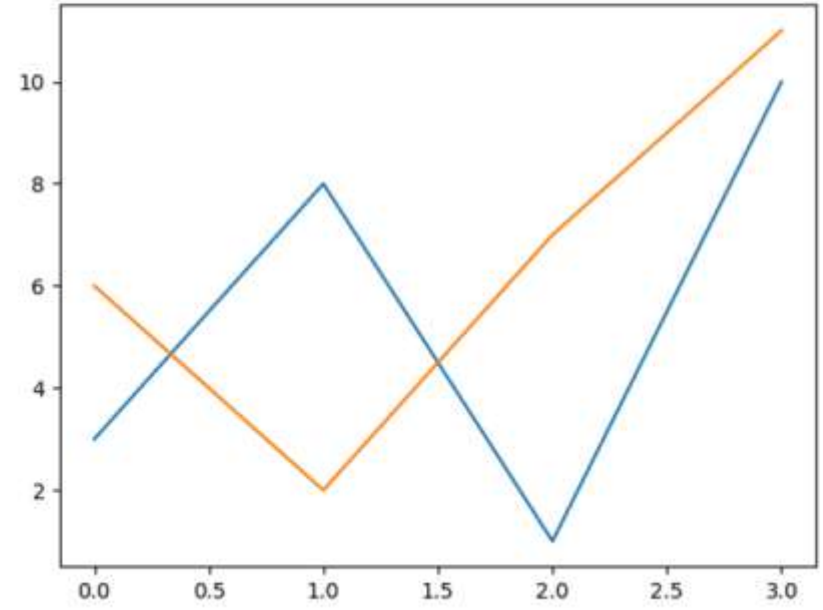
- Use linewidth or lw to change the width of a line. The value is a floating point number in points.

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

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linewidth = '20.5')
plt.show()
```



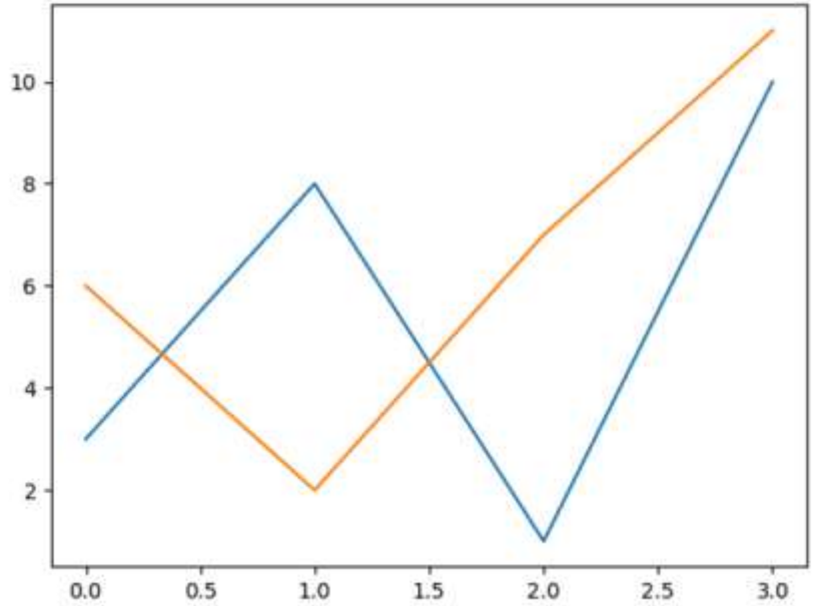
Multiple Lines

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

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)

plt.show()
```



Multiple Lines (Contd.)

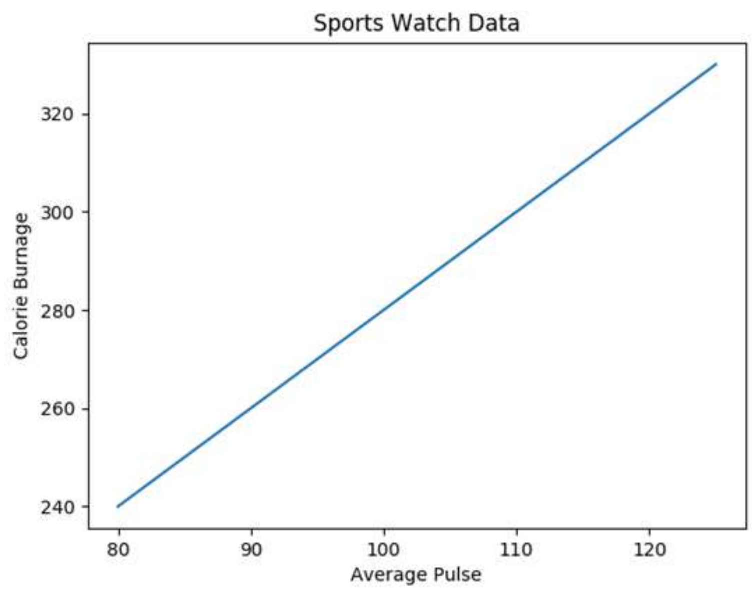
```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.array([0, 1, 2, 3])
y1 = np.array([3, 8, 1, 10])
x2 = np.array([0, 1, 2, 3])
y2 = np.array([6, 2, 7, 11])

plt.plot(x1, y1, x2, y2)

plt.show()
```



Labels and Title for a Plot

- Use xlabel() and ylabel() functions to set a label for the x- and y-axis.
- Use title() function to set a title for the plot.

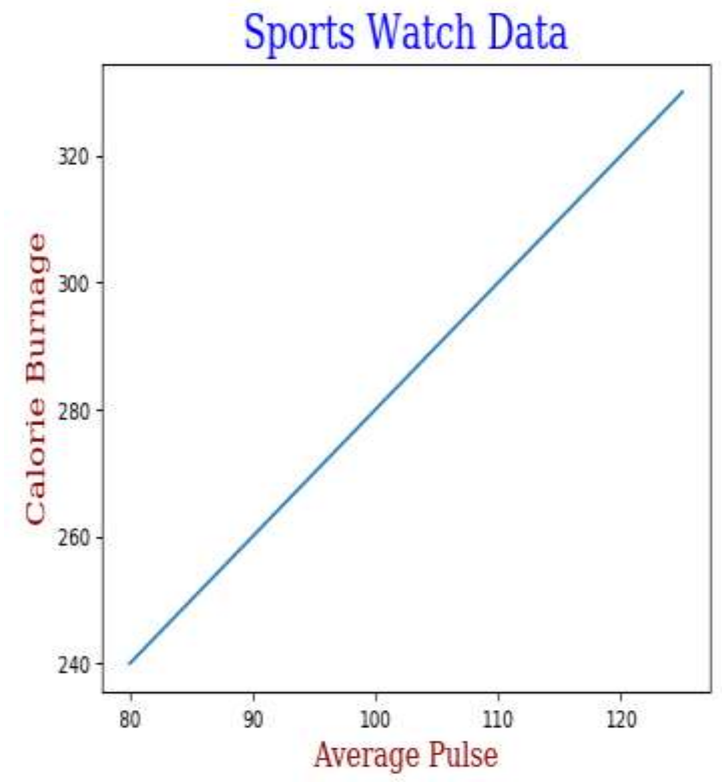
```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)
plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.show()
```



Set Font Properties for Title and Labels

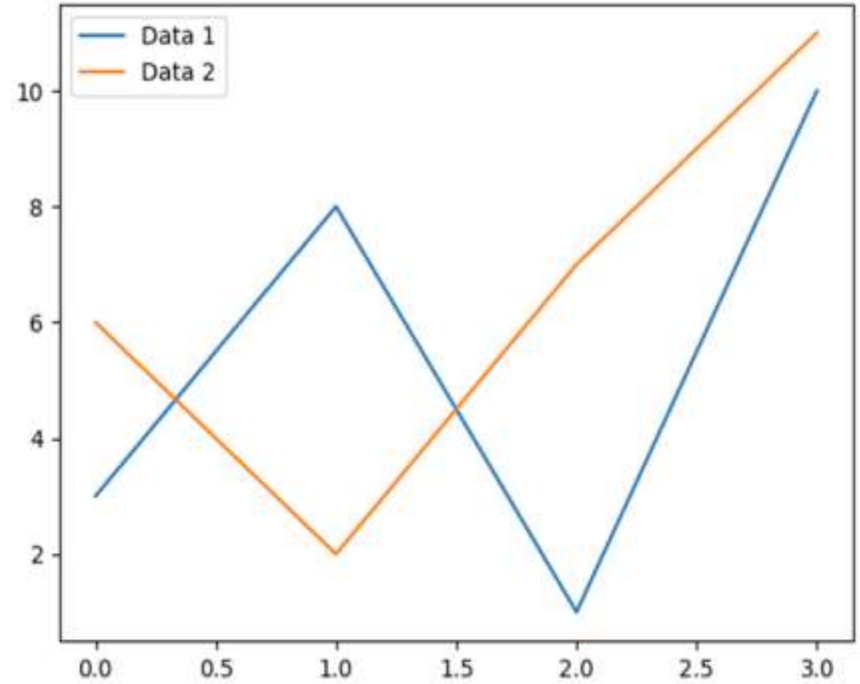
- Use the fontdict parameter in xlabel(), ylabel(), and title() to set font properties for the title and labels.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])
font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}
plt.title("Sports Watch Data", fontdict = font1)
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)

plt.plot(x, y)
plt.show()


#Position the title
plt.title("Sports Watch Data", loc = 'left')
```



Legend

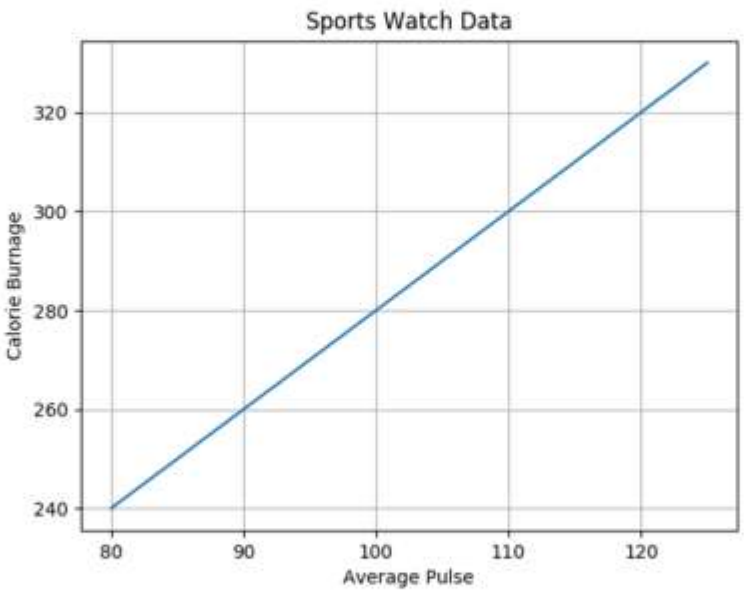
```
import matplotlib.pyplot as plt
import numpy as np
y1 = np.array([3,8,1,10])
y2 = np.array([6,2,7,11])
plt.plot(y1)
plt.plot(y2)
plt.legend(["Data 1", "Data 2"])
plt.show()
```



Grid Lines to a Plot

- Use the grid() function to add grid lines to the plot.

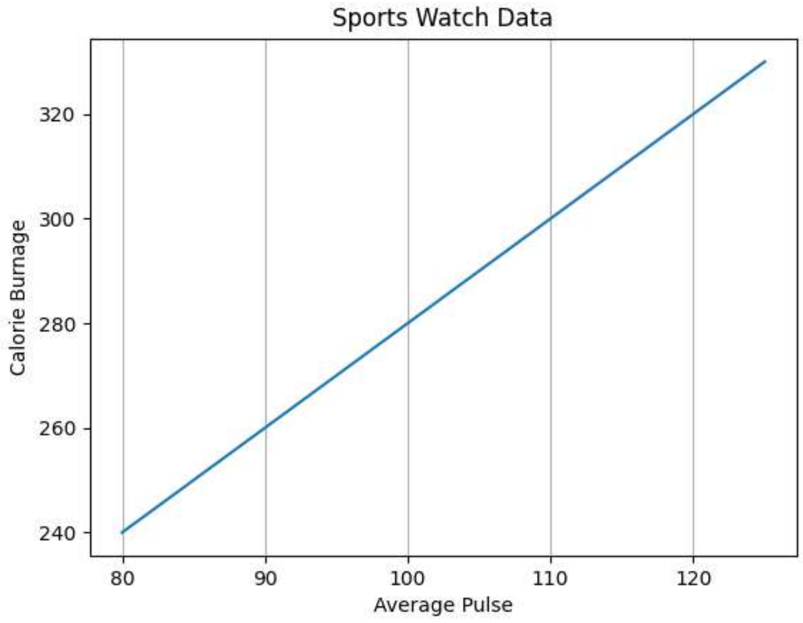
```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])
plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid()

plt.show()
```



Customizing Grids

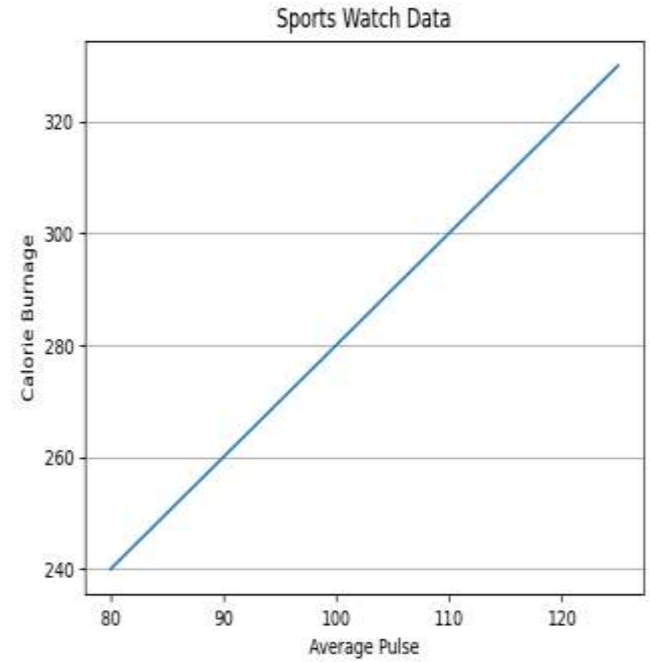
- Use the axis parameter in the grid() function to specify which grid lines to display.
- Legal values are: 'x', 'y', and 'both'. Default value is 'both'.

```
import numpy as np
import matplotlib.pyplot as plt
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])
plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")
plt.plot(x, y)
plt.grid(axis = 'x')
plt.show()
```



grid lines for the y-axis

```
plt.grid(axis = 'y')
```



- Set the line properties of the grid

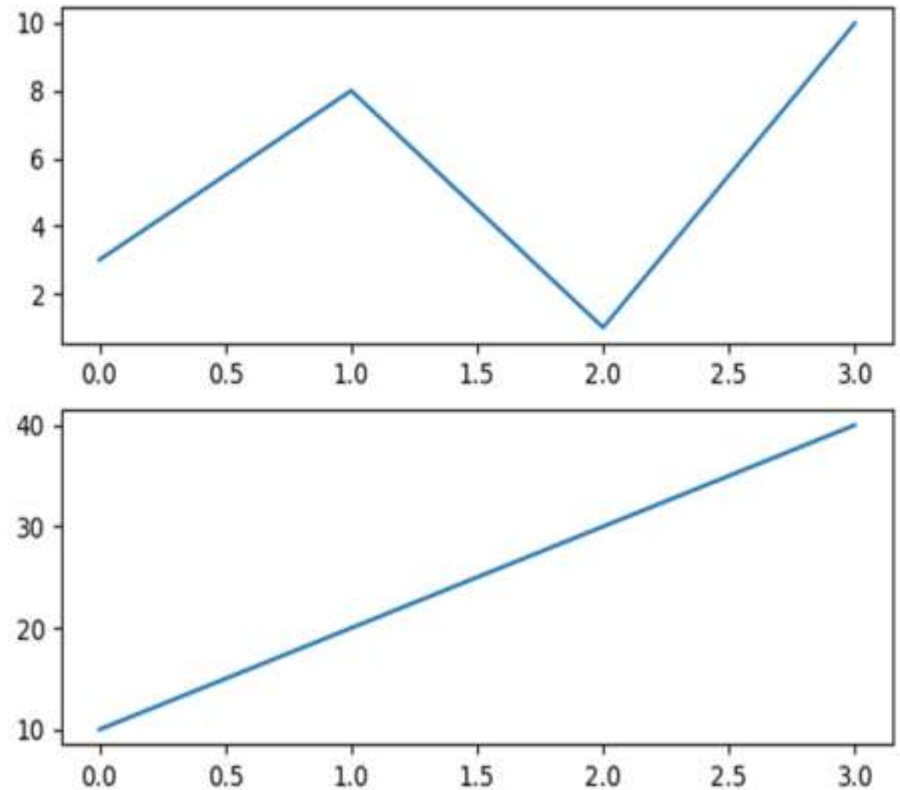
```
plt.grid(color = 'green', linestyle = '--', linewidth = 0.5)
```

Subplot

- You can draw multiple plots in one figure with the subplot() function.
- The subplot() function takes three arguments that describes the layout of the figure.
- The layout is organized in rows and columns, which are represented by the first and second argument.
- The third argument represents the index of the current plot.

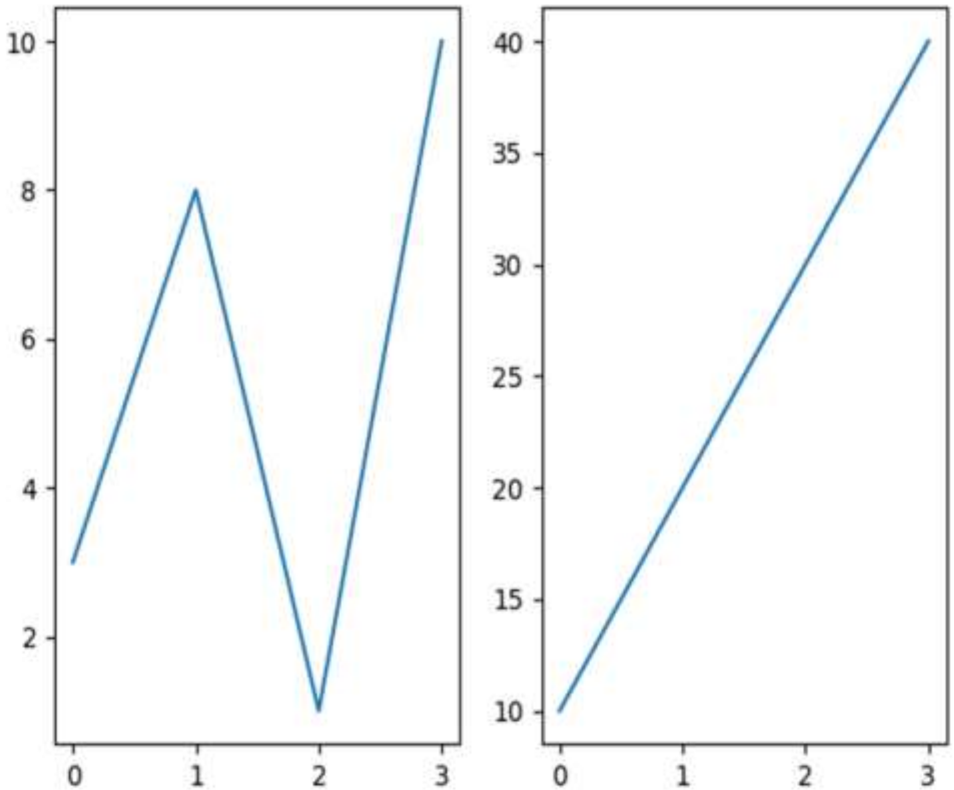
Subplot: Vertical Split

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

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 1, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 1, 2)
plt.plot(x,y)

plt.show()
```



Subplot: Horizontal Split

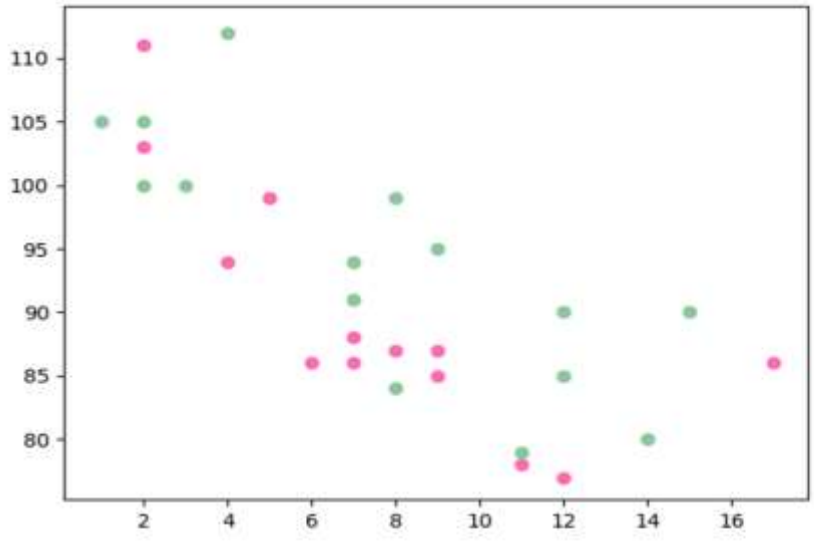
```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)

plt.show()
```



Creating Scatter Plots

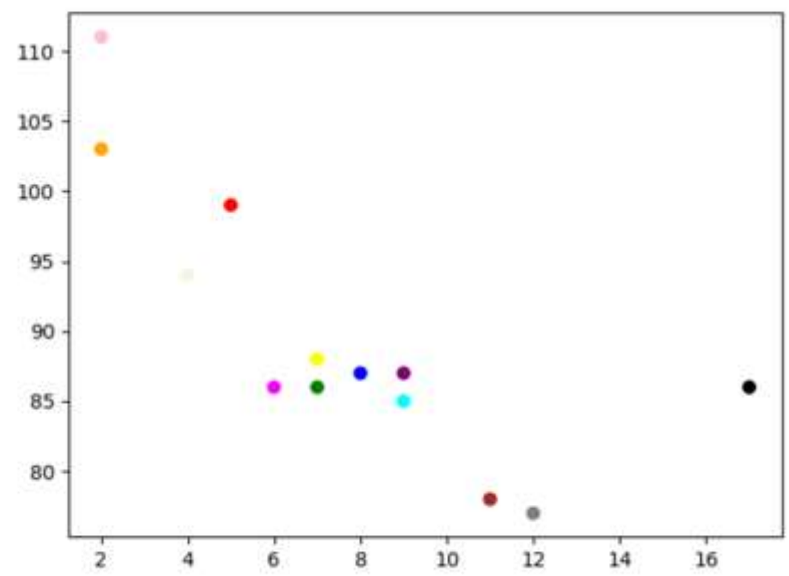
- The scatter() function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y, color = 'hotpink')
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y, color = '#88c999')

plt.show()
```



Color Each Dot in Scatter Plot

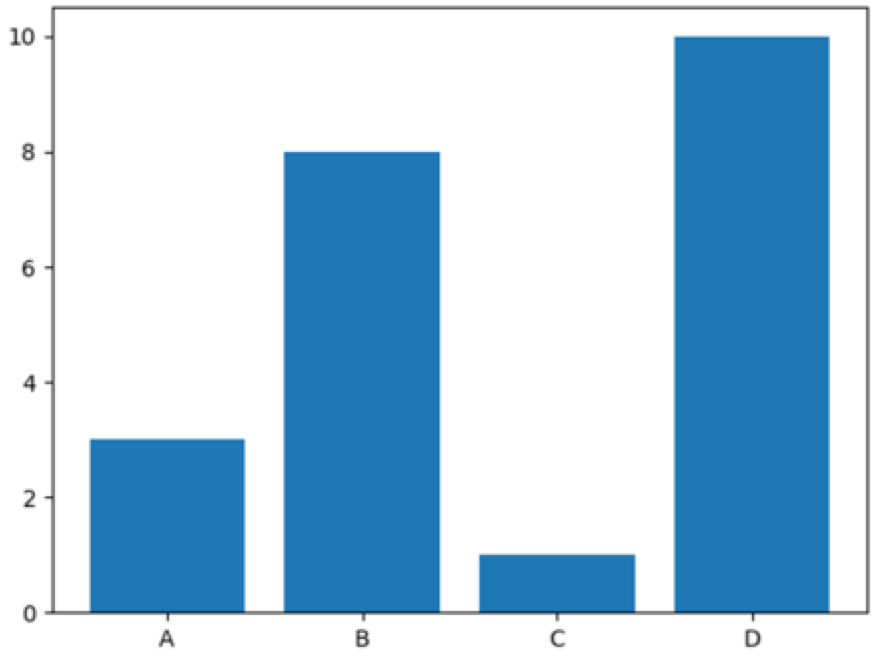
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
colors =
np.array(["red","green","blue","yellow","pink","black","orange","purple
","beige","brown","gray","cyan","magenta"])
plt.scatter(x, y, c=colors)
plt.show()
```



Bar Plot

- Use the bar() function to draw bar graphs.

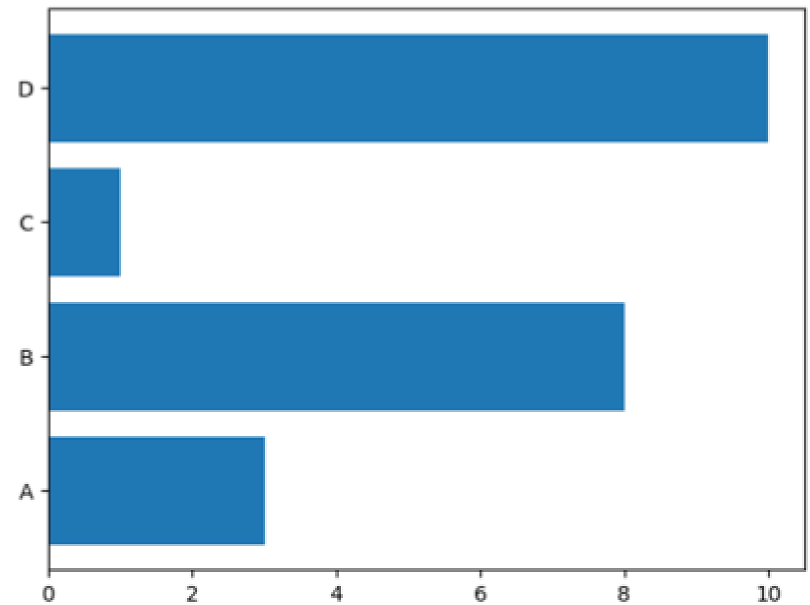
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x,y)
plt.show()
```



Horizontal Bars

- Use the barh() function, when bars are displayed horizontally instead of vertically.

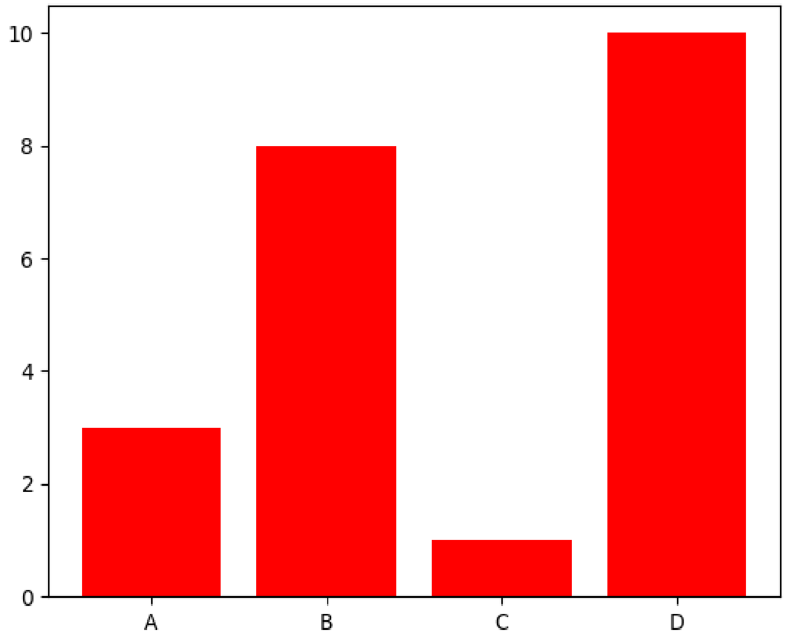
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.barh(x, y)
plt.show()
```



Bar Color

- The bar() and barh() use the argument color to set the color of the bars.

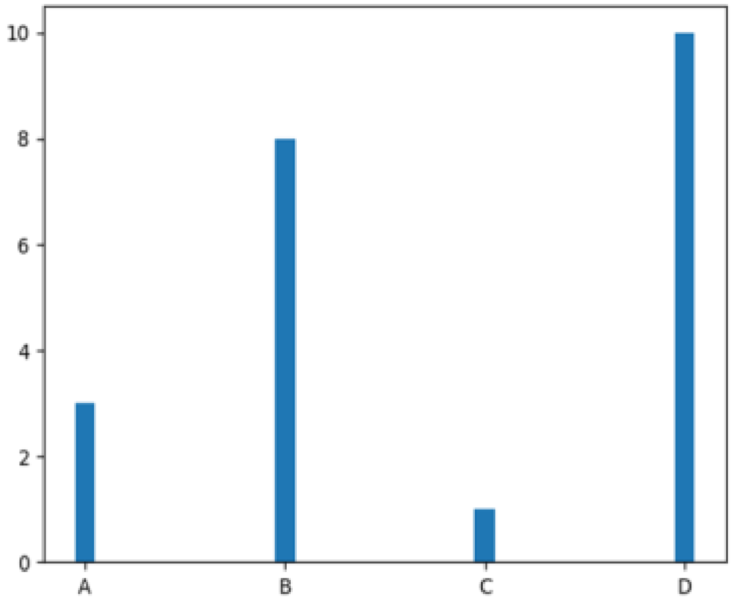
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x, y, color = “red")
plt.show()
```



Bar Width

- The bar() uses the argument width to set the width of the bars.

```
import matplotlib.pyplot as plt
import numpy as np
x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])
plt.bar(x, y, width = 0.1)
plt.show()
```

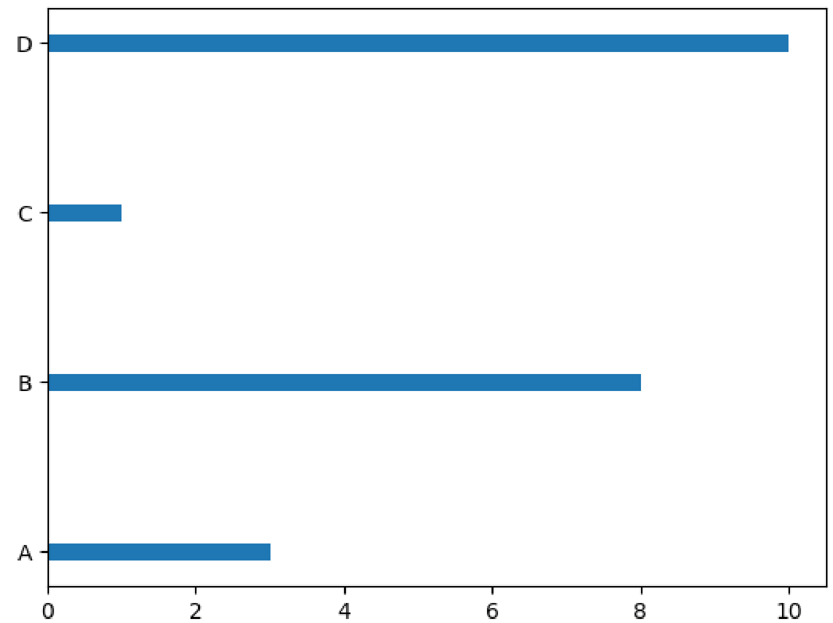


- The default width value is 0.8.
- For horizontal bars, use height instead of width.

Bar Height

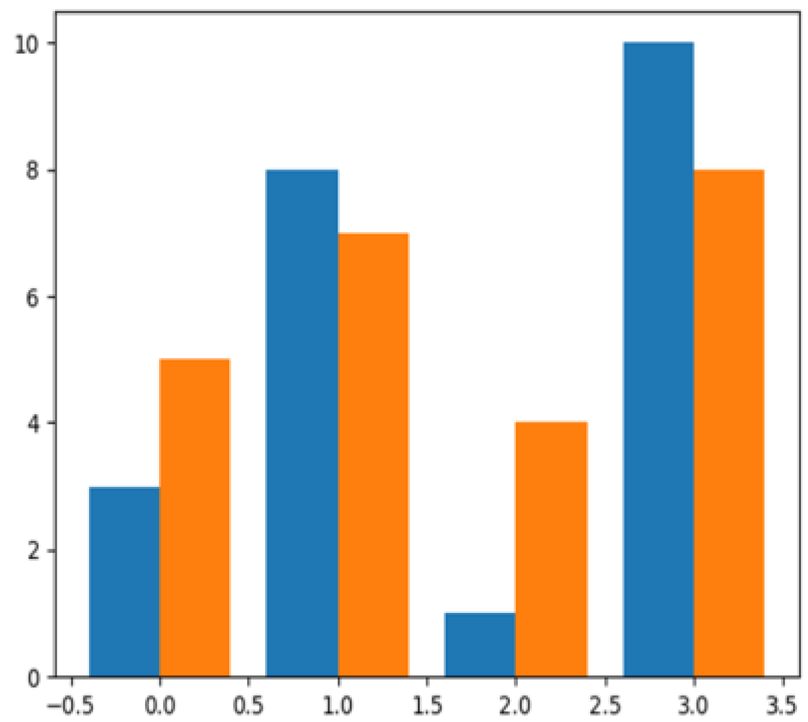
- The barh() takes the argument height to set the height of the bars.

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

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.barh(x, y, height= 0.1)
plt.show()
```



Multiple Bar Plot

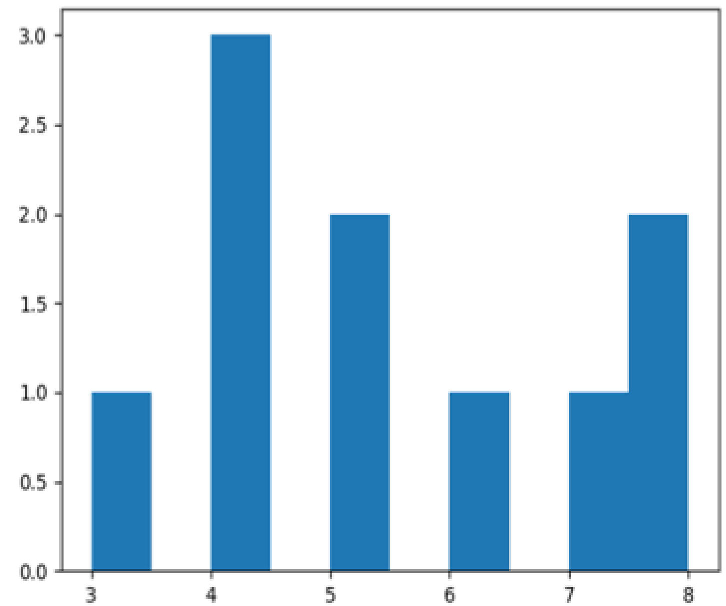
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array(["A", "B", "C", "D"])
y1 = np.array([3, 8, 1, 10])
y2= np.array([5, 7, 4, 8])

xl=np.arange(len(x))
print(xl)
plt.bar(xl-0.2, y1, width=0.4)
plt.bar(xl+0.2, y2, width=0.4)

plt.show()
```



Histogram

- A histogram is a graph showing *frequency* distributions.
- It is a graph showing the number of observations within each given interval.
- Use the hist() function to create histograms.
- hist() function use an array of numbers to create a histogram, the array is sent into the function as an argument.

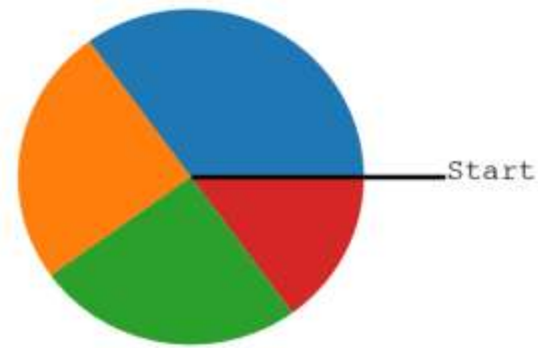
```
import matplotlib.pyplot as plt
import numpy as np
x = np.array([8, 4, 5, 4, 6, 7, 3, 4, 8, 5])
plt.hist(x)
plt.show()
```



Pie Charts

- Use the pie() function to draw pie charts.

```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
plt.pie(y)
plt.show()
```



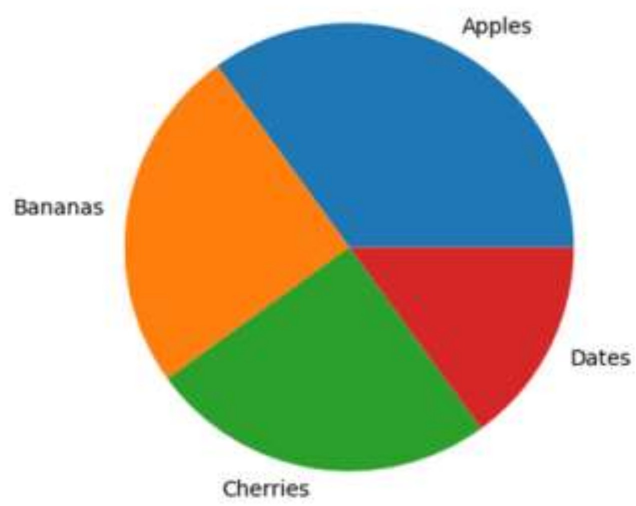
- Pie chart draws one piece (called a wedge) for each value in the array.
- By default, the plotting of the first wedge starts from the x-axis and moves *counterclockwise.*

Note: The size of each wedge is determined by using the formula:

The value divided by the sum of all values: x/sum(x)

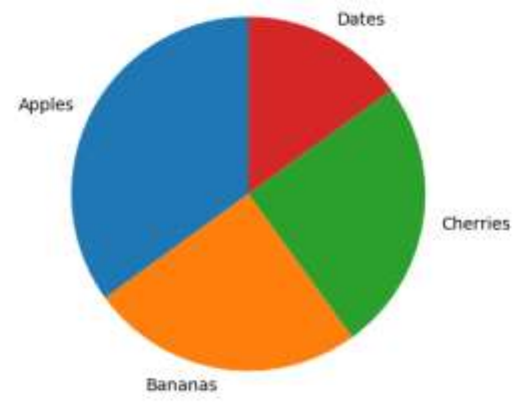
Pie Charts with Labels

```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(y, labels = mylabels)
plt.show()
```



Pie Charts with Start Angle

- The default start angle is at the x-axis, but it can be changed by specifying a startangle parameter.
- The startangle parameter is defined with an angle in degrees, default angle is 0.

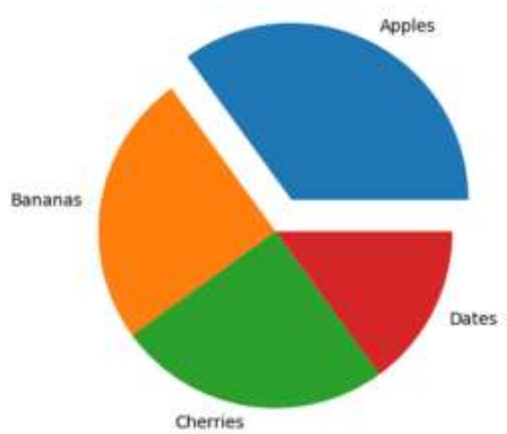



```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(y, labels = mylabels, startangle = 90)
plt.show()
```

Explode

- In order to make one of the wedges to stand out, use explode parameter.
- The explode parameter, if specified, and not None, must be an array with one value for each wedge.
- Each value represents how far from the center each wedge is displayed.

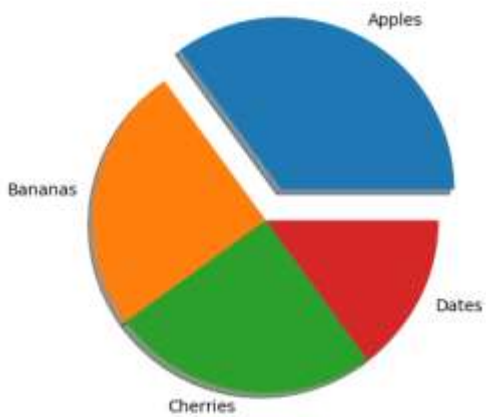
```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]
plt.pie(y, labels = mylabels,
            explode = myexplode)
plt.show()
```



Shadow

- Add a shadow to the pie chart by setting the shadow parameter to True.

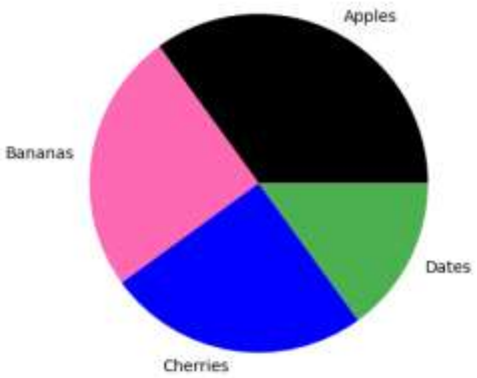
```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]
plt.pie(y, labels = mylabels,
    explode = myexplode, shadow = True)
plt.show()
```



Colors

- Set the color of each wedge with the colors parameter.
- The colors parameter, if specified, must be an array with one value for each wedge.

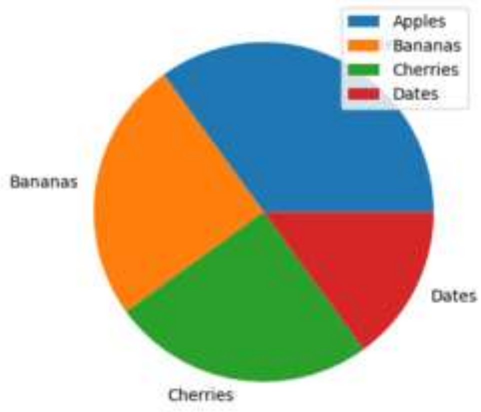
```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas",
                "Cherries", "Dates"]
mycolors = ["black", "hotpink", "b",
               "#4CAF50"]
plt.pie(y, labels = mylabels, colors = mycolors)
plt.show()
```



Pie Charts with Legend

- To add a list of explanation for each wedge, use the legend() function.

```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(y, labels = mylabels)
plt.legend()
plt.show()
```



Legend with Header

- To add a header to the legend, add the title parameter to the legend() function.

```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(y, labels = mylabels)
plt.legend(title = "Four Fruits:")
plt.show()
```