8. Python for Data Analysis

8.1. NumPy:

8.2. Introduction to NumPy

What is NumPy?

NumPy stands for **numeric python** which is a python package for the computation and processing of the **multidimensional** and **single dimensional array elements.**

Travis Oliphant created **NumPy package** in 2005 by injecting the features of the ancestor **module Numeric** into another **module Numarray.**

It is an extension module of Python which is mostly written in C. It provides various functions which are capable of performing the numeric computations with **a high speed**.

NumPy provides various **powerful data structures**, implementing **multi-dimensional arrays** and **matrices.** These data structures are used for the optimal computations regarding arrays and matrices.

The need of NumPy

With the revolution of data science, data analysis libraries like **NumPy, SciPy, Pandas,** etc. have seen a lot of growth. With a much easier syntax than other programming languages, python is the first choice language for the **data scientist.**

NumPy provides **a convenient** and **efficient** way to handle the **vast amount of data.** NumPy is also very convenient with **Matrix multiplication** and **data reshaping.** NumPy is fast which makes it reasonable to work with a large set of data.

There are the following advantages of using NumPy for data analysis.

1. NumPy performs array-oriented computing.
2. It efficiently implements the multidimensional arrays.
3. It performs scientific computations.
4. It is capable of performing **Fourier Transform** and **reshaping the data** stored in multidimensional arrays.
5. NumPy provides the **in-built functions** for **linear algebra** and **random number** generation.

Nowadays, **NumPy** in combination with **SciPy** and **Matplotlib** is used as the replacement to **MATLAB** as Python is more complete and easier programming language than **MATLAB.**

NumPy Environment Setup

NumPy doesn't come bundled with Python. We have to install it using **the python pip** installer. Execute the following command.

```
$ pip install numpy
```

It is best practice to install NumPy with the full SciPy stack. The binary distribution of the SciPy stack is specific to the operating systems.

Windows

On the Windows operating system, The SciPy stack is provided by the Anaconda which is a free distribution of the Python SciPy package.

It can be downloaded from the official website: https://www.anaconda.com/. It is also available for Linux and Mac.

The CanoPy also comes with the full SciPy stack which is available as free as well as commercial license. We can download it by visiting the link: https://www.enthought.com/products/canopy/

The Python (x, y) is also available free with the full SciPy distribution. Download it by visiting the link: https://python-xy.github.io/

Linux

In Linux, the different package managers are used to install the SciPy stack. The package managers are specific to the different distributions of Linux. Let's look at each one of them.

Ubuntu

Execute the following command on the terminal.

```
$ sudo apt-get install python-numpy
$ python-scipy python-matplotlibipythonipythonnotebook python-pandas
$ python-sympy python-nose
```

Redhat

On Redhat, the following commands are executed to install the Python SciPy package stack.

```
$ sudo yum install numpyscipy python-matplotlibipython
$ python-pandas sympy python-nose atlas-devel
```

To verify the installation, open the Python prompt by executing python command on the terminal (cmd in the case of windows) and try to import the module NumPy. If it doesn't give the error, then it is installed successfully.

8.3. Creating arrays, Using arrays and Scalars

8.4. Indexing Arrays, Array Transposition

Creating Numpy array from existing data

NumPy provides us the way to create an array by using the existing data.

numpy.array

This routine is used to create an array by using the existing data in the form of lists, or tuples. This routine is useful in the scenario where we need to convert a python sequence into the numpy array object.

The syntax to use the **array()** routine is given below.

numpy.array(sequence, dtype = None, order = None)

It accepts the following parameters.

1. **sequence:** It is the python sequence which is to be converted into the python array.
2. **dtype:** It is the data type of each item of the array.
3. **order:** It can be set to **C** or **F**. The default is **C**.

Example: creating numpy array using the list

```
import numpy as np
l = [1,2,3,4,5,6,7]
a = np.array(l);
print(type(a))
print(a)
```

Output:

```
<class 'numpy.ndarray'>
[1 2 3 4 5 6 7]
```

Example: creating a numpy array using Tuple

```
import numpy as np
l = (1, 2, 3, 4, 5, 6, 7)
a = np.array(l);
print(type(a))
print(a)
```

Output:

```
<class 'numpy.ndarray'>

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

Example: creating a numpy array using more than one list

```
import numpy as np
l = [[1,2,3,4,5,6,7],[8,9]]
a = np.array(l);
print(type(a))
print(a)
```

Output:

```
<class 'numpy.ndarray'>
[[1, 2, 3, 4, 5, 6, 7]
 [8, 9]]
```

numpy.frombuffer

This function is used to create an array by using the specified buffer. The syntax to use this buffer is given below.

numpy.frombuffer(buffer, dtype = float, count = -1, offset = 0)

It accepts the following parameters.

- **buffer:** It represents an object that exposes a buffer interface.
- **dtype:** It represents the data type of the returned data type array. The default value is 0.
- **count:** It represents the length of the returned ndarray. The default value is -1.
- **offset:** It represents the starting position to read from. The default value is 0.

Example: creating a numpy array using frombuffer

```
import numpy as np
l = b'hello world'
print(type(l))
a = np.frombuffer(l, dtype = "S1")
print(a)
print(type(a))
```

Output:

```
<class 'bytes'>
[b'h' b'e' b'l' b'l' b'o' b' ' b'w' b'o' b'r' b'l' b'd']
<class 'numpy.ndarray'>
```

numpy.fromiter

This routine is used to create a ndarray by using **an iterable object**. It returns a one-dimensional ndarray object.

The syntax is given below.

numpy.fromiter(iterable, dtype, count = - 1)

It accepts the following parameters.

1. **Iterable:** It represents an iterable object.
2. **dtype:** It represents the data type of the resultant array items.
3. **count:** It represents the number of items to read from the buffer in the array.

Example creating a numpy array using fromiter

```
import numpy as np
list = [0, 2, 4, 6]
it = iter(list)
x = np.fromiter(it, dtype = float)
print(x)
print(type(x))
```

Output:

```
[0. 2. 4. 6.]
<class 'numpy.ndarray'>
```

NumPy Ndarray

Ndarray is the **n-dimensional array object** defined in the **numpy** which stores the collection of the **similar type of elements**. In other words, we can define a ndarray as the collection of the data type (dtype) objects.

The ndarray object can be accessed by using the **0 based indexing**. Each element of the Array object contains the same size in the memory.

Creating a ndarray object

The ndarray object can be created by using the array routine of the numpy module. For this purpose, we need to import the numpy.

```
>>> import numpy as np
>>> a = np.array
>>> print(a)
```

We can also pass a collection object into the array routine to create the equivalent n-dimensional array. The syntax is given below.

```
>>> numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)
```

The parameters are described in the following table.

SN	Parameter	Description
1	**object**	It represents the collection object. It can be a list, tuple, dictionary, set, etc.
2	**dtype**	We can change the data type of the array elements by changing this option to the specified type. **The default is none.**
3	**copy**	It is optional. By default, it is true which means the object is copied.
4	**order**	It is optional. By default, it is C – style. There can be 4 possible values assigned to this option. These are 'K', 'A', 'C', 'F'. Specify the memory layout of the array. C – row major (C – language style) F – column major (F – language style)
5	**subok**	The returned array will be **base class array by default**. We can change this to make the subclasses passes through by setting this **option to true.**

6	**ndmin**	It represents the minimum dimensions of the resultant array.

To create an array using the list, use the following syntax.

```
>>> a = numpy.array([1, 2, 3])
```

Example:

```
>>> import numpy as np
>>> a = np.array([1,2,3,4,5], dtype = float)
>>> print(a)
```

Output:

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

To create a multi-dimensional array object, use the following syntax.

```
>>> a = numpy.array([[1, 2, 3], [4, 5, 6]])
```

Example:

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

Output:

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

To change the data type of the array elements, mention the name of the data type along with the collection.

```
>>> a = numpy.array([1, 3, 5, 7], complex)
```

Example:

```
>>> import numpy as np
>>> a = np.array([1,2,3,4,5], dtype = complex)
>>> print(a)
```

Output:

```
[1.+0.j 2.+0.j 3.+0.j 4.+0.j 5.+0.j]
```

Finding the dimensions of the Array

The **ndim** function can be used to find the dimensions of the array.

```
>>> import numpy as np
>>> a = np.array([[1, 2, 3, 4], [4, 5, 6, 7], [9, 10, 11, 23]])
>>> print(a.ndim)
>>> print(a)
```

Finding the size of each array element

The **itemsize** function is used to get the size of each array item. It returns the number of bytes taken by each array element.

Consider the following example.

Example:

```
#finding the size of each item in the array
import numpy as np
a = np.array([[1,2,3]])
print("Each item contains", a.itemsize, "bytes")
```

Output:

```
Each item contains 8 bytes.
```

Finding the data type of each array item

To check the data type of each array item, the dtype function is used. Consider the following example to check the data type of the array items.

Example:

```
#finding the data type of each array item
import numpy as np
a = np.array([[1,2,3]])
print("Each item is of the type", a.dtype)
```

Output:

```
Each item is of the type int64
```

Finding the shape and size of the array

To get the shape and size of the array, the size and shape function associated with the numpy array is used.

Consider the following example.

Example:

```
import numpy as np
a = np.array([[1,2,3,4,5,6,7]])
print("Array Size:",a.size)
print("Shape:",a.shape)
```

Output:

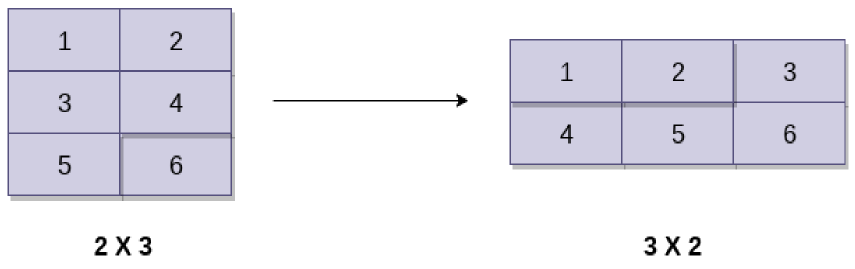
```
Array Size: 7
Shape: (1, 7)
```

Reshaping the array objects

By the shape of the array, we mean the number of rows and columns of a multi-dimensional array. However, the numpy module provides us the way to reshape the array by changing the number of rows and columns of the multi-dimensional array.

The reshape() function associated with the ndarray object is used to reshape the array. It accepts the two parameters indicating the row and columns of the new shape of the array.

Let's reshape the array given in the following image.



Example:

```
import numpy as np
a = np.array([[1,2],[3,4],[5,6]])
print("printing the original array..")
print(a)
a=a.reshape(2,3)
```

```
print("printing the reshaped array..")
print(a)
```

Output:

```
printing the original array..
[[1 2]
 [3 4]
 [5 6]]
printing the reshaped array..
[[1 2 3]
 [4 5 6]]
```

Slicing in the Array

Slicing in the NumPy array is the way to extract a range of elements from an array. Slicing in the array is performed in the same way as it is performed in the python list.

Consider the following example to print a particular element of the array.

Example:

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

Output:

```
2
5
```

The above program prints the 2nd element from the 0th index and 0th element from the 2nd index of the array.

Linspace

The linspace() function returns the evenly spaced values over the given interval. The following example returns the 10 evenly separated values over the given interval 5-15

Example:

```
import numpy as np
a = np.linspace(5,15,10)  #prints 10 values which are evenly spaced over the given interval 5-15
```

```
print(a)
```

Output:

```
[ 5.          6.11111111  7.22222222  8.33333333  9.44444444 10.55555556
 11.66666667 12.77777778 13.88888889 15.        ]
```

Finding – the maximum, minimum, and sum of the array elements

The NumPy provides the max(), min(), and sum() functions which are used to find the maximum, minimum, and sum of the array elements respectively.

Consider the following example.

Example:

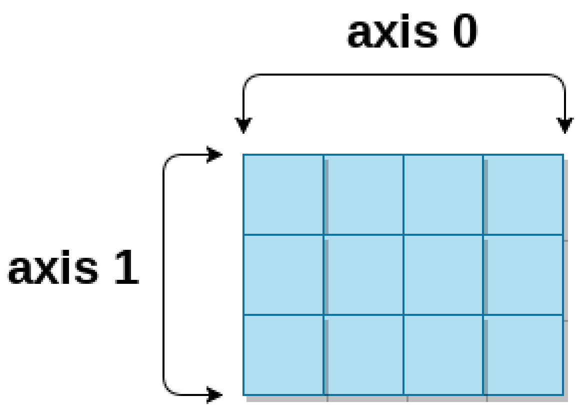
```
import numpy as np
a = np.array([1,2,3,10,15,4])
print("The array:",a)
print("The maximum element:",a.max())
print("The minimum element:",a.min())
print("The sum of the elements:",a.sum())
```

Output:

```
The array: [ 1  2  3 10 15  4]
The maximum element: 15
The minimum element: 1
The sum of the elements: 35
```

NumPy Array Axis

A NumPy multi-dimensional array is represented by the axis where **axis-0 represents the columns** and **axis-1 represents the rows.** We can mention the axis to perform row-level or column-level calculations like the addition of row or column elements.



NumPy Array

To calculate the maximum element among each column, the minimum element among each row, and the addition of all the row elements, consider the following example.

Example:

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
print("The array:",a)
print("The maximum elements of columns:",a.max(axis = 0))
print("The minimum element of rows",a.min(axis = 1))
print("The sum of all rows",a.sum(axis = 1))
```

Output:

```
The array: [[1  2  30]
            [10  15  4]]
The maximum elements of columns: [10 15 30]
The minimum element of rows [1 4]
The sum of all rows [33 29]
```

Finding square root and standard deviation

The **sqrt()** and **std()** functions associated with the numpy array are used to find the **square root** and **standard deviation** of the array elements respectively.

Standard deviation means how much each element of the array varies from the mean value of the numpy array.

Consider the following example.

Example:

```
import numpy as np
a = np.array([[1,2,30], [10,15,4]])
print(np.sqrt(a))
print(np.std(a))
```

Output:

```
[[1.         1.41421356 5.47722558]
 [3.16227766 3.87298335 2.        ]]
10.044346115546242
```

Arithmetic operations on the array

The numpy module allows us to perform the arithmetic operations on multi-dimensional arrays directly.

In the following example, the arithmetic operations are performed on the two multi-dimensional arrays a and b.

Example:

```
import numpy as np
a = np.array([[1,2,30],[10,15,4]])
b = np.array([[1,2,3],[12, 19, 29]])
print("Sum of array a and b\n",a+b)
print("Product of array a and b\n",a*b)
print("Division of array a and b\n",a/b)
```

Array Concatenation

The numpy provides us with the vertical stacking and horizontal stacking which allows us to concatenate two multi-dimensional arrays vertically or horizontally.

Consider the following example.

Example:

```
import numpy as np
a = np.array([[1,2,30], [10,15,4]])
b = np.array([[1,2,3], [12, 19, 29]])
print("Arrays vertically concatenated\n", np.vstack((a,b)));
print("Arrays horizontally concatenated\n", np.hstack((a,b)))
```

Output:

```
Arrays vertically concatenated
 [[ 1  2 30]
 [10 15  4]
 [ 1  2  3]
 [12 19 29]]
Arrays horizontally concatenated
 [[ 1  2 30  1  2  3]
 [10 15  4 12 19 29]]
```

NumPy Datatypes

The NumPy provides a higher range of numeric data types than that provided by the Python. A list of numeric data types is given in the following table.

S. No.	Data type	Description
1	**bool_**	It represents the boolean value indicating true or false. It is stored as a byte.
2	**int_**	It is the default type of integer. It is identical to long type in C that contains 64 bit or 32-bit integer.
3	**intc**	It is similar to the C integer (c int) as it represents 32 or 64-bit int.
4	**intp**	It represents the integers which are used for indexing.
5	**int8**	It is the 8-bit integer identical to a byte. The range of the value is -128 to 127.
6	**int16**	It is the 2-byte (16-bit) integer. The range is -32768 to 32767.

7	**int32**	It is the 4-byte (32-bit) integer. The range is -2147483648 to 2147483647.
8	**int64**	It is the 8-byte (64-bit) integer. The range is -9223372036854775808 to 9223372036854775807.
9	**uint8**	It is the 1-byte (8-bit) unsigned integer.
10	**uint16**	It is the 2-byte (16-bit) unsigned integer.
11	**uint32**	It is the 4-byte (32-bit) unsigned integer.
12	**uint64**	It is the 8 bytes (64-bit) unsigned integer.
13	**float_**	It is identical to float64.
14	**float16**	It is the half-precision float. 5 bits are reserved for the exponent. 10 bits are reserved for mantissa, and 1 bit is reserved for the sign.
15	**float32**	It is a single precision float. 8 bits are reserved for the exponent, 23 bits are reserved for mantissa, and 1 bit is reserved for the sign.
16	**float64**	It is the double precision float. 11 bits are reserved for the exponent, 52 bits are reserved for mantissa, 1 bit is used for the sign.
17	**complex_**	It is identical to complex128.
18	**complex64**	It is used to represent the complex number where real and imaginary part shares 32 bits each.
19	**complex128**	It is used to represent the complex number where real and

		imaginary part shares 64 bits each.

NumPy dtype

All the items of a numpy array are data type objects also known as numpy dtypes. A data type object implements the fixed size of memory corresponding to an array.

We can create a dtype object by using the following syntax.

numpy.dtype(object, align, copy)

The constructor accepts the following object.

Object: It represents the object which is to be converted to the data type.

Align: It can be set to any boolean value. If true, then it adds extra padding to make it equivalent to a C struct.

Copy: It creates another copy of the dtype object.

Example 1:

```
import numpy as np
d = np.dtype(np.int32)
print(d)
```

Output:

```
int32
```

Example 2:

```
import numpy as np
d = np.int32
print(d)
```

Output:

```
<class 'numpy.int32'>
```

Creating a Structured data type

We can create a map-like (dictionary) data type which contains the mapping between the values. For example, it can contain the mapping between employees and salaries or the students and the age, etc.

Consider the following example.

Example 1:

```
import numpy as np
```

```
d = np.dtype([('salary', np.float)])
print(d)
```
Output:
```
[('salary', '<="" pre="">
```
Example 2
```
import numpy as np
d=np.dtype([('salary',np.float)])
a = np.array([(10000.12,), (20000.50,)], dtype=d)
print(a['salary'])
```
Output:
```
[(10000.12,) (20000.5 ,)]
```

8.5. Universal Array Function

8.6. Array Input and Output

Python - numpy.append()

The **numpy.append()** function is available in NumPy package. As the name suggests, append means adding something. **The numpy.append() function is used to add or append new values to an existing numpy array.** This function adds the new values at the end of the array.

The numpy append() function is used to merge two arrays. It returns a new array, and the original array remains unchanged.

Syntax

numpy.append(arr, values, axis=None)

Parameters

There are the following parameters of the append() function:

1) arr: array_like

This is a ndarray. The new values are appended to a copy of this array. This parameter is required and plays an important role in numpy.append() function.

2) values: array_like

This parameter defines the values which are appended to a copy of a ndarray. One thing is to be noticed here that these values must be of the correct shape as the original ndarray, excluding the axis. If the axis is not defined, then the values can be in any shape and will flatten before use.

3) axis: int(optional)

This parameter defines the axis along which values are appended. When the axis is not given to them, both ndarray and values are flattened before use.

Returns

This function returns a copy of ndarray with values appended to the axis.

Example 1: np.append()

```
import numpy as np
a = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
b = np.array([[11, 21, 31], [42, 52, 62], [73, 83, 93]])
c = np.append(a,b)
c
```

Output:

```
array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 11, 21, 31, 42, 52, 62, 73, 83, 93])
```

In the above code

- We have imported numpy with alias name np.
- We have created an array 'a' using np.array() function.
- Then we have created another array 'b' using the same np.array() function.
- We have declared the variable 'c' and assigned the returned value of np.append() function.
- We have passed the array 'a' and 'b' in the function.
- Lastly, we tried to print the value of arr.

In the output, values of both arrays, i.e., 'a' and 'b', have been shown in the flattened form, and the original array remained same.

Example 2: np.append({a1,a2,...}, axis=0)

```
import numpy as np
a = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
b = np.array([[11, 21, 31], [42, 52, 62], [73, 83, 93]])
c = np.append(a,b,axis=0)
c
```

Output:

```
array([[ 10,  20,  30],
       [ 40,  50,  60],
       [ 70,  80,  90],
       [11, 21, 31],
       [42, 52, 62],
       [73, 83, 93]])
```

In the above code

- We have imported numpy with alias name np.
- We have created an array 'a' using np.array() function.
- Then we have created another array 'b' using the same np.array() function.
- We have declared the variable 'c' and assigned the returned value of np.append() function.
- We have passed the array 'a' and 'b' in the function, and we have also passed the axis as 0.
- Lastly, we tried to print the value of arr.

In the output, values of both arrays, i.e., 'a' and 'b', have been shown vertically in a single array, and the original array remained the same.

Example 3: np.append({a1,a2,...}, axis=1)

```
import numpy as np
a = np.array([[10, 20, 30], [40, 50, 60], [70, 80, 90]])
b = np.array([[11, 21, 31], [42, 52, 62], [73, 83, 93]])
c = np.append(a,b,axis=1)
c
```

Output:

```
array([[ 10,  20,  30, 11, 21, 31],
       [ 40,  50,  60, 42, 52, 62],
       [ 70,  80,  90, 73, 83, 93]])
```

Python - numpy.reshape()

The numpy.reshape() function is available in NumPy package. As the name suggests, reshape means 'changes in shape'. The numpy.reshape() function helps us to get a new shape to an array without changing its data.

Sometimes, we need to reshape the data from wide to long. So in this situation, we have to reshape the array using reshape() function.

Syntax

numpy.reshape(arr, new_shape, order='C')

Parameters

There are the following parameters of reshape() function:

1) arr: array_like

This is a ndarray. This is the source array which we want to reshape. This parameter is essential and plays a vital role in numpy.reshape() function.

2) new_shape: int or tuple of ints

The shape in which we want to convert our original array should be compatible with the original array. If an integer, the result will be a 1-D array of that length. One shape dimension can be -1. Here, the value is approximated by the length of the array and the remaining dimensions.

3) order: {'C', 'F', 'A'}, optional

These indexes order parameter plays a crucial role in reshape() function. These index orders are used to read the elements of source array and place the elements into the reshaped array using this index order.

1. **The index order 'C'** means to read/write the elements which are using a C-like index order where the last axis index is changing fastest, back to the first axis index changing slowest.
2. **The index order 'F'** means to read/write the elements which are using the Fortran-like index order, where the last axis index changing slowest and the first axis index changing fastest.
3. **The 'C' and 'F'** order take no amount of the memory layout of the underlying array and only refer to the order of indexing.
4. **The index order 'A'** means to read/write the elements in Fortran-like index order, when arr is contiguous in memory, otherwise use C-like order.

Returns

This function returns a ndarray. It is a new view object if possible; otherwise, it will be a copy. There is no guarantee of the memory layout of the returned array.

Example 1: C-like index ordering

```
import numpy as np
x = np.arange(12)
y = np.reshape(x, (4,3))
x
y
```

Output:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
array([[ 0,  1,  2,  3],
              [ 4,  5,  6,  7],
              [ 8,  9, 10, 11]])
```

In the above code

- We have imported numpy with alias name np.
- We have created an array 'a' using np.arrange() function.
- We have declared the variable 'y' and assigned the returned value of the np.reshape() function.
- We have passed the array 'x' and the shape in the function.
- Lastly, we tried to print the value of arr.

In the output, the array has been represented as three rows and four columns.

Example 2: Equivalent to C ravel then C reshape

```
import numpy as np
x=np.arange(12)
y=np.reshape(np.ravel(x),(3,4))
x
y
```

The ravel() function is used for creating a contiguous flattened array. A one-dimensional array that contains the elements of the input, is returned. A copy is made only when it is needed.

Output:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
array([[ 0,  1,  2,  3],
            [ 4,  5,  6,  7],
            [ 8,  9, 10, 11]])
```

Example 3: Fortran-like index ordering

```
import numpy as np
x = np.arange(12)
y = np.reshape(x, (4, 3), order='F')
x
y
```

Output:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

In the above code

- We have imported numpy with alias name np.
- We have created an array 'a' using np.arrange() function.
- We have declared the variable 'y' and assigned the returned value of np.reshape() function.
- We have passed the array 'x' and the shape and Fortran-like index order in the function.
- Lastly, we tried to print the value of arr.

In the output, the array has been represented as four rows and three columns.

Example 4: Fortran-like index ordering

```
import numpy as np
x=np.arange(12)
y=np.reshape(np.ravel(x, order='F'), (4, 3), order='F')
x
y
```

Output:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
array([[ 0,  4,  8],
            [ 1,  5,  9],
            [ 2,  6, 10],
            [ 3,  7, 11]])
```

Example 5: The unspecified value is inferred to be 2

```
import numpy as np
x = np.arange(12)
y = np.reshape(x, (2, -1))
x
y
```

In the above code

- We have imported numpy with alias name np.
- We have created an array 'a' using np.arrange() function.
- We have declared the variable 'y' and assigned the returned value of the np.reshape() function.
- We have passed the array 'x' and the shape (unspecified value) in the function.
- Lastly, we tried to print the value of arr.

In the output, the array has been represented as two rows and five columns.

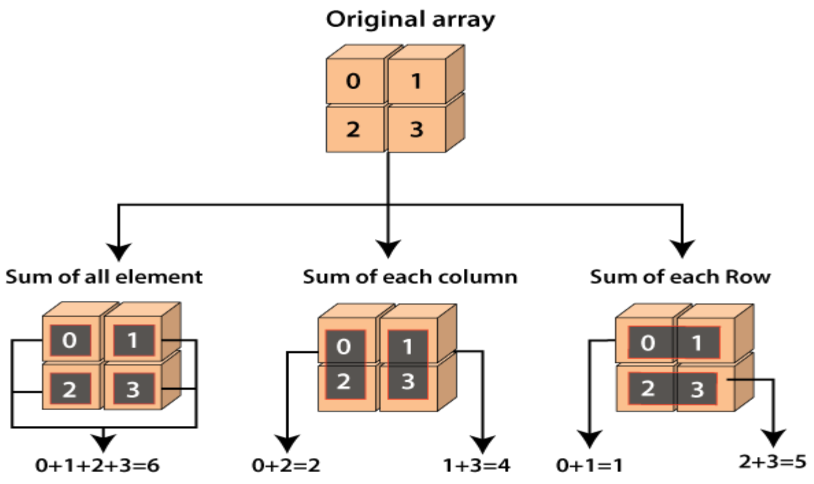
Output:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
array([[ 0,  1,  2,  3,  4,  5],
              [ 6,  7,  8,  9, 10, 11]])
```

Python - numpy.sum()

The numpy.sum() function is available in the NumPy package of Python. This function is used to compute the sum of all elements, the sum of each row, and the sum of each column of a given array.

Essentially, this sum ups the elements of an array, takes the elements within a ndarray, and adds them together. It is also possible to add rows and column elements of an array. The output will be in the form of an array object.



Syntax

There is the following syntax of numpy.sum() function:

numpy.sum(arr, axis=None, dtype=None, out=None, keepdims=<no value>, initial=<no value>)

Parameters

1) arr: array_like

This is a ndarray. This is the source array whose elements we want to sum. This parameter is essential and plays a vital role in numpy.sum() function.

2) axis: int or None or tuple of ints(optional)

This parameter defines the axis along which a sum is performed. The default axis is None, which will sum all the elements of the array. When the axis is negative, it counts from the last to the first axis. In version 1.7.0, a sum is performed on all axis specified in the tuple instead of a single axis or all axis as before when an axis is a tuple of ints.

3) dtype: dtype(optional)

This parameter defines the type of the accumulator and the returned array in which the elements are summed. By default, the dtype of arr is used unless arr has an integer dtype of less precision than the default platform integer. In such a case, when arr is signed, then the platform integer is used, and when arr is unsigned, then an unsigned integer of the same precision as the platform integer is used.

4) out: ndarray(optional)

This parameter defines the alternative output array in which the result will be placed. This resulting array must have the same shape as the expected output. The type of output values will be cast, when necessary.

5) keepdims: bool(option)

This parameter defines a Boolean value. When this parameter is set to True, the axis which is reduced is left in the result as dimensions with size one. With the help of this option, the result will be broadcast correctly against the input array. The keepdims will not be passed to the sum method of sub-classes of a ndarray, when the default value is passed, but not in case of non-default value. If the sub-class method does not implement keepdims, then any exception can be raised.

6) initial: scalar

This parameter defines the starting value for the sum.

Returns

This function returns an array of the same shape as arr with the specified axis removed. When arr is a 0-d array, or when the axis is None, a scalar is returned. A reference to **out** is returned, when an array output is specified.

Example 1: numpy.array()

```
import numpy as np
a = np.array([0.4,0.5])
b = np.sum(a)
b
```

Output:

```
0.9
```

In the above code

- We have imported numpy with alias name 'np'.
- We have created an array 'a' using np.array() function.
- We have declared variable 'b' and assigned the returned value of np.sum() function.
- We have passed the array 'a' in the function.
- Lastly, we tried to print the value of b.

In the output, the sum of all the elements of the array has been shown.

Example 2:

```
import numpy as np
a = np.array([0.4,0.5,0.9,6.1])
b = np.sum(a, dtype=np.int32)
b
```

Output:

```
6
```

In the above code

- We have imported numpy with alias name 'np'.
- We have created an array 'a' using np.array() function.
- We have declared variable 'b' and assigned the returned value of np.sum() function.
- We have passed the array 'a' and data type of int32 in the function.
- Lastly, we tried to print the value of b.

In the output, the sum only of integer numbers, not floating-point values has been displayed.

Example 3:

```
import numpy as np
a = np.array([[1,4],[3,5]])
b = np.sum(a)
b
```

In the above code

Output:

```
13
```

Example 4:

```
import numpy as np
a = np.array([[1,4],[3,5]])
b = np.sum(a,axis=0)
b
```

Output:

```
array([4, 9])
```

In the above code

- We have imported numpy with alias name np.

- We have created an array 'a' using np.array() function.
- We have declared variable 'b' and assigned the returned value of np.sum() function.
- We have passed the array 'a' and axis=0 in the function.
- Lastly, we tried to print the value of b.

In the output, the sum of the column elements has been calculated accordingly.

Example 5:

```
import numpy as np
a = np.array([[1,4],[3,5]])
b = np.sum(a,axis=1)
b
```

Output:

```
array([5, 8])
```

Example 6:

```
import numpy as np
b = np.sum([15], initial=8)
b
```

Output:

```
23
```

In the above code

- We have imported numpy with alias name np.
- We have declared variable 'b' and assigned the returned value of np.sum() function.
- We have passed the number of elements and initial value in the function.
- Lastly, we tried to print the value of b.

In the output, the initial value has been added to the last element in the sequence of elements and then performed the sum of all the elements.

Python - numpy.random()

The random is a module present in the NumPy library. This module contains the functions which are used for generating random numbers. This module contains some simple random data generation methods, some permutation and distribution functions, and random generator functions.

All the functions in a random module are as follows:

Simple random data

There are the following functions of simple random data:

1) np.random.rand(d0, d1, ..., dn)

This function of random module is used to generate random numbers or values in a given shape.

Example:

```
import numpy as np
a=np.random.rand(5,2)
a
```

Output:

```
array([[0.74710182, 0.13306399],
       [0.01463718, 0.47618842],
       [0.98980426, 0.48390004],
       [0.58661785, 0.62895758],
       [0.38432729, 0.90384119]])
```

2) np.random.randn(d0, d1, ..., dn)

This function of random module return a sample from the "standard normal" distribution.

Example:

```
import numpy as np
a = np.random.randn(2,2)
a
```

Output:

```
array([[ 1.43327469, -0.02019121],
    [ 1.54626422,  1.05831067]])
b = np.random.randn()
b
-0.3080190768904835
```

3) np.random.randint(low[, high, size, dtype])

This function of random module is used to generate random integers from inclusive(low) to exclusive(high).

Example:

```
import numpy as np
a=np.random.randint(3, size=10)
a
```

Output:

```
array([1, 1, 1, 2, 0, 0, 0, 0, 0, 0])
```

4) np.random.randint(low[, high, size])

This function of random module is used to generate random integers number of type np.int between low and high.

Example:

```
import numpy as np
a = np.random.randint(3)
a
b = type(np.random.randint(3))
b
c = np.random.randint(5, size=(3,2))
c
```

Output:

```
2
<class ‘int’>
array([[1, 1],
       [2, 5],
       [1, 3]])
```

5) np.random.random_sample([size])

This function of random module is used to generate random floats number in the half-open interval [0.0, 1.0).

Example:

```
import numpy as np
a = np.random.random_sample()
```

```
a
b = type(np.random.random_sample())
b
c = np.random.random_sample((5,))
c
```

Output:

```
0.09250360565571492
<type 'float'>
array([0.34665418, 0.47027209, 0.75944969, 0.37991244, 0.14159746])
```

6) np.random.random([size])

This function of random module is used to generate random floats number in the half-open interval [0.0, 1.0).

Example:

```
import numpy as np
a = np.random.random()
a
b = type(np.random.random())
b
c=np.random.random((5,))
c
```

Output:

```
0.008786953974334155
<type 'float'>
array([0.05530122, 0.59133394, 0.17258794, 0.6912388 , 0.33412534])
```

7) np.random.ranf([size])

This function of random module is used to generate random floats number in the half-open interval [0.0, 1.0).

Example:

```
import numpy as np
a = np.random.ranf()
a
```

```
b = type(np.random.ranf())
b
c = np.random.ranf((5,))
c
```

Output:

```
0.2907792098474542
<type 'float'>
array([0.34084881, 0.07268237, 0.38161256, 0.46494681, 0.88071377])
```

8) np.random.sample([size])

This function of random module is used to generate random floats number in the half-open interval [0.0, 1.0).

Example:

```
import numpy as np
a = np.random.sample()
a
b = type(np.random.sample())
b
c = np.random.sample((5,))
c
```

Output:

```
0.012298209913766511
<type 'float'>
array([0.71878544, 0.11486169, 0.38189074, 0.14303308, 0.07217287])
```

9) np.random.choice(a[, size, replace, p])

This function of random module is used to generate random sample from a given 1-D array.

Example:

```
import numpy as np
a = np.random.choice(5,3)
a
b = np.random.choice(5,3, p=[0.2, 0.1, 0.4, 0.2, 0.1])
b
```

Output:

```
array([0, 3, 4])
array([2, 2, 2], dtype=int64)
```

10) np.random.bytes(length)

This function of random module is used to generate random bytes.

Example:

```
import numpy as np
a = np.random.bytes(7)
a
```

Output:

```
'nQ\x08\x83\xf9\xde\x8a'
```

Permutations

There are the following functions of permutations:

1) np.random.shuffle()

This function is used for modifying a sequence in-place by shuffling its contents.

Example:

```
import numpy as np
a  =np.arange(12)
a
np.random.shuffle(a)
a
```

Output:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
array([10,  3,  2,  4,  5,  8,  0,  9,  1, 11,  7,  6])
```

2) np.random.permutation()

This function permute a sequence randomly or return a permuted range.

Example:

```
import numpy as np
a = np.random.permutation(12)
a
```

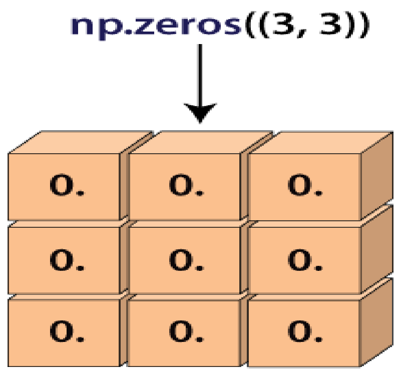
Output:

```
array([ 8,  7,  3, 11,  6,  0,  9, 10,  2,  5,  4,  1])
```

Python - numpy.zeros()

The numpy.zeros() function is one of the most significant functions which is used in machine learning programs widely. This function is used to generate an array containing zeros.

The numpy.zeros() function provide a new array of given shape and type, which is filled with zeros.



Syntax

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

Parameters

shape: int or tuple of ints

This parameter is used to define the dimensions of the array. This parameter is used for the shape in which we want to create an array, such as (3,2) or 2.

dtype: data-type(optional)

This parameter is used to define the desired data-type for the array. By default, the data-type is numpy.float64. This parameter is not essential for defining.

order: {'C','F'}(optional)

This parameter is used to define the order in which we want to store data in memory either row-major(C-style) or column-major(Fortran-style)

Return

This function returns a ndarray. The output array is the array with specified shape, dtype, order, and contains zeros.

Example 1: numpy.zeros() without dtype and order

```
import numpy as np
```

```
a = np.zeros(6)
a
```

Output:

```
array([0., 0., 0., 0., 0., 0.])
```

In the above code

- We have imported numpy with alias name np.
- We have declared the variable 'a' and assigned the returned value of np.zeros() function.
- We have passed an integer value in the function.
- Lastly, we tried to print the value of 'a'.

In the output, an array with floating-point integers(zeros) has been shown.

Example 2: numpy.zeros() without order

```
import numpy as np
a = np.zeros((6,), dtype=int)
a
```

Output:

```
array([0, 0, 0, 0, 0, 0])
```

Example 3: numpy.zeros() with shape

```
import numpy as np
a=np.zeros((6,2))
a
```

Output:

```
array([[0., 0.],
          [0., 0.],
          [0., 0.],
          [0., 0.],
          [0., 0.],
          [0., 0.]])
```

In the above code

- We have imported numpy with alias name np.
- We have declared the variable 'a' and assigned the returned value of np.zeros() function.
- We have passed the shape for the array elements.

- Lastly, we tried to print the value of 'a'.

In the output, an array of given shape has been shown.

Example 4: numpy.zeros() with the shape

```
import numpy as np
s1=(3,2)
a = np.zeros(s1)
a
```

Output:

```
array([[0., 0.],
       [0., 0.],
       [0., 0.]])
```

Example 5: numpy.zeros() with custom dtype

```
import numpy as np
a = np.zeros((3,), dtype=[('x', 'i4'), ('y', 'i4')])
a
```

Output:

```
array([(0, 0), (0, 0), (0, 0)], dtype=[('x', '<i4'), ('y', '<i4')])
```

In the above code

- We have imported numpy with alias name np.
- We have declared the variable 'a' and assigned the returned value of np.zeros() function.
- We have passed the shape and custom data type in the function.
- Lastly, we tried to print the value of 'a'.

In the output, an array contains zeros with custom data-type has been shown.

Python - numpy.log()

The numpy.log() is a mathematical function that is used to calculate the natural logarithm of x(x belongs to all the input array elements). It is the inverse of the exponential function as well as an element-wise natural logarithm. The natural logarithm log is the reverse of the exponential function, so that log(exp(x))=x. The logarithm in base e is the natural logarithm.

Syntax

numpy.log(x, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None , subok=True[, signature, extobj]) = <ufunc 'log'>

Parameters

x: array_like

This parameter defines the input value for the numpy.log() function.

out: ndarray, None, or tuple of ndarray and None(optional)

This parameter is used to define the location in which the result is stored. If we define this parameter, it must have a shape similar to the input broadcast; otherwise, a freshly-allocated array is returned. A tuple has a length equal to the number of outputs.

where: array_like(optional)

It is a condition that is broadcast over the input. At this location, where the condition is True, the out array will be set to the ufunc(universal function) result; otherwise, it will retain its original value.

casting: {'no','equiv','safe','same_kind','unsafe'}(optional)

This parameter controls the kind of data casting that may occur. The 'no' means the data types should not be cast at all. The 'equiv' means only byte-order changes are allowed. The 'safe' means the only cast, which can allow the preserved value. The 'same_kind' means only safe casts or casts within a kind. The 'unsafe' means any data conversions may be done.

order: {'K', 'C', 'F', 'A'}(optional)

This parameter specifies the calculation iteration order/ memory layout of the output array. By default, the order will be K. The order 'C' means the output should be C-contiguous. The order 'F' means F-contiguous, and 'A' means F-contiguous if the inputs are F-contiguous and if inputs are in C-contiguous, then 'A' means C-contiguous. 'K' means to match the element ordering of the inputs(as closely as possible).

dtype: data-type(optional)

It overrides the dtype of the calculation and output arrays.

subok: bool(optional)

By default, this parameter is set to true. If we set it to false, the output will always be a strict array, not a subtype.

signature

This argument allows us to provide a specific signature to the 1-d loop 'for', used in the underlying calculation.

extobj

This parameter is a list of length 1, 2, or 3 specifying the ufunc buffer-size, the error mode integer, and the error callback function.

Returns

This function returns a ndarray that contains the natural logarithmic value of x, which belongs to all elements of the input array.

Example 1:

```
import numpy as np
a  =np.array([2, 4, 6, 3**8])
a
b = np.log(a)
b
c = np.log2(a)
c
d = np.log10(a)
d
```

Output:

```
array([   2,    4,    6, 6561])
array([0.69314718, 1.38629436, 1.79175947, 8.78889831])
array([ 1.        ,  2.        ,  2.5849625 , 12.67970001])
array([0.30103   , 0.60205999, 0.77815125, 3.81697004])
```

In the above mentioned code

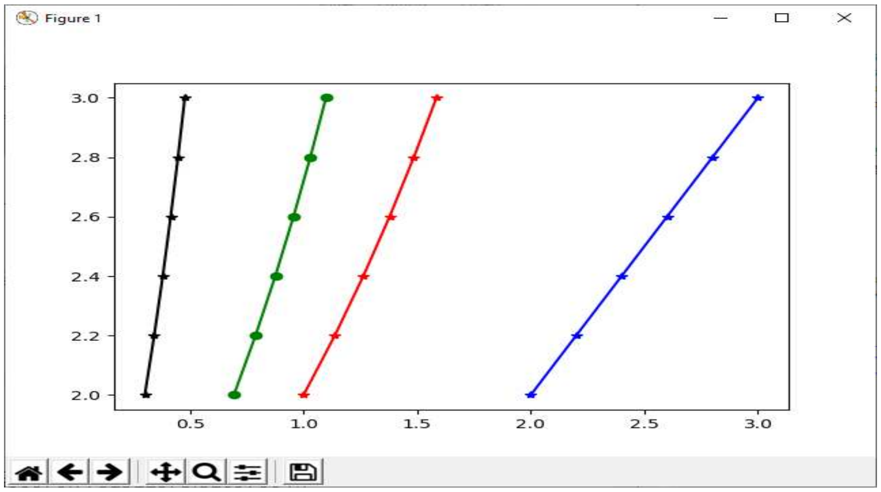
- We have imported numpy with alias name np.
- We have created an array 'a' using np.array() function.
- We have declared variable b, c, and, d and assigned the returned value of np.log(), np.log2(), and np.log10() functions respectively.
- We have passed the array 'a' in all the functions.
- Lastly, we tried to print the value of b, c, and d.

In the output, a ndarray has been shown, contains the log, log2, and log10 values of all the elements of the source array.

Example 2:

```
import numpy as np
import matplotlib.pyplot as plt
arr = [2, 2.2, 2.4, 2.6,2.8, 3]
result1 = np.log(arr)
result2 = np.log2(arr)
result3 = np.log10(arr)
plt.plot(arr,arr, color='blue', marker="*")
plt.plot(result1,arr, color='green', marker="o")
plt.plot(result2,arr, color='red', marker="*")
plt.plot(result3,arr, color='black', marker="*")
plt.show()
```

Output:



In the above code

- We have imported numpy with alias name np.
- We have also imported matplotlib.pyplot with alias name plt.
- Next, we have created an array 'arr' using np.array() function.
- After that we declared variable result1, result2, result3 and assigned the returned values of np.log(), np.log2(), and np.log10() functions respectively.
- We have passed the array 'arr' in all the functions.

- Lastly, we tried to plot the values of 'arr', result1, result2, and result3.

In the output, a graph with four straight lines with different colors has been shown.

Example 3:

```
import numpy as np
x=np.log([2, np.e, np.e**3, 0])
x
```

Output:

```
__main__:1: RuntimeWarning: divide by zero encountered in log
array([0.69314718, 1.        , 3.        ,        -inf])
```

In the above code

- Firstly, we have imported numpy with alias name np.
- We have declared the variable 'x' and assigned the returned value of np.log() functions.
- We have passed different values in the function, such as integer value, np.e, and np.e**2.
- Lastly, we tried to print the value of 'x'.

In the output, a ndarray has been shown, contains the log values of the elements of the source array.

Python - numpy.transpose()

The numpy.transpose() function is one of the most important functions in matrix multiplication. This function permutes or reserves the dimension of the given array and returns the modified array.

The numpy.transpose() function changes the row elements into column elements and the column elements into row elements. The output of this function is a modified array of the original one.

Syntax

numpy.transpose(arr, axis=None)

Parameters

arr: array_like

It is an ndarray. It is the source array whose elements we want to transpose. This parameter is essential and plays a vital role in numpy.transpose() function.

axis: List of ints()

If we didn't specify the axis, then by default, it reverses the dimensions otherwise permute the axis according to the given values.

Return

This function returns a ndarray. The output array is the source array, with its axis permuted. A view is returned whenever possible.

Example 1: numpy.transpose()

```
import numpy as np
a= np.arange(6).reshape((2,3))
a
b=np.transpose(a)
b
```

Output:

```
array([[0, 1, 2],
       [3, 4, 5]])
array([[0, 3],
        [1, 4],
        [2, 5]])
```

In the above code

- We have imported numpy with alias name np.
- We have created an array 'a' using np.arange() function and gave a shape using reshape() function.
- We have declared the variable 'b' and assigned the returned value of np.transpose() function.
- We have passed the array 'a' in the function.
- Lastly, we tried to print the value of b.

In the output, the transposed array of the original array has been shown.

Example 2: numpy.transpose() with axis

```
import numpy as np
a= np.array([[1, 2], [4, 5], [7, 8]])
a
b=np.transpose(a, (1,0))
```

```
b
```

Output:

```
array([[1, 2],
       [4, 5],
       [7, 8]])
array([[1, 4, 7],
        [2, 5, 8]])
```

In the above code

- We have imported numpy with alias name np.
- We have created an array 'a' using np.array() function.
- We have declared the variable 'b' and assigned the returned value of np.transpose() function.
- We have passed the array 'a' and the axis in the function.
- Lastly, we tried to print the value of b.

In the output, the transposed array of the original array has been shown.

Example 3: Reposition elements using numpy.transpose()

```
import numpy as np
a=np.ones((12,32,123,64))
b=np.transpose(a,(1,3,0,2)).shape
b
c=np.transpose(a,(0,3,1,2)).shape
c
```

Output:

```
(32L, 64L, 12L, 123L)
(12L, 64L, 32L, 123L)
```

- We have imported numpy with alias name np.
- We have created an array 'a' using np.ones() function.
- We have declared the variable 'b' and 'c' and assigned the returned value of np.transpose() function.
- We have passed the array 'a' and the positions of the array elements in the function.
- Lastly, we tried to print the value of b and c.

In the output, an array has been shown whose elements are located at the defined position in the array.

8.7. Pandas:

8.8. What are pandas? Where it is used?

8.9. Series in pandas, pandas DataFrames, Index objects, ReIndex

8.10. Drop Entry, Selecting Entries

8.11. Data Alignment, Rank and Sort

8.12. Summary Statics, Missing Data, Index Hierarchy

8.7. Pandas:

8.8. What are pandas? Where it is used?

8.9. Series in pandas, pandas DataFrames, Index objects, ReIndex

Python Pandas - Introduction

Pandas is defined as an open-source library that provides high-performance data manipulation in Python. The name of Pandas is derived from the word **Pan**el **da**ta, which means **an Econometrics from Multidimensional data**. It is used for data analysis in Python and developed by **Wes McKinney** in **2008**.

Data analysis requires lots of processing, such as **restructuring, cleaning** or **merging**, etc. Different tools are available for fast data processing, such as **Numpy, Scipy, Cython**, and **Panda**. But we prefer Pandas because working with Pandas is fast, simple and more **expressive** than other tools.

Pandas is built on top of the** Numpy **package, means** Numpy **is required for operating the Pandas.

Before Pandas, Python was capable for data preparation, but it only provided limited support for data analysis. So, Pandas came into the picture and enhanced the capabilities of data analysis. It can perform five significant steps required for processing and analysis of data irrespective of the origin of the data, i.e., **load, manipulate, prepare, model, and analyze**.

Key Features of Pandas

- It has a fast and efficient **DataFrame** object with the **default** and **customized indexing**.
- Used for reshaping and pivoting of the data sets.
- Group by data for aggregations and transformations.
- It is used for data alignment and integration of the missing data.
- Provide the functionality of Time Series.
- Process a variety of data sets in different formats like matrix data, tabular heterogeneous, time series.

- Handle multiple operations of the data sets such as subsetting, slicing, filtering, groupBy, re-ordering, and re-shaping.
- It integrates with the other libraries such as SciPy, and scikit-learn.
- Provides fast performance, and If you want to speed it, even more, you can use the **Cython**.

Benefits of Pandas

The benefits of pandas over using other language are as follows:

- **Data Representation:** It represents the data in a form that is suited for data analysis through its DataFrame and Series.
- **Clear code:** The clear API of the Pandas allows you to focus on the core part of the code. So, it provides clear and concise code for the user.

In Short ….

Why Use Pandas?

Pandas allows us to analyze big data and make conclusions based on statistical theories. Pandas can clean messy data sets, and make them readable and relevant. Relevant data is very important in **Data Science**.

Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

What Can Pandas Do?

Pandas gives you answers about the data. Like:

- Is there a correlation between two or more columns?
- What is average value?
- Max value?
- Min value?

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called *cleaning* the data.

Pandas Getting Started

Installation of Pandas

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

Install it using this command:

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

If this command fails, then use a python distribution that already has Pandas installed like, Anaconda, Spyder etc.

Import Pandas

Once Pandas is installed, import it in your applications by adding the import keyword:

```
import pandas
```

Now Pandas is imported and ready to use.

Example

```
import pandas
mydataset = {
  'cars': ["BMW", "Volvo", "Ford"],
  'passings': [3, 7, 2]
}
df = pandas.DataFrame(mydataset)
print(df)
```

Pandas as pd

Pandas is usually imported under the pd alias.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the as keyword while importing:

```
import pandas as pd
```

Now the Pandas package can be referred to as pd instead of pandas.

Example

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

Checking Pandas Version

The version string is stored under __version__ attribute.

Example

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

Python Pandas - Data Structure

Pandas deals with the following three data structures

- **Series**
- **DataFrame**
- **Panel**

These data structures are built on top of **Numpy array**, which means they are fast.

Dimension & Description

The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure. For example, **DataFrame is a container of Series, Panel is a container of DataFrame.**

Data Structure	Dimensions	Description
Series	**1**	**1D** labeled **homogeneous array**, **size-immutable.**

Data Frames	**2**	General **2D** labeled, **size-mutable** tabular structure with potentially **heterogeneously** typed columns.
Panel	**3**	General **3D** labeled, **size-mutable** array.

Building and handling two or more dimensional arrays is a tedious task, burden is placed on the user to consider the orientation of the data set when writing functions. But using Pandas data structures, the mental effort of the user is reduced.
For example, with tabular data (DataFrame) it is more semantically helpful to think of the **index** (the rows) and the **columns** rather than axis 0 and axis 1.

Mutability

All Pandas data structures are **value mutable (can be changed)** and except **Series** all are **size mutable**. Series is **size immutable.**
Note − **DataFrame** is widely used and one of the most **important data structures**. **Panel** is used **much less.**

Series

Series is **a one-dimensional** array like structure with **homogeneous data.** For example, the following series is **a collection of integers 10, 23, 56, …**

10	**23**	**56**	**17**	**52**	**61**	**73**	**90**	**26**	**72**

Key Points

- Homogeneous data
- Size Immutable
- Values of Data Mutable

DataFrame

DataFrame is **a two-dimensional** array with **heterogeneous data.** For example,

Name	Age	Gender	Rating
Ankush	32	Male	3.45
Indu	28	Female	4.6
Abhijeet	45	Male	3.9
Surekha	38	Female	2.78

The table represents the data of **a sales team** of an organization with their **overall performance rating.** The data is represented in **rows** and **columns.** Each **column** represents **an attribute** and each **row** represents **a person.**

Data Type of Columns

The data types of the four columns are as follows

Column	Type
Name	String
Age	Integer
Gender	String
Rating	Float

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

Panel

Panel is a three-dimensional data structure with heterogeneous data. It is hard to represent the panel in graphical representation. But a panel can be illustrated as a container of DataFrame.

Key Points

- Heterogeneous data
- Size Mutable
- Data Mutable

****

Note: Out of these...

Pandas provide two data structures for processing the data, i.e., **Series** and **DataFrame,** Whereas Panel data structure is not available in latest versions of Python – pandas.

****

Python Pandas – Series

Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called index.

pandas.Series

A pandas Series can be created using the following constructor −

Syntax:

pandas.Series(data, index, dtype, copy)

The parameters of the constructor are as follows

Sr. No	Parameter & Description
1	**data** data takes various forms like ndarray, list, constants.
2	**index** Index values must be **unique** and **hashable**, same length as data. Default **np.arrange(n)** if **no index is passed**.

3	**dtype** dtype is for data type. If None, data type will be inferred
4	**copy** Copy data. Default False

A series can be created using various inputs like

- **Array**
- **Dict**
- **Scalar value or constant**

Create an Empty Series

A basic series, which can be created is an Empty Series.

Example:

```
#import the pandas library and aliasing as pd
import pandas as pd
s = pd.Series()
print(s)
```

Output:

```
Series([], dtype: float64)
```

Create a Series from ndarray

If data is an ndarray, then index passed must be of the same length. If no index is passed, then by default index will be **range(n)** where **n** is array length, i.e., [0,1,2,3…. **range(len(array))-1].**

Example 1:

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a','b','c','d'])
s = pd.Series(data)
print (s)
```

Output:

```
0   a
1   b
2   c
3   d
dtype: object
```

We did not pass any index, so by default, it assigned the indexes ranging from 0 to **len(data)-1**, i.e., 0 to 3.

Example 2:

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = np.array(['a', 'b', 'c', 'd'])
s = pd.Series(data, index=[100, 101, 102, 103])
print (s)
```

Output:

```
100  a
101  b
102  c
103  d
dtype: object
```

We passed the index values here. Now we can see the customized indexed values in the output.

Create a Series from dict

A **dict** can be passed as input and if no index is specified, then the dictionary keys are taken in a sorted order to construct index. If **index** is passed, the values in data corresponding to the labels in the index will be pulled out.

Example 1

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
```

```
s = pd.Series(data)
print (s)
```

Output:

```
a 0.0
b 1.0
c 2.0
dtype: float64
```

Observe − Dictionary keys are used to construct index.

Example 2:

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data, index=['b', 'c', 'd', 'a'])
print (s)
```

Output:

```
b 1.0
c 2.0
d NaN
a 0.0
dtype: float64
```

Observe − Index order is persisted and the missing element is filled with NaN (Not a Number).

Create a Series from Scalar

If data is a scalar value, an index must be provided. The value will be repeated to match the length of **index**

```
#import the pandas library and aliasing as pd
import pandas as pd
import numpy as np
s = pd.Series(5, index=[0, 1, 2, 3])
print (s)
```

Its **output** is as follows −

```
0  5
1  5
2  5
3  5
dtype: int64
```

Accessing Data from Series with Position

Data in the series can be accessed similar to that in an **ndarray.**

Example 1

Retrieve the first element. As we already know, the counting starts from zero for the array, which means the first element is stored at zeroth position and so on.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
#retrieve the first element
print (s[0])
```

Output:

```
1
```

Example 2

Retrieve the first three elements in the Series. If a : is inserted in front of it, all items from that index onwards will be extracted. If two parameters (with : between them) is used, items between the two indexes (not including the stop index)

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
#retrieve the first three element
print (s[:3])
```

Output:

```
a  1
b  2
c  3
dtype: int64
```

Example 3

Retrieve the last three elements.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
#retrieve the last three element
print (s[-3:])
```

Output:

```
c  3
d  4
e  5
dtype: int64
```

Retrieve Data Using Label (Index)

A Series is like a fixed-size **dict** in that you can get and set values by index label.

Example 1

Retrieve a single element using index label value.

```
import pandas as pd

s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])

# retrieve a single element

print (s['a'])
```

Output:

```
1
```

Example 2

Retrieve multiple elements using a list of index label values.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
#retrieve multiple elements
print (s[['a','c','d']])
```

Output:

```
a  1
c  3
d  4
dtype: int64
```

Example 3

If a label is not contained, an exception is raised.

```
import pandas as pd
s = pd.Series([1,2,3,4,5],index = ['a','b','c','d','e'])
#retrieve multiple elements
print (s['f'])
```

Output:

…

KeyError: 'f'

Python Pandas - DataFrame

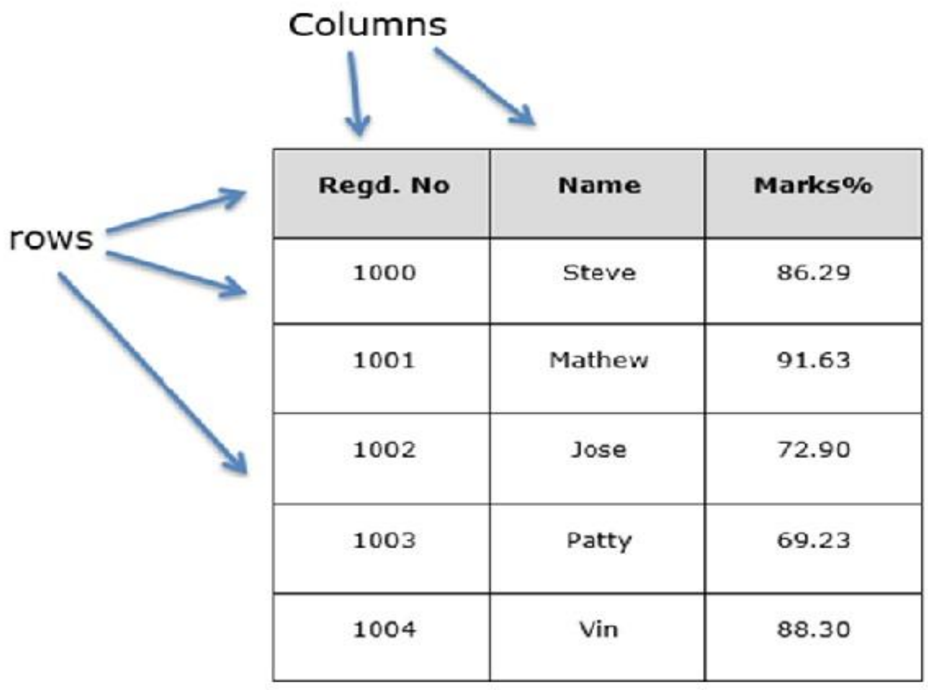
A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns.

Features of DataFrame

- Potentially columns are of different types
- Size – Mutable
- Labeled axes (rows and columns)
- Can Perform Arithmetic operations on rows and columns

Structure

Let us assume that we are creating a data frame with student's data.



Regd. No	Name	Marks%
1000	Steve	86.29
1001	Mathew	91.63
1002	Jose	72.90
1003	Patty	69.23
1004	Vin	88.30

You can think of it as an SQL table or a spreadsheet data representation.

pandas.DataFrame

A pandas DataFrame can be created using the following constructor

Syntax:

pandas.DataFrame(data, index, columns, dtype, copy)

The parameters of the constructor are as follows

Sr. No.	**Parameter & Description**
1	**data** data takes various forms like **ndarray, series, map, lists, dict, constants and also another DataFrame**.
2	**index** For the row labels, the Index to be used for the resulting frame is Optional Default np.arange(n) if no index is passed.
3	**columns** For column labels, **the optional default syntax** is - **np.arange(n)**. This is only true if no index is passed.
4	**dtype** Data type of each column.
5	**copy** This command (or whatever it is) is used for copying of data, if the default is False.

Create DataFrame

A pandas DataFrame can be created using various inputs like

- **Lists**
- **dict**
- **Series**
- **Numpy ndarrays**
- **Another DataFrame**

In the subsequent sections of this chapter, we will see how to create a DataFrame using these inputs.

Create an Empty DataFrame

A basic DataFrame, which can be created is an Empty Dataframe.

Example

```
#import the pandas library and aliasing as pd
import pandas as pd
df = pd.DataFrame()
print (df)
```

Output:

```
Empty DataFrame
Columns: []
Index: []
```

Create a DataFrame from Lists

The DataFrame can be created using a single list or a list of lists.

Example 1

```
import pandas as pd
data = [1,2,3,4,5]
df = pd.DataFrame(data)
print (df)
```

Output:

```
   0
0  1
1  2
2  3
3  4
4  5
```

Example 2

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
```

```
print (df)
```

Output:

```
     Name     Age
0    Alex     10
1    Bob      12
2    Clarke    13
```

Example 3

```
import pandas as pd
data = [['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'],dtype=float)
print (df)
```

Output:

```
     Name     Age
0    Alex     10.0
1    Bob      12.0
2    Clarke   13.0
```

Note − Observe, the **dtype** parameter changes the type of Age column to floating point.

Create a DataFrame from Dict of ndarrays / Lists

All the **ndarrays** must be of same length. If index is passed, then the length of the index should equal to the length of the arrays.

If no index is passed, then by default, index will be range(n), where **n** is the array length.

Example 1

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data)
print (df)
```

Output:

```
     Age    Name
0    28     Tom
1    34     Jack
```

```
2    29      Steve
3    42      Ricky
```

Note − Observe the values 0,1,2,3. They are the default index assigned to each using the function range(n).

Example 2

Let us now create an indexed DataFrame using arrays.

```
import pandas as pd
data = {'Name':['Tom', 'Jack', 'Steve', 'Ricky'],'Age':[28,34,29,42]}
df = pd.DataFrame(data, index=['rank1','rank2','rank3','rank4'])
print (df)
```

Output:

```
        Age    Name
rank1    28      Tom
rank2    34     Jack
rank3    29    Steve
rank4    42    Ricky
```

Note − Observe, the **index** parameter assigns an index to each row.

Create a DataFrame from List of Dicts

List of Dictionaries can be passed as input data to create a DataFrame. The dictionary keys are by default taken as column names.

Example 1

The following example shows how to create a DataFrame by passing a list of dictionaries.

```
import pandas as pd
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data)
print (df)
```

Output:

```
    a   b     c
0   1   2     NaN
1   5   10   20.0
```

Note − Observe, NaN (Not a Number) is appended in missing areas.

Example 2

The following example shows how to create a DataFrame by passing a list of dictionaries and the row indices.

```
import pandas as pd
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
df = pd.DataFrame(data, index=['first', 'second'])
print (df)
```

Output:

```
        a   b      c
first   1   2    NaN
second  5   10   20.0
```

Example 3

The following example shows how to create a DataFrame with a list of dictionaries, row indices, and column indices.

```
import pandas as pd
data = [{'a': 1, 'b': 2},{'a': 5, 'b': 10, 'c': 20}]
#With two column indices, values same as dictionary keys
df1 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b'])
#With two column indices with one index with other name
df2 = pd.DataFrame(data, index=['first', 'second'], columns=['a', 'b1'])
print (df1)
print (df2)
```

Output:

```
#df1 output
         a  b
first    1  2
second   5  10
#df2 output
         a  b1
first    1  NaN
```

```
second   5  NaN
```

Note − Observe, df2 DataFrame is created with a column index other than the dictionary key; thus, appended the NaN's in place. Whereas, df1 is created with column indices same as dictionary keys, so NaN's appended.

Create a DataFrame from Dict of Series

Dictionary of Series can be passed to form a DataFrame. The resultant index is the union of all the series indexes passed.

Example

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
  'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print (df)
```

Output:

```
    one   two
a   1.0   1
b   2.0   2
c   3.0   3
d   NaN   4
```

Note − Observe, for the series one, there is no label **'d'** passed, but in the result, for the **d** label, NaN is appended with NaN.

Let us now understand **column selection, addition**, and **deletion** through examples.

Column Selection

We will understand this by selecting a column from the DataFrame.

Example

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
  'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
```

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

Output:

```
a     1.0
b     2.0
c     3.0
d     NaN
Name: one, dtype: float64
```

Column Addition

We will understand this by adding a new column to an existing data frame.

Example

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
  'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
# Adding a new column to an existing DataFrame object with column label by passing new
series
print ("Adding a new column by passing as Series:")
df['three']=pd.Series([10,20,30],index=['a','b','c'])
print (df)
print ("Adding a new column using the existing columns in DataFrame:")
df['four']=df['one']+df['three']
print (df)
```

Output:

```
Adding a new column by passing as Series:
    one   two   three
a   1.0   1   10.0
b   2.0   2   20.0
c   3.0   3   30.0
d   NaN   4   NaN
Adding a new column using the existing columns in DataFrame:
     one  two  three   four
a    1.0   1   10.0    11.0
b    2.0   2   20.0    22.0
c    3.0   3   30.0    33.0
d    NaN   4    NaN     NaN
```

Column Deletion

Columns can be deleted or popped; let us take an example to understand how.

Example

```
# Using the previous DataFrame, we will delete a column
# using del function
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
  'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd']),
  'three' : pd.Series([10,20,30], index=['a','b','c'])}
df = pd.DataFrame(d)
print ("Our dataframe is:")
print (df)
# using del function
print ("Deleting the first column using DEL function:")
del df['one']
print (df)
# using pop function
print ("Deleting another column using POP function:")
df.pop('two')
print (df)
```

Output:

```
Our dataframe is:
     one  three  two
a    1.0   10.0   1
b    2.0   20.0   2
```

```
c    3.0    30.0   3
d    NaN     NaN   4
```

Deleting the first column using DEL function:

```
     three    two
a     10.0     1
b     20.0     2
c     30.0     3
d     NaN      4
```

Deleting another column using POP function:

```
   three
a  10.0
b  20.0
c  30.0
d  NaN
```

Row Selection, Addition, and Deletion

We will now understand row selection, addition and deletion through examples. Let us begin with the concept of selection.

Selection by Label

Rows can be selected by passing row label to a **loc** function.

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print (df.loc['b'])
```

Output:

```
one 2.0
two 2.0
Name: b, dtype: float64
```

The result is a series with labels as column names of the DataFrame. And, the Name of the series is the label with which it is retrieved.

Selection by integer location

Rows can be selected by passing integer location to an **iloc** function.

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print (df.iloc[2])
```

Output:

```
one   3.0
two   3.0
Name: c, dtype: float64
```

Slice Rows

Multiple rows can be selected using ‘ : ’ operator.

```
import pandas as pd
d = {'one' : pd.Series([1, 2, 3], index=['a', 'b', 'c']),
   'two' : pd.Series([1, 2, 3, 4], index=['a', 'b', 'c', 'd'])}
df = pd.DataFrame(d)
print (df[2:4])
```

Output:

```
   one  two
c  3.0    3
d  NaN    4
```

Addition of Rows

Add new rows to a DataFrame using the **append** function. This function will append the rows at the end.

```
import pandas as pd
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])
df = df.append(df2)
print (df)
```

Output:

```
  a  b
0  1  2
1  3  4
0  5  6
1  7  8
```

Deletion of Rows

Use index label to delete or drop rows from a DataFrame. If label is duplicated, then multiple rows will be dropped.

If you observe, in the above example, the labels are duplicate. Let us drop a label and will see how many rows will get dropped.

```
import pandas as pd
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])
df = df.append(df2)
# Drop rows with label 0
df = df.drop(0)
print (df)
```

Output:

```
  a b
1 3 4
1 7 8
```

In the above example, two rows were dropped because those two contain the same label 0.

Python - Pandas Index

Pandas Index is defined as a vital tool that selects particular rows and columns of data from a DataFrame. Its task is to organize the data and to provide fast accessing of data. It can also be called as a **Subset Selection**.

The values are in **bold** font in the index, and the individual value of the index is called a **label**.

If we want to compare the data accessing time with and without indexing, we can use **%%timeit** for comparing the time required for various access-operations.

We can also define an index like an address through which any data can be accessed across the Series or DataFrame. A DataFrame is a combination of three different components, the **index**, **columns,** and the **data**.

Axis and axes

An axis is defined as a common terminology that refers to rows and columns, whereas axes are collection of these rows and columns.

Creating index

First, we have to take a csv file that consist some data used for indexing.

```
# importing pandas package
import pandas as pd
data = pd.read_csv("Ex_data.csv")
print(data)
```

Output:

```
  Duration Pulse Maxpulse Calories
0     60      110        130     409.1
1     60      117        145     479.0
2     60      103        135     340.0
3     45      109        175     282.4
4     45      117        148     406.0
5     60      102        127     300.0
6     60      110        136     374.0
7     45      104        134     253.3
8     30      109        133     195.1
9     60      98         124     269.0
```

Example1:

```
# importing pandas package
import pandas as pd
# making data frame from csv file
info = pd.read_csv("Ex_data.csv", index_col ="Duration")
# retrieving multiple columns by indexing operator
a = info[["Pulse", "Maxpulse"]]
print(a)
```

Output:

```
          Pulse  Maxpulse
Duration
      60    110   130
      60    117   145
       60    103   135
      45    109   175
      45    117   148
      60    102   127
      60    110   136
      45    104   134
      30    109   133
      60    98    124
```

Example2:

```
# importing pandas package
importpandas as pd
# making data frame from csv file
info =pd.read_csv("Ex_data.csv", index_col ="Duration")
# retrieving columns by indexing operator
a =info["Calories"]
print(a)
```

Output:

```
Duration
  60   409.1
  60   479.0
  60   340.0
  45   282.4
  45   406.0
  60   300.0
  60   374.0
  45   253.3
  30   195.1
  60   269.0
  Name: Calories, dtype: float64
```

Set index

The '**set_index**' is used to set the DataFrame index using existing columns. An index can replace the existing index and can also expand the existing index.

It set a list, Series or DataFrame as the index of the DataFrame.

```
info = pd.DataFrame({'Name': ['Parker', 'Terry', 'Smith', 'William'],
'Year': [2011, 2009, 2014, 2010],
'Leaves': [10, 15, 9, 4]})
Print(info)
info.set_index('Name')
info.set_index(['year', 'Name'])
info.set_index([pd.Index([1, 2, 3, 4]), 'year'])
a = pd.Series([1, 2, 3, 4])
info.set_index([a, a**2])
```

Output:

```
          Name      Year   Leaves
1   1     Parker    2011     10
2   4     Terry     2009     15
3   9     Smith     2014     9
```

```
4   16     William     2010     4
```

Reset index

We can also reset the index using the '**reset_index**' command. Let's look at the '**cm**' DataFrame again.

Example:

```
info = pd.DataFrame([('William', 'C'),
('Smith', 'Java'),
('Parker', 'Python'),
('Phill', np.nan)],
index=[1, 2, 3, 4],
columns=('name', 'Language'))
info
info.reset_index()
```

Output:

```
   index      name     Language
0        1    William    C
1        2    Smith      Java
2        3    Parker     Python
3        4    Phill      NaN
```

Python - Pandas Reindex

The main task of the Pandas reindex is to conform DataFrame to a new index with optional filling logic and to place NA/NaN in that location where the values are not present in the previous index. It returns a new object unless the new index is produced as an equivalent to the current one, and the value of **copy** becomes **False**.

Reindexing is used to change the index of the rows and columns of the DataFrame. We can reindex the single or multiple rows by using the reindex() method. Default values in the new index are assigned NaN if it is not present in the DataFrame.

Syntax:

DataFrame.reindex(labels=None, index=None, columns=None, axis=None, method=None, copy=True, level=None, fill_value=nan, limit=None, tolerance=None)

Parameters:

labels: It is an optional parameter that refers to the new labels or the index to conform to the axis that is specified by the 'axis'.

index, columns: It is also an optional parameter that refers to the new labels or the index. It generally prefers an index object for avoiding the duplicate data.

axis: It is also an optional parameter that targets the axis and can be either the axis name or the numbers.

method: It is also an optional parameter that is to be used for filling the holes in the reindexed DataFrame. It can only be applied to the DataFrame or Series with a monotonically increasing/decreasing order.

None: It is a default value that does not fill the gaps.

pad / ffill: It is used to propagate the last valid observation forward to the next valid observation.

backfill / bfill: To fill the gap, It uses the next valid observation.

nearest: To fill the gap, it uses the next valid observation.

copy: Its default value is True and returns a new object as a boolean value, even if the passed indexes are the same.

level : It is used to broadcast across the level, and match index values on the passed MultiIndex level.

fill_value: Its default value is np.NaN and used to fill existing missing (NaN) values. It needs any new element for successful DataFrame alignment, with this value before computation.

limit: It defines the maximum number of consecutive elements that are to be forward or backward fill.

tolerance: It is also an optional parameter that determines the maximum distance between original and new labels for inexact matches. At the matching locations, the values of the index should most satisfy the equation abs(index[indexer] ? target) <= tolerance.

Returns:

It returns reindexed DataFrame.

Example 1:

The below example shows the working of **reindex()** function to reindex the dataframe. In the new index, default values are assigned NaN in the new index that does not have corresponding records in the DataFrame.

Note: We can use fill_value for filling the missing values.

```
import pandas as pd
# Create dataframe
info = pd.DataFrame({"P" : [4, 7, 1, 8, 9],
                "Q" : [6, 8, 10, 15, 11],
                "R" : [17, 13, 12, 16, 14],
                "S" : [15, 19, 7, 21, 9]},
                index =["Parker", "William", "Smith", "Terry", "Phill"])
# Print dataframe
info
```

Output:

```
         P  Q  R  S
Parker   4  6 17 15
William  7  8 13 19
Smith    1  10 12 7
Terry    8  15 16 21
Phill    9  11 14 9
```

Now, we can use the dataframe.reindex() function to reindex the dataframe.

```
# reindexing with new index values
info.reindex(["A", "B", "C", "D", "E"])
```

Output:

```
        P      Q      R      S
A      NaN    NaN    NaN    NaN
B      NaN    NaN    NaN    NaN
C      NaN    NaN    NaN    NaN
D      NaN    NaN    NaN    NaN
E      NaN    NaN    NaN    NaN
```

Notice that the new indexes are populated with NaN values. We can fill in the missing values using the fill_value parameter.

```
# filling the missing values by 100
info.reindex(["A", "B", "C", "D", "E"], fill_value =100)
```

Output:

```
        P      Q      R      S
A      100    100    100    100
B      100    100    100    100
C      100    100    100    100
D      100    100    100    100
E      100    100    100    100
```

Python Pandas - Cleaning Data

Data Cleaning

Data cleaning means fixing bad data in your data set.

Bad data could be:

- Empty cells
- Data in wrong format
- Wrong data
- Duplicates

In this tutorial you will learn how to deal with all of them.

Our Data Set

In the next chapters we will use this data set:

```
     Duration          Date  Pulse  Maxpulse  Calories
  0        60  '2020/12/01'    110       130     409.1
  1        60  '2020/12/02'    117       145     479.0
  2        60  '2020/12/03'    103       135     340.0
  3        45  '2020/12/04'    109       175     282.4
  4        45  '2020/12/05'    117       148     406.0
  5        60  '2020/12/06'    102       127     300.0
  6        60  '2020/12/07'    110       136     374.0
  7       450  '2020/12/08'    104       134     253.3
  8        30  '2020/12/09'    109       133     195.1
  9        60  '2020/12/10'     98       124     269.0
  10       60  '2020/12/11'    103       147     329.3
  11       60  '2020/12/12'    100       120     250.7
  12       60  '2020/12/12'    100       120     250.7
  13       60  '2020/12/13'    106       128     345.3
  14       60  '2020/12/14'    104       132     379.3
  15       60  '2020/12/15'     98       123     275.0
  16       60  '2020/12/16'     98       120     215.2
  17         60  '2020/12/17'    100        120      300.0
  18         45  '2020/12/18'     90        112        NaN
  19         60  '2020/12/19'    103        123      323.0
  20         45  '2020/12/20'     97        125      243.0
  21         60  '2020/12/21'    108        131      364.2
  22         45           NaN    100        119      282.0
  23         60  '2020/12/23'    130        101      300.0
  24         45  '2020/12/24'    105        132      246.0
  25         60  '2020/12/25'    102        126      334.5
  26         60    2020/12/26    100        120      250.0
  27         60  '2020/12/27'     92        118      241.0
  28         60  '2020/12/28'    103        132        NaN
  29         60  '2020/12/29'    100        132      280.0
  30         60  '2020/12/30'    102        129      380.3
  31         60  '2020/12/31'     92        115      243.0
```

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).

Empty Cells

Empty cells can potentially give you a wrong result when you analyze data.

Remove Rows

One way to deal with empty cells is to remove rows that contain empty cells.

This is usually OK, since 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('dirtydata.csv')
new_df = df.dropna()
print(new_df.to_string())
```

In our cleaning examples we will be using a CSV file called 'dirtydata.csv'.

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

If you want 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('dirtydata.csv')
df.dropna(inplace = True)
print(df.to_string())
```

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

Replace Empty Values

Another way of dealing with empty cells is to insert a *new* value instead.

This way you do not have to delete entire rows just because of some empty cells.

The **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('dirtydata.csv')
df.fillna(130, inplace = True)
```

Replace Only For a Specified Columns

The example above replaces all empty cells in the whole Data Frame.

To only 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('dirtydata.csv')
df["Calories"].fillna(130, inplace = True)
```

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('dirtydata.csv')
x = df["Calories"].mean()
df["Calories"].fillna(x, inplace = True)
```

Mean = the average value (the sum of all values divided by number of values).

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

```
import pandas as pd
df = pd.read_csv('dirtydata.csv')
x = df["Calories"].median()
df["Calories"].fillna(x, inplace = True)
```

Median = the value in the middle, after you have sorted all values ascending.

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

```
import pandas as pd

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

x = df["Calories"].mode()[0]

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

Mode = the value that appears most frequently.

Python Pandas – Cleaning: Data of Wrong Format

Data of Wrong Format

Cells with data of wrong format can make it difficult, or even impossible, to analyze data.

To fix it, you have two options: remove the rows, or convert all cells in the columns into the same format.

Convert Into a Correct Format

In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date:

```
    Duration          Date  Pulse  Maxpulse  Calories
0         60  '2020/12/01'    110       130     409.1
1         60  '2020/12/02'    117       145     479.0
2         60  '2020/12/03'    103       135     340.0
3         45  '2020/12/04'    109       175     282.4
4         45  '2020/12/05'    117       148     406.0
5         60  '2020/12/06'    102       127     300.0
6         60  '2020/12/07'    110       136     374.0
7        450  '2020/12/08'    104       134     253.3
8         30  '2020/12/09'    109       133     195.1
9         60  '2020/12/10'     98       124     269.0
10        60  '2020/12/11'    103       147     329.3
11        60  '2020/12/12'    100       120     250.7
12        60  '2020/12/12'    100       120     250.7
13        60  '2020/12/13'    106       128     345.3
14        60  '2020/12/14'    104       132     379.3
15        60  '2020/12/15'     98       123     275.0
16        60  '2020/12/16'     98       120     215.2
17        60  '2020/12/17'    100       120     300.0
18        45  '2020/12/18'     90       112       NaN
19        60  '2020/12/19'    103       123     323.0
20        45  '2020/12/20'     97       125     243.0
21        60  '2020/12/21'    108       131     364.2
22        45           NaN    100       119     282.0
23        60  '2020/12/23'    130       101     300.0
24        45  '2020/12/24'    105       132     246.0
25        60  '2020/12/25'    102       126     334.5
26        60      20201226    100       120     250.0
27        60  '2020/12/27'     92       118     241.0
28        60  '2020/12/28'    103       132       NaN
29        60  '2020/12/29'    100       132     280.0
```

```
30        60  '2020/12/30'    102       129      380.3
31        60  '2020/12/31'     92       115      243.0
```

Let’s try to convert all cells in the 'Date' column into dates.

Pandas has a to_datetime() method for this:

Example: Convert to date:

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

Output:

```
    Duration          Date  Pulse  Maxpulse  Calories
0         60  '2020/12/01'    110       130     409.1
1         60  '2020/12/02'    117       145     479.0
2         60  '2020/12/03'    103       135     340.0
3         45  '2020/12/04'    109       175     282.4
4         45  '2020/12/05'    117       148     406.0
5         60  '2020/12/06'    102       127     300.0
6         60  '2020/12/07'    110       136     374.0
7        450  '2020/12/08'    104       134     253.3
8         30  '2020/12/09'    109       133     195.1
9         60  '2020/12/10'     98       124     269.0
10        60  '2020/12/11'    103       147     329.3
11        60  '2020/12/12'    100       120     250.7
12        60  '2020/12/12'    100       120     250.7
13        60  '2020/12/13'    106       128     345.3
14        60  '2020/12/14'    104       132     379.3
15        60  '2020/12/15'     98       123     275.0
16        60  '2020/12/16'     98       120     215.2
17        60  '2020/12/17'    100       120     300.0
18        45  '2020/12/18'     90       112       NaN
19        60  '2020/12/19'    103       123     323.0
20        45  '2020/12/20'     97       125     243.0
21        60  '2020/12/21'    108       131     364.2
22        45           NaT    100       119     282.0
23        60  '2020/12/23'    130       101     300.0
24        45  '2020/12/24'    105       132     246.0
25        60  '2020/12/25'    102       126     334.5
26        60  '2020/12/26'    100       120     250.0
27        60  '2020/12/27'     92       118     241.0
28        60  '2020/12/28'    103       132       NaN
29        60  '2020/12/29'    100       132     280.0
30        60  '2020/12/30'    102       129     380.3
```

```
  31        60  '2020/12/31'     92       115     243.0
```

As you can see from the result, the date in row 26 was fixed, but the empty date in row 22 got a NaT (Not a Time) value, in other words an empty value. One way to deal with empty values is simply removing the entire row.

Removing Rows

The result from the converting in the example above gave us a NaT value, which can be handled as a NULL value, and we can remove the row by using the dropna() method.

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

```
df.dropna(subset=['Date'], inplace = True)
```

Python Pandas – Cleaning: Wrong Data (fixing)

Wrong Data

“Wrong data” does not have to be “empty cells” or “wrong format”, it can just be wrong, like if someone registered “199” instead of “1.99”.

Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.

If you take a look at our data set, you can see that in row 7, the duration is 450, but for all the other rows the duration is between 30 and 60.

Note: It doesn’t have to be wrong, but taking in consideration that this is the data set of someone’s workout sessions, we conclude with the fact that this person did not work out in **450** minutes.

```
     Duration          Date  Pulse  Maxpulse  Calories
0          60  '2020/12/01'    110       130     409.1
1          60  '2020/12/02'    117       145     479.0
2          60  '2020/12/03'    103       135     340.0
3          45  '2020/12/04'    109       175     282.4
4          45  '2020/12/05'    117       148     406.0
5          60  '2020/12/06'    102       127     300.0
6          60  '2020/12/07'    110       136     374.0
7         450  '2020/12/08'    104       134     253.3
8          30  '2020/12/09'    109       133     195.1
9          60  '2020/12/10'     98       124     269.0
10         60  '2020/12/11'    103       147     329.3
11         60  '2020/12/12'    100       120     250.7
12         60  '2020/12/12'    100       120     250.7
13         60  '2020/12/13'    106       128     345.3
14         60  '2020/12/14'    104       132     379.3
15         60  '2020/12/15'     98       123     275.0
16         60  '2020/12/16'     98       120     215.2
17         60  '2020/12/17'    100       120     300.0
18         45  '2020/12/18'     90       112       NaN
19         60  '2020/12/19'    103       123     323.0
20         45  '2020/12/20'     97       125     243.0
21         60  '2020/12/21'    108       131     364.2
22         45           NaN    100       119     282.0
23         60  '2020/12/23'    130       101     300.0
24         45  '2020/12/24'    105       132     246.0
25         60  '2020/12/25'    102       126     334.5
26         60      20201226    100       120     250.0
27         60  '2020/12/27'     92       118     241.0
28         60  '2020/12/28'    103       132       NaN
```

```
  29         60  '2020/12/29'    100        132      280.0
  30         60  '2020/12/30'    102        129      380.3
  31         60  '2020/12/31'     92        115      243.0
```

How can we fix wrong values, like the one for "Duration" in row 7?

Replacing Values

One way to fix wrong values is to replace them with something else.

In our example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7:

Example: Set "Duration" = 45 in row 7:

```
df.loc[7, 'Duration'] = 45
```

For small data sets you might be able to replace the wrong data one by one, but not for big data sets.

To replace wrong data for larger data sets you can create some rules, e.g. 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.

```
for x in df.index:
  if df.loc[x, "Duration"] > 120:
    df.loc[x, "Duration"] = 120
```

Removing Rows

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

This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses.

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

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

Python Pandas – Cleaning: Removing Duplicates

Discovering Duplicates

Duplicate rows are rows that have been registered more than one time.

```
    Duration          Date  Pulse  Maxpulse  Calories
0         60  '2020/12/01'    110       130     409.1
1         60  '2020/12/02'    117       145     479.0
2         60  '2020/12/03'    103       135     340.0
3         45  '2020/12/04'    109       175     282.4
4         45  '2020/12/05'    117       148     406.0
5         60  '2020/12/06'    102       127     300.0
6         60  '2020/12/07'    110       136     374.0
7        450  '2020/12/08'    104       134     253.3
8         30  '2020/12/09'    109       133     195.1
9         60  '2020/12/10'     98       124     269.0
10        60  '2020/12/11'    103       147     329.3
11        60  '2020/12/12'    100       120     250.7
12        60  '2020/12/12'    100       120     250.7
13        60  '2020/12/13'    106       128     345.3
14        60  '2020/12/14'    104       132     379.3
15        60  '2020/12/15'     98       123     275.0
16        60  '2020/12/16'     98       120     215.2
17        60  '2020/12/17'    100       120     300.0
18        45  '2020/12/18'     90       112       NaN
19        60  '2020/12/19'    103       123     323.0
20        45  '2020/12/20'     97       125     243.0
21        60  '2020/12/21'    108       131     364.2
22        45           NaN    100       119     282.0
23        60  '2020/12/23'    130       101     300.0
24        45  '2020/12/24'    105       132     246.0
25        60  '2020/12/25'    102       126     334.5
26        60      20201226    100       120     250.0
27        60  '2020/12/27'     92       118     241.0
28        60  '2020/12/28'    103       132       NaN
29        60  '2020/12/29'    100       132     280.0
30        60  '2020/12/30'    102       129     380.3
31        60  '2020/12/31'     92       115     243.0
```

By taking a look at our test data set, we can assume that row 11 and 12 are duplicates.

To discover duplicates, we can use the duplicated() method.

The duplicated() method returns a Boolean values for each row:

Example: Returns True for every row that is a duplicate, otherwise False:

```
print(df.duplicated())
```

Removing Duplicates

To remove duplicates, use the drop_duplicates() method.

Example: Remove all duplicates:

```
df.drop_duplicates(inplace = True)
```

Python Pandas - Data Correlations

Finding Relationships

A great aspect of the Pandas module is the corr() method.

The corr() method calculates the relationship between each column in your data set.

The examples in this page uses a CSV file called: 'data.csv'.

Example: Show the relationship between the columns:

df.corr()

Output:

```
           Duration     Pulse  Maxpulse  Calories
Duration   1.000000 -0.155408  0.009403  0.922721
Pulse     -0.155408  1.000000  0.786535  0.025120
Maxpulse   0.009403  0.786535  1.000000  0.203814
Calories   0.922721  0.025120  0.203814  1.000000
```

Note: The corr() method ignores "not numeric" columns.

Result Explained:

The Result of the corr() method is a table with a lot of numbers that represents how well the relationship is between two columns.

The number varies from -1 to 1.

1 means that there is a 1 to 1 relationship (**a perfect correlation**), and for this data set, each time a value went up in the first column, the other one went up as well.

0.9 is also a good relationship, and if you increase one value, the other will probably increase as well.

-0.9 would be just as good relationship as 0.9, but if you increase one value, the other will probably go down.

0.2 means NOT a good relationship, meaning that if one value goes up does not mean that the other will.

What is a good correlation?

It depends on the use, but I think it is safe to say you have to have at least 0.6 (or -0.6) to call it a good correlation.

Perfect Correlation:

We can see that "Duration" and "Duration" got the number 1.000000, which makes sense, each column always has a perfect relationship with itself.

Good Correlation:

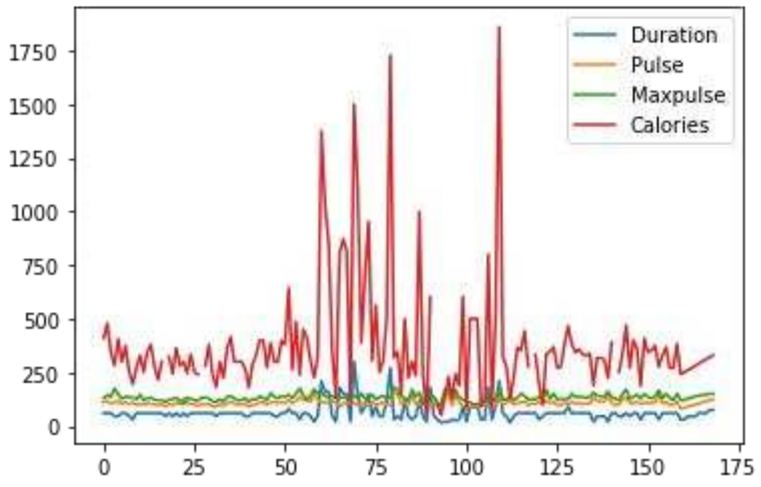
"Duration" and "Calories" got a 0.922721 correlation, which is a very good correlation, and we can predict that the longer you work out, the more calories you burn, and the other way around: if you burned a lot of calories, you probably had a long work out.

Bad Correlation:

"Duration" and "Maxpulse" got a 0.009403 correlation, which is a very bad correlation, meaning that we cannot predict the max pulse by just looking at the duration of the work out, and vice versa.

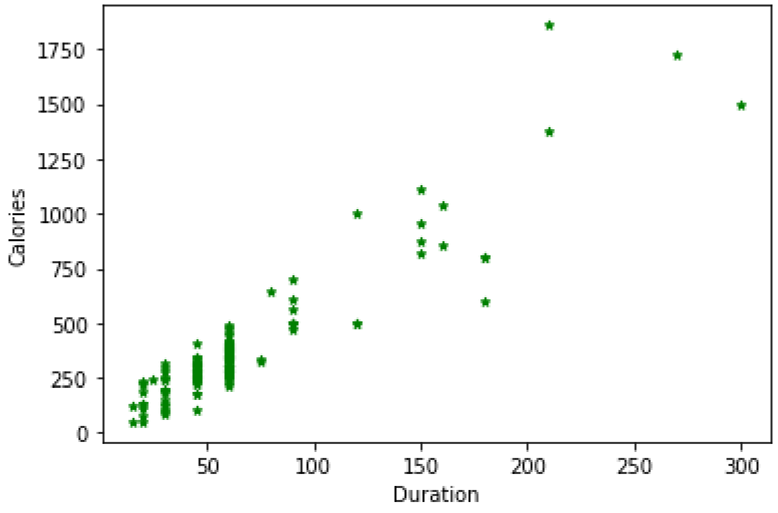
Example1: Data Visualization of 'data.csv' for all columns by using matplotlib.

```
# Data Visualization of 'data.csv' for all columns.
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('data.csv')
df.plot()
plt.show()
```



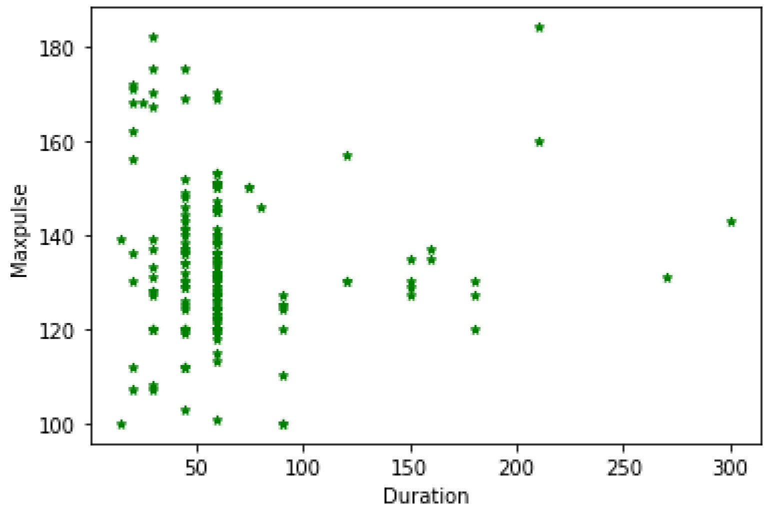
Example 2: A scatterplot where there are is relationship between the columns Duration and Calories.

```
# We learned that the correlation between "Duration" and "Calories" was 0.922721,
# and we conluded with the fact that higher duration means more calories burned.
# By looking at the scatterplot, I would be able to prove it.
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('data.csv')
df.plot(kind = 'scatter', x = 'Duration', y = 'Calories', marker = '*', color = 'g')
plt.show()
```



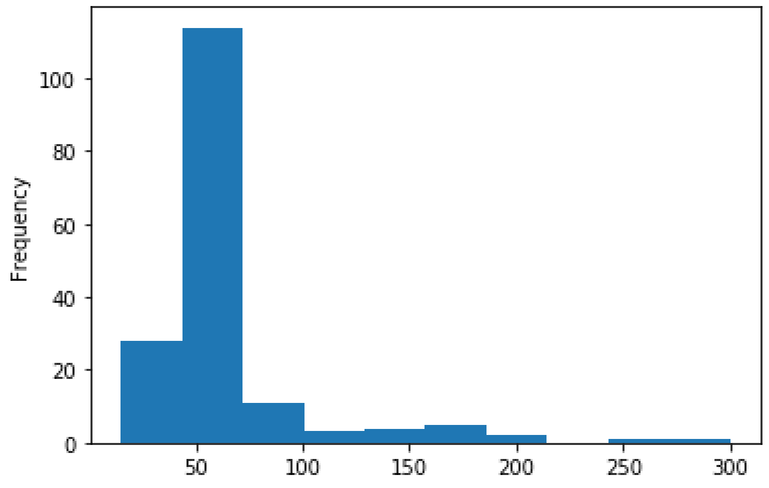
# Example 3: A scatterplot where there are no relationship between the columns:

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

df = pd.read_csv('data.csv')
df.plot(kind = 'scatter', x = 'Duration', y = 'Maxpulse', marker = '*', color = 'green')
plt.show()
```



Example 4: A Histogram that there are over 100 workouts that lasted between 50 and 60 minutes.

```
# The histogram tells us that there are over 100 workouts that lasted between 50 and 60 minutes.
import pandas as pd
import matplotlib.pyplot as plt

df = pd.read_csv('data.csv')
df["Duration"].plot(kind = 'hist')
plt.show()
```



Pandas - head() method

```
# head() method is used to return top n (5 by default) rows of a data frame or series.
import pandas as pd
df = pd.read_csv("data.csv")
print(df.head())
```

Pandas - tail() method

```
# tail() method is used to return last n (5 by default) rows of a data frame or series.
print(df.tail())
```

Pandas - info() method

```
# The info() function is used to print a concise summary of a DataFrame.
print(df.info())
```

Pandas – describe() method

```
# The describe() function is used to generate descriptive statistics
# that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding NaN.
print(df.describe())
```

Pandas – duplicated() method

```
# duplicated() function indicate duplicate Series values.
# The duplicated values are indicated as True values in the resulting Series.
# Either all duplicates, all except the first or all except the last occurrence of duplicates can be indicated.
print(df.duplicated())
```

Pandas – copy() method

```
df1 = df.copy(deep = True)
df2 = df.copy(deep = False)
# When 'deep = True' (default), a new object will be created with a copy of the calling object's data and indices.
```

# Modifications to the data or indices of the copy will not be reflected in the original object.

# When 'deep = False', a new object will be created without copying the calling object's data or index (only references)

# Modifications to the data of the original will be reflected in the shallow copy (and vice versa).

Pandas – iloc() method

```
# Purely integer-location based on indexing for selection by position
print(df.iloc[125])
print(df.iloc[[120, 122, 160]])
# Slicing
print(df.iloc[:10])
print(df.iloc[160:])
```

Pandas – loc() method

```
# primarily label based, for selection by position, but may also be used with a boolean array.
print(df.loc[[10, 150]])
```

Example: Writing DataFrame to .csv file

```
import pandas as pd
data = {'Name': ['Anand Deshpande', 'Vinod Murty', 'Dinesh Bansal','Ashish Kulkarni', ' Kamlesh
Mishra', ' Prashant Sinha'],
        'Hire Date': ['03/15/14', '06/01/15', '05/12/14', '11/01/13', '08/12/14', '05/23/13'],
        'Salary': [50000.0, 65000.0, 45000.0, 70000.0, 48000.0, 66000.0],
        'Leaves Remaining': [10, 8, 10, 3, 7, 8]
       }
info = pd.DataFrame(data)
print('DataFrame Values:\n', info)
# default CSV
csv_data = info.to_csv()
f = open('emp_data.csv','w')
f.write(csv_data)
f.close()
print('\nCSV String Values:\n', csv_data)
```

8.13. Matplotlib:

8.14. Python for Data Visualization

8.15. Introduction to Matplotlib

8.16. Visualization Tools

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 we can use it freely. Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Matplotlib Getting Started

Installation of Matplotlib

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

Install it using this command:

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

If this command fails, then use a python distribution that already has Matplotlib installed, like Anaconda, Spyder etc.

Import Matplotlib

Once Matplotlib is installed, import it in your applications by adding the import *module* statement:

```
import matplotlib
```

Now Matplotlib is imported and ready to use:

Checking Matplotlib Version

The version string is stored under __version__ attribute.

Example:

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

Matplotlib Pyplot

Pyplot

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias:

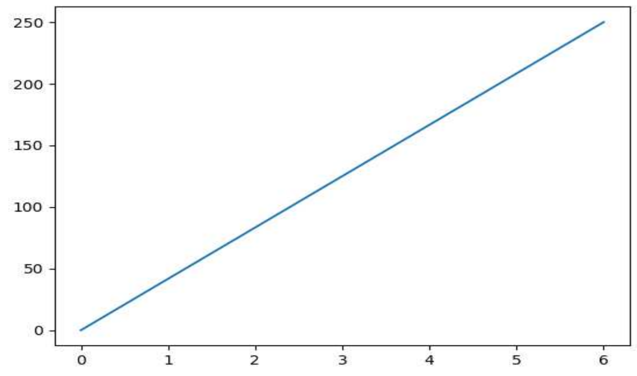
```
import matplotlib.pyplot as plt
```

Now the Pyplot package can be referred to as plt.

Example: Draw a line in a diagram from position (0, 0) to position (6, 250):

```
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()
```

Result:



The **x-axis** is the horizontal axis.

The **y-axis** is the vertical axis.

Plotting Without Line

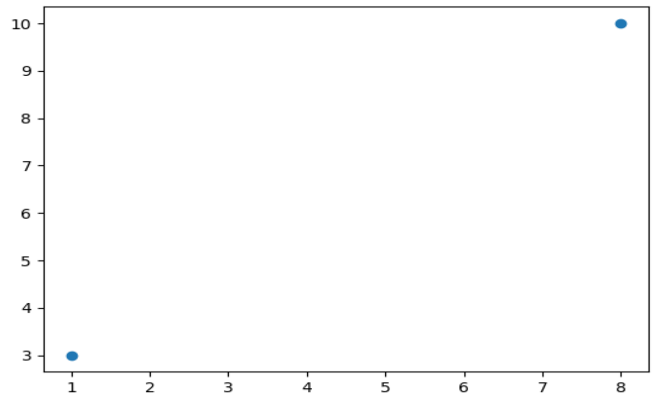
To plot only the markers, you can use *shortcut string notation* parameter 'o', which means 'rings'.

Example: Draw two points in the diagram, one at position (1, 3) and one in position (8, 10):

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

```
xpoints = np.array([1, 8])
ypoints = np.array([3, 10])
plt.plot(xpoints, ypoints, 'o')
plt.show()
```

Result:



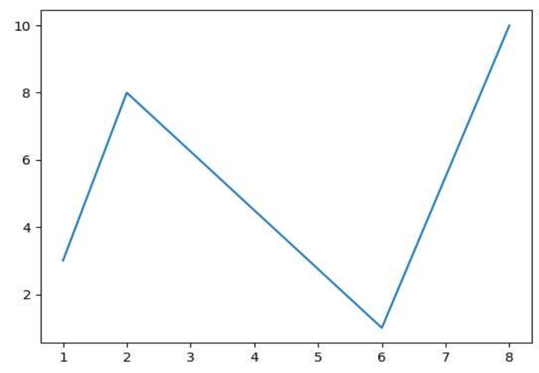
Multiple Points

You can plot as many points as you like, just make sure you have the same number of points in both axis.

Example: Draw a line in a diagram from position (1, 3) to (2, 8) then to (6, 1) and finally to position (8, 10):

```
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()
```

Result:



Default X-Points

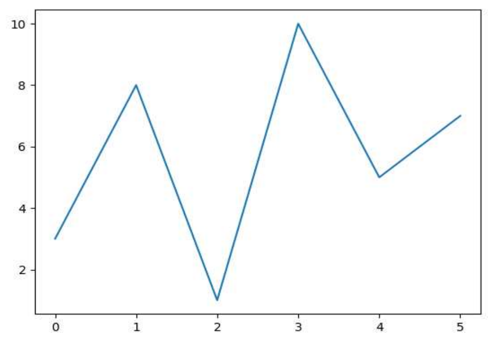
If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, (etc. depending on the length of the y-points.

So, if we take the same example as above, and leave out the x-points, the diagram will look like this:

Example: Plotting without x-points:

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

Result:



The **x-points** in the example above is [0, 1, 2, 3, 4, 5].

Matplotlib Markers

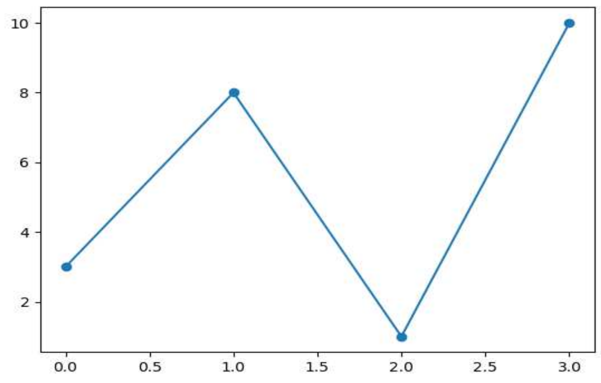
Markers

You can use the keyword argument marker to emphasize each point with a specified marker:

Example: Mark each point with a circle:

```
import matplotlib.pyplot as plt

import numpy as np

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

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

plt.show()
```

Result:



Matplotlib Labels and Title

Create Labels for a Plot

With Pyplot, you can use the **xlabel()** and **ylabel()** functions to set a label for the x- and y-axis.

Set Font Properties for Title and Labels and Position of Title

You can use the fontdict parameter in xlabel(), ylabel(), and title() to set font properties for the title and labels.

Example: Set font properties for the title and labels and also position of the title.

```
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, loc= ‘left’)
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)
plt.plot(x, y)
plt.show()
```

Matplotlib Adding Grid Lines

Add Grid Lines to a Plot

With Pyplot, you can use the grid() function to add grid lines to the plot.

Example: **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()
```

Matplotlib Subplots

Display Multiple Plots

With the **subplots()** function you can draw multiple plots in one figure:

You can draw as many plots you like on one figure, just describe the number of rows, columns, and the index of the plot.

Title

You can add a title to each plot with the **title()** function:

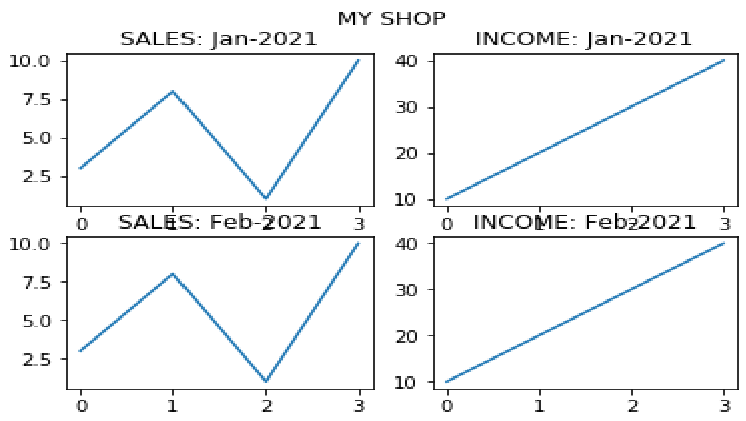
Super Title

You can add a title to the entire figure with the **suptitle()** function:

Example: Draw 4 plots: with title and supertitle

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

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 2, 1)
plt.plot(x,y)
plt.title("SALES: Jan-2021")

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 2, 2)
plt.plot(x,y)
plt.title("INCOME: Jan-2021")

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])
plt.subplot(2, 2, 3)
plt.plot(x,y)
plt.title("SALES: Feb-2021")

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])
plt.subplot(2, 2, 4)
plt.plot(x,y)
plt.title("INCOME: Feb-2021")
plt.suptitle("MY SHOP")
plt.show()
```

Result:



Matplotlib Scatter

Creating Scatter Plots

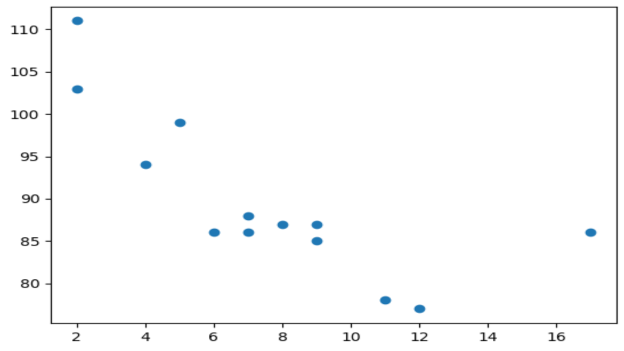
With Pyplot, you can use the scatter() function to draw a scatter plot.

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:

Example: A simple 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])
plt.scatter(x, y)
plt.show()
```

Result:



Matplotlib Bars

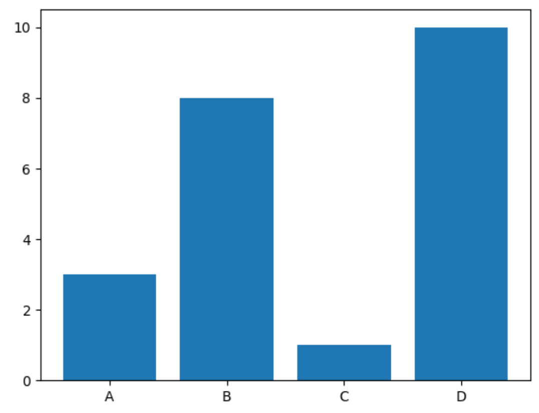
Creating Bars

With Pyplot, you can use the bar() function to draw bar graphs:

Example: Draw 4 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 = 0.4)

plt.show()
```

Result:



The bar() function takes arguments that describes the layout of the bars.

The categories and their values represented by the *first* and *second* argument as arrays.

Horizontal Bars

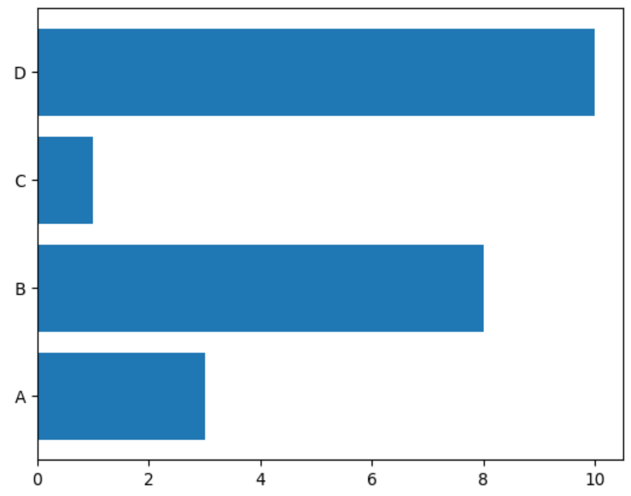
If you want the bars to be displayed horizontally instead of vertically, use the barh() function:

Example: Draw 4 horizontal 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 = 0.4)
plt.show()
```

Result:



Matplotlib Histograms

Histogram

A histogram is a graph showing *frequency* distributions.

It is a graph showing the number of observations within each given interval.

Example: Say you ask for the height of 250 people, you might end up with a histogram like this:

You can read from the histogram that there are approximately:

2 - people from 140 to 145cm

5 - people from 145 to 150cm

15 - people from 151 to 156cm

31 - people from 157 to 162cm

46 - people from 163 to 168cm

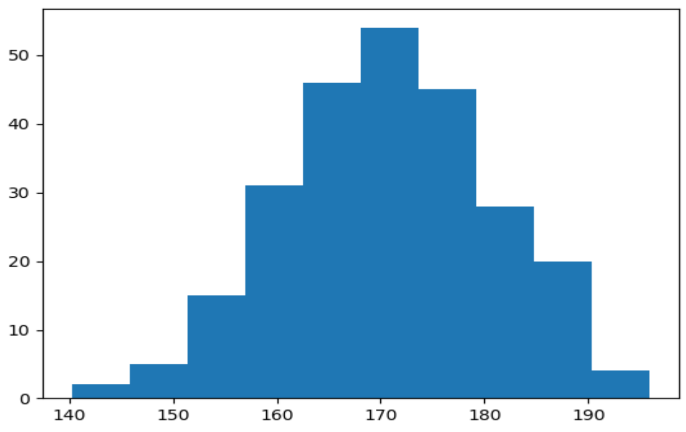
53 - people from 168 to 173cm

45 - people from 173 to 178cm

28 - people from 179 to 184cm

21 - people from 185 to 190cm

4 - people from 190 to 195cm



Create Histogram

In Matplotlib, we use the hist() function to create histograms.

The hist() function will use an array of numbers to create a histogram, the array is sent into the function as an argument.

For simplicity we use NumPy to randomly generate an array with 250 values, where the values will concentrate around 170, and the standard deviation is 10.

Example: A Normal Data Distribution by NumPy:

```
import numpy as np
x = np.random.normal(170, 10, 250)
print(x)
```

Result: This will generate a *random* result, and could look like this:

```
[167.62255766 175.32495609 152.84661337 165.50264047 163.17457988
 162.29867872 172.83638413 168.67303667 164.57361342 180.81120541
 170.57782187 167.53075749 176.15356275 176.95378312 158.4125473
 187.8842668  159.03730075 166.69284332 160.73882029 152.22378865
 164.01255164 163.95288674 176.58146832 173.19849526 169.40206527
 166.88861903 149.90348576 148.39039643 177.90349066 166.72462233
 177.44776004 170.93335636 173.26312881 174.76534435 162.28791953
 166.77301551 160.53785202 170.67972019 159.11594186 165.36992993
 178.38979253 171.52158489 173.32636678 159.63894401 151.95735707
 175.71274153 165.00458544 164.80607211 177.50988211 149.28106703
 179.43586267 181.98365273 170.98196794 179.1093176  176.91855744
 168.32092784 162.33939782 165.18364866 160.52300507 174.14316386
```

```
   163.01947601 172.01767945 173.33491959 169.75842718 198.04834503
   192.82490521 164.54557943 206.36247244 165.47748898 195.26377975
   164.37569092 156.15175531 162.15564208 179.34100362 167.22138242
   147.23667125 162.86940215 167.84986671 172.99302505 166.77279814
   196.6137667  159.79012341 166.5840824  170.68645637 165.62204521
   174.5559345  165.0079216  187.92545129 166.86186393 179.78383824
   161.0973573  167.44890343 157.38075812 151.35412246 171.3107829
   162.57149341 182.49985133 163.24700057 168.72639903 169.05309467
   167.19232875 161.06405208 176.87667712 165.48750185 179.68799986
   158.7913483  170.22465411 182.66432721 173.5675715  176.85646836
   157.31299754 174.88959677 183.78323508 174.36814558 182.55474697
   180.03359793 180.53094948 161.09560099 172.29179934 161.22665588
   171.88382477 159.04626132 169.43886536 163.75793589 157.73710983
   174.68921523 176.19843414 167.39315397 181.17128255 174.2674597
   186.05053154 177.06516302 171.78523683 166.14875436 163.31607668
   174.01429569 194.98819875 169.75129209 164.25748789 180.25773528
   170.44784934 157.81966006 171.33315907 174.71390637 160.55423274
   163.92896899 177.29159542 168.30674234 165.42853878 176.46256226
   162.61719142 166.60810831 165.83648812 184.83238352 188.99833856
   161.3054697  175.30396693 175.28109026 171.54765201 162.08762813
   164.53011089 189.86213299 170.83784593 163.25869004 198.68079225
   166.95154328 152.03381334 152.25444225 149.75522816 161.79200594
   162.13535052 183.37298831 165.40405341 155.59224806 172.68678385
   179.35359654 174.19668349 163.46176882 168.26621173 162.97527574
   192.80170974 151.29673582 178.65251432 163.17266558 165.11172588
   183.11107905 169.69556831 166.35149789 178.74419135 166.28562032
   169.96465166 178.24368042 175.3035525  170.16496554 158.80682882
   187.10006553 178.90542991 171.65790645 183.19289193 168.17446717
   155.84544031 177.96091745 186.28887898 187.89867406 163.26716924
   169.71242393 152.9410412  158.68101969 171.12655559 178.1482624
   187.45272185 173.02872935 163.8047623  169.95676819 179.36887054
   157.01955088 185.58143864 170.19037101 157.221245   168.90639755
   178.7045601  168.64074373 172.37416382 165.61890535 163.40873027
   168.98683006 149.48186389 172.20815568 172.82947206 173.71584064
   189.42642762 172.79575803 177.00005573 169.24498561 171.55576698
   161.36400372 176.47928342 163.02642822 165.09656415 186.70951892
   153.27990317 165.59289527 180.34566865 189.19506385 183.10723435
   173.48070474 170.28701875 157.24642079 157.9096498  176.4248199
]
```

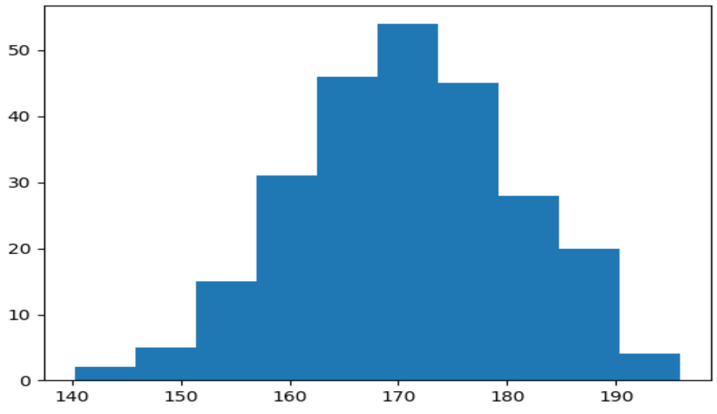
The hist() function will read the array and produce a histogram:

Example: A simple histogram:

```
import matplotlib.pyplot as plt

import numpy as np

x = np.random.normal(170, 10, 250)
```

```
plt.hist(x)
plt.show()
```

Result:



Matplotlib Pie Charts

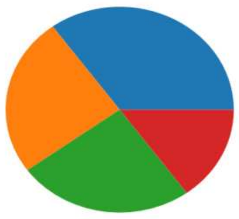
Creating Pie Charts

With Pyplot, you can use the pie() function to draw pie charts:

Example: A simple pie chart:

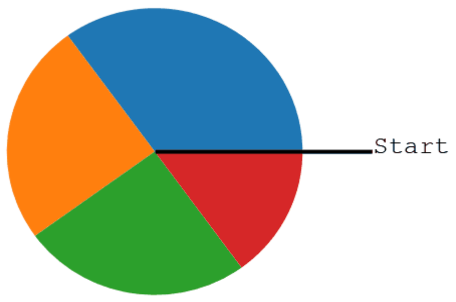
```
import matplotlib.pyplot as plt
import numpy as np
y = np.array([35, 25, 25, 15])
plt.pie(y)
plt.show()
```

Result:



As you can see the pie chart draws one piece (called a wedge) for each value in the array (in this case [35, 25, 25, 15]).

By default the plotting of the first wedge starts from the x-axis and move ***counterclockwise*****:**



Note: The size of each wedge is determined by comparing the value with all the other values, by using this formula:

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

Labels

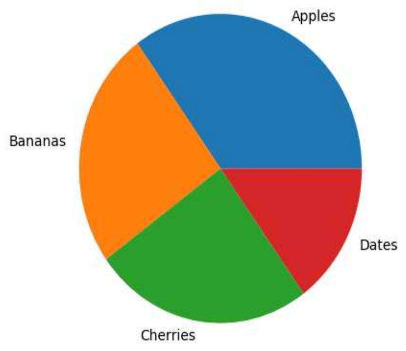
Add labels to the pie chart with the label parameter.

The label parameter must be an array with one label for each wedge:

Example: A simple pie chart:

```
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()
```

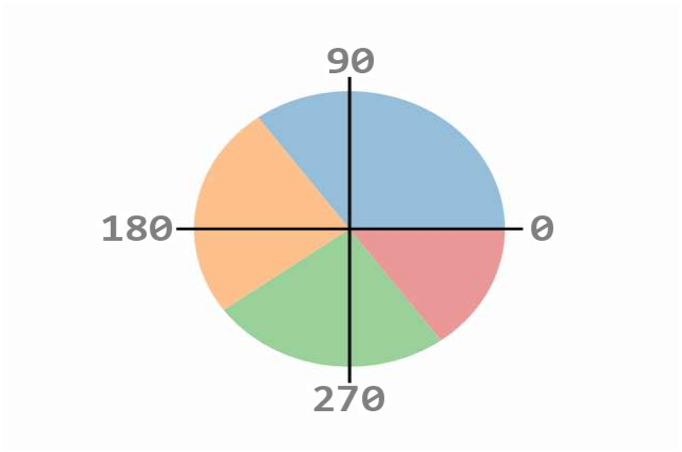
Result:



Start Angle

As mentioned the default start angle is at the x-axis, but you can change the start angle by specifying a startangle parameter.

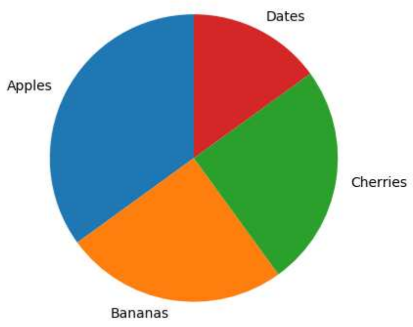
The startangle parameter is defined with an angle in degrees, default angle is 0:



Example: Start the first wedge at 90 degrees:

```
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()
```

Result:



Explode

Maybe you want one of the wedges to stand out? The explode parameter allows you to do that.

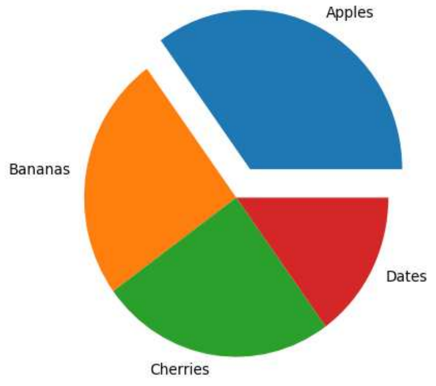
The explode parameter, if specified, and not None, must be an array with one value for each wedge.

Each value represents how far from the center each wedge is displayed:

Example: Pull the "Apples" wedge 0.2 from the center of the pie:

```
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()
```

Result:



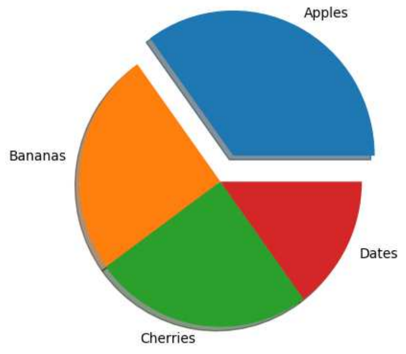
Shadow

Add a shadow to the pie chart by setting the shadows parameter to True:

Example: Add a shadow:

```
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()
```

Result:



Colors

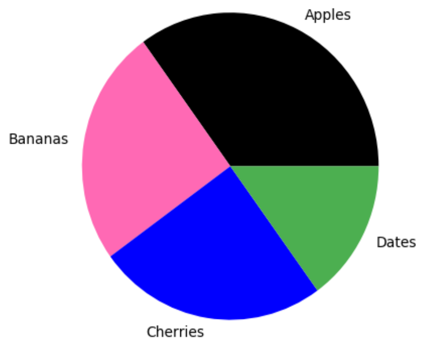
You can set the color of each wedge with the colors parameter.

The colors parameter, if specified, must be an array with one value for each wedge:

Example: Specify a new color 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()
```

Result:



You can use Hexadecimal color values, any of the 140 supported color names, or one of these shortcuts:

'r' - Red

'g' - Green

'b' - Blue

'c' - Cyan

'm' - Magenta

'y' - Yellow

'k' - Black

'w' - White

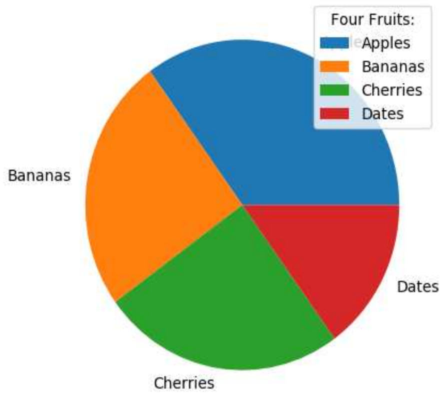
Legend With Header

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

Example: Add a legend with a header:

```
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()
```

Result:


Four Fruits:
Apples
Bananas
Cherries
Dates
Bananas
Dates
Cherries