What is the NumPy library in Python?

NumPy stands for Numerical Python and is one of the most useful scientific libraries in Python programming. It provides support for large multidimensional array objects and various tools to work with them. Various other libraries like Pandas, Matplotlib, and Scikit-learn are built on top of this amazing library.

Arrays are a collection of elements/values, that can have one or more dimensions. An array of one dimension is called a Vector while having two dimensions is called a Matrix. NumPy arrays are called ndarray or N-dimensional arrays and they store elements of the same type and size. It is known for its high-performance and provides efficient storage and data operations as arrays grow in size. NumPy comes pre-installed when you download Anaconda.

But if you want to install NumPy separately on your machine, just type the below command on your terminal:

pip install numpy

Now you need to import the library:

import numpy as np

Python Lists vs NumPy Arrays – What's the Difference?

If you're familiar with Python, you might be wondering why use NumPy arrays when we already have Python lists? After all, these Python lists act as an array that can store elements of various types. This is a perfectly valid question and the answer to this is hidden in the way Python stores an object in memory. A Python object is actually a pointer to a memory location that stores all the details about the object, like bytes and the value. Although this extra information is what makes Python a dynamically typed language, it also comes at a cost which becomes apparent when storing a large collection of objects, like in an array.

Python lists are essentially an array of pointers, each pointing to a location that contains the information related to the element. This adds a lot of overhead in terms of memory and computation. And most of this information is rendered redundant when all the objects stored in the list are of the same type! To overcome this problem, we use NumPy arrays that contain only homogeneous elements, i.e. elements having the same data type. This makes it more efficient at storing and manipulating the array. This difference becomes apparent when the array has a large number of elements, say thousands or millions. Also, with NumPy arrays, you can perform element-wise operations, something which is not possible using Python lists!

This is the reason why NumPy arrays are preferred over Python lists when performing mathematical operations on a large amount of data.

1. Creating a NumPy Array

a. Basic ndarray
b. Array of zeros
c. Array of ones
d. Random numbers in ndarray
e. An array of your choice
f. Imatrix in NumPy
g. Evenly spaced ndarray

Creating a NumPy Array

(a) Basic ndarray:

NumPy arrays are very easy to create given the complex problems they solve. To create a very basic ndarray, you use the np.array() method. All you have to pass are the values of the array as a list:

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

Output:
```
array([1, 2, 3, 4])
```

This array contains integer values. You can specify the type of data in the dtype argument:
```
np.array([1,2,3,4],dtype=np.float32)
```

Output:
```
array([1., 2., 3., 4.], dtype=float32)
```
Since NumPy arrays can contain only homogeneous datatypes, values will be upcast if the types do not match:

```
np.array([1,2.0,3,4])
```

Output:
```
array([1., 2., 3., 4.])
```
Here, NumPy has upcast integer values to float values.

NumPy arrays can be **multi-dimensional** too.

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

Output:
```
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```
Here, we created a 2-dimensional array of values.

Note: A matrix is just a rectangular array of numbers with shape N x M where N is the number of rows and M is the number of columns in the matrix. The one you just saw above is a 2 x 4 matrix.

(b) Array of zeros
NumPy lets you create an array of all zeros using the np.zeros() method. All you have to do is pass the shape of the desired array:

```
np.zeros(5)
```

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

The one above is a 1-D array while the one below is a 2-D array:
```
np.zeros((2,3))
```

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

(c) Array of ones
You could also create an array of all 1s using the np.ones() method:

```
np.ones(5,dtype=np.int32)
```

Output:
```
array([1, 1, 1, 1, 1])
```

(d) Random numbers in ndarrays

Another very commonly used method to create ndarrays is np.random.rand() method. It creates an array of a given shape with random values from [0,1):

```
# random
np.random.rand(2,3)
```

Output:

```
array([[0.95580785, 0.98378873, 0.65133872],
       [0.38330437, 0.16033608, 0.13826526]])
```

(e) An array of your choice

Or, in fact, you can create an array filled with any given value using the np.full() method. Just pass in the shape of the desired array and the value you want:

```
np.full((2,2),7)
```

Output:

```
array([[7, 7],
       [7, 7]])
```

(f) Imatrix in NumPy

Another great method is np.eye() that returns an array with 1s along its diagonal and 0s everywhere else.
An Identity matrix is a square matrix that has 1s along its main diagonal and 0s everywhere else. Below is an Identity matrix of shape 3 x 3.

Note: A square matrix has an N x N shape. This means it has the same number of rows and columns.

```
# identity matrix
np.eye(3)
```

Output:

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

However, NumPy gives you the flexibility to change the diagonal along which the values have to be 1s. You can either move it above the main diagonal:

```
# not an identity matrix
np.eye(3,k=1)
```

Output:

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

Or move it below the main diagonal:

```
np.eye(3,k=-2)
```

Output:

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

Note: A matrix is called the Identity matrix only when the 1s are along the main diagonal and not any other diagonal!

(g) Evenly spaced ndarray

You can quickly get an evenly spaced array of numbers using the np.arange() method:

```
np.arange(5)
```

Output:

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

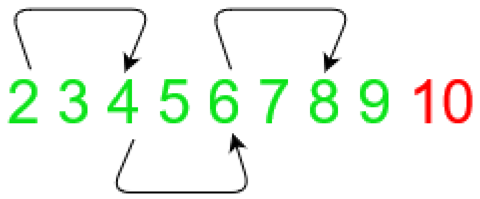
The start, end and step size of the interval of values can be explicitly defined by passing in three numbers as arguments for these values respectively. A point to be noted here is that the interval is defined as [start,end) where the last number will not be included in the array:

```
np.arange(2,10,2)
```

Output:

```
array([2, 4, 6, 8])
```

Alternate elements were printed because the step-size was defined as 2. Notice that 10 was not printed as it was the last element.



Another similar function is np.linspace(), but instead of step size, it takes in the number of samples that need to be retrieved from the interval. A point to note here is that the last number is included in the values returned unlike in the case of np.arange().

```
np.linspace(0,1,5)
```

Output:

```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

Great! Now you know how to create arrays using NumPy. But its also important to know the shape of the array.

2. The Shape and Reshaping of NumPy Array

a. Dimensions of NumPy array
b. Shape of NumPy array
c. Size of NumPy array
d. Reshaping a NumPy array
e. Flattening a NumPy array
f. Transpose of a NumPy array

Once you have created your ndarray, the next thing you would want to do is check the number of axes, shape, and the size of the ndarray.

(a) Dimensions of NumPy arrays

You can easily determine the number of dimensions or axes of a NumPy array using the ndims attribute:

```
# Number of axis
a = np.array([[5,10,15],[20,25,20]])
print('Array :','\n',a)
print('Dimensions :','\n',a.ndim)
```

Output:

```
Array:
 [[ 5 10 15]
 [20 25 20]]
Dimensions:
 2
```

This array has two dimensions: 2 rows and 3 columns.

(b) Shape of NumPy array

The shape is an attribute of the NumPy array that shows how many rows of elements are there along each dimension. You can further index the shape so returned by the ndarray to get value along each dimension:

```
a = np.array([[1,2,3],[4,5,6]])
print('Array :','\n',a)
print('Shape :','\n',a.shape)
print('Rows = ',a.shape[0])
print('Columns = ',a.shape[1])
```

Output:

```
Array:
 [[1 2 3]
 [4 5 6]]
Shape:
 (2, 3)
Rows = 2
Columns = 3
```

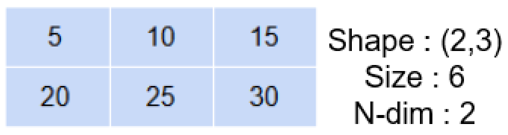
(c) Size of NumPy array

You can determine how many values there are in the array using the size attribute. It just multiplies the number of rows by the number of columns in the ndarray:

```
# size of array
a = np.array([[5,10,15],[20,25,20]])
print('Size of array :',a.size)
print('Manual determination of size of array :',a.shape[0]*a.shape[1])
```

Output:

```
Size of array: 6
Manual determination of size of array: 6
```



5	10	15
20	25	30

Shape : (2,3)
Size : 6
N-dim : 2

(d) Reshaping a NumPy array

Reshaping a ndarray can be done using the np.reshape() method. It changes the shape of the ndarray without changing the data within the ndarray:

```
# reshape
a = np.array([3,6,9,12])
np.reshape(a,(2,2))
```

Output:

```
array([[ 3,  6],
       [ 9, 12]])
```

Here, I reshaped the ndarray from a 1-D to a 2-D ndarray.

While reshaping, if you are unsure about the shape of any of the axis, just input -1. NumPy automatically calculates the shape when it sees a -1:

```
a = np.array([3,6,9,12,18,24])
print('Three rows :','\n',np.reshape(a,(3,-1)))
print('Three columns :','\n',np.reshape(a,(-1,3)))
```

Output:

```
Three rows :
 [[ 3  6]
 [ 9 12]
 [18 24]]
Three columns :
 [[ 3  6  9]
 [12 18 24]]
```

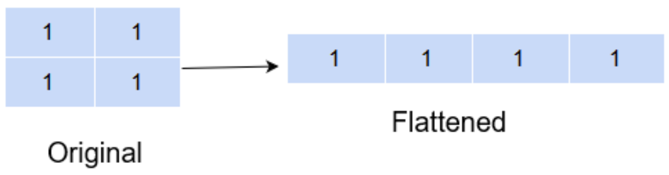
(e) Flattening a NumPy array

Sometimes when you have a multidimensional array and want to collapse it to a single-dimensional array, you can either use the flatten() method or the ravel() method:

```
a = np.ones((2,2))
b = a.flatten()
c = a.ravel()
print('Original shape :', a.shape)
print('Array :','\n', a)
print('Shape after flatten :',b.shape)
print('Array :','\n', b)
print('Shape after ravel :',c.shape)
print('Array :','\n', c)
```

Output:

```
Original shape : (2, 2)
Array :
 [[1. 1.]
 [1. 1.]]
Shape after flatten : (4,)
Array :
 [1. 1. 1. 1.]
Shape after ravel : (4,)
Array :
 [1. 1. 1. 1.]
```



But an important difference between flatten() and ravel() is that the former returns a copy of the original array while the latter returns a reference to the original array. This means any changes made to the array returned from ravel() will also be reflected in the original array while this will not be the case with flatten().

```
b[0] = 0
print(a)
[[1. 1.]
 [1. 1.]]
```

The change made was not reflected in the original array.

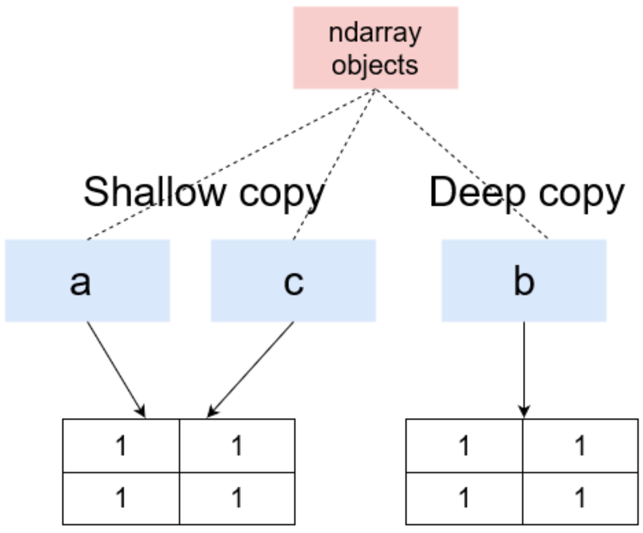
```
c[0] = 0
print(a)
[[0. 1.]
 [1. 1.]]
```

But here, the changed value is also reflected in the original ndarray.

What is happening here is that flatten() creates a Deep copy of the ndarray while ravel() creates a Shallow copy of the ndarray.

Deep copy means that a completely new ndarray is created in memory and the ndarray object returned by flatten() is now pointing to this memory location. Therefore, any changes made here will not be reflected in the original ndarray.

A Shallow copy, on the other hand, returns a reference to the original memory location. Meaning the object returned by ravel() is pointing to the same memory location as the original ndarray object. So, definitely, any changes made to this ndarray will also be reflected in the original ndarray too.



(f) Transpose of a NumPy array

Another very interesting reshaping method of NumPy is the transpose() method. It takes the input array and swaps the rows with the column values, and the column values with the values of the rows:

```
a = np.array([[1,2,3],
[4,5,6]])
b = np.transpose(a)
print('Original','\n','Shape',a.shape,'\n',a)
print('Expand along columns:','\n','Shape',b.shape,'\n',b)
```

Output:

```
Original
 Shape (2, 3)
 [[1 2 3]
 [4 5 6]]
Expand along columns:
 Shape (3, 2)
 [[1 4]
 [2 5]
 [3 6]]
```

On transposing a 2 x 3 array, we got a 3 x 2 array. Transpose has a lot of significance in linear algebra.

3. Expanding and Squeezing a NumPy Array

a. Expanding a NumPy array

b. Squeezing a NumPy array

c. Sorting in NumPy Arrays

(a) Expanding a NumPy array

You can add a new axis to an array using the expand_dims() method by providing the array and the axis along which to expand:

```
# expand dimensions
a = np.array([1,2,3])
b = np.expand_dims(a,axis=0)
c = np.expand_dims(a,axis=1)
print('Original:','\n','Shape',a.shape,'\n',a)
print('Expand along columns:','\n','Shape',b.shape,'\n',b)
print('Expand along rows:','\n','Shape',c.shape,'\n',c)
```

Output:

```
Original:
 Shape (3,)
 [1 2 3]
Expand along columns:
 Shape (1, 3)
 [[1 2 3]]
Expand along rows:
 Shape (3, 1)
 [[1]
 [2]
 [3]]
```

(b) Squeezing a NumPy array

On the other hand, if you instead want to reduce the axis of the array, use the squeeze() method. It removes the axis that has a single entry. This means if you have created a 2 x 2 x 1 matrix, squeeze() will remove the third dimension from the matrix:

```
# squeeze
a = np.array([[[1,2,3],
[4,5,6]]])
b = np.squeeze(a, axis=0)
print('Original','\n','Shape',a.shape,'\n',a)
print('Squeeze array:','\n','Shape',b.shape,'\n',b)
```

Output:

```
Original
 Shape (1, 2, 3)
 [[[1 2 3]
  [4 5 6]]]
Squeeze array:
 Shape (2, 3)
 [[1 2 3]
 [4 5 6]]
```

However, if you already had a 2 x 2 matrix, using squeeze() in that case would give you an error:

```
# squeeze
a = np.array([[1,2,3],
[4,5,6]])
b = np.squeeze(a, axis=0)
print('Original','\n','Shape',a.shape,'\n',a)
print('Squeeze array:','\n','Shape',b.shape,'\n',b)
```

(c) Sorting in NumPy arrays

For any programmer, the time complexity of any algorithm is of prime essence. Sorting is an important and very basic operation that you might well use on a daily basis as a data scientist. So, it is important to use a good sorting algorithm with minimum time complexity.

The NumPy library is a legend when it comes to sorting elements of an array. It has a range of sorting functions that you can use to sort your array elements. It has implemented quicksort, heapsort, mergesort, and timesort for you under the hood when you use the sort() method:

```
a = np.array([1,4,2,5,3,6,8,7,9])
np.sort(a, kind='quicksort')
```

Output:

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

You can even sort the array along any axis you desire:

```
a = np.array([[5,6,7,4],
              [9,2,3,7]])# sort along the column
print('Sort along column :','\n',np.sort(a, kind='mergresort',axis=1))
# sort along the row
print('Sort along row :','\n',np.sort(a, kind='mergresort',axis=0))
```

Output:

```
Sort along column:
 [[4 5 6 7]
 [2 3 7 9]]
Sort along row:
 [[5 2 3 4]
 [9 6 7 7]]
```

4. Indexing and Slicing of NumPy Array

a. Slicing 1-D NumPy arrays
b. Slicing 2-D NumPy arrays
c. Slicing 3-D NumPy arrays
d. Negative slicing of NumPy arrays

So far, we have seen how to create a NumPy array and how to play around with its shape. In this section, we will see how to extract specific values from the array using indexing and slicing.

(a) Slicing 1-D NumPy arrays

Slicing means retrieving elements from one index to another index. All we have to do is to pass the starting and ending point in the index like this: [start: end].

However, you can even take it up a notch by passing the step-size. What is that? Well, suppose you wanted to print every other element from the array, you would define your step-size as 2, meaning get the element 2 places away from the present index.
Incorporating all this into a single index would look something like this: [start:end:step-size].

```
a = np.array([1,2,3,4,5,6])
print(a[1:5:2])
```

Output:

```
[2 4]
```

Notice that the last element did not get considered. This is because slicing includes the start index but excludes the end index.

A way around this is to write the next higher index to the final index value you want to retrieve:

```
a = np.array([1,2,3,4,5,6])
print(a[1:6:2])
```

Output:

```
[2 4 6]
```

If you don't specify the start or end index, it is taken as 0 or array size, respectively, as default. And the step-size by default is 1.

```
a = np.array([1,2,3,4,5,6])
print(a[:6:2])
print(a[1::2])
print(a[1:6:])
```

Output:

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

(b) Slicing 2-D NumPy arrays

Now, a 2-D array has rows and columns so it can get a little tricky to slice 2-D arrays. But once you understand it, you can slice any dimension array!

Before learning how to slice a 2-D array, let's have a look at how to retrieve an element from a 2-D array:

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

```
print(a[0,0])
print(a[1,2])
print(a[1,0])
```

Output:

```
1
6
4
```

Here, we provided the row value and column value to identify the element we wanted to extract. While in a 1-D array, we were only providing the column value since there was only 1 row.

So, to slice a 2-D array, you need to mention the slices for both, the row and the column:

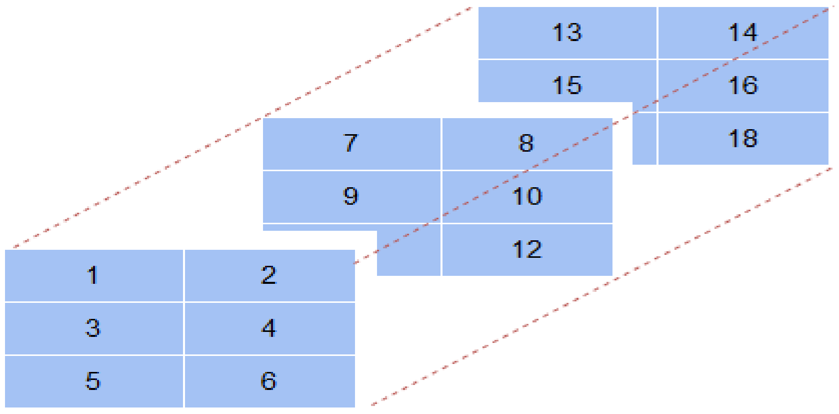
```
a = np.array([[1,2,3],[4,5,6]])
# print first row values
print('First row values :','\n',a[0:1,:])
# with step-size for columns
print('Alternate values from first row:','\n',a[0:1,::2])
#
print('Second column values :','\n',a[:,1::2])
print('Arbitrary values :','\n',a[0:1,1:3])
```

Output:

```
First row values :
 [[1 2 3]]
Alternate values from first row:
 [[1 3]]
Second column values :
 [[2]
 [5]]
Arbitrary values :
 [[2 3]]
```

(c) Slicing 3-D NumPy arrays

So far we haven't seen a 3-D array. Let's first visualize how a 3-D array looks like:



```
a = np.array([[[1,2],[3,4],[5,6]],# first axis array
[[7,8],[9,10],[11,12]],# second axis array
[[13,14],[15,16],[17,18]]])# third axis array
# 3-D array
print(a)
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]

 [[13 14]
  [15 16]
  [17 18]]]
```
In addition to the rows and columns, as in a 2-D array, a 3-D array also has a depth axis where it stacks one 2-D array behind the other. So, when you are slicing a 3-D array, you also need to mention which 2-D array you are slicing. This usually comes as the first value in the index:

```
# value
print('First array, first row, first column value :','\n',a[0,0,0])
print('First array last column :','\n',a[0,:,1])
print('First two rows for second and third arrays :','\n',a[1:,0:2,0:2])
```

Output:
```
First array, first row, first column value :
 1
First array last column :
 [2 4 6]
First two rows for second and third arrays :
 [[[ 7  8]
  [ 9 10]]

 [[13 14]
  [15 16]]]
```
If in case you wanted the values as a single dimension array, you can always use the flatten() method to do the job!

```
print('Printing as a single array :','\n',a[1:,0:2,0:2].flatten())
```

Output:
```
Printing as a single array :
 [ 7  8  9 10 13 14 15 16]
```

(d) Negative slicing of NumPy arrays

An interesting way to slice your array is to use negative slicing. Negative slicing prints elements from the end rather than the beginning. Have a look below:

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

Output:
```
[ 5 10]
```
Here, the last values for each row were printed. If, however, we wanted to extract from the end, we would have to explicitly provide a negative step-size otherwise the result would be an empty list.

```
print(a[:,-1:-3:-1])
```

Output:

```
[[ 5  4]
 [10  9]]
```

Having said that, the basic logic of slicing remains the same, i.e. the end index is never included in the output.

An interesting use of negative slicing is to reverse the original array.

```
a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print('Original array :','\n',a)
print('Reversed array :','\n',a[::-1,::-1])
```

Output:

```
Original array :
 [[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
Reversed array :
 [[10  9  8  7  6]
 [ 5  4  3  2  1]]
```

You can also use the flip() method to reverse an ndarray.

```
a = np.array([[1,2,3,4,5],
[6,7,8,9,10]])
print('Original array :','\n',a)
print('Reversed array vertically :','\n',np.flip(a,axis=1))
print('Reversed array horizontally :','\n',np.flip(a,axis=0))
```

Output:

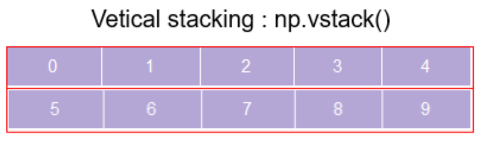
```
Original array :
 [[ 1  2  3  4  5]
 [ 6  7  8  9 10]]
Reversed array vertically :
 [[ 5  4  3  2  1]
 [10  9  8  7  6]]
Reversed array horizontally :
 [[ 6  7  8  9 10]
 [ 1  2  3  4  5]]
```

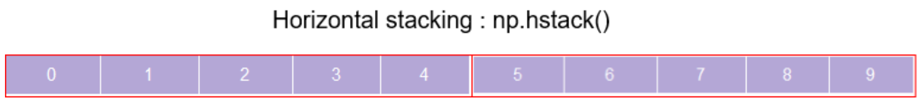
5. Stacking and Concatenating Numpy Arrays

a. Stacking ndarrays

You can create a new array by combining existing arrays. This you can do in two ways:

- Either combine the arrays vertically (i.e. along the rows) using the vstack() method, thereby increasing the number of rows in the resulting array
- Or combine the arrays in a horizontal fashion (i.e. along the columns) using the hstack(), thereby increasing the number of columns in the resultant array





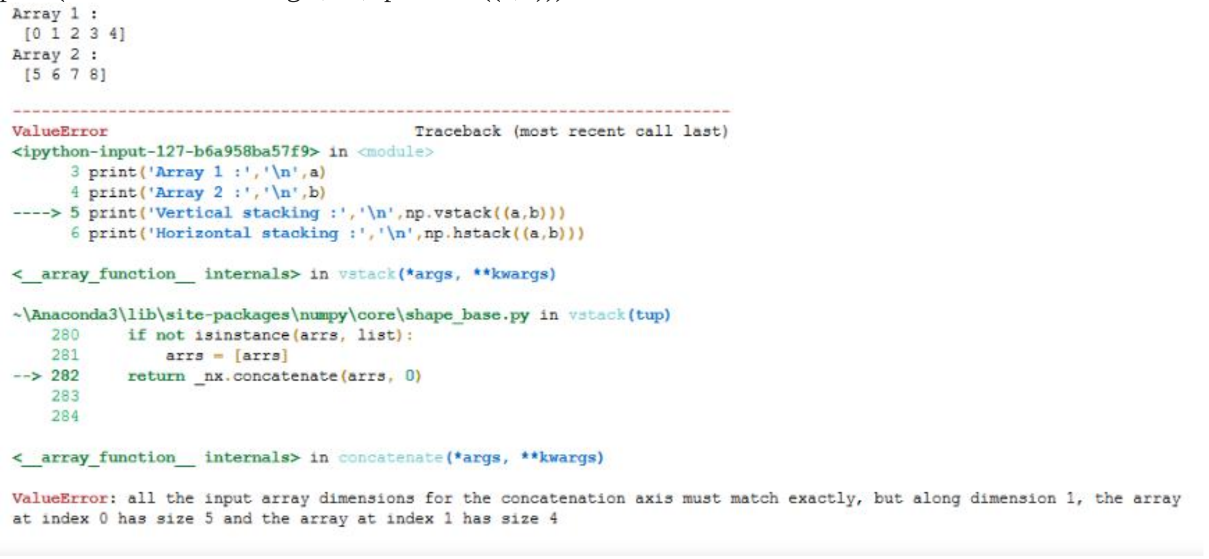
```
a = np.arange(0,5)
b = np.arange(5,10)
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Vertical stacking :','\n',np.vstack((a,b)))
print('Horizontal stacking :','\n',np.hstack((a,b)))
```

Output:

```
Array 1 :
 [0 1 2 3 4]
Array 2 :
 [5 6 7 8 9]
Vertical stacking :
 [[0 1 2 3 4]
 [5 6 7 8 9]]
Horizontal stacking :
 [0 1 2 3 4 5 6 7 8 9]
```

A point to note here is that the axis along which you are combining the array should have the same size otherwise you are bound to get an error!

```
a = np.arange(0,5)
b = np.arange(5,9)
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Vertical stacking :','\n',np.vstack((a,b)))
print('Horizontal stacking :','\n',np.hstack((a,b)))
```



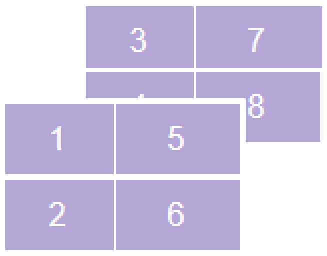
```
Array 1 :
 [0 1 2 3 4]
Array 2 :
 [5 6 7 8]

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-127-b6a958ba57f9> in <module>
      3 print('Array 1 :','\n',a)
      4 print('Array 2 :','\n',b)
----> 5 print('Vertical stacking :','\n',np.vstack((a,b)))
      6 print('Horizontal stacking :','\n',np.hstack((a,b)))

<__array_function__ internals> in vstack(*args, **kwargs)

~\Anaconda3\lib\site-packages\numpy\core\shape_base.py in vstack(tup)
    280     if not isinstance(arrs, list):
    281         arrs = [arrs]
--> 282     return _nx.concatenate(arrs, 0)
    283
    284

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input array dimensions for the concatenation axis must match exactly, but along dimension 1, the array
at index 0 has size 5 and the array at index 1 has size 4
```

Another interesting way to combine arrays is using the dstack() method. It combines array elements index by index and stacks them along the depth axis:

```
a = [[1,2],[3,4]]
b = [[5,6],[7,8]]
c = np.dstack((a,b))
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Dstack :','\n',c)
print(c.shape)
```

Output:

```
Array 1 :
 [[1, 2], [3, 4]]
Array 2 :
 [[5, 6], [7, 8]]
Dstack :
 [[[1 5]
  [2 6]]

 [[3 7]
  [4 8]]]
(2, 2, 2)
```



b. Concatenating ndarrays

While stacking arrays is one way of combining old arrays to get a new one, you could also use the concatenate() method where the passed arrays are joined along an existing axis:

```
a = np.arange(0,5).reshape(1,5)
b = np.arange(5,10).reshape(1,5)
print('Array 1 :','\n',a)
print('Array 2 :','\n',b)
print('Concatenate along rows :','\n',np.concatenate((a,b),axis=0))
print('Concatenate along columns :','\n',np.concatenate((a,b),axis=1))
```

Output:

```
Array 1 :
 [[0 1 2 3 4]]
Array 2 :
 [[5 6 7 8 9]]
Concatenate along rows:
 [[0 1 2 3 4]
 [5 6 7 8 9]]
Concatenate along columns:
 [[0 1 2 3 4 5 6 7 8 9]]
```

The drawback of this method is that the original array must have the axis along which you want to combine. Otherwise, get ready to be greeted by an error.

Another very useful function is the append method that adds new elements to the end of a ndarray. This is obviously useful when you already have an existing ndarray but want to add new values to it.

```
# append values to ndarray
a = np.array([[1,2],
             [3,4]])
```

```
np.append(a,[[5,6]], axis=0)
```

Output:

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

c. Broadcasting in Numpy Arrays

Broadcasting is one of the best features of ndarrays. It lets you perform arithmetics operations between ndarrays of different sizes or between an ndarray and a simple number!

Broadcasting essentially stretches the smaller ndarray so that it matches the shape of the larger ndarray:

```
a = np.arange(10,20,2)
b = np.array([[2],[2]])
print('Adding two different size arrays :','\n',a+b)
print('Multiplying an ndarray and a number :',a*2)
```

Output:

```
Adding two different size arrays:
 [[12 14 16 18 20]
 [12 14 16 18 20]]
Multiplying an ndarray and a number: [20 24 28 32 36]
```

Its working can be thought of like stretching or making copies of the scalar, the number, [2, 2, 2] to match the shape of the ndarray and then perform the operation element-wise. But no such copies are being made. It is just a way of thinking about how broadcasting is working.

This is very useful because it is more efficient to multiply an array with a scalar value rather than another array! It is important to note that two ndarrays can broadcast together only when they are compatible.

Ndarrays are compatible when:

Both have the same dimensions
Either of the ndarrays has a dimension of 1. The one having a dimension of 1 is broadcast to meet the size requirements of the larger ndarray
In case the arrays are not compatible, you will get a Value Error.

```
a = np.ones((3,3))
b = np.array([2])
a+b
```

Output:

```
array([[3., 3., 3.],
       [3., 3., 3.],
       [3., 3., 3.]])
```

Here, the second ndarray was stretched, hypothetically, to a 3 x 3 shape, and then the result was calculated.

6. Perform following operations using pandas

a. Creating dataframe

Introduction

Data science with pandas

Pandas are one of the most popular and powerful data science libraries in Python. It can be considered as the stepping stone for any aspiring data scientist who prefers to code in Python. Even though the library is easy to get started, it can certainly do a wide variety of data manipulation. This makes Pandas one of the handiest data science libraries in the developer’s community. Pandas basically allow the manipulation of large datasets and data frames. It can also be considered as one of the most efficient statistical tools for mathematical computations of tabular data.

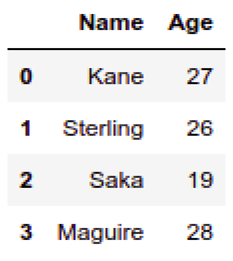
Today. We’ll cover some of the most important and recurring operations that we perform in Pandas. Make no mistake; there are tons of implementations and prospects of Pandas. Here we’ll try to cover some notable aspects only. We’ll use the analogy of Euro Cup 2020 in this tutorial. We’ll start off by creating our own minimal dataset.

Creating our small dataset

Let’s start off by creating a small sample dataset to try out various operations with Pandas. In this tutorial, we shall create a Football data frame that stores the record of 4 players each from Euro Cup 2020’s finalists – England and Italy.

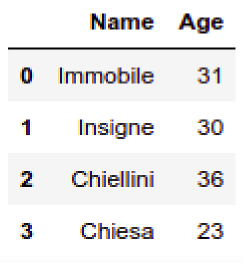
```
import pandas as pd
# Create team data
data_england = {'Name': ['Kane', 'Sterling', 'Saka', 'Maguire'], 'Age': [27, 26, 19, 28]}
data_italy = {'Name': ['Immobile', 'Insigne', 'Chiellini', 'Chiesa'], 'Age': [31, 30, 36, 23]}

# Create Dataframe
df_england = pd.DataFrame(data_england)
df_italy = pd.DataFrame(data_italy)
```

The England data frame looks something like this



	Name	Age
0	Kane	27
1	Sterling	26
2	Saka	19
3	Maguire	28

The Italy data frame looks something like this



	Name	Age
0	Immobile	31
1	Insigne	30
2	Chiellini	36
3	Chiesa	23

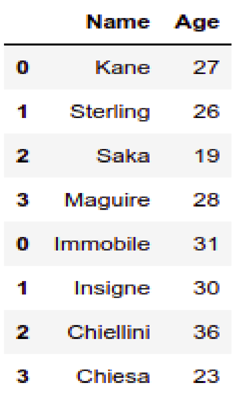
b. concat()

Let’s start by concatenating our two data frames. The word “concatenate” means to “link together in series”. Now that we have created two data frames, let’s try and “concat” them.

We do this by implementing the concat() function.

```
frames = [df_england, df_italy]
both_teams = pd.concat(frames)
both_teams
```

The result looks something like this:



	Name	Age
0	Kane	27
1	Sterling	26
2	Saka	19
3	Maguire	28
0	Immobile	31
1	Insigne	30
2	Chiellini	36
3	Chiesa	23

A similar operation could also be done using the append() function.

Try doing:

```
df_england.append(df_italy)
```

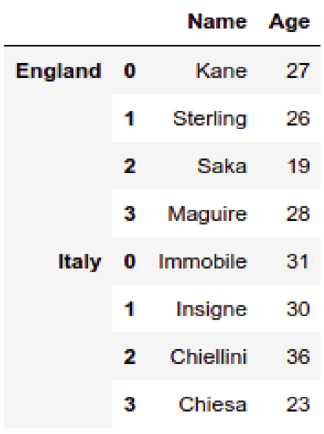
You’ll get the same result!

Now, imagine you wanted to label your original data frames with the associated countries of these players. You can do this by setting specific keys to your data frames.

Try doing:

```
pd.concat(frames, keys=["England", "Italy"])
```

And our result looks like this:



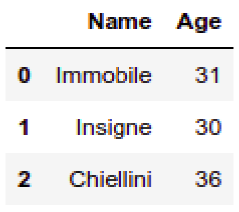
		Name	Age
England	0	Kane	27
	1	Sterling	26
	2	Saka	19
	3	Maguire	28
Italy	0	Immobile	31
	1	Insigne	30
	2	Chiellini	36
	3	Chiesa	23

c. Setting conditions

Conditional statements basically define conditions for data frame columns. There may be situations where you have to filter out various data by applying certain column conditions (numeric or non-numeric). For eg: In an Employee data frame, you might have to list out a bunch of people whose salary is more than Rs. 50000. Also, you might want to filter the people who live in New Delhi, or whose name starts with "A". Let's see a hands-on example.

Imagine we want to filter experienced players from our squad. Let's say, we want to filter those players whose age is greater than or equal to 30. In such case, try doing:

```
both_teams[both_teams["Age"] >= 30]
```

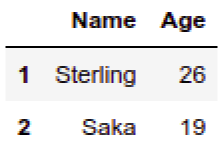


	Name	Age
0	Immobile	31
1	Insigne	30
2	Chiellini	36

Hmm! Looks like Italians are more experienced lads.

Now, let's try to do some string filtration. We want to filter those players whose name starts with "S". This implementation can be done by pandas' startswith() function. Let's try:

```
both_teams[both_teams["Name"].str.startswith('S')]
```



	Name	Age
1	Sterling	26
2	Saka	19

d. Adding a new column

Let's try adding more data to our df_england data frame.

```
club = ['Tottenham', 'Man City', 'Arsenal', 'Man Utd']
# 'Associated Club' is our new column name
df_england['Associated Clubs'] = club
df_england
```

This will add a new column 'Associated Club' to England's data frame.

	Name	**Age**	**Associated Clubs**
0	Kane	27	Tottenham
1	Sterling	26	Man City
2	Saka	19	Arsenal
3	Maguire	28	Man Utd

Let's try to repeat implementing the concat function after updating the data for England.

```
frames = [df_england, df_italy]
both_teams = pd.concat(frames)
both_teams
```

	Name	Age	Associated Clubs
0	Kane	27	Tottenham
1	Sterling	26	Man City
2	Saka	19	Arsenal
3	Maguire	28	Man Utd
0	Immobile	31	NaN
1	Insigne	30	NaN
2	Chiellini	36	NaN
3	Chiesa	23	NaN

Now, this is interesting! Pandas seem to have automatically appended the NaN values in the rows where ‘Associated Clubs’ weren’t explicitly mentioned. In this case, we had only updated ‘Associated Clubs’ data on England. The corresponding values for Italy were set to NaN.

7. Perform following operations using pandas

a. Filling NaN with string

Now, what if, instead of NaN, we want to include some other text? Let’s try adding “No record found” instead of NaN values.

```
both_teams['Associated Clubs'].fillna('No Data Found', inplace=True)
both_teams
```

	Name	Age	Associated Clubs
0	Kane	27	Tottenham
1	Sterling	26	Man City
2	Saka	19	Arsenal
3	Maguire	28	Man Utd
0	Immobile	31	No Data Found
1	Insigne	30	No Data Found
2	Chiellini	36	No Data Found
3	Chiesa	23	No Data Found

b. Sorting based on column values

Sorting operation is straightforward in Pandas. Sorting basically allows the data frame to be ordered by numbers or alphabets (in either increasing or decreasing order). Let’s try and sort the players according to their names.

```
both_teams.sort_values('Name')
```

	Name	Age	Associated Clubs
2	Chiellini	36	No Data Found
3	Chiesa	23	No Data Found
0	Immobile	31	No Data Found
1	Insigne	30	No Data Found
0	Kane	27	Tottenham
3	Maguire	28	Man Utd
2	Saka	19	Arsenal
1	Sterling	26	Man City

Fair enough, we sorted the data frame according to the names of the players. We did this by implementing the sort_values() function.

Let’s sort them by ages:

```
both_teams.sort_values('Age')
```

	Name	Age	Associated Clubs
2	Saka	19	Arsenal
3	Chiesa	23	No Data Found
1	Sterling	26	Man City
0	Kane	27	Tottenham
3	Maguire	28	Man Utd
1	Insigne	30	No Data Found
0	Immobile	31	No Data Found
2	Chiellini	36	No Data Found

Ah, yes! Arsenal’s Bukayo Saka is the youngest lad out there!
Can we also sort by the oldest players? Absolutely!

```
both_teams.sort_values('Age', ascending=False)
```

	Name	Age	Associated Clubs
2	Chiellini	36	No Data Found
0	Immobile	31	No Data Found
1	Insigne	30	No Data Found
3	Maguire	28	Man Utd
0	Kane	27	Tottenham
1	Sterling	26	Man City
3	Chiesa	23	No Data Found
2	Saka	19	Arsenal

c. Pandas “groupby”

Grouping is arguably the most important feature of Pandas. A groupby() function simply groups a particular column. Let’s see a simple example by creating a new data frame.

```
a = {
    'UserID': ['U1001', 'U1002', 'U1001', 'U1001', 'U1003'],
    'Transaction': [500, 300, 200, 300, 700]
}
df_a = pd.DataFrame(a)
df_a
```

	UserID	Transaction
0	U1001	500
1	U1002	300
2	U1001	200
3	U1001	300
4	U1003	700

Notice, we have two columns – UserID and Transaction. You can also see a repeating UserID (U1001). Let's apply a groupby() function to it.
df_a.groupby('UserID').sum()

	Transaction
UserID	
U1001	1000
U1002	300
U1003	700

The function grouped the similar UserIDs and took the sum of those IDs.
If you want to unravel a particular UserID, just try mentioning the value name through get_group().
df_a.groupby('UserID').get_group('U1001')

	UserID	**Transaction**
0	U1001	500
2	U1001	200
3	U1001	300

And this is how we grouped our UserIDs and also checked for a particular ID name.

8. Read the following file formats using pandas
a. Text files
b. CSV files
c. Excel files
d. JSON files

Introduction
I have recently come across a lot of aspiring data scientists wondering why it's so difficult to import different file formats in Python. Most of you might be familiar with the read_csv() function in Pandas but things get tricky from there.

How to read a JSON file in Python? How about an image file? How about multiple files all at once? These are questions you should know the answer to – but might find it difficult to grasp initially.

And mastering these file formats is critical to your success in the data science industry. You'll be working with all sorts of file formats collected from multiple data sources – that's the reality of the modern digital age we live in.
So in this article, I will introduce you to some of the most common file formats that a data scientist should know. We will learn how to read them in Python so that you are well prepared before you enter the battlefield!

Table of Contents

vii. Reading data from Pickle files in Python
viii. Web Scraping with Python
ix. Reading Image files using PIL
x. Read multiple files using Glob

Extracting from Zip Files in Python

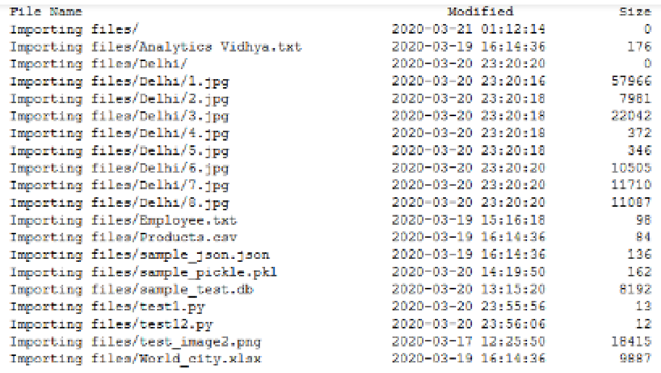
Zip files are a gift from the coding gods. It is like they have fallen from heaven to save our storage space and time. Old school programmers and computer users will certainly relate to how we used to copy gigantic installation files in Zip format!

But technically speaking, ZIP is an archive file format that supports lossless data compression. This means you don't have to worry about your data being lost in the compression-decompression process (Silicon Valley, anyone?).

Here, let's look at how you can open a ZIP folder with Python. For this, you will need the zip file library in Python.

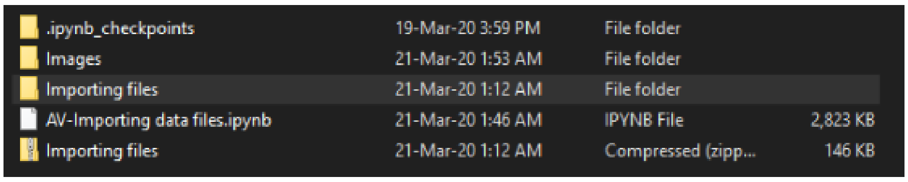
I have zipped all the files required for this article in a separate ZIP folder, so let's extract them!

```
# import zipfile
from zipfile import ZipFile
# path to the zipfile
file = './Importing files.zip'
# read zipfile and extract contents
with ZipFile (file, 'r') as zip:
   zip.printdir()
   zip.extractall()
```



```
File Name                                             Modified             Size
Importing files/                               2020-03-21 01:12:14            0
Importing files/Analytics Vidhya.txt           2020-03-19 16:14:36          176
Importing files/Delhi/                         2020-03-20 23:20:20            0
Importing files/Delhi/1.jpg                    2020-03-20 23:20:16        57966
Importing files/Delhi/2.jpg                    2020-03-20 23:20:18         7981
Importing files/Delhi/3.jpg                    2020-03-20 23:20:18        22042
Importing files/Delhi/4.jpg                    2020-03-20 23:20:18          372
Importing files/Delhi/5.jpg                    2020-03-20 23:20:18          346
Importing files/Delhi/6.jpg                    2020-03-20 23:20:20        10505
Importing files/Delhi/7.jpg                    2020-03-20 23:20:20        11710
Importing files/Delhi/8.jpg                    2020-03-20 23:20:20        11087
Importing files/Employee.txt                   2020-03-19 15:16:18           98
Importing files/Products.csv                   2020-03-19 16:14:36           84
Importing files/sample_json.json               2020-03-19 16:14:36          136
Importing files/sample_pickle.pkl              2020-03-20 14:19:50          162
Importing files/sample_test.db                 2020-03-20 13:15:20         8192
Importing files/test1.py                       2020-03-20 23:55:56           13
Importing files/test12.py                      2020-03-20 23:56:06           12
Importing files/test_image2.png                2020-03-17 12:25:50        18415
Importing files/World_city.xlsx                2020-03-19 16:14:36         9887
```

Once you run the above code, you can view the extracted files in the same folder as your Python script:



.ipynb_checkpoints	19-Mar-20 3:59 PM	File folder	
Images	21-Mar-20 1:53 AM	File folder	
Importing files	21-Mar-20 1:12 AM	File folder	
AV-Importing data files.ipynb	21-Mar-20 1:46 AM	IPYNB File	2,823 KB
Importing files	21-Mar-20 1:12 AM	Compressed (zipp...	146 KB

a. Reading Text Files in Python

Text files are one of the most common file formats to store data. Python makes it very easy to read data from text files.

Python provides the open() function to read files that take in the file path and the file access mode as its parameters. For reading a text file, the file access mode is ‘r’. I have mentioned the other access modes below:

‘w’ – writing to a file
‘r+’ or ‘w+’ – read and write to a file
‘a’ – appending to an already existing file
‘a+’ – append to a file after reading
Python provides us with three functions to read data from a text file:

read(n) – This function reads n bytes from the text files or reads the complete information from the file if no number is specified. It is smart enough to handle the delimiters when it encounters one and separates the sentences
readline(n) – This function allows you to read n bytes from the file but not more than one line of information
readlines() – This function reads the complete information in the file but unlike read(), it doesn’t bother about the delimiting character and prints them as well in a list format
Let us see how these functions differ in reading a text file:

```
# read text file
with open(r'./Importing files/Analytics Vidhya.txt','r') as f:
    print(f.read())
```

Welcome to an article on how to import files in python. We'll work with the following types of files:
1. Text
2. CSV
3. Excel
4. SQL
5. Web data
6. Image

The read() function imported all the data in the file in the correct structured form.

```
# read text file
with open(r'./Importing files/Analytics Vidhya.txt','r') as f:
    print(f.read(10))
```

Welcome to

By providing a number in the read() function, we were able to extract the specified amount of bytes from the file.

```
# read text file
with open(r'./Importing files/Analytics Vidhya.txt','r') as f:
    print(f.readline())
```

Welcome to an article on how to import files in python. We'll work with the following types of files:

Using readline(), only a single line from the text file was extracted.

```
# read text file
with open(r'./Importing files/Analytics Vidhya.txt','r') as f:
    print(f.readlines())
```

```
["Welcome to an article on how to import files in python. We'll work with the following types of files:\n", '1. Text\n', '2. CSV\n', '3. Excel\n', '4. SQL\n', '5. Web data\n', '6. Image']
```

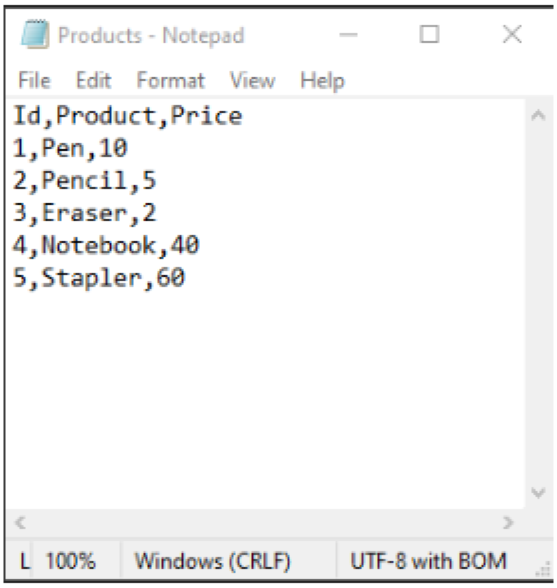
Here, the readline() function extracted all the text file data in a list format.

b. Reading CSV Files in Python

Ah, the good old CSV format. A CSV (or Comma Separated Value) file is the most common type of file that a data scientist will ever work with. These files use a "," as a delimiter to separate the values and each row in a CSV file is a data record.

These are useful to transfer data from one application to another and is probably the reason why they are so commonplace in the world of data science.

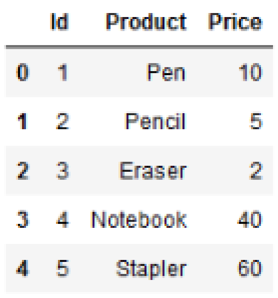
If you look at them in the Notepad, you will notice that the values are separated by commas:


Products - Notepad
File Edit Format View Help
L 100% Windows (CRLF) UTF-8 with BOM

```
Id,Product,Price
1,Pen,10
2,Pencil,5
3,Eraser,2
4,Notebook,40
5,Stapler,60
```

The Pandas library makes it very easy to read CSV files using the read_csv() function:

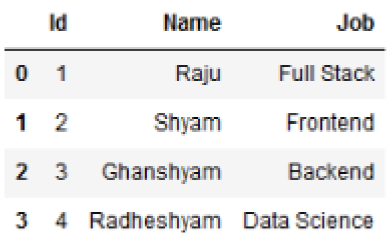
```
# import pandas
import pandas as pd

# read csv file into a DataFrame
df = pd.read_csv(r'./Importing files/Products.csv')
# display DataFrame
df
```



	Id	Product	Price
0	1	Pen	10
1	2	Pencil	5
2	3	Eraser	2
3	4	Notebook	40
4	5	Stapler	60

But CSV can run into problems if the values contain commas. This can be overcome by using different delimiters to separate information in the file, like '\t' or ';', etc. These can also be imported with the read_csv() function by specifying the delimiter in the parameter value as shown below while reading a TSV (Tab Separated Values) file:

```
import pandas as pd

df = pd.read_csv(r'./Importing files/Employee.txt',delimiter='\t')
df
```



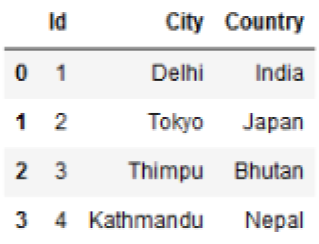
	Id	Name	Job
0	1	Raju	Full Stack
1	2	Shyam	Frontend
2	3	Ghanshyam	Backend
3	4	Radheshyam	Data Science

c. Reading Excel Files in Python

Most of you will be quite familiar with Excel files and why they are so widely used to store tabular data. So I'm going to jump right to the code and import an Excel file in Python using Pandas.

Pandas has a very handy function called read_excel() to read Excel files:

```
# read Excel file into a DataFrame
df = pd.read_excel(r'./Importing files/World_city.xlsx')
# print values
Df
```



	Id	City	Country
0	1	Delhi	India
1	2	Tokyo	Japan
2	3	Thimpu	Bhutan
3	4	Kathmandu	Nepal

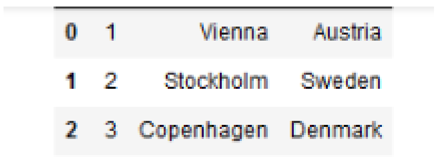
But an Excel file can contain multiple sheets, right? So how can we access them?

For this, we can use the Pandas' ExcelFile() function to print the names of all the sheets in the file:

```
# read Excel sheets in pandas
xl = pd.ExcelFile(r'./Importing files/World_city.xlsx')

# print sheet name
xl.sheet_names

['Asia','Europe','Australia']
```

After doing that, we can easily read data from any sheet we wish by providing its name in the sheet_name parameter in the read_excel() function:

```
# read Europe sheet
df = pd.read_excel(r'./Importing files/World_city.xlsx',sheet_name='Europe')
df
```



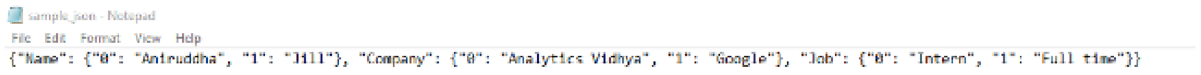
```
0  1     Vienna    Austria
1  2  Stockholm     Sweden
2  3 Copenhagen    Denmark
```

d. Working with JSON Files in Python

JSON (JavaScript Object Notation) files are lightweight and human-readable to store and exchange data. It is easy for machines to parse and generate these files and are based on the JavaScript programming language.

JSON files store data within {} similar to how a dictionary stores it in Python. But their major benefit is that they are language-independent, meaning they can be used with any programming language – be it Python, C or even Java!

This is how a JSON file looks:



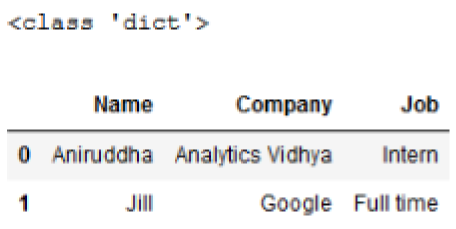
```
{"Name": {"0": "Aniruddha", "1": "Jill"}, "Company": {"0": "Analytics Vidhya", "1": "Google"}, "Job": {"0": "Intern", "1": "Full time"}}
```

Python provides a json module to read JSON files. You can read JSON files just like simple text files. However, the read function, in this case, is replaced by json.load() function that returns a JSON dictionary.

Once you have done that, you can easily convert it into a Pandas dataframe using the pandas.DataFrame() function:

```
import json

# open json file
with open('./Importing files/sample_json.json','r') as file:
    data = json.load(file)

# json dictionary
print(type(data))

# loading into a DataFrame
```

```
df_json = pd.DataFrame(data)
df_json
```

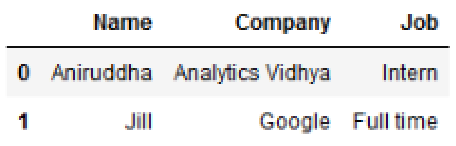


```
<class 'dict'>
```

	Name	Company	Job
0	Aniruddha	Analytics Vidhya	Intern
1	Jill	Google	Full time

But you can even load the JSON file directly into a dataframe using the pandas.read_json() function as shown below:

```
# reading directly into a DataFrame usind pd.read_json()
path = './Importing files/sample_json.json'
df = pd.read_json(path)
df
```



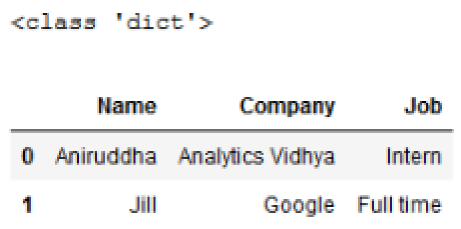
	Name	Company	Job
0	Aniruddha	Analytics Vidhya	Intern
1	Jill	Google	Full time

Reading Data from Pickle Files in Python

Pickle files are used to store the serialized form of Python objects. This means objects like list, set, tuple, dict, etc. are converted to a character stream before being stored on the disk. This allows you to continue working with the objects later on. These are particularly useful when you have trained your machine learning model and want to save them to make predictions later on.

So, if you serialized the files before saving them, you need to de-serialize them before you use them in your Python programs. This is done using the pickle.load() function in the pickle module. But when you open the pickle file with Python’s open() function, you need to provide the ‘rb’ parameter to read the binary file.

```
import pickle

with open('./Importing files/sample_pickle.pkl','rb') as file:
    data = pickle.load(file)

# pickle data
print(type(data))

df_pkl = pd.DataFrame(data)
df_pkl
```



```
<class 'dict'>

      Name        Company          Job
0  Aniruddha  Analytics Vidhya   Intern
1       Jill            Google  Full time
```

9. Read the following file formats
a. Pickle files
b. Image files using PIL
c. Multiple files using Glob
d. Importing data from database

a, b. Reading Image Files using PIL
The advent of Convolutional Neural Networks (CNN) has opened the flood gates to working in the computer vision domain and solving problems like object detection, object classification, generating new images and what not!

But before you jump on to working with these problems, you need to know how to open your images in Python. Let's see how we can do that by retrieving images from the webpage that we stored in our local folder.

You will need the Python PIL (Python Image Library) for this job.

Simply call the open() function in the Image module of PIL and pass in the path to your image:

```
from PIL import Image

# filename = r'C:\Users\Dell\Desktop\Analytics Vidhya\Delhi\1.jpg'
filename = r'./Importing files/Delhi/1.jpg'
Image.open(filename)
```

c. Read Multiple Files using Glob
And now, what if you want to read multiple files in one go? That's quite a common challenge in data science projects.

Python’s Glob module lets you traverse through multiple files in the same location. Using glob.glob(), we can import all the files from our local folder that match a special pattern.

These filename patterns can be made using different wildcards like “*” (for matching multiple characters), “?” (for matching any single character), or ‘[0-9]’ (for matching any number). Let’s see glob in action below.

When importing multiple .py files from the same directory as your Python script, we can use the “*” wildcard:

```
for i in glob.glob('.\Importing files\*.py'):
    print(i)
```

```
.\Import files\test1.py
.\Import files\test2.py
```

When importing only a 5 character long Python file, we can use the “?” wildcard:

```
for i in glob.glob('.\Importing files\?????.py'):
    print(i)
```

```
.\Import files\test1.py
```

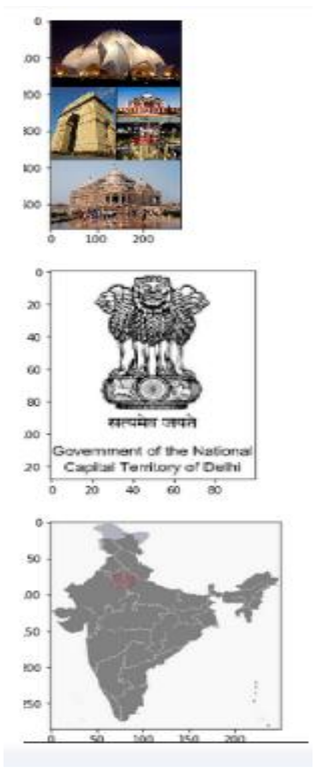
When importing an image file containing a number in the filename, we can use the “[0-9]” wildcard:

```
for i in glob.glob('./Importing files/test_image[0-9].png'):
    print(i)
```

```
.\Import files\test_image2.png
```

Earlier, we imported a few images from the Wikipedia page on Delhi and saved them in a local folder. I will retrieve these images using the glob module and then display them using the PIL library:

```
import cv2
import matplotlib.pyplot as plt

# import glob

filepath = r'./Importing files/Delhi'

images = glob.glob(filepath+'\*.jpg')

for i in images[:3]:
    im = Image.open(i)
    plt.imshow(im)
    plt.show()
```



d. Importing Data from a Database using Python

When you are working on a real-world project, you would need to connect your program to a database to retrieve data. There is no way around it (that's why learning SQL is an important part of your data science journey).

Data in databases is stored in the form of tables and these systems are known as Relational database management systems (RDBMS). However, connecting to RDBMS and retrieving the data from it can prove to be quite a challenging task. Here's the good news – we can easily do this using Python's built-in modules!

One of the most popular RDBMS is SQLite. It has many plus points:

Lightweight database and hence it is easy to use in embedded software
35% faster reading and writing compared to the File System
No intermediary server required. Reading and writing are done directly from the database files on the disk
Cross-platform database file format. This means a file written on one machine can be copied to and used on a different machine with a different architecture
There are many more reasons for its popularity. But for now, let's connect with an SQLite database and retrieve our data!

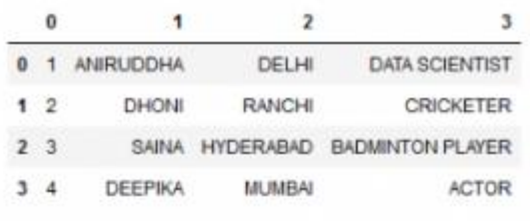
You will need to import the sqlite3 module to use SQLite. Then, you need to work through the following steps to access your data:

Create a connection with the database connect(). You need to pass the name of your database to access it. It returns a Connection object
Once you have done that, you need to create a cursor object using the cursor() function. This will allow you to implement SQL commands with which you can manipulate your data

You can execute the commands in SQL by calling the execute() function on the cursor object. Since we are retrieving data from the database, we will use the SELECT statement and store the query in an object
Store the data from the object into a dataframe by either calling fetchone(), for one row, or fecthall(), for all the rows, function on the object
And just like that, you have retrieved the data from the database into a Pandas dataframe!

A good practice is to save/commit your transactions using the commit() function even if you are only reading the data.

```
import pandas as pd
import sqlite3

# open engine connection
con=sqlite3.connect('./Importing files/sample_test.db')

# create a cursor object
cur = con.cursor()

# Perform query: rs
rs = cur.execute('select * from TEST')

# Save results of the query to DataFrame: df
df = pd.DataFrame(rs.fetchall())

# Close connection
con.commit()

# Print head of DataFrame df
Df
```



```
   0          1          2                 3
0  1  ANIRUDDHA      DELHI    DATA SCIENTIST
1  2      DHONI     RANCHI         CRICKETER
2  3      SAINA  HYDERABAD  BADMINTON PLAYER
3  4    DEEPIKA     MUMBAI             ACTOR
```

10. Demonstrate web scraping using python

Web Scraping using Python

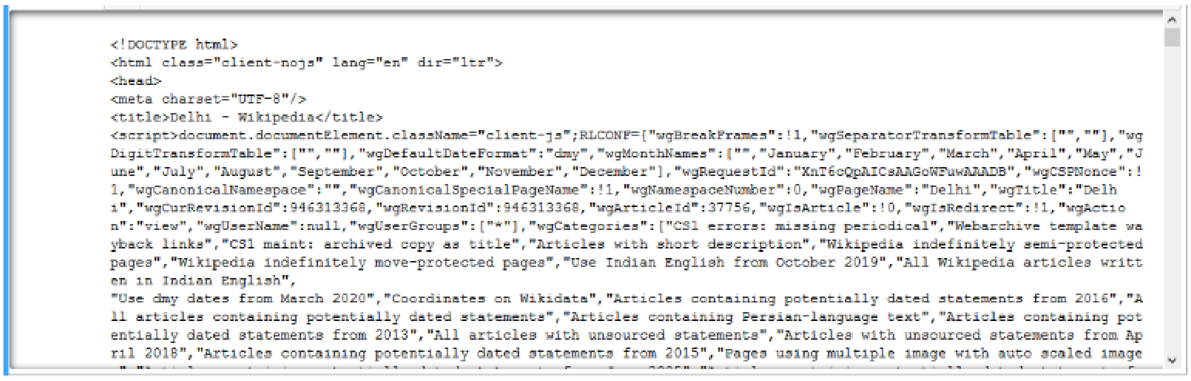
Web Scraping refers to extracting large amounts of data from the web. This is important for a data scientist who has to analyze large amounts of data.

Python provides a very handy module called requests to retrieve data from any website. The requests.get() function takes in a URL as its parameter and returns the HTML response as its output. The way it works is summarized in the following steps:

It packages the Get request to retrieve data from webpage
Sends the request to the server
Receives the HTML response and stores in a response object

For this example, I want to show you a bit about my city – Delhi. So, I will retrieve data from the Wikipedia page on Delhi:

```
import requests

# url = "https://weather.com/en-IN/weather/tenday/l/aff9460b9160c73ff01769fd83ae82cf37cb27fb7eb73c70b91257d413147b69"
url = "https://en.wikipedia.org/wiki/Delhi"

# response object
resp = requests.get(url)

# using text attribute of the response object, return the HTML of webpage as string
text = resp.text
print(text)
```



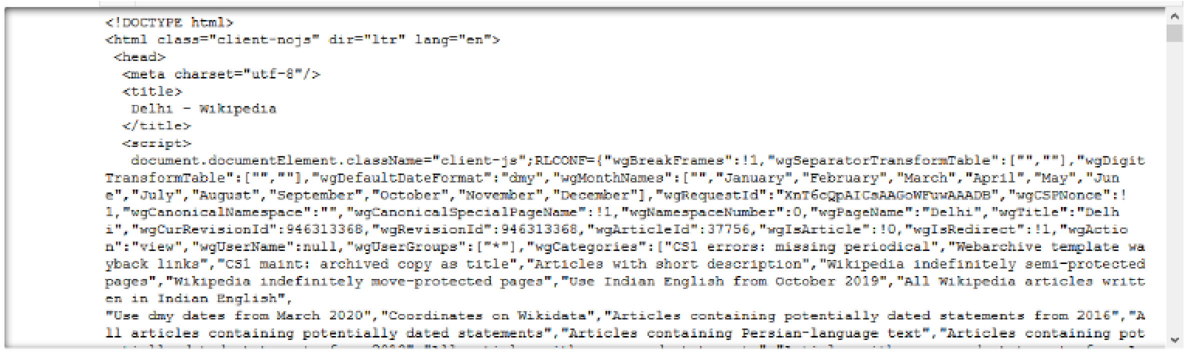
```
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>Delhi - Wikipedia</title>
<script>document.documentElement.className="client-js";RLCONF={"wgBreakFrames":!1,"wgSeparatorTransformTable":["",""],"wg
DigitTransformTable":["",""],"wgDefaultDateFormat":"dmy","wgMonthNames":["","January","February","March","April","May","J
une","July","August","September","October","November","December"],"wgRequestId":"XnT6cQpAICsAAGoWFuwAAADB","wgCSPNonce":!
1,"wgCanonicalNamespace":"","wgCanonicalSpecialPageName":!1,"wgNamespaceNumber":0,"wgPageName":"Delhi","wgTitle":"Delh
i","wgCurRevisionId":946313368,"wgRevisionId":946313368,"wgArticleId":37756,"wgIsArticle":!0,"wgIsRedirect":!1,"wgActio
n":"view","wgUserName":null,"wgUserGroups":["*"],"wgCategories":["CS1 errors: missing periodical","Webarchive template wa
yback links","CS1 maint: archived copy as title","Articles with short description","Wikipedia indefinitely semi-protected
pages","Wikipedia indefinitely move-protected pages","Use Indian English from October 2019","All Wikipedia articles writt
en in Indian English",
"Use dmy dates from March 2020","Coordinates on Wikidata","Articles containing potentially dated statements from 2016","A
ll articles containing potentially dated statements","Articles containing Persian-language text","Articles containing pot
entially dated statements from 2013","All articles with unsourced statements","Articles with unsourced statements from Ap
ril 2018","Articles containing potentially dated statements from 2015","Pages using multiple image with auto scaled image
```

But as you can see, the data is not very readable. The tree-like structure of the HTML content retrieved by our request is not very comprehensible. To improve this readability, Python has another wonderful library called BeautifulSoup.

BeautifulSoup is a Python library for parsing the tree-like structure of HTML and extracting data from the HTML document.

```
import requests
from bs4 import BeautifulSoup

# url
# url = "https://weather.com/en-IN/weather/tenday/l/aff9460b9160c73ff01769fd83ae82cf37cb27fb7eb73c70b91257d413147b69"
url = "https://en.wikipedia.org/wiki/Delhi"

# Package the request, send the request and catch the response: r
r = requests.get(url)

# Extracts the response as html: html_doc
html_doc = r.text

# Create a BeautifulSoup object from the HTML: soup
soup = BeautifulSoup(html_doc)
```

```
# Print the response
print(soup.prettify())
```



```
<!DOCTYPE html>
<html class="client-nojs" dir="ltr" lang="en">
 <head>
  <meta charset="utf-8"/>
  <title>
   Delhi - Wikipedia
  </title>
  <script>
   document.documentElement.className="client-js";RLCONF={"wgBreakFrames":!1,"wgSeparatorTransformTable":["",""],"wgDigit
TransformTable":["",""],"wgDefaultDateFormat":"dmy","wgMonthNames":["","January","February","March","April","May","Jun
e","July","August","September","October","November","December"],"wgRequestId":"XnT6cQpAICsAAGoWFuwAAADB","wgCSPNonce":!
1,"wgCanonicalNamespace":"","wgCanonicalSpecialPageName":!1,"wgNamespaceNumber":0,"wgPageName":"Delhi","wgTitle":"Delh
i","wgCurRevisionId":946313368,"wgRevisionId":946313368,"wgArticleId":37756,"wgIsArticle":!0,"wgIsRedirect":!1,"wgActio
n":"view","wgUserName":null,"wgUserGroups":["*"],"wgCategories":["CS1 errors: missing periodical","Webarchive template wa
yback links","CS1 maint: archived copy as title","Articles with short description","Wikipedia indefinitely semi-protected
pages","Wikipedia indefinitely move-protected pages","Use Indian English from October 2019","All Wikipedia articles writt
en in Indian English",
"Use dmy dates from March 2020","Coordinates on Wikidata","Articles containing potentially dated statements from 2016","A
ll articles containing potentially dated statements","Articles containing Persian-language text","Articles containing pot
```

You must have noticed the difference in the output. We have a more structured output in this case!

Now, we can extract the title of the webpage by calling the title() function of our soup object:

```
title = soup.title
title
```

```
<title> Delhi - wikipedia</title>
```

The webpage has a lot of pictures of the famous monuments in Delhi and other things related to Delhi. Let's try and store these in a local folder.

We will need the Python urllib library to retrieve the URL of the images that we want to store. It has a urllib.request() function that is used for opening and reading URLs. Calling the urlretrieve() function on this object allows us to download objects denoted by the URL to a local file:

```
import urllib

# function to save image from the passed URL
def download_img(url, i):

#     folder = r'C:\Users\Dell\Desktop\Analytics Vidhya\Delhi\\'
    folder = r'./Importing files/Delhi/'

    # define the file path to store images
    filepath = folder + str(i) +'.jpg'

    # retrieve the image from the URL and save in the folder
    urllib.request.urlretrieve(url,filepath)
```

The images are stored in the "img" tag in HTML. These can be found by calling find_all() on the soup object. After this, we can iterate over the image and get its source by calling the get() function on the image object. The rest is handled by our download function:

```
images = soup.find_all('img')
i = 1
```

```
for image in images[2:10]:
    try:
        download_img('https:'+image.get('src'), i)
        i = i+1
    except:
        continue
```

11. Perform following preprocessing techniques on loan prediction dataset

a. Feature Scaling
b. Feature Standardization
c. Label Encoding
d. One Hot Encoding

Introduction

This article primarily focuses on data pre-processing techniques in python. Learning algorithms have affinity towards certain data types on which they perform incredibly well. They are also known to give reckless predictions with unscaled or unstandardized features. Algorithm like XGBoost, specifically requires dummy encoded data while algorithm like decision tree doesn’t seem to care at all (sometimes)!

In simple words, pre-processing refers to the transformations applied to your data before feeding it to the algorithm. In python, scikit-learn library has a pre-built functionality under sklearn.preprocessing. There are many more options for pre-processing which we’ll explore.

After finishing this article, you will be equipped with the basic techniques of data pre-processing and their in-depth understanding. For your convenience, I’ve attached some resources for in-depth learning of machine learning algorithms and designed few exercises to get a good grip of the concepts.

Practical Guide on Data Preprocessing in Python using Scikit Learn

Available Data set

For this article, I have used a subset of the Loan Prediction (missing value observations are dropped) data set

Note : Testing data that you are provided is the subset of the training data from Loan Prediction problem.

Now, lets get started by importing important packages and the data set.

```
# Importing pandas
>> import pandas as pd
# Importing training data set
>> X_train=pd.read_csv('X_train.csv')
>> Y_train=pd.read_csv('Y_train.csv')
# Importing testing data set
>> X_test=pd.read_csv('X_test.csv')
>> Y_test=pd.read_csv('Y_test.csv')
```

Lets take a closer look at our data set.

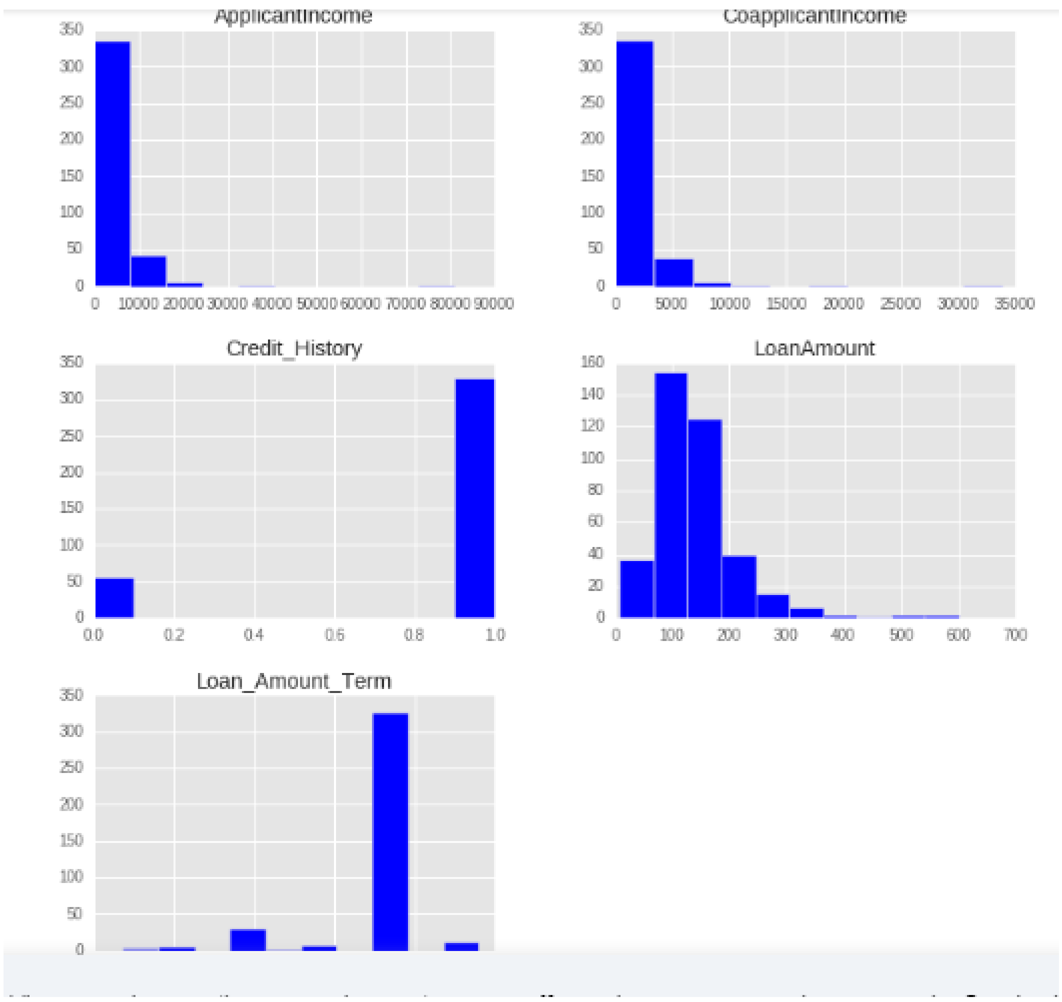
```
>> print (X_train.head())
```

```
      Loan_ID Gender Married Dependents Education Self_Employed
15   LP001032   Male      No          0  Graduate            No
248  LP001824   Male     Yes          1  Graduate            No
590  LP002928   Male     Yes          0  Graduate            No
246  LP001814   Male     Yes          2  Graduate            No
388  LP002244   Male     Yes          0  Graduate            No

     ApplicantIncome  CoapplicantIncome  LoanAmount  Loan_Amount_Term
15              4950                0.0       125.0             360.0
248             2882             1843.0       123.0             480.0
590             3000             3416.0        56.0             180.0
246             9703                0.0       112.0             360.0
388             2333             2417.0       136.0             360.0

     Credit_History Property_Area
15              1.0         Urban
248             1.0     Semiurban
590             1.0     Semiurban
246             1.0         Urban
388             1.0         Urban
```

Feature Scaling

Feature scaling is the method to limit the range of variables so that they can be compared on common grounds. It is performed on continuous variables. Lets plot the distribution of all the continuous variables in the data set.

```
>> import matplotlib.pyplot as plt
>> X_train[X_train.dtypes[(X_train.dtypes=="float64")|(X_train.dtypes=="int64")]
                    .index.values].hist(figsize=[11,11])
```



After understanding these plots, we infer that ApplicantIncome and CoapplicantIncome are in similar range (0-50000$) where as LoanAmount is in thousands and it ranges from 0 to 600$. The story for Loan_Amount_Term is completely different from other variables because its unit is months as opposed to other variables where the unit is dollars.

If we try to apply distance based methods such as kNN on these features, feature with the largest range will dominate the outcome results and we'll obtain less accurate predictions. We can overcome this trouble using feature scaling. Let's do it practically.

Resources: Check out this article on kNN for better understanding.

Lets try out kNN on our data set to see how well it will perform. Here is a live coding window to get you started. You can run the codes and get the output in this window itself:

```
# Importing libraries
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Importing training data set
X_train=pd.read_csv('X_train.csv')
Y_train=pd.read_csv('Y_train.csv')
print("\nShape of train set:", X_train.shape, Y_train.shape)
```

```
# Importing testing data set
X_test=pd.read_csv('X_test.csv')
Y_test=pd.read_csv('Y_test.csv')
print("\nShape of test set:", X_test.shape, Y_test.shape)

# Initializing and Fitting a k-NN model
knn=KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train[['ApplicantIncome','CoapplicantIncome','LoanAmount','Loan_Amount_Term'
,'Credit_History']],Y_train.values.ravel())

# Checking the performance of our model on the testing data set
print("\nAccuracy score on test set :",
accuracy_score(Y_test,knn.predict(X_test[['ApplicantIncome','CoapplicantIncome','LoanAm
ount', 'Loan_Amount_Term','Credit_History']])))
```

# We got around 61% of correct prediction which is not bad but in real world practices will this be enough ? Can we deploy this model in real world problem? To answer this question lets take a look at distribution of Loan_Status in train data set.

```
print("\nDistribution of Loan_Status in train set :")
print(Y_train.Target.value_counts()/Y_train.Target.count())
```

# There are 70% of approved loans, since there are more number of approved loans we will generate a prediction where all the loans are approved and lets go ahead and check the accuracy of our prediction

```
print("\nDistribution of Loan_Status in predictions on the test set :")
print(Y_test.Target.value_counts()/Y_test.Target.count())
```

## NOTE: You can run the rest of the code given below in this blog using this live coding window.

```
# Importing libraries
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Importing training data set
X_train=pd.read_csv('X_train.csv')
Y_train=pd.read_csv('Y_train.csv')
print("\nShape of train set:", X_train.shape, Y_train.shape)

# Importing testing data set
X_test=pd.read_csv('X_test.csv')
Y_test=pd.read_csv('Y_test.csv')
print("\nShape of test set:", X_test.shape, Y_test.shape)

# Initializing and Fitting a k-NN model
knn=KNeighborsClassifier(n_neighbors=5)
```

knn.fit(X_train[['ApplicantIncome','CoapplicantIncome','LoanAmount','Loan_Amount_Term','Credit_History']],Y_train.values.ravel())

# Checking the performance of our model on the testing data set
print("\nAccuracy score on test set :", accuracy_score(Y_test,knn.predict(X_test[['ApplicantIncome','CoapplicantIncome','LoanAmount', 'Loan_Amount_Term','Credit_History']])))

# We got around 61% of correct prediction which is not bad but in real world practices will this be enough ? Can we deploy this model in real world problem? To answer this question lets take a look at distribution of Loan_Status in train data set.

print("\nDistribution of Loan_Status in train set :")
print(Y_train.Target.value_counts()/Y_train.Target.count())

# There are 70% of approved loans, since there are more number of approved loans we will generate a prediction where all the loans are approved and lets go ahead and check the accuracy of our prediction

print("\nDistribution of Loan_Status in predictions on the test set :")
print(Y_test.Target.value_counts()/Y_test.Target.count())

## NOTE: You can run the rest of the code given below in this blog using this live coding window.

Wow !! we got an accuracy of 63% just by guessing, What is the meaning of this, getting better accuracy than our prediction model ?

This might be happening because of some insignificant variable with larger range will be dominating the objective function. We can remove this problem by scaling down all the features to a same range. sklearn provides a tool MinMaxScaler that will scale down all the features between 0 and 1. Mathematical formula for MinMaxScaler is.

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Lets try this tool on our problem.

# Importing MinMaxScaler and initializing it
>> from sklearn.preprocessing import MinMaxScaler
>> min_max=MinMaxScaler()
# Scaling down both train and test data set
>> X_train_minmax=min_max.fit_transform(X_train[['ApplicantIncome', 'CoapplicantIncome',
'LoanAmount', 'Loan_Amount_Term', 'Credit_History']])
>> X_test_minmax=min_max.fit_transform(X_test[['ApplicantIncome', 'CoapplicantIncome',
'LoanAmount', 'Loan_Amount_Term', 'Credit_History']])
Now, that we are done with scaling, lets apply kNN on our scaled data and check its accuracy.

```
# Fitting k-NN on our scaled data set
>> knn=KNeighborsClassifier(n_neighbors=5)
>> knn.fit(X_train_minmax,Y_train)
# Checking the model's accuracy
>> accuracy_score(Y_test,knn.predict(X_test_minmax))
Out : 0.75
```

Great !! Our accuracy has increased from 61% to 75%. This means that some of the features with larger range were dominating the prediction outcome in the domain of distance based methods(kNN).

It should be kept in mind while performing distance based methods we must attempt to scale the data, so that the feature with lesser significance might not end up dominating the objective function due to its larger range. In addition, features having different unit should also be scaled thus providing each feature equal initial weightage and at the end we will have a better prediction model.

Exercise 1
Try to do the same exercise with a logistic regression model(parameters : penalty=’l2′,C=0.01) and provide your accuracy before and after scaling in the comment section.

Feature Standardization
Before jumping to this section I suggest you to complete Exercise 1.

In the previous section, we worked on the Loan_Prediction data set and fitted a kNN learner on the data set. After scaling down the data, we have got an accuracy of 75% which is very considerably good. I tried the same exercise on Logistic Regression and I got the following result :

Before Scaling : 61%

After Scaling : 63%

The accuracy we got after scaling is close to the prediction which we made by guessing, which is not a very impressive achievement. So, what is happening here? Why hasn’t the accuracy increased by a satisfactory amount as it increased in kNN?

Resources : Go through this article on Logistic Regression for better understanding.

Here is the answer:

In logistic regression, each feature is assigned a weight or coefficient (Wi). If there is a feature with relatively large range and it is insignificant in the objective function then logistic regression will itself assign a very low value to its co-efficient, thus neutralizing the dominant effect of that particular feature, whereas distance based method such as kNN does not have this inbuilt strategy, thus it requires scaling.

Aren’t we forgetting something ? Our logistic model is still predicting with an accuracy almost closer to a guess.

Now, I’ll be introducing a new concept here called standardization. Many machine learning algorithms in sklearn requires standardized data which means having zero mean and unit variance.

Standardization (or Z-score normalization) is the process where the features are rescaled so that they’ll have the properties of a standard normal distribution with $\mu=0$ and $\sigma=1$, where μ is the mean (average) and σ is the standard deviation from the mean. Standard scores (also called z scores) of the samples are calculated as follows:

$$z = \frac{x - \mu}{\sigma}$$

Elements such as l1 ,l2 regularizer in linear models (logistic comes under this category) and RBF kernel in SVM in objective function of learners assumes that all the features are centered around zero and have variance in the same order.

Features having larger order of variance would dominate on the objective function as it happened in the previous section with the feature having large range. As we saw in the Exercise 1 that without any preprocessing on the data the accuracy was 61%, lets standardize our data apply logistic regression on that. Sklearn provides scale to standardize the data.

```
# Standardizing the train and test data
>> from sklearn.preprocessing import scale
>> X_train_scale=scale(X_train[['ApplicantIncome', 'CoapplicantIncome',
            'LoanAmount', 'Loan_Amount_Term', 'Credit_History']])
>> X_test_scale=scale(X_test[['ApplicantIncome', 'CoapplicantIncome',
           'LoanAmount', 'Loan_Amount_Term', 'Credit_History']])
# Fitting logistic regression on our standardized data set
>> from sklearn.linear_model import LogisticRegression
>> log=LogisticRegression(penalty='l2',C=.01)
>> log.fit(X_train_scale,Y_train)
# Checking the model's accuracy
>> accuracy_score(Y_test,log.predict(X_test_scale))
```

Out : 0.75

We again reached to our maximum score that was attained using kNN after scaling. This means standardizing the data when using a estimator having l1 or l2 regularization helps us to increase the accuracy of the prediction model. Other learners like kNN with euclidean distance measure, k-means, SVM, perceptron, neural networks, linear discriminant analysis, principal component analysis may perform better with standardized data.

Though, I suggest you to understand your data and what kind of algorithm you are going to apply on it; over the time you will be able to judge weather to standardize your data or not.

Note : Choosing between scaling and standardizing is a confusing choice, you have to dive deeper in your data and learner that you are going to use to reach the decision. For starters, you can try both the methods and check cross validation score for making a choice.

Resources : Go through this article on cross validation for better understanding.

Exercise 2

Try to do the same exercise with SVM model and provide your accuracy before and after standardization in the comment section.

Resources : Go through this article on support vector machines for better understanding.

Label Encoding

In previous sections, we did the pre-processing for continuous numeric features. But, our data set has other features too such as Gender, Married, Dependents, Self_Employed and Education. All these categorical features have string values. For example, Gender has two levels either Male or Female. Lets feed the features in our logistic regression model.

```
# Fitting a logistic regression model on whole data
>> log=LogisticRegression(penalty='l2',C=.01)
>> log.fit(X_train,Y_train)
# Checking the model's accuracy
>> accuracy_score(Y_test,log.predict(X_test))
Out : ValueError: could not convert string to float: Semiurban
```

We got an error saying that it cannot convert string to float. So, what’s actually happening here is learners like logistic regression, distance based methods such as kNN, support vector machines, tree based methods etc. in sklearn needs numeric arrays. Features having string values cannot be handled by these learners.

Sklearn provides a very efficient tool for encoding the levels of a categorical features into numeric values. LabelEncoder encode labels with value between 0 and n_classes-1.

Lets encode all the categorical features.

```
# Importing LabelEncoder and initializing it
>> from sklearn.preprocessing import LabelEncoder
>> le=LabelEncoder()
# Iterating over all the common columns in train and test
>> for col in X_test.columns.values:
    # Encoding only categorical variables
    if X_test[col].dtypes=='object':
    # Using whole data to form an exhaustive list of levels
    data=X_train[col].append(X_test[col])
    le.fit(data.values)
```

```
        X_train[col]=le.transform(X_train[col])
        X_test[col]=le.transform(X_test[col])
```

All our categorical features are encoded. You can look at your updated data set using X_train.head(). We are going to take a look at Gender frequency distribution before and after the encoding.

```
Before: Male 318
         Female 66
Name: Gender, dtype: int64
After: 1 318
        0 66
Name: Gender, dtype: int64
```

Now that we are done with label encoding, lets now run a logistic regression model on the data set with both categorical and continuous features.

```
# Standardizing the features
>> X_train_scale=scale(X_train)
>> X_test_scale=scale(X_test)
# Fitting the logistic regression model
>> log=LogisticRegression(penalty='l2',C=.01)
>> log.fit(X_train_scale,Y_train)
# Checking the models accuracy
>> accuracy_score(Y_test,log.predict(X_test_scale))
Out : 0.75
```

Its working now. But, the accuracy is still the same as we got with logistic regression after standardization from numeric features. This means categorical features we added are not very significant in our objective function.

Exercise 3

Try out decision tree classifier with all the features as independent variables and comment your accuracy.

Resources: Go through this article on decision trees for better understanding.

One-Hot Encoding

One-Hot Encoding transforms each categorical feature with n possible values into n binary features, with only one active.

Most of the ML algorithms either learn a single weight for each feature or it computes distance between the samples. Algorithms like linear models (such as logistic regression) belongs to the first category.

Lets take a look at an example from loan_prediction data set. Feature Dependents have 4 possible values 0,1,2 and 3+ which are then encoded without loss of generality to 0,1,2 and 3.

We, then have a weight “W” assigned for this feature in a linear classifier,which will make a decision based on the constraints W*Dependents + K > 0 or eqivalently W*Dependents < K.

Let f(w)= W*Dependents

Possible values that can be attained by the equation are 0, W, 2W and 3W. A problem with this equation is that the weight “W” cannot make decision based on four choices. It can reach to a decision in following ways:

- All leads to the same decision (all of them <K or vice versa)
- 3:1 division of the levels (Decision boundary at f(w)>2W)
- 2:2 division of the levels (Decision boundary at f(w)>W)

Here we can see that we are loosing many different possible decisions such as the case where “0” and “2W” should be given same label and “3W” and “W” are odd one out.

This problem can be solved by One-Hot-Encoding as it effectively changes the dimensionality of the feature “Dependents” from one to four, thus every value in the feature “Dependents” will have their own weights. Updated equation for the decison would be f'(w) < K.

where, f'(w) = W1*D_0 + W2*D_1 + W3*D_2 + W4*D_3

All four new variable has boolean values (0 or 1).

The same thing happens with distance based methods such as kNN. Without encoding, distance between “0” and “1” values of Dependents is 1 whereas distance between “0” and “3+” will be 3, which is not desirable as both the distances should be similar. After encoding, the values will be new features (sequence of columns is 0,1,2,3+) : [1,0,0,0] and [0,0,0,1] (initially we were finding distance between “0” and “3+”), now the distance would be √2.

For tree based methods, same situation (more than two values in a feature) might effect the outcome to extent but if methods like random forests are deep enough, it can handle the categorical variables without one-hot encoding.

Now, lets take look at the implementation of one-hot encoding with various algorithms.

Lets create a logistic regression model for classification without one-hot encoding.

```
# We are using scaled variable as we saw in previous section that
# scaling will effect the algo with l1 or l2 reguralizer
>> X_train_scale=scale(X_train)
>> X_test_scale=scale(X_test)
# Fitting a logistic regression model
>> log=LogisticRegression(penalty='l2',C=1)
>> log.fit(X_train_scale,Y_train)
# Checking the model's accuracy
>> accuracy_score(Y_test,log.predict(X_test_scale))
```

Out : 0.73958333333333337

Now we are going to encode the data.

```
>> from sklearn.preprocessing import OneHotEncoder
>> enc=OneHotEncoder(sparse=False)
>> X_train_1=X_train
>> X_test_1=X_test
>> columns=['Gender', 'Married', 'Dependents', 'Education','Self_Employed',
       'Credit_History', 'Property_Area']
>> for col in columns:
    # creating an exhaustive list of all possible categorical values
    data=X_train[[col]].append(X_test[[col]])
    enc.fit(data)
    # Fitting One Hot Encoding on train data
    temp = enc.transform(X_train[[col]])
    # Changing the encoded features into a data frame with new column names
    temp=pd.DataFrame(temp,columns=[(col+"_"+str(i)) for i in data[col]
        .value_counts().index])
    # In side by side concatenation index values should be same
    # Setting the index values similar to the X_train data frame
    temp=temp.set_index(X_train.index.values)
    # adding the new One Hot Encoded varibales to the train data frame
    X_train_1=pd.concat([X_train_1,temp],axis=1)
    # fitting One Hot Encoding on test data
    temp = enc.transform(X_test[[col]])
    # changing it into data frame and adding column names
    temp=pd.DataFrame(temp,columns=[(col+"_"+str(i)) for i in data[col]
        .value_counts().index])
    # Setting the index for proper concatenation
    temp=temp.set_index(X_test.index.values)
    # adding the new One Hot Encoded varibales to test data frame
    X_test_1=pd.concat([X_test_1,temp],axis=1)
```

Now, lets apply logistic regression model on one-hot encoded data.

```
# Standardizing the data set
>> X_train_scale=scale(X_train_1)
>> X_test_scale=scale(X_test_1)
# Fitting a logistic regression model
>> log=LogisticRegression(penalty='l2',C=1)
>> log.fit(X_train_scale,Y_train)
# Checking the model's accuracy
```

```
>> accuracy_score(Y_test,log.predict(X_test_scale))
```

Out : 0.75

Here, again we got the maximum accuracy as 0.75 that we have gotten so far. In this case, logistic regression regularization(C) parameter 1 whereas earlier we used C=0.01.

12. Perform following visualizations using matplotlib

a. Bar Graph
b. Pie Chart
c. Box Plot
d. Histogram
e. Line Chart and Subplots
f. Scatter Plot

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

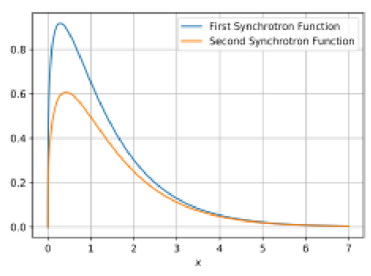
```
import matplotlib
print(matplotlib.__version__)
```

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK. There is also a procedural "pylab" interface based on a state machine (like OpenGL), designed to closely resemble that of MATLAB, though its use is discouraged. SciPy makes use of Matplotlib.

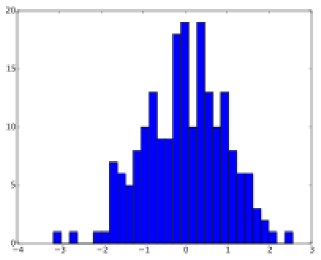
Matplotlib was originally written by John D. Hunter. Since then it has an active development community and is distributed under a BSD-style license. Michael Droettboom was nominated as matplotlib's lead developer shortly before John Hunter's death in August 2012 and was further joined by Thomas Caswell. Matplotlib is a NumFOCUS fiscally sponsored project.

Matplotlib 2.0.x supports Python versions 2.7 through 3.10. Python 3 support started with Matplotlib 1.2. Matplotlib 1.4 is the last version to support Python 2.6. Matplotlib has pledged not to support Python 2 past 2020 by signing the Python 3 Statement.

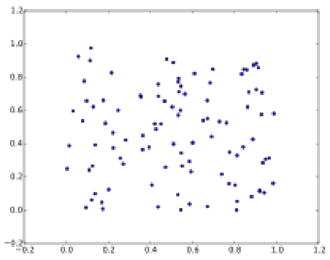
Examples



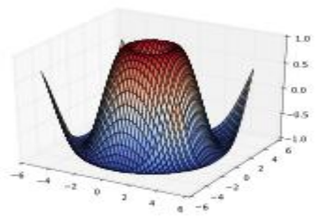
Line plot



Histogram



Scatter plot



3D plot

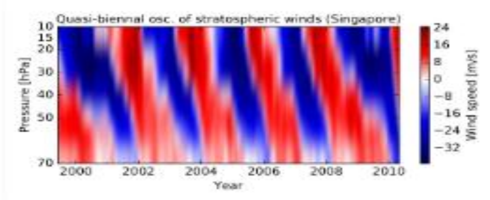
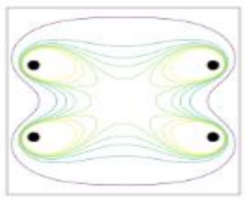
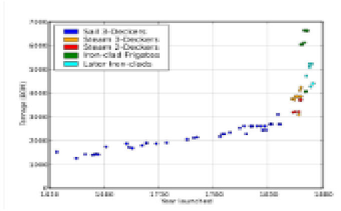


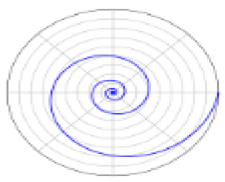
Image plot



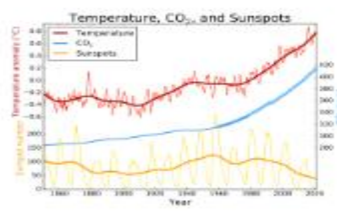
Contour plot



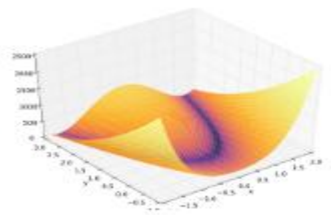
Scatter plot



Polar plot



Line plot



3-D plot

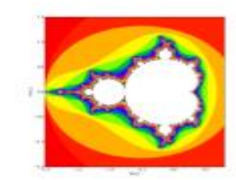


Image plot

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

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

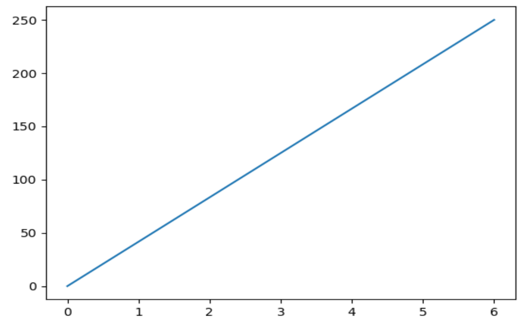
Now the Pyplot package can be referred to as plt.

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:



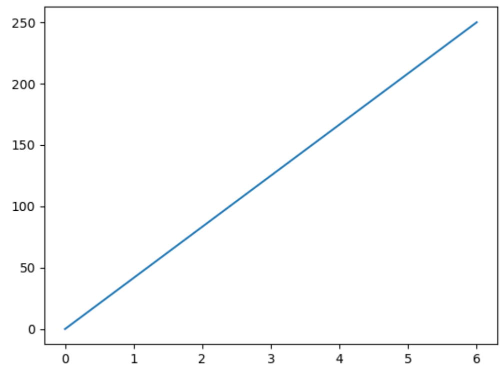
Matplotlib Plotting:

Plotting x and y points
The plot() function is used to draw points (markers) in a diagram.
By default, the plot() function draws a line from point to point.
The function takes parameters for specifying points in the diagram.
Parameter 1 is an array containing the points on the x-axis.
Parameter 2 is an array containing the points on the y-axis.
If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

Example
Draw a line in a diagram from position (1, 3) to 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)
plt.show()
```

Result:



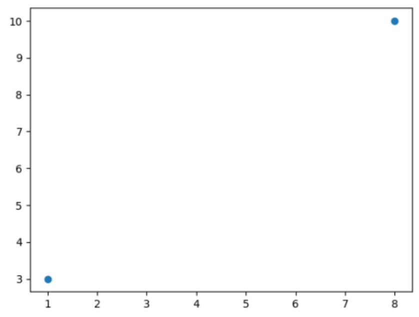
The x-axis is the horizontal axis.
The y-axis is the vertical axis.

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:

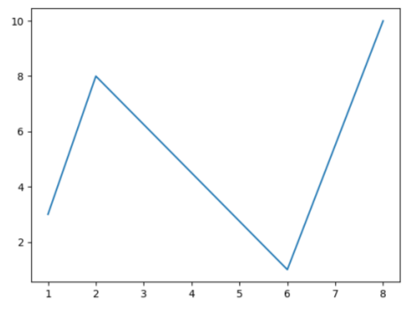


Multiple Points
You can plot as many points as you like, just make sure you have the same number of points in both axis.

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.

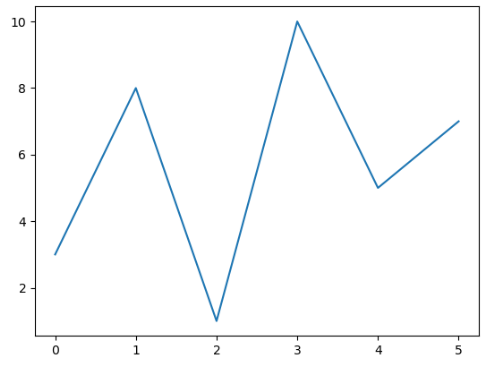
So, if we take the same example as above, and leave out the x-points, the diagram will look like this:

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:



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

Matplotlib Markers

Markers

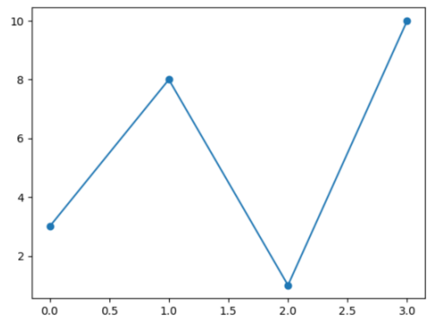
You can use the keyword argument marker to emphasize each point with a specified marker:

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:



Example

Mark each point with a star:

---

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

Marker Reference
You can choose any of these markers:

Marker	Description
'o'	Circle
'*'	Star
'.'	Point
','	Pixel
'x'	X
'X'	X (filled)
'+'	Plus
'P'	Plus (filled)
's'	Square
'D'	Diamond
'd'	Diamond (thin)
'p'	Pentagon
'H'	Hexagon
'h'	Hexagon
'v'	Triangle Down
'^'	Triangle Up
'<'	Triangle Left
'>'	Triangle Right
'1'	Tri Down
'2'	Tri Up
'3'	Tri Left
'4'	Tri Right
'|'	Vline
'_'	Hline

Format Strings fmt
You can use also use the shortcut string notation parameter to specify the marker.

This parameter is also called fmt, and is written with this syntax:

marker|line|color

Line Reference

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

Color Reference

Color Syntax	Description

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

Example
Set the size of the markers to 20:

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

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

plt.plot(ypoints, marker = 'o', ms = 20)
plt.show()
```

Result:

Marker Color
You can use the keyword argument markeredgecolor or the shorter mec to set the color of the edge of the markers:

Example
Set the EDGE color to red:

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

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

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

Result:

You can use the keyword argument markerfacecolor or the shorter mfc to set the color inside the edge of the markers:

Example
Set the FACE color to red:

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

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

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

Result:

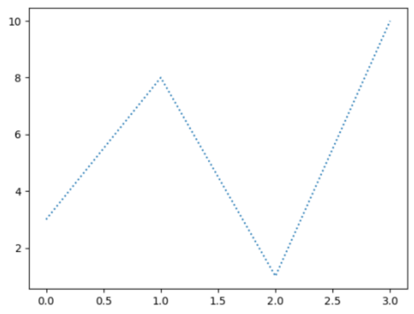
Matplotlib Line

Linestyle
You can use the keyword argument linestyle, or shorter ls, to change the style of the plotted line:

Example
Use a dotted line:

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

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

plt.plot(ypoints, linestyle = 'dotted') /// plt.plot(ypoints, ls = 'dotted')
plt.show()
```

Result:



Shorter Syntax
The line style can be written in a shorter syntax:

linestyle can be written as ls.

dotted can be written as :.

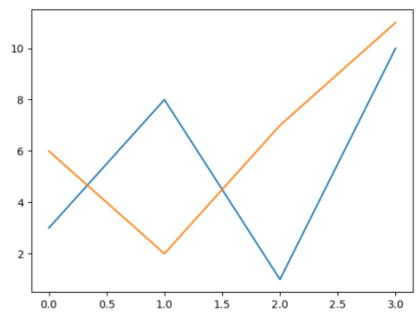
dashed can be written as --.

Multiple Lines
You can plot as many lines as you like by simply adding more plt.plot() functions:

Example
Draw two lines by specifying a plt.plot() function for each line:

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

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

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

plt.show()
```

Result:

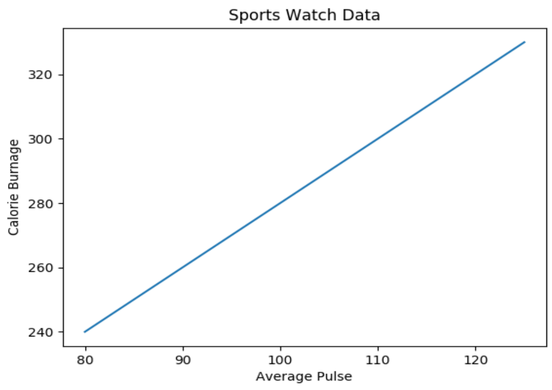


Matplotlib Labels and Title
Create Labels for a Plot
With Pyplot, you can use the xlabel() and ylabel() functions to set a label for the x- and y-axis.

Example
Add labels to the x- and y-axis:

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

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

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

plt.show()
```

Result:

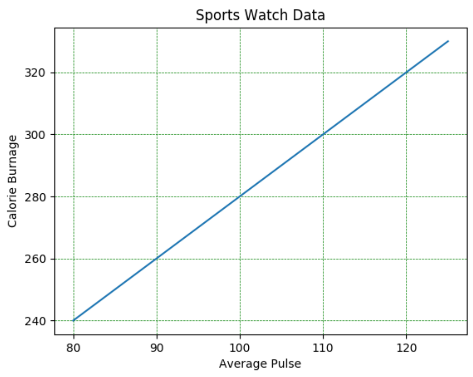


Matplotlib Adding Grid Lines
Add Grid Lines to a Plot
With Pyplot, you can use the grid() function to add grid lines to the plot.

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.grid(axis = 'x') // plt.grid(axis = 'y') //
                              plt.grid(color = 'green', linestyle = '--', linewidth = 0.5)

plt.show()
```

Result:



Matplotlib Subplots

Display Multiple Plots

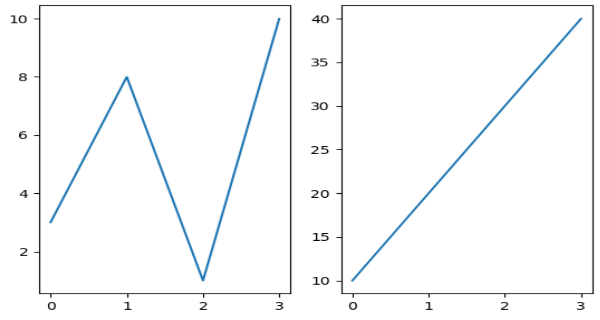
With the subplots() function you can draw multiple plots in one figure:

Example

Draw 2 plots:

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

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

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

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

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

plt.show()
```

Result:



The subplots() Function
The subplots() function takes three arguments that describes the layout of the figure.

The layout is organized in rows and columns, which are represented by the first and second argument.

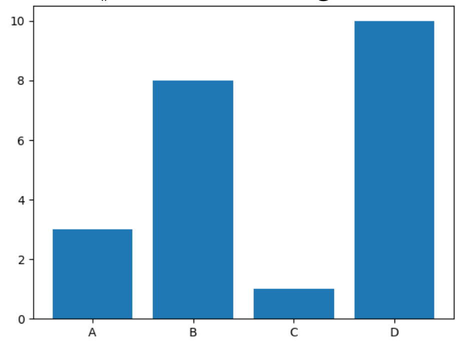
The third argument represents the index of the current plot.

```
plt.subplot(1, 2, 1)
#the figure has 1 row, 2 columns, and this plot is the first plot.

plt.subplot(1, 2, 2)
#the figure has 1 row, 2 columns, and this plot is the second plot.
```

So, if we want a figure with 2 rows an 1 column (meaning that the two plots will be displayed on top of each other instead of side-by-side), we can write the syntax like this

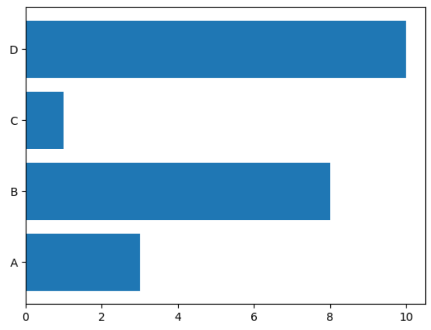
Creating Bars
With Pyplot, you can use the bar() function to draw bar graphs:

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

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

plt.bar(x,y, width = 0.1)
plt.show()
```

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


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

```
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, color = "red", height = 0.1)
plt.show()
```

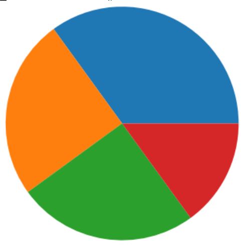


Creating Pie Charts

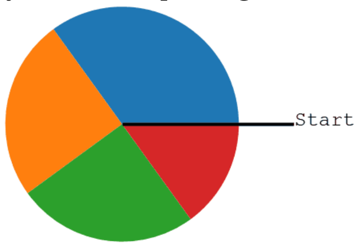
With Pyplot, you can use the pie() function to draw pie charts:

Example

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



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).

Explode & Shadow

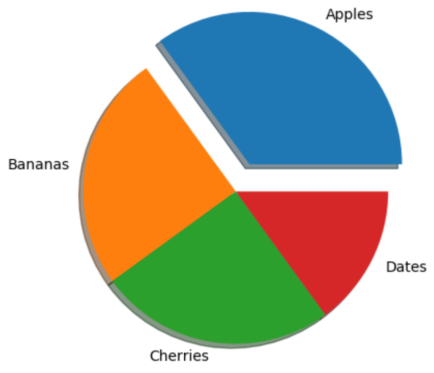
Maybe you want one of the wedges to stand out? The explode parameter allows you to do that.
The explode parameter, if specified, and not None, must be an array with one value for each wedge.
Each value represents how far from the center each wedge is displayed:

Shadow

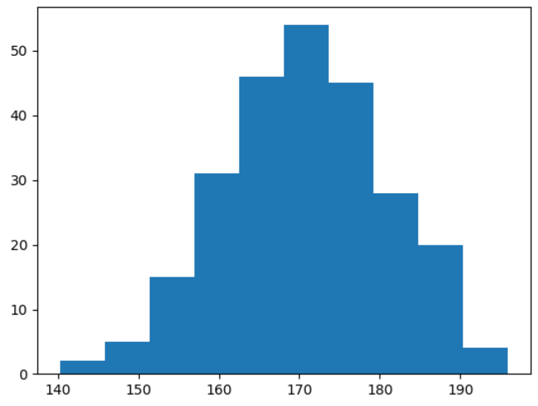
Add a shadow to the pie chart by setting the shadows parameter to True:

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

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
```

```
myexplode = [0.2, 0, 0, 0]

plt.pie(y, labels = mylabels, explode = myexplode, shadow = True)
plt.show()
```



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. Learn more about Normal Data Distribution in our Machine Learning Tutorial.

Example

A Normal Data Distribution by NumPy:

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

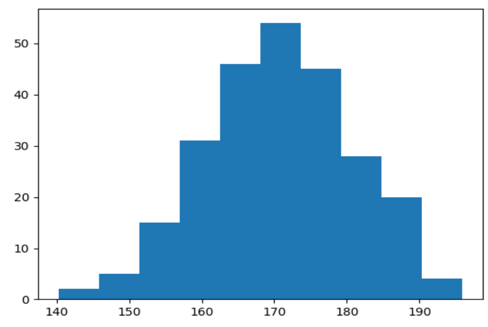
Result:

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 ]
```

```
import matplotlib.pyplot as plt
import numpy as np
x = np.random.normal(170, 10, 250)
plt.hist(x)
plt.show()
```

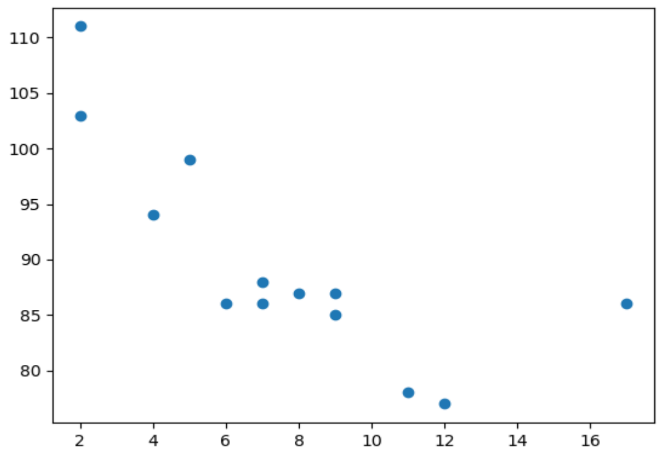


Creating Scatter Plots

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

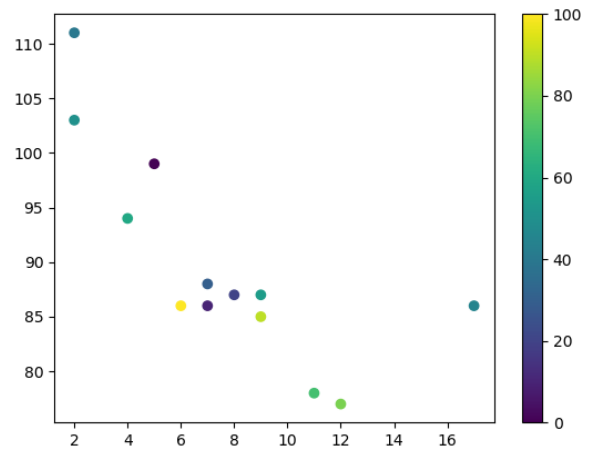
The scatter() function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

```
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()
```



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

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
colors = np.array([0, 10, 20, 30, 40, 45, 50, 55, 60, 70, 80, 90, 100])
```

```
plt.scatter(x, y, c=colors, cmap='viridis')

plt.colorbar()

plt.show()
```



Web References:

1. https://www.analyticsvidhya.com/blog/2020/04/the-ultimate-numpy-tutorial-for-data-science-beginners/

2. https://www.analyticsvidhya.com/blog/2021/07/data-science-with-pandas-2-minutes-guide-to-key-concepts/

3. https://www.analyticsvidhya.com/blog/2020/04/how-to-read-common-file-formats-python/

4.https://www.analyticsvidhya.com/blog/2016/07/practical-guide-data-preprocessing-python-scikit-learn/

5.https://www.analyticsvidhya.com/blog/2020/02/beginner-guide-matplotlib-data-visualization-explorationpython/