12.1 What are Modules in Python

- Modules refer to a file containing Python statements and definitions.
- A file containing Python code, for e.g.: abc.py, is called a module and its module name would be "abc".
- We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.
- We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Examples:

```
[ ] import math
    print(math.pi)
```

```
3.141592653589793
```

```
[ ] import datetime
    datetime.datetime.now()
```

```
datetime.datetime(2017, 10, 18, 20, 47, 20, 606228)
```

import with renaming

```
import math as m
print(m.pi)
```

```
3.141592653589793
```

from...import statement

We can import specific names form a module without importing the module as a whole.

```
[ ] from datetime import datetime
    datetime.now()
```

```
datetime.datetime(2017, 10, 18, 20, 47, 38, 17242)
```

import all names

```
[ ] from math import *
    print("Value of PI is " + str(pi))
```

```
Value of PI is 3.141592653589793
```

- math and datetime are examples of modules in python.
- We can also import a function by renaming as shown above.
- Instead of importing whole module we can import functions which are needed as shown below

dir() built in function

We can use the dir() function to find out names that are defined inside a module.

```
dir(example)
```

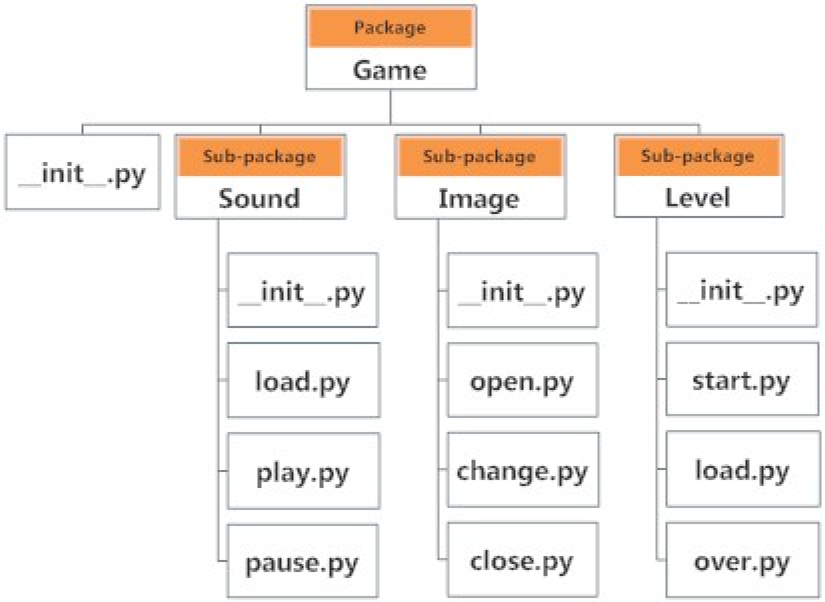
```
['__builtins__',
 '__cached__',
 '__doc__',
 '__file__',
 '__loader__',
 '__name__',
 '__package__',
 '__spec__',
 'add']
```

```
[ ] print(example.add.__doc__)
```

```
    This program adds two numbers and return the result
```

- The dir() function returns all properties and methods of the specified object, without the values.
- This function will return all the properties and methods, even built-in properties which are default for all objects.

- Packages are a way of structuring Python's module namespace by using "dotted module names".
- A directory must contain a file named **init**.py in order for Python to consider it as a package. This file can be left empty but we generally place the initialization code for that package in this file.



- Intuitively we can think of packages as folders in python and modules are .py files.For python to consider a folder as a package it needs to have __init__.py file in it.

importing module from a package

We can import modules from packages using the dot (.) operator.

```
[3] #import Gate.Image.open
```

- We can import a module from the package using . notation.

12.4 Introduction to NumPy

NumPy Arrays

python objects:

1. high-level number objects: integers, floating point

2. containers: lists (costless insertion and append), dictionaries (fast lookup)

Numpy provides:

1. extension package to Python for multi-dimensional arrays
2. closer to hardware (efficiency)
3. designed for scientific computation (convenience)
4. Also known as array oriented computing

1. Creating arrays

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

print(np.arange(10))
```

```
[0 1 2 3]
[0 1 2 3 4 5 6 7 8 9]
```

Why it is useful: Memory-efficient container that provides fast numerical operations.

```
#python lists
L = range(1000)
%timeit [i**2 for i in L]
```

```
307 µs ± 17.6 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

```
a = np.arange(1000)
%timeit a**2
```

```
1.35 µs ± 126 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

```
#1-D

a = np.array([0, 1, 2, 3])

a
```

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

```
# 2-D, 3-D....

b = np.array([[0, 1, 2], [3, 4, 5]])

b
```

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

```
c = np.array([[[0, 1], [2, 3]], [[4, 5], [6, 7]]])

c
```

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

       [[4, 5],
        [6, 7]]])
```

- We can create arrays in numpy as shown above.

1.2 Functions for creating arrays

```
#using arrange function

# arange is an array-valued version of the built-in Python range function

a = np.arange(10) # 0.... n-1
a
```

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

```
b = np.arange(1, 10, 2) #start, end (exclusive), step

b
```

```
array([1, 3, 5, 7, 9])
```

```
#using linspace

a = np.linspace(0, 1, 6) #start, end, number of points

a
```

```
array([ 0. ,  0.2,  0.4,  0.6,  0.8,  1. ])
```

```
#common arrays

a = np.ones((3, 3))

a
```

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

```
b = np.zeros((3, 3))

b
```

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

```
#create array using diag function

a = np.diag([1, 2, 3, 4]) #construct a diagonal array.

a
```

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

```
np.diag(a)   #Extract diagonal
```

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

```
#create array using random

#Create an array of the given shape and populate it with random samples from a uniform distribution over [0, 1).
a = np.random.rand(4)

a
```

```
array([ 0.85434586,  0.05106692,  0.37337949,  0.32093548])
```

- We can use functions for creating numpy arrays as shown above.

2. Basic DataTypes

You may have noticed that, in some instances, array elements are displayed with a **trailing dot (e.g. 2. vs 2)**. This is due to a difference in the **data-type** used:

```
a = np.arange(10)
a.dtype
```

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

```
#You can explicitly specify which data-type you want:
a = np.arange(10, dtype='float64')
a
```

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

```
#The default data type is float for zeros and ones function
a = np.zeros((3, 3))
print(a)
a.dtype
```

```
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]
dtype('float64')
```

Each built-in data type has a character code that uniquely identifies it.

'b' – boolean

'i' – (signed) integer

'u' – unsigned integer

'f' – floating-point

'c' – complex-floating point

'm' – timedelta

'M' – datetime

'O' – (Python) objects

'S', 'a' – (byte-)string

'U' – Unicode

'V' – raw data (void)

3. Indexing and Slicing

The items of an array can be accessed and assigned to the same way as other **Python sequences (e.g. lists)**:

```
a = np.arange(10)
print(a[5])  #indices begin at 0, like other Python sequences (and C/C++)
```

```
5
```

```
# For multidimensional arrays, indexes are tuples of integers:
a = np.diag([1, 2, 3])
print(a[2, 2])
```

```
3
```

```
a[2, 1] = 5 #assigning value
a
```

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

```
a = np.arange(10)
a
```

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

```
a[1:8:2] # [startindex: endindex(exclusive) : step]
```

```
array([1, 3, 5, 7])
```

```
#we can also combine assignment and slicing:
a = np.arange(10)
a[5:] = 10
a
```

```
array([ 0,  1,  2,  3,  4, 10, 10, 10, 10, 10])
```

```
b = np.arange(5)
a[5:] = b[::-1]  #assigning
a
```

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

- A slicing operation creates a view on the original array, which is just a way of accessing array data. Thus the original array is not copied in memory. You can use **np.may_share_memory()** to check if two arrays share the same memory block.

4. Copies and Views

When modifying the view, the original array is modified as well:

```
a = np.arange(10)
a
```

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

```
b = a[::2]
b
```

```
array([0, 2, 4, 6, 8])
```

```
np.shares_memory(a, b)
```

```
True
```

```
b[0] = 10
b
```

```
array([10,  2,  4,  6,  8])
```

```
a  #eventhough we modified b,  it updated 'a' because both shares same memory
```

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

- When we copy a new array gets created ,so changes made to the copied array don't affect the original array.

```

a = np.arange(10)

c = a[::2].copy()     #force a copy
c
```

```
array([0, 2, 4, 6, 8])
```

```
np.shares_memory(a, c)
```

```
False
```

```
c[0] = 10

a
```

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

5. Fancy Indexing

NumPy arrays can be indexed with slices, but also with boolean or integer arrays **(masks)**. This method is called **fancy indexing**. It creates copies not views.

Using Boolean Mask

```
[ ]  a = np.random.randint(0, 20, 15)
     a
```

```
array([18, 17,  1, 18,  5, 17,  0, 14, 12, 11,  4, 15, 16,  8,  7])
```

```
[ ]  mask = (a % 2 == 0)
```

```
[ ]  extract_from_a = a[mask]

     extract_from_a
```

```
array([18, 18,  0, 14, 12,  4, 16,  8])
```

Indexing with a mask can be very useful to assign a new value to a sub-array:

```
[ ]  a[mask] = -1
     a
```

```
array([-1, 17,  1, -1,  5, 17, -1, -1, -1, 11, -1, 15, -1, -1,  7])
```

Indexing with an array of integers

```
[ ]  a = np.arange(0, 100, 10)
     a
```

```
array([ 0, 10, 20, 30, 40, 50, 60, 70, 80, 90])
```

```
[ ]  #Indexing can be done with an array of integers, where the same index is repeated several time:
     a[[2, 3, 2, 4, 2]]
```

```
array([20, 30, 20, 40, 20])
```

```
     # New values can be assigned
     a[[9, 7]] = -200
     a
```

```
array([   0,   10,   20,   30,   40,   50,   60, -200,   80, -200])
```

12.5 Common Operations in NumPy

1.Elementwise Operations

1. Basic Operations

with scalars

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

a + 1
```

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

```
[ ]
a ** 2
```

```
array([ 1,  4,  9, 16])
```

All arithmetic operates elementwise

```
[ ]
b = np.ones(4) + 1

a - b
```

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

```
[ ]
a * b
```

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

```
# Matrix multiplication

c = np.diag([1, 2, 3, 4])

print(c * c)
print("*****************")
print(c.dot(c))
```

```
[[ 1  0  0  0]
 [ 0  4  0  0]
 [ 0  0  9  0]
 [ 0  0  0 16]]
*****************
[[ 1  0  0  0]
 [ 0  4  0  0]
 [ 0  0  9  0]
 [ 0  0  0 16]]
```

- Below are the comparison operations that we can perform using numpy

comparisions

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

```
array([False,  True, False,  True], dtype=bool)
```

```
a > b
```

```
array([False, False,  True, False], dtype=bool)
```

```
#array-wise comparisions
a = np.array([1, 2, 3, 4])
b = np.array([5, 2, 2, 4])
c = np.array([1, 2, 3, 4])

np.array_equal(a, b)
```

```
False
```

```
np.array_equal(a, c)
```

```
True
```

Below are the logical operations that we can perform using numpy

Logical Operations

```
a = np.array([1, 1, 0, 0], dtype=bool)
b = np.array([1, 0, 1, 0], dtype=bool)

np.logical_or(a, b)
```

```
array([ True,  True,  True, False], dtype=bool)
```

```
np.logical_and(a, b)
```

```
array([ True, False, False, False], dtype=bool)
```

Transcendental functions:

```
a = np.arange(5)

np.sin(a)
```

```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ])
```

```
np.log(a)
```

```
/Users/satishatcha/.virtualenvs/course/lib/python2.7/site-packages/ipykernel_launcher.py
  """Entry point for launching an IPython kernel.
array([       -inf,  0.        ,  0.69314718,  1.09861229,  1.38629436])
```

```
np.exp(a)   #evaluates e^x for each element in a given input
```

```
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692,  54.59815003])
```

2.Basic Reductions

Below are the basic reductions we can do using numpy.

computing sums

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

```
10
```

```
[ ]  #sum by rows and by columns

     x = np.array([[1, 1], [2, 2]])
     x
```

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

```
[ ]  x.sum(axis=0)   #columns first dimension
```

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

```
[ ]  x.sum(axis=1)  #rows (second dimension)
```

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

Other reductions

```
[ ]  x = np.array([1, 3, 2])
     x.min()
```

```
1
```

```
[ ]  x.max()
```

```
3
```

```
[ ]  x.argmin()# index of minimum element
```

```
0
```

```
[ ]  x.argmax()# index of maximum element
```

```
1
```

Statistics

```
[ ]  x = np.array([1, 2, 3, 1])
     y = np.array([[1, 2, 3], [5, 6, 1]])
     x.mean()
```

```
1.75
```

```
[ ]  np.median(x)
```

```
1.5
```

```
[ ]  np.median(y, axis=-1) # last axis
```

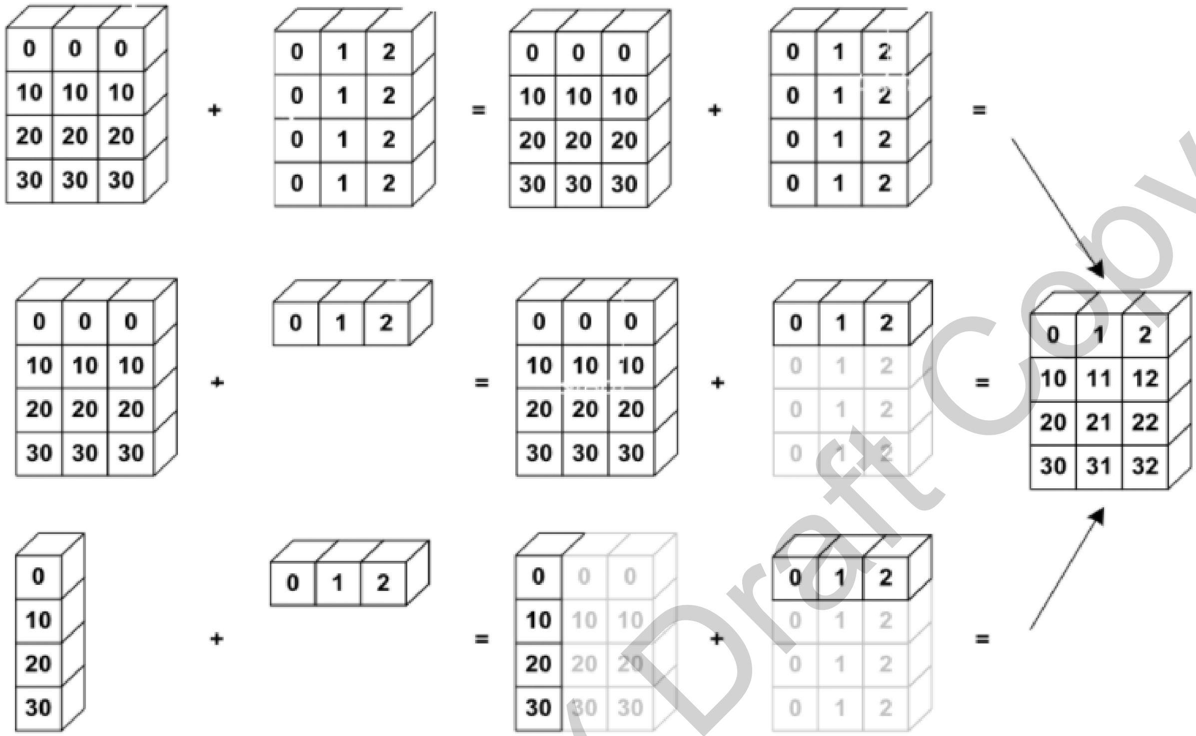
```
array([ 2.,  5.])
```

```
[ ]  x.std()          # full population standard dev.
```

```
0.82915619758884995
```

3.Broadcasting

- Basic operations on numpy arrays (addition, etc.) are elementwise
- This works on arrays of the same size. Nevertheless, It's also possible to do operations on arrays of different sizes if NumPy can transform these arrays so that they all have the same size: this conversion is called broadcasting.
- The image below gives an example of broadcasting:


0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
0 1 2
0 1 2
0 1 2
=
0 1 2
+
=
+
=
0 1 2
10 11 12
20 21 22
30 31 32
0
10
20
30
+
0 1 2
=
+
=

```python
a = np.tile(np.arange(0, 40, 10), (3,1))
print(a)

print("*************")
a=a.T
print(a)
```

```
[[ 0 10 20 30]
 [ 0 10 20 30]
 [ 0 10 20 30]]
*************
[[ 0  0  0]
 [10 10 10]
 [20 20 20]
 [30 30 30]]
```

```python

b = np.array([0, 1, 2])
b
```

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

```python

a + b
```

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

```python
a = np.arange(0, 40, 10)
a.shape

```

```
(4,)
```

```python
a = a[:, np.newaxis]  # adds a new axis -> 2D array
a.shape
```

```
(4, 1)
```

```python
a
```

```
array([[ 0],
       [10],
       [20],
       [30]])
```

```python
a + b
```

```
array([[ 0,  1,  2],
       [10, 11, 12],
       [20, 21, 22],
       [30, 31, 32]])
```

4.Array Shape Manipulation

1.Flattening

```
[ ]  a = np.array([[1, 2, 3], [4, 5, 6]])
     a.ravel() #Return a contiguous flattened array.
```

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

```
a.T #Transpose
```

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

```
[ ]  a.T.ravel()
```

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

2.Reshaping

The inverse operation to flattening:

```
[ ]  print(a.shape)
     print(a)
```

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

```
[ ]  b = a.ravel()
     print(b)
```

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

```
[ ]  b = b.reshape((2, 3))
     b
```

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

3.Adding a Dimension

- Indexing with the np.newaxis object allows us to add an axis to an array
- newaxis is used to increase the dimension of the existing array by one more dimension, when used once. Thus,

  1D array will become 2D array

  2D array will become 3D array

  3D array will become 4D array and so on

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

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

```
z[:, np.newaxis]
```

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

4.Dimension Shuffling

```
a = np.arange(4*3*2).reshape(4, 3, 2)
a.shape
```

```
(4, 3, 2)
```

```
a
```

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

       [[ 6,  7],
        [ 8,  9],
        [10, 11]],

       [[12, 13],
        [14, 15],
        [16, 17]],

       [[18, 19],
        [20, 21],
        [22, 23]]])
```

```
a[0, 2, 1]
```

```
5
```

5.Resizing

```python
a = np.arange(4)
a.resize((8,))
a
```

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

However, it must not be referred to somewhere else:

```python
b = a
a.resize((4,))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-68-702766c88583> in <module>()
      1 b = a
----> 2 a.resize((4,))

ValueError: cannot resize an array that references or is referenced
by another array in this way.  Use the resize function
```

SEARCH STACK OVERFLOW

6.Sorting Data

```
#Sorting along an axis:
a = np.array([[5, 4, 6], [2, 3, 2]])
b = np.sort(a, axis=1)
b
```

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

```
#in-place sort
a.sort(axis=1)
a
```

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

```
#sorting with fancy indexing
a = np.array([4, 3, 1, 2])
j = np.argsort(a)
j
```

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

```
a[j]
```

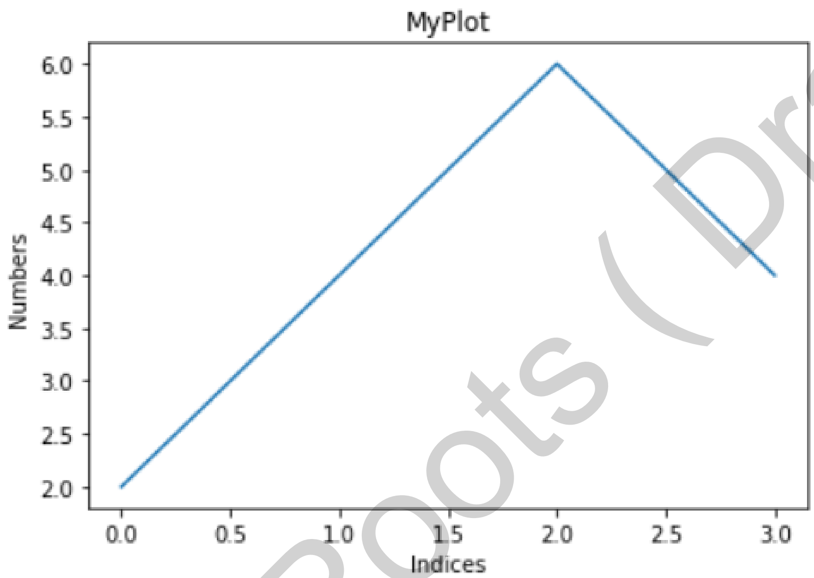
```
array([1, 2, 3, 4])
```

12.6 Getting started with Matplotlib

1.Plotting

- **matplotlib.pyplot** is a collection of command style functions that make matplotlib work like MATLAB.
- Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

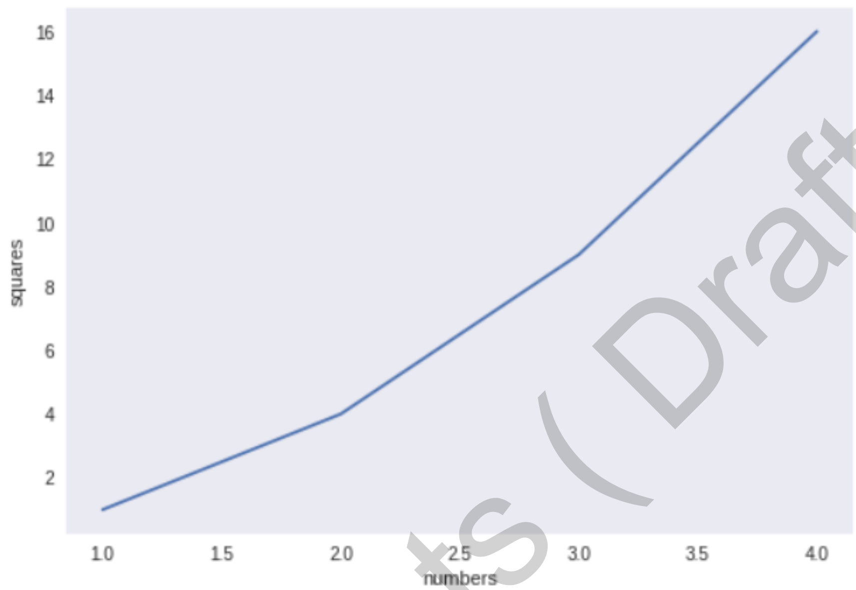
```
plt.plot([2,4, 6, 4])
plt.ylabel("Numbers")
plt.xlabel('Indices')
plt.title('MyPlot')
plt.show()
```



- If you provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0,1,2,3].

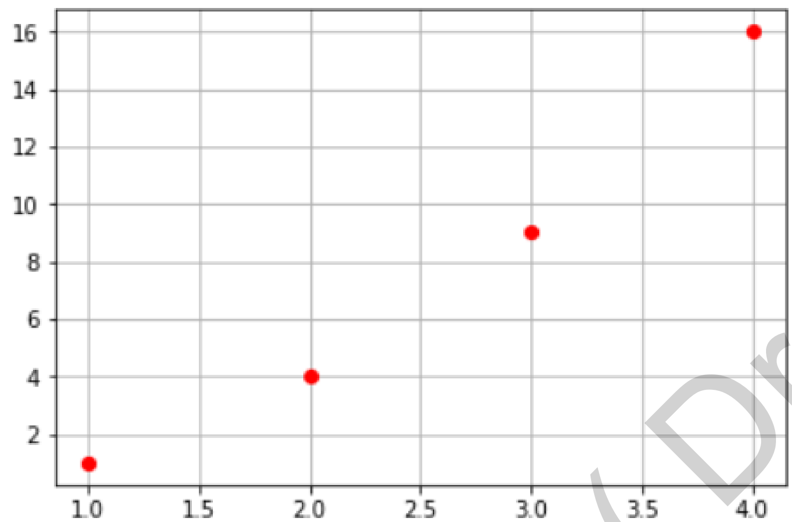
plot x versus y

```
[ ]  plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
     plt.ylabel('squares')
     plt.xlabel('numbers')
     plt.grid() # grid on

     plt.show()
```



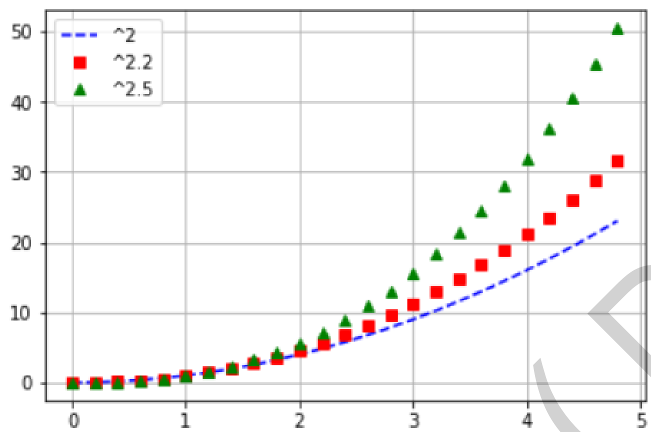
- For every x, y pair of arguments, there is an optional third argument as shown below which is the **format string** that indicates the color and line type of the plot.

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
plt.grid()

plt.show()
```



- If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use **numpy arrays**. In fact, all sequences are converted to numpy arrays internally.

```
import numpy as np
t = np.arange(0., 5., 0.2)

#blue dashes, red squares and green triangles
plt.plot(t, t**2, 'b--', label='^2')#   'rs',   'g^')
plt.plot(t,t**2.2, 'rs', label='^2.2')
plt.plot(t, t**2.5, 'g^', label='^2.5')
plt.grid()
plt.legend() # add legend based on line labels
plt.show()
```

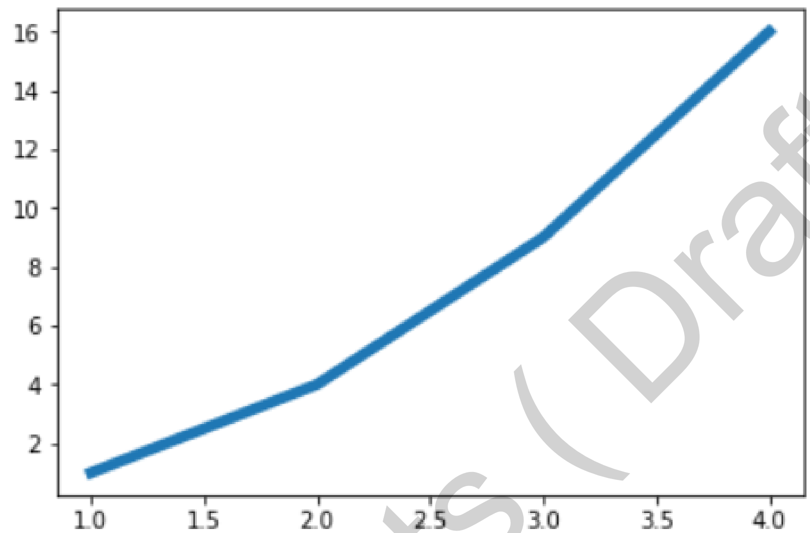


2.Controlling line properties

Lines have many attributes that you can set: linewidth, dash style etc

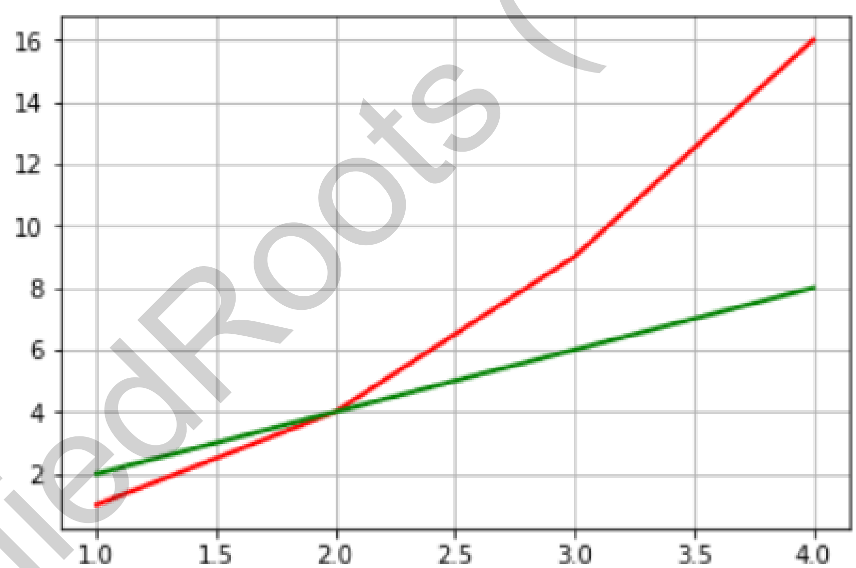
use keyword args

```
x = [1, 2, 3, 4]
y = [1, 4, 9, 16]
plt.plot(x, y, linewidth=5.0)
plt.show()
```



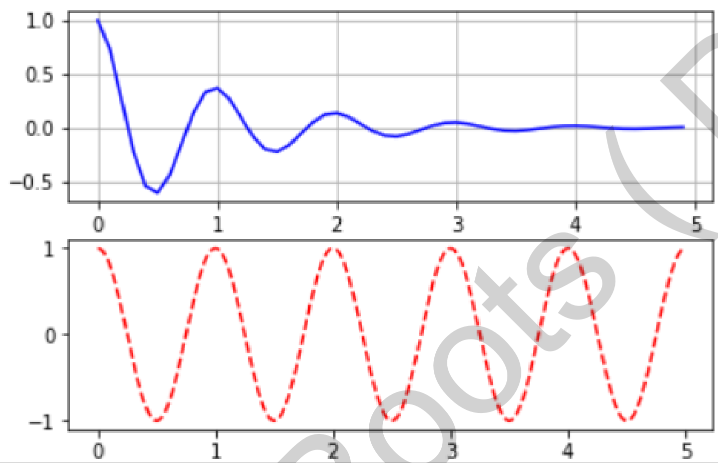
use the setp()

```
x1 = [1, 2, 3, 4]
y1 = [1, 4, 9, 16]
x2 = [1, 2, 3, 4]
y2 = [2, 4, 6, 8]
lines = plt.plot(x1, y1, x2, y2)

# use keyword args
plt.setp(lines[0], color='r', linewidth=2.0)

# or MATLAB style string value pairs
plt.setp(lines[1], 'color', 'g', 'linewidth', 2.0)

plt.grid()
```



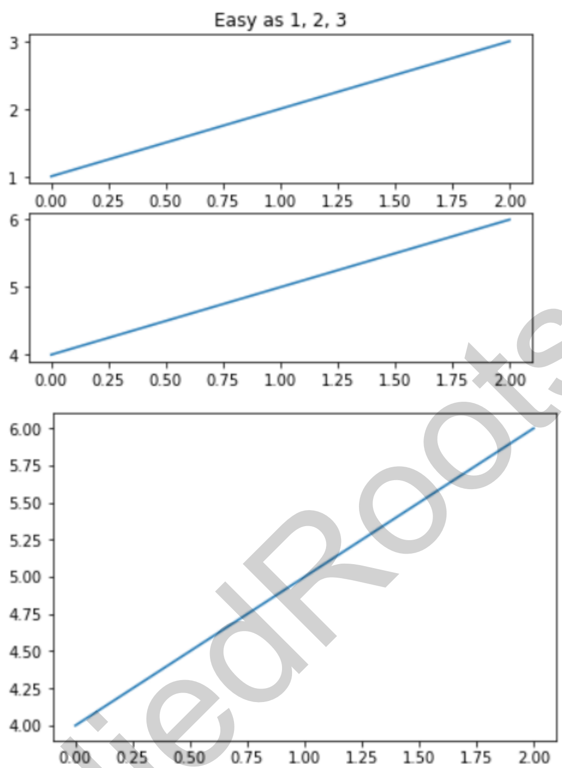
3 .Working with multiple figures and axes

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

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

plt.figure(1)
# The subplot() command specifies numrows, numcols,
# fignum where fignum ranges from 1 to numrows*numcols.
plt.subplot(211)
plt.grid()
plt.plot(t1, f(t1), 'b-')

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



matplotlib.pyplot .subplots creates a figure and a grid of subplots with a single call, while providing reasonable control over how the individual plots are created.

```python
plt.figure(1)                # the first figure
plt.subplot(211)             # the first subplot in the first figure
plt.plot([1, 2, 3])
plt.subplot(212)             # the second subplot in the first figure
plt.plot([4, 5, 6])


plt.figure(2)                # a second figure
plt.plot([4, 5, 6])          # creates a subplot(111) by default

plt.figure(1)                # figure 1 current; subplot(212) still current
plt.subplot(211)             # make subplot(211) in figure1 current
plt.title('Easy as 1, 2, 3') # subplot 211 title
plt.show()
```



12.7 Getting started with Pandas

What is pandas-python? Introduction and Installation

- Pandas is python module that makes data science easy and effective
- Weather dataset

Questions?

1. What was the maximum temperature in new york in the month of january?
2. On which days did it rain?
3. What was the average speed of wind during the month?

The data frame shown below is the data at hand and we try to answer the above using the data.

```
df
```

Out[4]:

	EST	Temperature	DewPoint	Humidity	Sea Level PressureIn	VisibilityMiles	WindSpeedMPH	Precip
0	1/1/2016	38	23	52	30.03	10	8.0	
1	1/2/2016	36	18	46	30.02	10	7.0	
2	1/3/2016	40	21	47	29.86	10	8.0	
3	1/4/2016	25	9	44	30.05	10	9.0	
4	1/5/2016	20	-3	41	30.57	10	5.0	
5	1/6/2016	33	4	35	30.50	10	4.0	
6	1/7/2016	39	11	33	30.28	10	2.0	
7	1/8/2016	39	29	64	30.20	10	4.0	
8	1/9/2016	44	38	77	30.16	9	8.0	

```
In [4]: # now we will see in pandas

        import pandas as pd
        df = pd.read_csv('nyc_weather.csv')
        df
```

We read csv files using pandas as shown.

```
 [6]: #get the maximum temparature
      df['Temperature'].max()

t[6]: 50
```

Using max function we can obtain maximum value from a column in pandas dataframe

```
n [7]: #to know which day it rains
       df['EST'][df['Events'] == 'Rain']

ut[7]: 8      1/9/2016
       9      1/10/2016
       15     1/16/2016
       26     1/27/2016
       Name: EST, dtype: object
```

We used a boolean expression which evaluates to be true only for those rows where the Events column has Rain. Finally the EST column will be returned.

```
n [8]: #3. average wind speed
       df['WindSpeedMPH'].mean()

ut[8]: 6.892857142857143
```

We can use the mean( ) function on the WindSpeedMPH column in the dataframe and get average speed.

Installation

pip3 install pandas

For installing pandas we use above command.

12.8 Understanding DataFrames in Pandas

Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

Introduction to Pandas dataframe

Data frame is a main object in pandas. It is used to represent data with rows and columns

Data frame is a data structure represent the data in tabular or excel spreadsheet like data)

creating dataframe:

```
import pandas as pd
df = pd.read_csv("weather_data.csv")   #read weather.csv data
df
```

	day	temperature	windspeed	event
0	1/1/2017	32	6	Rain
1	1/2/2017	35	7	Sunny
2	1/3/2017	28	2	Snow
3	1/4/2017	24	7	Snow
4	1/5/2017	32	4	Rain
5	1/6/2017	31	2	Sunny

We use pandas to load our data into a dataframe.

```python
#list of tuples

weather_data = [('1/1/2017', 32, 6, 'Rain'),
                ('1/2/2017', 35, 7, 'Sunny'),
                ('1/3/2017', 28, 2, 'Snow'),
                ('1/4/2017', 24, 7, 'Snow'),
                ('1/5/2017', 32, 4, 'Rain'),
                ('1/6/2017', 31, 2, 'Sunny')
               ]
df = pd.DataFrame(weather_data, columns=['day', 'temperature', 'windspeed', 'event'])
df
```

	day	temp	windspeed	event
0	1/1/2017	32	6	Rain
1	1/2/2017	35	7	Sunny
2	1/3/2017	28	2	Snow
3	1/4/2017	24	7	Snow
4	1/5/2017	32	4	Rain
5	1/6/2017	31	2	Sunny

Using list of tuples we can create pandas dataframe as shown above

```
#get dimentions of the table

df.shape    #total number of rows and columns
```

```
(6, 4)
```

```
#if you want to see initial some rows then use head command (default 5 rows)
df.head()
```

	day	temperature	windspeed	event
0	1/1/2017	32	6	Rain
1	1/2/2017	35	7	Sunny
2	1/3/2017	28	2	Snow
3	1/4/2017	24	7	Snow
4	1/5/2017	32	4	Rain

```
#if you want to see last few rows then use tail command (default last 5 rows will print)
df.tail()
```

	day	temperature	windspeed	event
1	1/2/2017	35	7	Sunny
2	1/3/2017	28	2	Snow
3	1/4/2017	24	7	Snow

- Using the shape attribute we can find numbers of rows and columns in the dataframe.
- head() gives top five rows in a dataframe, similarly tail() gives button 5 rows of the dataframe

```
#slicing
df[2:5]
```

	day	temperature	windspeed	event
2	1/3/2017	28	2	Snow
3	1/4/2017	24	7	Snow
4	1/5/2017	32	4	Rain

```
df.columns    #print columns in a table
```

```
Index(['day', 'temperature', 'windspeed', 'event'], dtype='object')
```

- We can use slicing and obtain required rows in the dataframe.
- Columns attribute gives the column names of the dataframe

```
df.day      #print particular column data
```

```
0    1/1/2017
1    1/2/2017
2    1/3/2017
3    1/4/2017
4    1/5/2017
5    1/6/2017
Name: day, dtype: object
```

```
#another way of accessing column
df['day'] #df.day (both are same)
```

```
0    1/1/2017
1    1/2/2017
2    1/3/2017
3    1/4/2017
4    1/5/2017
5    1/6/2017
Name: day, dtype: object
```

```
#get 2 or more columns
df[['day', 'event']]
```

	day	event
0	1/1/2017	Rain
1	1/2/2017	Sunny
2	1/3/2017	Snow

- We can get particular column data or 2 or more columns as shown above.

```
#print max temperature
df['temperature'].max()
```

```
35
```

```
#print max temperature
df['temperature'].min()
```

```
24
```

```
[ ]
#print max temperature
df['temperature'].describe()
```

```
count     6.000000
mean     30.333333
std       3.829708
min      24.000000
25%      28.750000
50%      31.500000
75%      32.000000
max      35.000000
Name: temperature, dtype: float64
```

We can use functions like min(),max() on a particular column of dataframe to obtain min or max value.Alternatively we can use describe() function which gives us overall statistics of a particular column.

```
# select rows which has maximum temperature
df[df.temperature == df.temperature.max()]

```

	day	temperature	windspeed	event
1	1/2/2017	35	7	Sunny

```
#select only day column which has maximum temperature
df.day[df.temperature == df.temperature.max()]

```

```
1    1/2/2017
Name: day, dtype: object
```

We can also select particular rows based on the conditions we specify as shown above.

12.9 Common Operations on Data Frames

1.Read and write CSV and XLS files

```
In [3]: import pandas as pd
        df = pd.read_csv('weather_data.csv')
        df
```

Out[3]:

	day	temperature	windspeed	event
0	1/1/2017	32	6	Rain
1	1/2/2017	35	7	Sunny
2	1/3/2017	28	2	Snow
3	1/4/2017	24	7	Snow
4	1/5/2017	32	4	Rain

```
In [3]: #INSTALL: pip3 install xlrd

        #read excel file
        df = pd.read_excel('weather_data.xlsx')
        df
```

Out[3]:

	day	temperature	windspeed	event
0	1/1/2017	32	6	Rain
1	1/2/2017	35	7	Sunny
2	1/3/2017	28	2	Snow
3	1/4/2017	24	7	Snow
4	1/5/2017	32	4	Rain
5	1/6/2017	31	2	Sunny

We can read csv and excel files into pandas dataframes as shown above.

```
#write DF to csv
df.to_csv('new.csv')
df.to_csv('new_noIndex.csv', index=False)
```

```
# INSTALL: pip3 install openpyxl

#write DF to Excel
df.to_excel('new.xlsx', sheet_name='weather_data')
```

Similarly we can store data into csv or excel files as shown above.

2.GROUP-BY

```
g = df.groupby('city')
g
```

```
<pandas.core.groupby.DataFrameGroupBy object at 0x106d495f8>
```

```
for city, city_df in g:
    print(city)
    print(city_df)
```

```
mumbai
        day    city  temperature  windspeed  event
4  1/1/2017  mumbai           90          5  Sunny
5  1/2/2017  mumbai           85         12    Fog
6  1/3/2017  mumbai           87         15    Fog
7  1/4/2017  mumbai           92          5   Rain
new york
        day      city  temperature  windspeed  event
0  1/1/2017  new york           32          6   Rain
1  1/2/2017  new york           36          7  Sunny
2  1/3/2017  new york           28         12   Snow
3  1/4/2017  new york           33          7  Sunny
paris
         day   city  temperature  windspeed   event
8   1/1/2017  paris           45         20   Sunny
9   1/2/2017  paris           50         13  Cloudy
10  1/3/2017  paris           54          8  Cloudy
11  1/4/2017  paris           42         10  Cloudy
```

When we used groupby( ) on city column ,it groups all the rows where city name is same and group them together.Its similar to groupby command in SQL

```python
#or to get specific group
g.get_group('new york')

```

	day	city	temperature	windspeed	event
0	1/1/2017	new york	32	6	Rain
1	1/2/2017	new york	36	7	Sunny
2	1/3/2017	new york	28	12	Snow
3	1/4/2017	new york	33	7	Sunny

```python
#Find maximum temperature in each of the cities
print(g.max())
```

```
                 day  temperature  windspeed  event
city
mumbai    1/4/2017           92         15  Sunny
new york  1/4/2017           36         12  Sunny
paris     1/4/2017           54         20  Sunny
```

```
print(g.mean())

```

```
          temperature  windspeed
city
mumbai          88.50       9.25
new york        32.25       8.00
paris           47.75      12.75
```

```
print(g.describe())
```

```
         temperature
               count   mean       std   min    25%   50%    75%   max
city
mumbai           4.0  88.50  3.109126  85.0  86.50  88.5  90.50  92.0
new york         4.0  32.25  3.304038  28.0  31.00  32.5  33.75  36.0
paris            4.0  47.75  5.315073  42.0  44.25  47.5  51.00  54.0

         windspeed
             count   mean       std  min    25%   50%    75%   max
city
mumbai         4.0   9.25  5.057997  5.0  5.00   8.5  12.75  15.0
new york       4.0   8.00  2.708013  6.0  6.75   7.0   8.25  12.0
paris          4.0  12.75  5.251984  8.0  9.50  11.5  14.75  20.0
```

Using the groupby object that we obtained we can obtain data of each group,mean etc as shown above.

3.Concatenate Data Frames

```python
import pandas as pd
india_weather = pd.DataFrame({
    "city": ["mumbai","delhi","banglore"],
    "temperature": [32,45,30],
    "humidity": [80, 60, 78]
})

india_weather
```

	city	humidity	temperature
0	mumbai	80	32
1	delhi	60	45
2	banglore	78	30

```python
us_weather = pd.DataFrame({
    "city": ["new york","chicago","orlando"],
    "temperature": [21,14,35],
    "humidity": [68, 65, 75]
})
us_weather
```

	city	humidity	temperature
0	new york	68	21
1	chicago	65	14
2	orlando	75	35

- Let's create two data frames as shown above

```
#concate two dataframes
df = pd.concat([india_weather, us_weather])
df
```

	city	humidity	temperature
0	mumbai	80	32
1	delhi	60	45
2	banglore	78	30
0	new york	68	21
1	chicago	65	14
2	orlando	75	35

```
#if you want continuous index
df = pd.concat([india_weather, us_weather], ignore_index=True)
df
```

	city	humidity	temperature
0	mumbai	80	32
1	delhi	60	45
2	banglore	78	30
3	new york	68	21
4	chicago	65	14
5	orlando	75	35

- We can concatenate two dataframes using concat( ) as shown above ,for getting a unique index while concatenating we set ignore_index parameter to true.
- We can Combine dataframe objects horizontally along the x axis by passing in axis=1as shown below.

```
df = pd.concat([india_weather, us_weather],axis=1)
df
```

	city	humidity	temperature	city	humidity	temperature
0	mumbai	80	32	new york	68	21
1	delhi	60	45	chicago	65	14
2	banglore	78	30	orlando	75	35

4.Merge DataFrames

Consider two data frames as shown below

```
temperature_df = pd.DataFrame({
    "city": ["mumbai","delhi","banglore", 'hyderabad'],
    "temperature": [32,45,30,40]})
temperature_df
```

	city	temperature
0	mumbai	32
1	delhi	45
2	banglore	30
3	hyderabad	40

```
humidity_df = pd.DataFrame({
    "city": ["delhi","mumbai","banglore"],
    "humidity": [68, 65, 75]})
humidity_df
```

	city	humidity
0	delhi	68
1	mumbai	65
2	banglore	75

```
#merge two dataframes with out explicitly mention index
df = pd.merge(temperature_df, humidity_df, on='city')
df
```

	city	temperature	humidity
0	mumbai	32	65
1	delhi	45	68
2	banglore	30	75

```
#OUTER-JOIN
df = pd.merge(temperature_df, humidity_df, on='city', how='outer')
df
```

	city	temperature	humidity
0	mumbai	32	65.0
1	delhi	45	68.0
2	banglore	30	75.0
3	hyderabad	40	NaN

- We can merge two dataframes as shown above ,it is similar to join in SQL.

5.Numerical Indexing (.loc vs iloc)

```
df = pd.DataFrame([1,2,3,4,5,6,7,8,9,19], index=[49,48,47,46,45, 1, 2, 3, 4, 5])
df
```

	0
49	1
48	2
47	3
46	4
45	5
1	6
2	7
3	8
4	9
5	19

We can index our data frame numerically as shown above ,it creates an index column and by default the index will be the row number.

46	4
45	5
1	6
2	7
3	8
4	9
5	19

index

```
In [23]: s.loc[46]
Out[23]: 4
```

row no

```
In [24]: s.iloc[3]
Out[24]: 4
```

- We can use loc( ) and by specifying the index we can get the value as shown above.
- We can use iloc( ) and by specifying row number we can get the value.