----------TABLE OF CONTENTS----------

```
11.3  From a list of dicts
11.4  From a dict of tuples
11 5  Alternate Constructors
11.6  Column selection, addition, deletion
11.7  Row selection, addition, deletion
11.8  Indexing / Selection
11.9  Data Alignment And Arithmetic
11.10 Transposing
```

12.Data Viewing

1. Introduction

- Python is a powerful programming language ideal for scripting and rapid application development.
- It is used in web development (like: Django and Bottle)
- Used for scientific and mathematical computing (Orange, SymPy, NumPy) to desktop graphical user Interfaces (Pygame, Panda3D).
- Python has gathered alot of appreciation as a choice of language for Data Analytics.
- Open Source and Free to install.
- Awesome online community.
- Very easy to learn.
- It can become common language for data science.
- It can also become a common language for production of web based analytics products.
- As it is interpreted programming language so it is easier to implement.
- Compilation is not required, execution process can be done directly.
- Python is a general-purpose programming language that is becoming more and more popular for doing data science.
- Companies worldwide are using Python to harvest insights from their data and get a competitive edge.

```
            ~ **Now a days python is gaining more popularity all around the world** ~

          This tutorial introduces you to the basic concepts and features of Python 3.
          After reading the tutorial, you will be able to read and write basic Python
programs
          and explore Python in depth on your own.

          This tutorial is intended for people who have knowledge of other programming lan
guages
          and want to get started with Python quickly.
```

2. Data Types

- The data stored in memory can be of many types. For example

```
        a person's age is stored as a numeric value and
        his or her address is stored as alphanumeric characters.
```

- Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.
- We can use the **type()** function to know which class a variable or a value belongs to
- And **isinstance()** function to check if it belongs to a particular class.

Python support 4 types of data types

- int (signed integers)
- long (long integers, they can also be represented in octal and hexadecimal)
- float (floating point real values)
- complex (complex numbers)

Example

In [3]:

```
a = 5
b = 123+43j

# Output: <class 'int'>
print(type(a))
print(type(1243))

# Outputs: <class 'float'>
print(type(5.0))
print(type(0.0123))

# Output: <class 'Complex'>
print(type(3+3j))
print(type(b))
```

```
<class 'int'>
<class 'int'>
<class 'float'>
<class 'float'>
<class 'complex'>
<class 'complex'>
```

Following are the data types in Python

2.1 Variables

- Python variables do not need explicit declaration to reserve memory space.
- The declaration happens automatically when you assign a value to a variable.
- The equal sign (=) is used to assign values to variables.
- The operand to the left of the = operator is the name of the variable
- And the operand to the right of the = operator is the value stored in the variable.
- You can delete a single object or multiple objects by using the del statement.

Examples

In [45]:

```
counter = 100.00
miles = 100

print (counter)
print (miles)
```

```
100.0
100
```

In [46]:

```
a = 5
print("a =", 5)
```

```
a = 5
```

In [21]:

```
asd = "High five"
print("asd =", asd)
```

```
asd = High five
```

In [23]:

```
city = "Karachi"
print (city)
```

```
Karachi
```

In [15]:

```
#Deleting singl or multiple variables

var = 123
vara = 124
varb = 0.4

del var

del vara, varb

#display an error after deletion
vara+varb
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-15-92a0ff19b17f> in <module>
     10
     11 #display an error after deletion
---> 12 vara+varb

NameError: name 'vara' is not defined
```

2.2 Numbers

- Number data types store numeric values.
- Number objects are created when you assign a value to them.
- This data type stores numeric values.
- They are immutable data types (Means changing the value of Number data type will results in a newly allocated object).
- As you may now complex number consists of two parts real and imaginary, and is denoted by

Examples

In [48]:

```
2+12
```

Out[48]:

```
14
```

In [49]:

```
12 + 3.00 #float conversion
```

Out[49]:

```
15.0
```

In [50]:

```
4**3 #exonent notation
```

Out[50]:

```
64
```

In [7]:

```
x = 2 + 3j # where 2 is the real part of number and 3 is imaginary part of number
print(x)
print(type(x))
```

```
(2+3j)
<class 'complex'>
```

2.3 Strings

- Strings in Python are contiguous set of characters represented in the quotation marks.
- Single or double quotes are used to enclose string.
- Python doesn't support char data type. It will be as String of length one in Python.
- In short, strings are immutable sequence of characters. There are a lot of methods to ease manipulation and creation of strings

Why

```
Python strings are "immutable" which means they cannot be changed after they are created
(Java strings also use this immutable style). Since strings can't be changed, we construct
*new* strings as we go to represent computed values.
```

When

```
When we want to represent text rather numbers
To input name of anything
```

Where

```
 String is used where we need input or hardcoded value of name of anything or any input
where characters are used.
 String. A string is a data type used in programming, such as an integer and floating point
unit, but is used
 to represent text rather than numbers.
 It is comprised of a set of characters that can also contain spaces and numbers. For examp
le, the word "hamburger"
 and the phrase "I ate 3 hamburgers" are both strings.
```

Examples

In [17]:

```
str = 'Hello World!'
print (str)          # Prints whole string
```

```
Hello World!
```

In [18]:

```
print('Lahore')
```

```
Lahore
```

In [19]:

```
num = '123'
print(num)
```

```
123
```

In [20]:

```
print('Pakistan Zindabad_____Pak Armed Forces Paindabad')
```

```
Pakistan Zindabad_____Pak Armed Forces Paindabad
```

Special Operation

- The slice operator ([ ] and [:]) is used to subset of string with indexes starting
- In python strings are immuteale , thus we can update it using, the plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.

In [5]:

```
str = 'Hello World!'
print (str[0])        # Prints first character of the string
```

```
H
```

In [6]:

```
str = 'Hello World!'
print (str[2:5])      #  characters starting from 3rd to 5th
```

```
llo
```

In [7]:

```
str = 'Hello World!'
print (str[2:] )      #starting from 3rd character
```

```
llo World!
```

In [8]:

```
str = 'Hello World!'
print (str * 2)       # string two times
```

```
Hello World!Hello World!
```

In [9]:

```
str = 'Hello World!'
print (str + " TEST") #  concatenated string
```

```
Hello World! TEST
```

Deletion

*As strings in Python are immuteable so we shall use del word to delete whole string

In [11]:

```
string='fauji'
print(string)
del string
print(string)
```

```
fauji
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-11-355df168f7d7> in <module>
      2 print(string)
      3 del string
----> 4 print(string)

NameError: name 'string' is not defined
```

In [22]:

```
string2='comsats'
print(string2)
string3=' uni'
print(string2+string3)
del string2
print(string2+string3)
```

```
comsats
comsats uni
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-22-28af23919767> in <module>
      4 print(string2+string3)
      5 del string2
----> 6 print(string2+string3)

NameError: name 'string2' is not defined
```

In [28]:

```
strg= 'Machine' + ' learning'
print (strg)
del strg
print(strg)
```

```
Machine learning
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-28-3f7591304868> in <module>
      2 print (strg)
      3 del strg
----> 4 print(strg)

NameError: name 'strg' is not defined
```

String formating

Here is a feature in Python to formate a string by using '%' sign and with this sign specified format is used like

- %s for string
- %d for intiger
- %f for floating point values
- single % is used to show result / it refer toward variable whose string should be used as result

In [36]:

```
age = 22
name = 'Hamza'
religion = 'ISLAM'

print('I am %s. My age is %d and my religio is %s'%(name,age,religion))
```

```
I am Hamza. My age is 22 and my religio is ISLAM
```

In [41]:

```
weight= 12.5
name = 'sugar'
print('weight of %s is %f'%(name,weight) )
```

```
weight of sugar is 12.500000
```

In [47]:

```
#String formating can also be used for list
mylist = [123.0,123,2]
print('My list is %s'%mylist)
```

```
My list is [123.0, 123, 2]
```

2.4. List

- List is used to store different value together
- [] is used for list
- indexes starts from 0
- comma is used to seperate different values
- Differnt type of values can be stored in list . i.e string , float , integer at same time
- Lists can have list in them called as nested lists

Why

```
We use list beacuse It is mutable and we can have any number of items and they may be of di
fferent types.
Also, a list can even have another list as an item.
```

When

```
Use lists if you have a collection of data that does not need random access. Try to choose
lists when you need a simple,       iterable collection that is modified frequently.
When we care about order then we use list
We use list when we want to store differnt types of data in a sequence
When dupliacates are nor forbidden
```

Where

```
Lists: You can use these when you need a collection of similar/non similar items
We use list where we want to store data for exaple Your many cats' names and their ages.
```

In [49]:

```
list1 = []
print(list1)
```

```
[]
```

In [51]:

```
list2 = [1.2,123,'Hi!']
print(list2)
```

```
[1.2, 123, 'Hi!']
```

In [54]:

```
list3 = ['XYZ',123, [1,2,3]]
print(list3)
```

```
['XYZ', 123, [1, 2, 3]]
```

Accessing

- [] double brackets can be used to declare list
- index number can be used to acces particular item from list
- range of list can be acccessed using [:]

In [64]:

```
listt = ['ali', 'ismail', 'shahid' ,1,2,3 ]
print(listt[2])
```

```
shahid
```

In [62]:

```
listt = ['ali', 'ismail', 'shahid' ,1,2,3 ]
print(listt[1 :5 ])
```

```
['ismail', 'shahid', 1, 2]
```

In [63]:

```
listt = ['ali', 'ismail', 'shahid' ,1,2,3 ]
print(listt[2 : ])
```

```
['shahid', 1, 2, 3]
```

Updating

- we can update list beacuse it is muteable unlike string
- we can use '=' to update list
- index can be given to add iytem at particular position
- append and extend can also be used to update list
- ' + ' is used to concatenate lists
- ' * ' is used to display list multiple times

```
listt = ['ali', 'ismail', 'shahid' ,1,2,3 ]
listt[2] = 'akmal'
print(listt)
```

In [68]:

```
listt = ['ali', 'ismail', 'shahid' ,1,2,3 ]
listt[0:2] = [1,2,3]
print(listt)
```

```
[1, 2, 3, 'shahid', 1, 2, 3]
```

In [74]:

```
listt = ['ali', 'ismail', 'shahid' ,1,2,3 ]
listt.append ('Adeel')
print(listt)
```

```
['ali', 'ismail', 'shahid', 1, 2, 3, 'Adeel']
```

In [78]:

```
listt = ['ali', 'ismail', 'shahid' ,1,2,3 ]
listt.extend (['Machine learning'])
print(listt)
```

```
['ali', 'ismail', 'shahid', 1, 2, 3, 'Machine learning']
```

In [81]:

```
listt = ['Pakistan', ]

print(listt + ['Lahore'])
```

```
['Pakistan', 'Lahore']
```

In [85]:

```
listt = ['Pakistan Zindabad' ]

print(listt *2)
```

```
['Pakistan Zindabad', 'Pakistan Zindabad']
```

Deletion

- we can delete or empty list using del , pop, remove and clear
- del can be used to delete particular item at given index and to delete whole list
- pop can be used to delete at index and to pop item from end of list
- remove can be used to remove given item
- clear can be used to clear whole list

In [88]:

```
listt = ['cs','bba','chem','phy','archi']

del listt[2]
print(listt)

del listt[1:5]
print(listt)

del listt
print(listt)
```

```
['cs', 'bba', 'phy', 'archi']
['cs']

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-88-6c438ac353b7> in <module>
      8
      9 del listt
---> 10 print(listt)

NameError: name 'listt' is not defined
```

In [103]:

```
listt = ['lahore','karachi',13,'fsd','multan',12,12.2]
listt.remove(12)
print(listt)
```

```
['lahore', 'karachi', 13, 'fsd', 'multan', 12.2]
```

In [99]:

```
listt = ['lahore','karachi',13,'fsd','multan',12,12.2]

print(listt.pop(4)) # display poped item
print(listt)
```

```
multan
['lahore', 'karachi', 13, 'fsd', 12, 12.2]
```

In [105]:

```
listt = ['lahore','karachi',13,'fsd','multan',12,12.2]


print(listt.pop()) #display poped item

print(listt)

listt.clear()
print(listt)  #cleared list
```

```
12.2
['lahore', 'karachi', 13, 'fsd', 'multan', 12]
[]
```

2.5 Dictionaries

- Dictionary is unordered collection
- Other have only value is collects like list tuple etc
- Dictionary have key : value pair
- Each key : value pair is seperated by comma (,)
- It is like array
- Key of dictionary is immutable like string , number etc
- Curly brackets {} are used for dictionaries

Why

```
Python dictionary is an implementation of a hash table and is a key-value store. It is not
ordered and it requires that         the keys are hashtable.
For speedy key(membership) checking
```

When

```
When you need a logical association between a key:value pair.
When you need fast lookup for your data, based on a custom key.
When your data is being constantly modified. Remember, dictionaries are mutable.
When we don't care aboput order then we use dictionaries
When we want to associate a single value to each key then we use dictionaries
```

Where

```
We use dictionaries where we want to store data like roll numbers of differnt students
their names or differnt booksISBN and their name(with key:value format)
```

Accessing

- we can acces dictionary using name of key key in square brackets along with dictionary name

- we can acces dictionary using name of key key in square brackets along with dictionary name
- dictionary can be crreated usiny dict() function.

In [108]:

```
dictt = {'country': 'Pakistan', 'continent': 'Asia', 'independece': 1947}
print (dictt['country'])
print (dictt)
```

```
Pakistan
{'country': 'Pakistan', 'continent': 'Asia', 'independece': 1947}
```

In [113]:

```
dict = {'Name':"Hamza", 'Depart': "BCS", 'Batch': 'FA16'}
print (dict['Batch'])
```

```
FA16
```

In [115]:

```
dictt = {'Name':"Hamza", 'height': 173 , 'weight': 65.5,'color': 'brown','age': 22}
print (dictt)
```

```
{'Name': 'Hamza', 'height': 173, 'weight': 65.5, 'color': 'brown', 'age': 22}
```

Updating

- dictonary is mutable
- we can edit or add item in it
- If key is avaiable then it is updated otherwise new key : value is added
- We can use update function to updated or append dictionary
- update function can also be used to append one dictionary to another dictionary

In [3]:

```
dict = {'name':'Adeel', 'age': 35}

dict['age'] = 27
print(dict)

dict['address'] = 'Downtown'
print(dict)
```

```
{'name': 'Adeel', 'age': 27}
{'name': 'Adeel', 'age': 27, 'address': 'Downtown'}
```

In [19]:

```
dict = {'Name': 'Hamza', 'Age': 22, 'degree': 'BS'}
dict['Age'] = 23;
dict['University'] = "CUI";

print (dict)
```

```
{'Name': 'Hamza', 'Age': 23, 'degree': 'BS', 'University': 'CUI'}
```

In [20]:

```
dict = {'Name': 'Hamza', 'Age': 22, 'degree': 'BS'}
dict.update({'Age':23})
dict.update({'University': "CUI"})

print (dict)
```

```
{'Name': 'Hamza', 'Age': 23, 'degree': 'BS', 'University': 'CUI'}
```

Deleting

- del() can be used to delete individual item or delete whole dictionary
- clear() can be used to clear whole dictionaryu
- pop() can be used to remove an item with provided key and return value
- popitem() is usd to remnove key and return arbitrary value

In [25]:

```
dict = {'name':'Adeel', 'age': 35}

print(dict)
del dict['age']
print(dict)
```

```
{'name': 'Adeel', 'age': 35}
{'name': 'Adeel'}
```

In [31]:

```
dict = {'name':'Adeel', 'age': 35}

print(dict)
dict.clear()
print(dict)
```

```
{'name': 'Adeel', 'age': 35}
{}
```

In [34]:

```
dict = {'name':'Adeel', 'age': 35 ,'Institute':'CUI'}

print(dict)
dict.popitem()
print(dict)
```

```
{'name': 'Adeel', 'age': 35, 'Institute': 'CUI'}
{'name': 'Adeel', 'age': 35}
```

In [42]:

```
dict = {'name':'Adeel', 'age': 35}

print(dict)
dict.pop('name')
print(dict)
```

```
{'name': 'Adeel', 'age': 35}
{'age': 35}
```

Membership Test

- In dictionary we can do membership test
- Membership test can be done only on key not value
- It returns true if existed else false

In [5]:

```
dict = {'one': 1, 'three': 9, 'five': 25, 'seven': 49, 'nine': 81}
```

```
print('one' in dict)

print('two' not in dict)

print(9 in dict)
```

```
True
True
False
```

2.6. Tuple

- tuple is similar to list
- But difference is that tuple is immutable while list can be changed
- Tumple can be created just putting coma in elements and paranthesis around all elements
- For even a single value we have to write comma after that value
- its indices starts from 0
- It can consist of of multiple data types

Why

```
We use tuple because is is immutable and and it is faster
It makes code fast and safer
```

When

```
When we need fixed size sequence then we use tuple and faster result
And when we don't need to chnage or edit sequence(when your data cannot change.)
```

Where

```
Tuples are generally used for smaller groups of similar items, things like coordinate
systems
```

Accessing

- Tuple can be accessed through slicing , contcatenatio, indexexing and negative indexing
- Tuples can be used inside tuple (nested tuple)

In [43]:

```
tuple = (1, "Hello", 3.4)
print(tuple)

tuple = ("Ali", [8.3, 2.4, 6], (1, 'Ali', 3))
print(tuple)
```

```
(1, 'Hello', 3.4)
('Ali', [8.3, 2.4, 6], (1, 'Ali', 3))
```

In [25]:

```
tuple = ("Ali", [8.3, 2.4, 6], (1, 'Ali', 3))
print(tuple[1])
```

```
[8.3, 2.4, 6]
```

In [26]:

```
tuple = ("Ali", [8.3, 2.4, 6], (1, 'Ali', 3))
```

```
print(tuple[1 : 3])
```

```
([8.3, 2.4, 6], (1, 'Ali', 3))
```

In [27]:

```
tuple = ("Ali", [8.3, 2.4, 6], (1, 'Ali', 3))
print(tuple[1][2])
```

```
6
```

In [19]:

```
tuple = ("Ali", [8.3, 2.4, 6], (1, 'Ali', 3))
print(my_tuple[-1])
```

```
(1, 'Ali', 3)
```

Updating

- Tuples are imutable , so we can not edit it
- We can combine two tuples and make a new tuple
- Nested elements of tuple can be changes
- Tuple cannot be changed but reassigned
- Tuple can be repeared using *

In [22]:

```
tup1 = (12.0, 34.,'qsd56')
tup2 = ('abc', 'xyz')
tup3 = tup1 + tup2; print (tup3)
```

```
(12.0, 34.0, 'qsd56', 'abc', 'xyz')
```

In [31]:

```
tuple = ("Ali", [8.3, 2.4, 6], [1, 'Ali', 3])
tuple[1][2]=312
print(tuple)
```

```
('Ali', [8.3, 2.4, 312], [1, 'Ali', 3])
```

In [50]:

```
tuple = ("Ali", [8.3, 2.4, 6], 1, 'Ali', 3)
print(tuple)

tuple = ("Ali",1, 'Ali', 3)
print(tuple)
```

```
('Ali', [8.3, 2.4, 6], 1, 'Ali', 3)
('Ali', 1, 'Ali', 3)
```

Deleting

- Tuple is immutablle
- del is used to remove entire tuple because we cannot remove individual item

In [51]:

```
tuple = ('pet','ra','got','gun','rim')
```

```
del tuple
print(tuple)
```

```
<class 'tuple'>
```

In [55]:

```
tuple = ("Ali", [8.3, 2.4, 6], 1, 'Ali', 3)
print(tuple)
del tuple
print(tuple)
```

```
('Ali', [8.3, 2.4, 6], 1, 'Ali', 3)
<class 'tuple'>
```

In [58]:

```
tuple = ("Ali", 'saqib', 'Faris', 3)
print(tuple)
del tuple
print(tuple)
```

```
('Ali', 'saqib', 'Faris', 3)
<class 'tuple'>
```

2.7 Sets

- collection of unordered items.
- No duplicates in collection.
- It is immutable.
- Set itself is mutable.We can add or remove items from it.
- Sets are used to do operations like union, intersection, symmetric difference etc.
- Empty set is written as {}. items in set are seperated by comma (,) .
- We can make a set from a list using set() function.
- Data type can be found using type() function.

Why

```
We use set because the major advantage as opposed to a list, is that it has a highly
optimized method for checking
whether a specific element is contained in the set.
```

When

```
Use a set if you need uniqueness for the elements
```

Where

```
We use sets in real life when we want to organize books, organizing audio playlist etc
```

In [68]:

```
listt = [1,3,2,3 ]
sett= set(listt)
print(sett)
```

```
{1, 2, 3}
```

In [73]:

```
        listt = [1,3,2,3 ]
        sett= set(listt)
        print(type(sett))
```

```
<class 'set'>
```

In [75]:

```
        listt = [1,'ali',3,2,'adeel',3 ]
        sett= set(listt)
        print(sett)
```

```
{1, 2, 3, 'ali', 'adeel'}
```

Updating

- add() is used to add single value
- update() is used for adding multiple values.
- update() function can take tuple, strings, list or other set as argument.

In [80]:

```
                set = {1,3,'asd',5,'7'}
                set.add(2)
                print(set)
```

```
{1, 2, 3, 5, '7', 'asd'}
{1, 2, 3, 5, 'ali', 'adeel', '7', 'asd'}
```

In [88]:

```
                set = {1,3,'asd',5,'7'}
                set.update([2,3,4])
                print(set)
```

```
{1, 2, 3, 4, 5, '7', 'asd'}
```

In [91]:

```
                set = {1,3,'asd',5,'7'}
                listt = [1,'ali',3,2,'adeel',3 ]
                set.update(listt)
                print(set)
```

```
{1, 2, 3, 5, 'ali', 'adeel', '7', 'asd'}
```

Deleting

- discard() and remove() are used to to delete particular item from set.
- discard() will not give an error if item doesn't exists in set.
- remove() will give an error if item doesn't exists in set.
- clear is used to remove all items
- pop is used to return and remove item

In [114]:

```
            set = {1,'ali',3,2,'adeel',3 }
            set.discard(1)
            print(set)

            #weill not give an error
            set.discard(8)
```

```
            print(set)
```

```
{1, 2, 3, 'ali', 'adeel'}
```

In [116]:

```
            set = {1,'ali',3,2,'adeel',3 }

            set.remove(2)
            print(set)

            #Will eraise an error message
            set.remove(8)
            print(set)
```

```
{1, 3, 'ali', 'adeel'}
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-116-e94f3a7456d9> in <module>
      5
      6 #Will eraise an error message
----> 7 set.remove(8)
      8 print(set)
      9

KeyError: 8
```

In [110]:

```
            set = {1,'ali',3,2,'adeel',3 }

            print(set.pop())
```

```
1
```

In [107]:

```
            set = {1,'ali',3,2,'adeel',3 }

            set.clear()
            print(set)
```

```
set()
```

Set Operaqtions

- Set can be used tom perform union , intersection , difference and symmetric difference

In [121]:

```
            # UNUION
            # Display all elemnts of both sets
            A = {'a', 's', 'd', 'f','g' }
            B = {'f', 'g', 'h', 'j', 'k'}
            print(A | B)
```

```
{'g', 'a', 'd', 'f', 'h', 'k', 's', 'j'}
```

In [122]:

```
            # Intersection
```

```
# Dislay common elemens from both sets
A = {'a', 's', 'd', 'f','g' }
B = {'f', 'g', 'h', 'j', 'k'}
print(A | B)
```

```
{'g', 'a', 'd', 'f', 'h', 'k', 's', 'j'}
```

In [123]:

```
# Differnce
# Display elemts only in A not i B
A = {'a', 's', 'd', 'f','g' }
B = {'f', 'g', 'h', 'j', 'k'}
print(A - B)
```

```
{'d', 'a', 's'}
```

In [124]:

```
# Symmetric Difference
# Display all elemts of both sents except those which are common
A = {'a', 's', 'd', 'f','g' }
B = {'f', 'g', 'h', 'j', 'k'}
print(A ^ B)
```

```
{'a', 's', 'k', 'd', 'j', 'h'}
```

3. Comparision Operators

- Comparision operators are basically relational operators, they decide relation between values
- Comparision operators are used for strings and numbers
- In string it checks the size of string and capital/small alphabets also matters in comparision (ASCII values of character are used for comparision)
- It returns true or false in output
- Comparision operators are mostly used in If--Else and loops

In [3]:

```
2>3
```

Out[3]:

```
False
```

In [4]:

```
23 < 43
```

Out[4]:

```
True
```

In [7]:

```
23!=32
```

Out[7]:

```
True
```

In [10]:

```
33 == 34
```

Out[10]:

False

In [12]:

```
33<=33
```

Out[12]:

True

In [15]:

```
45>=43
```

Out[15]:

True

In [17]:

```
'Turkey' == 'Istambol'
```

Out[17]:

False

In [23]:

```
'Turkey' <= 'turk'
```

Out[23]:

True

In [26]:

```
'Turkey' >= 'Turk'
```

Out[26]:

True

In [29]:

```
'Turkey' >= 'turk'
```

Out[29]:

False

4. IF-Else Statements

- IF-Else is used for dicision making
- It is used in code when we want to run particular part of code instead of running whole code
- Only that part of IF-Else executes whose ccondition become fullfiled
- elif is used for else if

In [86]:

```
num = 2
```

```
if num > 1:
    print("greater")
elif num < 1:
    print("Lesser")
else :
    print("nothing")
```

```
greater
```

In [87]:

```
num = -3312
if num > 0:
    print("positive NUMBER")
else:
    print("NEGATIVE NUMBER")
```

```
NEGATIVE NUMBER
```

In [68]:

```
if (2623 % 2) == 0 :
    print("EVEN")
else:
    print("ODD")
```

```
ODD
```

5.For And While Loop

- When we want to perform iterations , we uses loops
- Three types of loops for, while and do-while
- For example when we want to take inputs many times etc
- For loops is used when particular number of iterations needs to perform
- While loop is used when we dont know , how many times iteratuons would be performed
- nested loops can also be used for example we can use for , while or do-while loop in between for , while or do while loop
- Normally code executes sequentially but there are some circumstancse when we need to run a patch of code several times so we use loop
- We can use range function to limit the iterations

In [66]:

```
fact = 1
N = 5
for i in range (1,N+1):
    fact*=i

print (fact)
```

```
120
```

In [67]:

```
numbers = [41, 1, 18, 4, 32]
sum = 0

for val in numbers:
    sum = sum+val

print("Sum = ", sum)
```

```
Sum =  96
```

In [77]:

```
list = ['Lahore', 'karachi', 'Islamabad','Peshawer','']

for i in range(len(list)):
    print( list[i])
```

```
Lahore
karachi
Islamabad
Peshawer
```

In [79]:

```
n = 5
fact = 1

i = 1
while i <=n :
    fact*=i
    i=i+1


print(fact)
```

```
120
```

Loop Control Statements

- **break** terminates loop and move forward toward the immediate following loop
- **Continue** it skips the value of the current iteration based on condition and then other iteration will go on normally

In [83]:

```
for val in "pakistano":
    if val == "o":
        break
    print(val)
```

```
p
a
k
i
s
t
a
n
```

In [6]:

```
for val in "pakisltan":
    if val == "l":
        continue
    print(val)
```

```
p
a
k
i
s
t
a
n
```

6. Functions

- Function is used to perform specific task.
- Functions is reuseable, once it is crreated then i can be used anywhere
- It can also reduce length of code
- Functions can be defined as **def functionName(Parameters):**
- Functions is used to make program manageable
- Keyword **def** marks the start of function header
- we can pass value through **parameters**. They are optional
- **colon (:)** used to mark the end of function header
- Optional documentation string **(docstring)** to describe what the function does
- An optional **return statement** to return a value from the function.It is also optional
- Tuple / List etc can be passed as parameter in function
- a Function can be declared which will call another function again and again untill condition satisfied , this is called as **recursive function**
- There are many **built in** functions in Python

```
def functionname( parameters ):
"function_docstring"
function_suite
return [expression]
```

In [7]:

```
def printme( str ):
        print (str)
        return;
printme("I am Hamza")
```

```
I am Hamza
```

In [11]:

```
def abslt(num):

    if num >= 0:
        return num
    else:
        return -num

print(abslt(32))
print(abslt(-24))
```

```
32
24
```

In [13]:

```
def add(num1,num2):
    num1+num2

print(abslt(32+23))
print(abslt(-24+54))
```

```
55
30
```

In [30]:

```
#Tuple is passesd as parameter
def greet(*names):

    for name in names:
       print(name)
```

```
names = ["Muhammad","Ali","Manan","Nabeel"]

greet(names)
```

```
['Muhammad', 'Ali', 'Manan', 'Nabeel']
```

In [29]:

```
# Recursive Function for factorial
def factorial(x):

    if x == 1:
        return 1
    else:
        return (x * factorial(x-1))

num = 4
print("Factorial = ",factorial(num))
```

```
Factorial =  24
```

Built-In Functions

There are some built-in functions in Python, which are already defined in python

- **abs()** returns absolute value of a number
- **all()** returns true when all elements in iterable is true
- **any()** Checks if any Element of an Iterable is True
- **ascii()** Returns String Containing Printable Representation
- **bin()** converts integer to binary string
- **bool()** Converts a Value to Boolean
- **bytearray()** returns array of given byte size
- **bytes()** returns immutable bytes object
- **callable()** Checks if the Object is Callable
- **chr()** Returns a Character (a string) from an Integer
- **classmethod()** returns class method for given function
- **compile()** Returns a Python code object
- **complex()** Creates a Complex Number
- **delattr()** Deletes Attribute From the Object
- **dict()** Creates a Dictionary
- **dir()** Tries to Return Attributes of Object
- **divmod()** Returns a Tuple of Quotient and Remainder
- **enumerate()** Returns an Enumerate Object
- **eval()** Runs Python Code Within Program
- **exec()** Executes Dynamically Created Program
- **filter()** constructs iterator from elements which are true
- **float()** returns floating point number from number, string
- **format()** returns formatted representation of a value
- **frozenset()** returns immutable frozenset object
- **getattr()** returns value of named attribute of an object
- **globals()** returns dictionary of current global symbol table
- **hasattr()** returns whether object has named attribute
- **hash()** returns hash value of an object
- **help()** Invokes the built-in Help System
- **hex()** Converts to Integer to Hexadecimal
- **id()** Returns Identify of an Object
- **input()** reads and returns a line of string
- **int()** returns integer from a number or string
- **isinstance()** Checks if a Object is an Instance of Class
- **issubclass()** Checks if a Object is Subclass of a Class
- **iter()** returns iterator for an object
- **len()** Returns Length of an Object
- **list()** Function creates list in Python
- **locals()** Returns dictionary of a current local symbol table

- **map()** Applies Function and Returns a List
- **max()** returns largest element
- **memoryview()** returns memory view of an argument
- **min()** returns smallest element
- **next()** Retrieves Next Element from Iterator
- **object()** Creates a Featureless Object
- **oct()** converts integer to octal
- **open()** Returns a File object
- **ord()** returns Unicode code point for Unicode character
- **pow()** returns x to the power of y
- **print()** Prints the Given Object
- **property()** returns a property attribute
- **range()** return sequence of integers between start and stop
- **repr()** returns printable representation of an object
- **reversed()** returns reversed iterator of a sequence
- **round()** rounds a floating point number to ndigits places.
- **set()** returns a Python set
- **setattr()** sets value of an attribute of object
- **slice()** creates a slice object specified by range()
- **sorted()** returns sorted list from a given iterable
- **staticmethod()** creates static method from a function
- **str()** returns informal representation of an object
- **sum()** Add items of an Iterable
- **super()** Allow you to Refer Parent Class by super
- **tuple()** Function Creates a Tuple
- **type()** Returns Type of an Object
- **vars()** Returns **dict** attribute of a class
- **zip()** Returns an Iterator of Tuples
- **import**() Advanced Function Called by import

7. Lambda Function

- In Python, anonymous function is a function that is defined without a name.
- While normal functions are defined using the **def** keyword, in Python anonymous functions are defined using the **lambda** keyword.
- Anonymous functions are also called lambda functions.
- It has following syntax

```
**lambda arguments: expression**
```

- Lambda functions can have any number of arguments but only one expression. The expression is evaluated and returned.
- Lambda functions can be used wherever function objects are required.
-

In [32]:

```
times3 = lambda var:var*3
times3(10)
```

Out[32]:

```
30
```

In [33]:

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

```
15
```

In [41]:

```
x = lambda a, b : a * b
print(x(5, 6))
```

```
30
```

In [45]:

```
        x = lambda a, b : a**b

    print(x(5, 5))
```

```
3125
```

7.1 map()

- Map() function is used with two arguments. Its syntax is as follow

```
r = map(func, seq)
```

- The first argument func is the name of a function and
- the second a sequence (e.g. a list,tumple,set). seq. map() applies the function func to all the elements of the sequence .
- And returns a modified sequence changed by func.

In [77]:

```
sentence = 'It is raining cats and dogs'
words = sentence.split()
print (words)
lengths = map(lambda word: len(word), words)
print(tuple(lengths))
```

```
['It', 'is', 'raining', 'cats', 'and', 'dogs']
(2, 2, 7, 4, 3, 4)
```

In [78]:

```
def calculateSquare(n):
  return n*n

numbers = (1, 2, 3, 4,23,5)
result = map(calculateSquare, numbers)
print(tuple(result))
```

```
(1, 4, 9, 16, 529, 25)
```

In [76]:

```
num1 = [7,5,3]
num2 = [4,2,1]

result = map(lambda n1, n2: n1+n2, num1, num2)
print(tuple(result))
```

```
{11, 4, 7}
```

In [83]:

```
strg = ['shock','Knock','Rock','Crowd']
result = map(lambda x: x+'ed', strg)
print(tuple(result))
```

```
('shocked', 'Knocked', 'Rocked', 'Crowded')
```

7.2 filter()

- The filter() method constructs an iterator from elements of an iterable for which a function returns true
- The syntax is as follow

```
filter(function, iterable)
```

- The function filter(f,l) needs a function f as its first argument. f returns a Boolean value, i.e. either True or False.
- This function will be applied to every element of the ITERABLE .

In [99]:

```
list = [1, 5, 4, 6, 8, 11, 3, 12]

print( tuple(filter(lambda x: (x+3 == 6) , my_list)))
```

```
(3,)
```

In [93]:

```
fib = [0,1,1,2,3,5,8,13,21,34,55]
result1 = filter(lambda x: x % 2, fib)
tuple(result1)
```

Out[93]:

```
(1, 1, 3, 5, 13, 21, 55)
```

In [101]:

```
fib = [0,1,1,2,3,5,8,13,21,34,55]
result2 = filter(lambda x: x % 2 == 0, fib)
tuple(result2)
```

Out[101]:

```
(0, 2, 8, 34)
```

In [105]:

```
strg = ['shock','Knock','Rock','Crowd']
print( tuple(filter(lambda x: x+'ed' =='Rocked', strg) ))
```

```
('Rock',)
```

8. File I/O

- We are going to discuss all basic functions of I/O
- Python has several functions for creating, reading, updating, and deleting files.

Reading Input From Keyboard

- Python provides two built-in functions to read a line of text from standard input, which by default comes from the keyboard. **These functions are**

```
raw_input
input
```

- The raw_input([prompt]) function reads one line from standard input and returns it as a string (removing the trailing newline).
- The input([prompt]) function is equivalent to raw_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

In [26]:

In [26]:

```
#to check type of input
name = input("What is your name? ")
type(name)
```

```
What is your name? Hamza
```

Out[26]:

```
str
```

In [27]:

```
#whatever you entered that will convertted to string
str = input("Enter your input: ");
print ("you entered : ", str)
type(str)
```

```
Enter your input: CUI
you entered :  CUI
```

Out[27]:

```
str
```

In [28]:

```
#int input
i = int(input("Enter your input: "));
print ("you entered : ", i)
type(i)
```

```
Enter your input: 123
you entered :  123
```

Out[28]:

```
int
```

In [29]:

```
#float input
f = float(input("Enter your input: "));
print ("you entered : ", f)
type(f)
```

```
Enter your input: 123.123
you entered :  123.123
```

Out[29]:

```
float
```

In [2]:

```
#float input
f = complex(input("Enter your input: "));
print ("you entered : ", f)
type(f)
```

```
Enter your input: 12+2j
you entered :  (12+2j)
```

Out[2]:

```
complex
```

I/O from or to Text File

- **"r"** - Read - Default value. Opens a file for reading, error if the file does not exist
- **"r+"**- Read/Write - opens a file read and write mode.
- **"a"** - Append - Opens a file for appending, creates the file if it does not exist
- **a+** - Append and Read - opens a file in append and read mode.
- **"w"** - Write - Opens a file for writing, creates the file if it does not exist
- **"x"** - Create - Creates the specified file, returns an error if the file exists

In addition you can specify if the file should be handled as binary or text mode

- **"t"** - Text - Default value. Text mode
- **"b"** - Binary - Binary mode (e.g. images)

The Syntax is as follow

```
f = open("demofile.txt", "rt")
f = open("demofile.txt")
```

In [98]:

```
fileOpen = open("ML.txt", "w")
str = fileOpen.write("Hamza");
fileOpen.close()

fileOpen = open("ML.txt", "r+")
str = fileOpen.read();
print(str)
fileOpen.close()
```

```
Hamza
```

In [66]:

```
fileOpen = open("ML.txt", "a+")
fileOpen.write("\nCUI Lahore");
fileOpen.close()

fileOpen = open("ML.txt", "r+")
string = fileOpen.read();
print (string)
fileOpen.close()
```

```
Hamza
CUI Lahore
```

In [60]:

```
ML = open("ML.txt", "wb")
print ("Name of the file: ", ML.name)

ML.close()
```

```
Name of the file:  ML.txt
```

In [70]:

```
# Read Limited length of file
fileOpen = open("ML.txt", "r")
str = fileOpen.read(7)
print(str)
fileOpen.close()
```

```
Hamza
C
```

In [68]:

```
ML = open("ML.txt", "rb")
print ("Closed of not ", ML.closed)

ML.close()
print ("Closed of not ", ML.closed)
```

```
Closed of not  False
Closed of not  True
```

File Position

For findind file position we use **seek()** and **tell()**

- The method tell() returns the current position of the file read/write pointer within the file.
- The method seek() sets the file's current position at the offset. The whence argument is optional and defaults to 0, which means absolute file positioning, other values are 1 which means seek relative to the current position and 2 means seek relative to the file's end.

In [81]:

```
# Read Limited length of file
fileOpen = open("ML.txt", "r")
str = fileOpen.read(7)
print(str)
print("Position is ",fileOpen.tell())
fileOpen.close()
```

```
Hamza
C
Position is  8
```

In [80]:

```
# Read Limited length of file
fileOpen = open("ML.txt", "r")
str = fileOpen.read(4)
print(str)
print("Position is ",fileOpen.tell())
fileOpen.close()
```

```
Hamz
Position is  4
```

In [82]:

```
# Read Limited length of file
fileOpen = open("ML.txt", "r")
str = fileOpen.read()
print(str)
print("Position is ",fileOpen.tell())
fileOpen.close()
```

```
Hamza
CUI Lahore
Position is  17
```

In [83]:

```
# Read Limited length of file
fileOpen = open("ML.txt", "r")
str = fileOpen.read(7)
print(str)

# setting position of pointer at 2
print("New Position is ",fileOpen.seek(2))
str = fileOpen.read(12)
```

```
            print(str)
            fileOpen.close()
```

```
Hamza
C
New Position is  2
mza
CUI Laho
```

In [87]:

```
            # Read Limited length of file
            fileOpen = open("ML.txt", "r")
            # setting position of pointer at 6
            print("New Position is ",fileOpen.seek(6))
            str = fileOpen.read(12)
            print(str)
            fileOpen.close()
```

```
New Position is  6

CUI Lahore
```

In [99]:

```
            import os
            os.rename("ML.txt", "LM.txt")
            ML = open("LM.txt", "rb")
            print ("File name is  ", ML.name)
            ML.close()
```

```
File name is   LM.txt
```

In [100]:

```
            os.remove("LM.txt")
```

----------9. Panda's introduction----------

- Pandas is an open-source Python Library
- Providing high performance data manipulation and analysis tool using its powerful data structures.
- The name Pandas is derived from the word Panel Data
- Developer **Wes McKinney In 2008**, started developing pandas when in need of high performance, flexible tool for analysis of data.
- Prior to Pandas Python was majorly used for data munging and preparation.
- It had very little contribution towards data analysis. Pandas solved this problem. Using Pandas.
- we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data

```
load
prepare
manipulate
model
analyze.
```

- Python with Pandas is used in a wide range of fields including academic and commercial domains including

```
finance
economics
Statistics
analytics  etc
```

FEATURES

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.

- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

The Data Structures provided by Pandas are of two distinct types

```
1- Pandas Series
2- Pandas DataFrame
```

10. Series

- A Series is a one-dimensional object that can hold any data type such as integers, floats and strings
- A series is very similar to NumPy array.
- The difference between the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location
- The axis labels are collectively referred to as the index

A pandas Series can be created using the following constructor

```
pandas.Series( data, index = index)
            OR
pandas.Series(data,index = index)
```

```
data  -  data takes various forms like ndarray, list, constants

Index -  Index values must be unique and hashable,same length as data.Default np.arrange(n)
if no index is passed

dtype -  dtype is for data type. If None, data type will be inferred

copy  -  Copy data. Default False
```

A series can be created using various inputs like

- ndArray
- Dict
- Scalar value or constant

10.1 From ndArray

- If data is an ndarray, index must be the same length as data.
- If no index is passed, one will be created having values [0, ..., len(data) - 1].

In [3]:

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

s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
s
```

Out[3]:

```
a    0.755784
b   -0.379226
c   -1.665673
d    0.262326
e    0.655381
dtype: float64
```

In [6]:

```
import pandas as pd
import numpy as np

data = np.array(['a','b','c','d'])
s = pd.Series(data,index=[100,101,102,103])
s
```

Out[6]:

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

In [8]:

```
import pandas as pd
import numpy as np

data = np.array(['a','b','c','d'])
s = pd.Series(data)
s
```

Out[8]:

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

In [10]:

```
import pandas as pd
import numpy as np

data = np.array(['g','e','e','k','s','f', 'o','r','g','e','e','k','s'])
ser = pd.Series(data)

print(ser[2:7])
```

```
2    e
3    k
4    s
5    f
6    o
dtype: object
```

In [14]:

```
import pandas as pd
import numpy as np

data = np.array(['g','e','e','k','g','e','e','k','s'])
ser = pd.Series(data)

print(ser[5:])
```

```
5    e
6    e
7    k
8    s
dtype: object
```

10.2 From Dict

- If data is a dict, if index is passed the values in data corresponding to the labels in the index will be pulled out.
- If index is not passed then it will be constructed from the sorted keys of the dict, if possible.

In [18]:

```
import pandas as pd

dictionary = {'D' : 10, 'B' : 20, 'C' : 30}
series = pd.Series(dictionary)

print(series)
```

```
D    10
B    20
C    30
dtype: int64
```

In [22]:

```
import pandas as pd

dictionary = {'name' : 'hamza', 'Reg#' : 'FA16-BCS-259', 'Institute' : 'CUI'}
series = pd.Series(dictionary)

print(series)
```

```
name                hamza
Reg#          FA16-BCS-259
Institute              CUI
dtype: object
```

In [6]:

```
import pandas as pd

dictionary = {'name' : 'hamza', 'Reg#' : 'FA16-BCS-259', 'Institute' : 'CUI'}
series = pd.Series(dictionary,index=['name','age','Reg#','Institute','City'])

print(series)
```

```
name                hamza
age                   NaN
Reg#          FA16-BCS-259
Institute              CUI
City                  NaN
dtype: object
```

10.3 From Scaler Value

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

In [55]:

```
import pandas as pd
series = pd.Series(12,index=['name','age','Reg#','Institute','City'])

print(series)
```

```
name         12
age          12
Reg#         12
Institute    12
City         12
dtype: int64
```

In [38]:

```
import pandas as pd
series = pd.Series(12,index=['z','a','b','c','d'])

print(series)
```

```
z    12
a    12
b    12
c    12
d    12
dtype: int64
```

In [37]:

```
import pandas as pd
series = pd.Series(12,index=[1,2,3,4])

print(series)
```

```
1    12
2    12
3    12
4    12
dtype: int64
```

10.4 Series is ndArray -like

- It acts very similarly to a ndarray. It is a valid argument to most NumPy functions.
- However, things like slicing also slice the index.

In [47]:

```
import pandas as pd
data = np.array([1,42,34,62,23,52,3])
series = pd.Series(data)
series[series < series.median()]
```

Out[47]:

```
0     1
4    23
6     3
dtype: int32
```

In [49]:

```
import pandas as pd
data = np.array([1,42,34,62,23,52,3])
series = pd.Series(data)
series[series < 50]
```

Out[49]:

```
0     1
1    42
2    34
4    23
6     3
dtype: int32
```

In [60]:

```
import pandas as pd
data = np.array([1,1,2,1,4,1,4,2,3])
series = pd.Series(data)
series[series == 1 ]
```

Out[60]:

```
0    1
1    1
3    1
5    1
dtype: int32
```

In [65]:

```
import pandas as pd

dictionary = {'name' : 'hamza', 'Reg#' : 'FA16-BCS-259', 'Institute' : 'CUI'}
series = pd.Series(dictionary,index=['1','age','Reg#','Institute','City'])

series[2:3]
```

Out[65]:

```
Reg#    FA16-BCS-259
dtype: object
```

10.6 Vectorized operations and label alignment with Series

- When doing data analysis, as with raw NumPy arrays looping through Series value-by-value is usually not necessary.
- Series can also be passed into most NumPy methods expecting an ndarray.

In [67]:

```
import pandas as pd
data = np.array([1,1,2,1,4,1,4,2,3])
series = pd.Series(data)
series+series
```

Out[67]:

```
0    2
1    2
2    4
3    2
4    8
5    2
6    8
7    4
8    6
dtype: int32
```

In [69]:

```
import pandas as pd
data = np.array([1,1,2,1,4,1,4,2,3])
series = pd.Series(data)
series*series
```

Out[69]:

```
0     1
1     1
2     4
3     1
4    16
5     1
6    16
7     4
8     9
dtype: int32
```

In [71]:

```
import pandas as pd
data = np.array([1,23,123,14,1,4,2,3])
series = pd.Series(data)
series**3
```

Out[71]:

```
0          1
1      12167
2    1860867
3       2744
4          1
5         64
6          8
7         27
dtype: int32
```

In [74]:

```
import pandas as pd
data = np.array(['block','smash','Rock','smook'])
series = pd.Series(data)
series+'ed'
```

Out[74]:

```
0    blocked
1    smashed
2     Rocked
3    smooked
dtype: object
```

In [2]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data = np.array(['block','smash','Rock','smook'])
s = pd.Series(data, index=['a', 'b', 'c', 'd'])

s['a']
```

Out[2]:

```
'block'
```

11. Data Frames

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

```
a. Potentially columns are of different types
b. Size - Mutable
c. Labeled axes (rows and columns)
d. Can Perform Arithmetic operations on rows and columns
```

- DataFrames are the workhorse of pandas and are directly inspired by the R programming language.
- Like Series, DataFrame accepts many different kinds of input

```
1.Dict of 1D ndarrays, lists, dicts, or Series
2.2-D numpy.ndarray
3.Structured or record ndarray
4.A Series
5.Another DataFrame
```

A pandas DataFrame can be created using the following constructor

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

- Along with the data, you can optionally pass index (row labels) and columns (column labels) arguments.

- If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame.
- Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.
- If axis labels are not passed, they will be constructed from the input data based on common sense rules

11.1 From dict of Series or dict

- The result index will be the union of the indexes of the various Series.
- If there are any nested dicts, these will be first converted to Series.
- If no columns are passed, the columns will be the sorted list of dict keys.

In [3]:

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

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

df = pd.DataFrame(d)
df
```

Out[3]:

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

In [4]:

```
d = {
    'Country' : pd.Series(['Pakistan','Afghanistan','Iran','Saudia','Turkey'], index=[1
,2,3,4,5]),
    'Population' : pd.Series([220000000,300000000,230000000,1120000000], index=[1,2,3,4
])
}

df = pd.DataFrame(d)
df
```

Out[4]:

	Country	Population
1	Pakistan	2.200000e+08
2	Afghanistan	3.000000e+08
3	Iran	2.300000e+08
4	Saudia	1.120000e+09
5	Turkey	NaN

In [5]:

```
# a data frame will be constructed for given row labels
pd.DataFrame(d, index=[1,3,4,5])
```

Out[5]:

	Country	Population

	Country	Population
1	Pakistan	2.200000e+08
3	Iran	2.300000e+08
4	Saudia	1.120000e+09
5	Turkey	NaN

In [6]:

```
# following example shows a data frame when we give coloumn labels
pd.DataFrame(d, index=[1,2,4], columns=['Country', 'Population'])
```

Out[6]:

	Country	Population
1	Pakistan	220000000
2	Afghanistan	300000000
4	Saudia	1120000000

In [7]:

```
# following example shows a data frame when we give coloumn labels
pd.DataFrame(d, index=[1,2,4], columns=['Country'])
```

Out[7]:

	Country
1	Pakistan
2	Afghanistan
4	Saudia

In [13]:

```
df.columns
```

Out[13]:

```
Index(['Country', 'Population'], dtype='object')
```

In [9]:

```
#Column Selection
df['Population']
```

Out[9]:

```
1    2.200000e+08
2    3.000000e+08
3    2.300000e+08
4    1.120000e+09
5             NaN
Name: Population, dtype: float64
```

11.2 From dict of ndArrays / Lists

- The ndarrays must all be the same length.
- If an index is passed, it must clearly also be the same length as the arrays.
- If no index is passed, the result will be range(n), where n is the array length

In [20]:

```
import pandas as pd
data = {
```

```
                    'Name':['Tom', 'Jack', 'Steve', 'Ricky'],
                    'Age':[28,34,29,42],
                    'Height':[166,156,177,173]
                }
                df = pd.DataFrame(data)
                df
```

Out[20]:

	Name	Age	Height
0	Tom	28	166
1	Jack	34	156
2	Steve	29	177
3	Ricky	42	173

In [25]:

```
            #we can also creat dict from OrderDict
            sales ={   'account': ['Jones LLC', 'Alpha Co', 'Blue Inc'],
                      'Jan': [150, 200, 50],
                      'Feb':  [200, 210, 90],
                      'Mar': [140, 215, 95]
                    }
            df = pd.DataFrame.from_dict(sales)
            df
```

Out[25]:

	account	Jan	Feb	Mar
0	Jones LLC	150	200	140
1	Alpha Co	200	210	215
2	Blue Inc	50	90	95

In [27]:

```
            import pandas as pd
            data = {
                    'Name':['Tom', 'Jack', 'Steve', 'Ricky'],
                    'Age':[28,34,29,42],
                    'Height':[166,156,177,173]
            }
            df = pd.DataFrame(data,index=[1,2,3,4])
            df
```

Out[27]:

	Name	Age	Height
1	Tom	28	166
2	Jack	34	156
3	Steve	29	177
4	Ricky	42	173

In [28]:

```
            # If indexs are given then it would be same length as arrays
            pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
```

Out[28]:

	Country	Population
a	NaN	NaN
b	NaN	NaN

	Country	Population
c	NaN	NaN
d	NaN	NaN

In [29]:

```
# If indexs are given then it would be same length as arrays
pd.DataFrame(data, index=[1,2,3,4])
```

Out[29]:

	Name	Age	Height
1	Tom	28	166
2	Jack	34	156
3	Steve	29	177
4	Ricky	42	173

11.3 From a list of dicts

- List of Dictionaries can be passed as input data to create a DataFrame.
- The dictionary keys are by default taken as column names.
- If no index is passed, the result will be range(n), where n is the list length

In [28]:

```
sales = [{'account': 'Jones LLC', 'Jan': 150, 'Feb': 200, 'Mar': 140},
         {'account': 'Alpha Co',  'Jan': 200, 'Feb': 210, 'Mar': 215},
         {'account': 'Blue Inc',  'Jan': 50,  'Feb': 90,  'Mar': 95 }]
df = pd.DataFrame(sales,index=[1,2,3])
df
```

Out[28]:

	Feb	Jan	Mar	account
1	200	150	140	Jones LLC
2	210	200	215	Alpha Co
3	90	50	95	Blue Inc

In [31]:

```
data2 = [
                {'name': 'Ali', 'Reg#': 'fa16-bcs-259'}, {'name': 'Jamshed', 'Reg#': 'fa16-
bcs-123', 'City': 'Lahore'}
]
pd.DataFrame(data2)
```

Out[31]:

	City	Reg#	name
0	NaN	fa16-bcs-259	Ali
1	Lahore	fa16--bcs-123	Jamshed

In [33]:

```
data2 = [
                {'name': 'Ali', 'Reg#': 'fa16-bcs-259'}, {'name': 'Jamshed', 'Reg#': 'fa16-
bcs-123', 'City': 'Lahore'}
]
pd.DataFrame(data2,index=['First','Second'])
```

Out[33]:

	City	Reg#	name
First	NaN	fa16-bcs-259	Ali
Second	Lahore	fa16--bcs-123	Jamshed

In [35]:

```
# passing list of dicts as data and columns (columns labels)
pd.DataFrame(data2, columns=['name', 'b'])
```

Out[35]:

	name	b
0	Ali	NaN
1	Jamshed	NaN

In [37]:

```
# passing list of dicts as data and columns (columns labels)
pd.DataFrame(data2, columns=['name', 'City'])
```

Out[37]:

	name	City
0	Ali	NaN
1	Jamshed	Lahore

In [39]:

```
# passing list of dicts as data and columns (columns labels)
pd.DataFrame(data2, index=['a', 'b'])
```

Out[39]:

	City	Reg#	name
a	NaN	fa16-bcs-259	Ali
b	Lahore	fa16--bcs-123	Jamshed

11.4. From dict of tuples

- Multi indexed datafeames can be created using dict of tuples

In [49]:

```
pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
              ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
              ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
              ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
              ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}
             })
```

Out[49]:

		a			b	
		b	a	c	a	b
A	**B**	1.0	4.0	5.0	8.0	10.0
	C	2.0	3.0	6.0	7.0	NaN

D	NaN	NaN	NaN	NaN	9.0

In [48]:

```
pd.DataFrame({('1', 'age'): {('1', 'Adeel'): 40, ('1', 'Hamza'):22},
              ('1', 'height'): {('1', 'Adeel'): 167, ('1', 'Hamza'): 172},
              ('1', 'weight'): {('1', 'Adeel'): 80, ('1', 'Hamza'): 66},
              ('2', 'age'):    {('1', 'Adeel'): 40, ('1', 'Hamza'):22},
              ('2', 'height'): {('1', 'Adeel'): 167, ('1', 'Hamza'): 172},
              ('2', 'weight'): {('1', 'Adeel'): 80, ('1', 'Hamza'): 66}
             })
```

Out[48]:

		1			2		
		age	height	weight	age	height	weight
1	Adeel	40	167	80	40	167	80
	Hamza	22	172	66	22	172	66

In [54]:

```
pd.DataFrame({('1', 'a'): {1: 410,2:32,3:14,5:23},
              ('1', 'b'): {1: 410,2:32,3:14,4:23},
              ('1', 'c'): {3: 33},
              ('2', 'a'): {1: 40},
              ('2', 'b'): {1: 167,2: 410,3:32,3:14,5:23},
              ('2', 'c'): {4 : 830},
              })
```

Out[54]:

	1			2		
	a	b	c	a	b	c
1	410.0	410.0	NaN	40.0	167.0	NaN
2	32.0	32.0	NaN	NaN	410.0	NaN
3	14.0	14.0	33.0	NaN	14.0	NaN
4	NaN	23.0	NaN	NaN	NaN	830.0
5	23.0	NaN	NaN	NaN	23.0	NaN

11.5 Alternate Constructor

DataFrame.from_dict

- It takes a dict of dicts or a dict of array-like sequences and returns a DataFrame.
- It operates like the DataFrame constructor except for the orient parameter which is 'columns' by default, but which can be set to 'index' in order to use the dict keys as row labels.

In [38]:

```
sales = {'account': ['Jones LLC', 'Alpha Co', 'Blue Inc'],
         'Jan': [150, 200, 50],
         'Feb': [200, 210, 90],
         'Mar': [140, 215, 95]}
df = pd.DataFrame.from_dict(sales)
df
```

Out[38]:

	account	Jan	Feb	Mar
0	Jones LLC	150	200	140
1	Alpha Co	200	210	215

	account	Jan	Feb	Mar
1	Alpha Co	200	210	215
2	Blue Inc	50	90	95

DataFrame.from_records

- It takes a list of tuples or an ndarray with structured dtype.
- Works analogously to the normal DataFrame constructor except that index maybe be a specific field of the structured dtype to use as the index.

In [52]:

```
            sales = [('Jones LLC', 150, 200, 50),
                     ('Alpha Co', 200, 210, 90),
                     ('Blue Inc', 140, 215, 95)]
            labels = ['account', 'Jan', 'Feb', 'Mar']
            df = pd.DataFrame.from_records(sales, columns=labels)
            print(df)
            sales = np.zeros((3,), dtype=[('account', 'i4'),('Jan', 'f4'),('Feb', 'a10'),('Mar', 'a
10')])
            sales
```

```
     account  Jan  Feb  Mar
0  Jones LLC  150  200   50
1   Alpha Co  200  210   90
2   Blue Inc  140  215   95
```

Out[52]:

```
array([(0, 0., b'', b''), (0, 0., b'', b''), (0, 0., b'', b'')],
      dtype=[('account', '<i4'), ('Jan', '<f4'), ('Feb', 'S10'), ('Mar', 'S10')])
```

In [53]:

```
            import pandas as pd
            import numpy as np

            data = np.array(['g','e','e','k','g','e','e','k','s'])
            ser = pd.Series(data,index=['A','B','C','D','E','F','G','H','I'])
            print(ser)
            data = np.zeros((2,), dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10')])
            data
```

```
A    g
B    e
C    e
D    k
E    g
F    e
G    e
H    k
I    s
dtype: object
```

Out[53]:

```
array([(0, 0., b''), (0, 0., b'')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
```

In [54]:

```
            pd.DataFrame.from_records(data, index='C')
```

Out[54]:

C	A	B
b"	0	0.0
b"	0	0.0

A B

In [55]:

```
data = np.zeros((3,), dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10'),('D', 'f4')])
data
pd.DataFrame.from_records(data, index='D')
```

Out[55]:

D	A	B	C
0.0	0	0.0	b''
0.0	0	0.0	b''
0.0	0	0.0	b''

In [56]:

```
pd.DataFrame.from_records(data, index='A')
```

Out[56]:

A	B	C	D
0	0.0	b''	0.0
0	0.0	b''	0.0
0	0.0	b''	0.0

DataFrame.from_items

- DataFrame.from_items works analogously to the form of the dict constructor that takes a sequence of (key, value) pairs, where the keys are column (or row, in the case of orient='index') names, and the value are the column values (or row values).
- This can be useful for constructing a DataFrame with the columns in a particular order without having to pass an explicit list of columns

In [57]:

```
pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])])
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: FutureWarning: from_items is d
eprecated. Please use DataFrame.from_dict(dict(items), ...) instead.
DataFrame.from_dict(OrderedDict(items)) may be used to preserve the key order.
  """Entry point for launching an IPython kernel.
```

Out[57]:

	A	B
0	1	4
1	2	5
2	3	6

In [59]:

```
sales = [('account', ['Jones LLC', 'Alpha Co', 'Blue Inc']),
         ('Jan', [150, 200, 50]),
         ('Feb', [200, 210, 90]),
         ('Mar', [140, 215, 95]),
        ]
df = pd.DataFrame.from_items(sales)
df
```

```
C:\ProgramData\Anaconda3\lib\site-packages\ipykernel_launcher.py:6: FutureWarning: from_items is d
eprecated. Please use DataFrame.from_dict(dict(items), ...) instead.
DataFrame.from_dict(OrderedDict(items)) may be used to preserve the key order.
```

Out[59]:

	account	Jan	Feb	Mar
0	Jones LLC	150	200	140
1	Alpha Co	200	210	215
2	Blue Inc	50	90	95

11.6 Column selection, addition, deletion

- We can treat dataframes semantically like a dict of liek indexes and series object
- We can do following operations on dataframes' columns

```
Selection
addition
deletion
```

In [110]:

```
import pandas as pd
import numpy as np
#Column Selection
d = {
    'Country' : pd.Series(['Pakistan','Afghanistan','Iran','Saudia','Turkey'], index=[1
,2,3,4,5]),
    'Population' : pd.Series([220000000,300000000,230000000,1120000000], index=[1,2,3,4
])
}

df = pd.DataFrame(d)
df['Country']
```

Out[110]:

```
1       Pakistan
2    Afghanistan
3           Iran
4         Saudia
5         Turkey
Name: Country, dtype: object
```

In [111]:

```
df['Population']
```

Out[111]:

```
1    2.200000e+08
2    3.000000e+08
3    2.300000e+08
4    1.120000e+09
5             NaN
Name: Population, dtype: float64
```

In [112]:

```
import pandas as pd
#Adding new column
d = {
    'Country' : pd.Series(['Pakistan','Afghanistan','Iran','Saudia','Turkey'], index=[1
,2,3,4,5]),
    'Population' : pd.Series([220000000,300000000,230000000,1120000000], index=[1,2,3,4
```

```
])
}
df['Ranking']=pd.Series([1,2,3,4,5], index=[1,2,3,4,5])
df
```

Out[112]:

	Country	Population	Ranking
1	Pakistan	2.200000e+08	1
2	Afghanistan	3.000000e+08	2
3	Iran	2.300000e+08	3
4	Saudia	1.120000e+09	4
5	Turkey	NaN	5

In [113]:

```
import pandas as pd
#Adding new column which is result of multiplicaion of two columns
d = {
    'Country' : pd.Series(['Pakistan','Afghanistan','Iran','Saudia','Turkey'], index=[1
,2,3,4,5]),

    'Population' : pd.Series([220000000,300000000,230000000,1120000000], index=[1,2,3,4
])
}
df['Ranking']=pd.Series([1,3,4,2,5], index=[1,2,3,4,5])
df['New']=df['Ranking']*df['Population']
df
```

Out[113]:

	Country	Population	Ranking	New
1	Pakistan	2.200000e+08	1	2.200000e+08
2	Afghanistan	3.000000e+08	3	9.000000e+08
3	Iran	2.300000e+08	4	9.200000e+08
4	Saudia	1.120000e+09	2	2.240000e+09
5	Turkey	NaN	5	NaN

In [114]:

```
import pandas as pd
#Setting flag value to true, where values are satisfied
d = {
    'Country' : pd.Series(['Pakistan','Afghanistan','Iran','Saudia','Turkey'], index=[1
,2,3,4,5]),

    'Population' : pd.Series([220000000,300000000,230000000,1120000000], index=[1,2,3,4
])
}
df['Ranking']=pd.Series([1,3,4,2,5], index=[1,2,3,4,5])
df['flag'] = df['Ranking'] > 3
df
```

Out[114]:

	Country	Population	Ranking	New	flag
1	Pakistan	2.200000e+08	1	2.200000e+08	False
2	Afghanistan	3.000000e+08	3	9.000000e+08	False
3	Iran	2.300000e+08	4	9.200000e+08	True
4	Saudia	1.120000e+09	2	2.240000e+09	False
5	Turkey	NaN	5	NaN	True

In [115]:

```
#poping whole column
Population=df.pop('Population')
df
```

Out[115]:

	Country	Ranking	New	flag
1	Pakistan	1	2.200000e+08	False
2	Afghanistan	3	9.000000e+08	False
3	Iran	4	9.200000e+08	True
4	Saudia	2	2.240000e+09	False
5	Turkey	5	NaN	True

In [116]:

```
#Deleting column
del df['flag']
df
```

Out[116]:

	Country	Ranking	New
1	Pakistan	1	2.200000e+08
2	Afghanistan	3	9.000000e+08
3	Iran	4	9.200000e+08
4	Saudia	2	2.240000e+09
5	Turkey	5	NaN

In [117]:

```
# following example will take values from coloumn one until give range and will populate the new c
oloumn
df['Populate'] = df['Country'][:3]
df
```

Out[117]:

	Country	Ranking	New	Populate
1	Pakistan	1	2.200000e+08	Pakistan
2	Afghanistan	3	9.000000e+08	Afghanistan
3	Iran	4	9.200000e+08	Iran
4	Saudia	2	2.240000e+09	NaN
5	Turkey	5	NaN	NaN

In [118]:

```
# following example will take values from coloumn one until give range and will populate the new c
oloumn
df['Populate'] = df['Country'][3:5]
df
```

Out[118]:

	Country	Ranking	New	Populate
1	Pakistan	1	2.200000e+08	NaN
2	Afghanistan	3	9.000000e+08	NaN
3	Iran	4	9.200000e+08	NaN

	Country	Ranking	New	Populate
4	Saudia	2	2.240000e+09	Saudia
5	Turkey	5	NaN	Turkey

In [119]:

```
# inset column at particular position
df.insert(1, 'Population', df['New'])
df
```

Out[119]:

	Country	Population	Ranking	New	Populate
1	Pakistan	2.200000e+08	1	2.200000e+08	NaN
2	Afghanistan	9.000000e+08	3	9.000000e+08	NaN
3	Iran	9.200000e+08	4	9.200000e+08	NaN
4	Saudia	2.240000e+09	2	2.240000e+09	Saudia
5	Turkey	NaN	5	NaN	Turkey

In [120]:

```
# inset column at particular position
df.insert(3, 'Test', df['Ranking'])
df
```

Out[120]:

	Country	Population	Ranking	Test	New	Populate
1	Pakistan	2.200000e+08	1	1	2.200000e+08	NaN
2	Afghanistan	9.000000e+08	3	3	9.000000e+08	NaN
3	Iran	9.200000e+08	4	4	9.200000e+08	NaN
4	Saudia	2.240000e+09	2	2	2.240000e+09	Saudia
5	Turkey	NaN	5	5	NaN	Turkey

11.7. Row Selection, Addition, and Deletion

- We can also perform followong operations on rows

```
Selection
Addition
Deletion
```

In [121]:

```
#Rows can be selected by passing row label to a loc function.
df.loc[2]
```

Out[121]:

```
Country       Afghanistan
Population            9e+08
Ranking                   3
Test                      3
New                   9e+08
Populate                NaN
Name: 2, dtype: object
```

In [127]:

```
# Rows can be selected by passing integer location to an iloc function.
df = pd.DataFrame(d)
```

```
df.iloc[2]
```

Out[127]:

```
Country          Iran
Population    2.3e+08
Name: 3, dtype: object
```

In [128]:

```
# multiple rows can be selected using slicing operator :
df[2:4]
```

Out[128]:

	Country	Population
3	Iran	2.300000e+08
4	Saudia	1.120000e+09

In [129]:

```
df[:4]
```

Out[129]:

	Country	Population
1	Pakistan	2.200000e+08
2	Afghanistan	3.000000e+08
3	Iran	2.300000e+08
4	Saudia	1.120000e+09

In [134]:

```
# Appending rows
df = pd.DataFrame([[1, 2], [3, 4]], columns = ['a','b'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])

df = df.append(df2)
df
```

Out[134]:

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

In [150]:

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

df = df.append(df2)
df = df.drop(0)
df
```

Out[150]:

	a	b
1	3	4

	a	b
1	7	8

11.8 Indexing / Selection

- Row selection, for example, returns a Series whose index is the columns of the DataFrame:
- Basic indexing are as follow

In [155]:

```
import pandas as pd
            import numpy as np
            d1 = {
                'Country' : pd.Series(['Pakistan','Afghanistan','Iran','Saudia','Turkey'], index=[1
,2,3,4,5]),
                'Population' : pd.Series([220000000,300000000,230000000,1120000000], index=[1,2,3,4
])
            }

            df=pd.DataFrame(d1)
            df.loc[3]
```

Out[155]:

```
Country          Iran
Population    2.3e+08
Name: 3, dtype: object
```

In [162]:

```
            df.iloc[4]
```

Out[162]:

```
Country       Turkey
Population       NaN
Name: 5, dtype: object
```

In [159]:

```
            df[1:4]
```

Out[159]:

	Country	Population
2	Afghanistan	3.000000e+08
3	Iran	2.300000e+08
4	Saudia	1.120000e+09

In [160]:

```
            df[3:]
```

Out[160]:

	Country	Population
4	Saudia	1.120000e+09
5	Turkey	NaN

11.9 Data alignment and arithmetic

- Data alignment between DataFrame objects automatically align on both the columns and the index (row labels).
- Again, the resulting object will have the union of the column and row labels

In [12]:

```
import pandas as pd
import numpy as np
```

In [18]:

```
df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])
df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
df+df2
```

Out[18]:

	A	B	C	D
0	-3.137594	-1.330863	1.519771	NaN
1	2.502827	0.146704	-0.694219	NaN
2	2.839812	-1.286170	0.845201	NaN
3	1.030850	-1.672526	1.546588	NaN
4	3.487300	-2.090674	-2.466920	NaN
5	0.247054	-0.473206	-0.736531	NaN
6	0.527707	-0.635002	-0.921714	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN

In [20]:

```
df = pd.DataFrame([[1, 2,12], [3, 4,5]], columns = ['a','b','c'])
df2 = pd.DataFrame([[5, 6], [7, 8]], columns = ['a','b'])
df+df2
```

Out[20]:

	a	b	c
0	6	8	NaN
1	10	12	NaN

In [30]:

```
df = pd.DataFrame(np.random.randn(15, 4), columns=['A', 'B', 'C', 'D'])
df2 = pd.DataFrame(np.random.randn(12, 3), columns=['A', 'B', 'C'])
df+df2
```

Out[30]:

	A	B	C	D
0	1.132605	0.600413	0.966921	NaN
1	-1.358268	-0.824680	-0.756372	NaN
2	-0.007583	-3.300363	-0.567649	NaN
3	-2.665667	0.805339	-0.846012	NaN
4	1.156127	0.911009	0.810051	NaN
5	-1.920775	1.025162	-0.950633	NaN
6	-1.765272	0.289123	-2.044669	NaN
7	1.163973	0.082579	0.431350	NaN
8	1.516732	1.405450	1.236736	NaN

	A	B	C	D
9	-0.957546	-1.199114	2.318394	NaN
10	-0.648246	-0.877498	2.365698	NaN
11	-0.448516	0.716052	1.816975	NaN
12	NaN	NaN	NaN	NaN
13	NaN	NaN	NaN	NaN
14	NaN	NaN	NaN	NaN

In [41]:

```
df = pd.DataFrame(np.random.randn(5, 4), columns=['A', 'B', 'C', 'D'])
df2 = pd.DataFrame(np.random.randn(8, 3), columns=['A', 'B', 'C'])
df+df2
```

Out[41]:

	A	B	C	D
0	0.609726	-1.699829	-0.203226	NaN
1	0.466975	0.966766	0.587446	NaN
2	-0.267552	1.317713	2.916945	NaN
3	-1.118670	0.158354	2.448717	NaN
4	2.551752	-2.410683	2.261066	NaN
5	1.446704	-0.807515	0.576831	NaN
6	-1.252395	1.552488	-1.074472	NaN
7	1.872534	2.246399	3.518105	NaN
8	NaN	NaN	NaN	NaN

- when doing an operation between DataFrame and Series, the default behavior is to align the Series index on the DataFrame columns, thus broadcasting row-wise.

In [44]:

```
df - df.iloc[4]
```

Out[44]:

	A	B	C	D
0	-2.846796	0.604873	0.097995	2.456236
1	-3.072377	2.921180	0.993911	2.055733
2	-1.842660	2.313930	1.897424	1.051044
3	-1.891378	1.311931	1.016807	2.931201
4	0.000000	0.000000	0.000000	0.000000
5	-2.519660	0.469201	0.337175	2.159540
6	-5.430880	2.953710	0.447716	1.903906
7	-1.375381	3.389040	1.982614	0.727133
8	-3.492068	2.198879	0.589105	0.810441

In [46]:

```
df - df.loc[2]
```

Out[46]:

	A	B	C	D
0	-1.004135	-1.709057	-1.799429	1.405192

	A	B	C	D
1	1.229717	0.607250	0.903514	1.004689
2	0.000000	0.000000	0.000000	0.000000
3	-0.048718	-1.001999	-0.880618	1.880157
4	1.842660	-2.313930	-1.897424	-1.051044
5	-0.677000	-1.844728	-1.560249	1.108496
6	-3.588220	0.639780	-1.449708	0.852862
7	0.467279	1.075111	0.085190	-0.323910
8	-1.649408	-0.115051	-1.308320	-0.240602

In [47]:

```
df*31+3
```

Out[47]:

	A	B	C	D
0	-7.643170	-31.232636	8.961250	32.423670
1	-14.636193	40.572883	36.734617	20.008058
2	23.485029	21.748127	64.743545	-11.137292
3	21.974786	-9.313838	37.444399	47.147575
4	80.607495	-49.983690	5.923390	-43.719655
5	2.498044	-35.438452	16.375824	23.226099
6	-87.749786	41.581322	19.802593	15.301441
7	37.970690	55.076559	67.384432	-21.178518
8	-27.646626	18.181554	24.185638	-18.595969

In [48]:

```
df**3
```

Out[48]:

	A	B	C	D
0	-0.040470	-1.346592	0.007111	0.855077
1	-0.184132	1.780486	1.288672	0.165150
2	0.288552	0.221202	7.901137	-0.094845
3	0.229322	-0.062675	1.371737	2.888255
4	15.690107	-4.992770	0.000839	-3.423054
5	-0.000004	-1.906394	0.080330	0.277748
6	-25.087174	1.927730	0.159237	0.062486
7	1.435580	4.740692	8.958958	-0.474465
8	-0.966191	0.117453	0.319183	-0.338091

In [53]:

```
df*3/33
```

Out[53]:

	A	B	C	D
0	-0.031212	-0.100389	0.017482	0.086286
1	-0.051719	0.110184	0.098928	0.049877
2	0.060073	0.054980	0.181066	-0.041458
3	0.055645	-0.036111	0.101010	0.129465

	A	B	C	D
4	0.227588	0.155377	0.008573	0.137008
5	-0.001472	-0.112723	0.039225	0.059314
6	-0.266128	0.113142	0.049274	0.036075
7	0.102553	0.152717	0.188811	-0.070905
8	-0.089873	0.044521	0.062128	-0.063331

In [54]:

```
df-23
```

Out[54]:

	A	B	C	D
0	-23.343328	-24.104279	-22.807702	-22.050849
1	-23.568909	-21.787972	-21.911787	-22.451353
2	-22.339193	-22.395222	-21.008273	-23.456042
3	-22.387910	-23.397221	-21.888890	-21.575885
4	-20.496532	-24.709151	-22.905697	-24.507086
5	-23.016192	-24.239950	-22.568522	-22.347545
6	-25.927412	-21.755441	-22.457981	-22.603179
7	-21.871913	-21.320111	-20.923083	-23.779952
8	-23.988601	-22.510272	-22.316592	-23.696644

In [55]:

```
df%3
```

Out[55]:

	A	B	C	D
0	2.656672	1.895721	0.192298	0.949151
1	2.431091	1.212028	1.088213	0.548647
2	0.660807	0.604778	1.991727	2.543958
3	0.612090	2.602779	1.111110	1.424115
4	2.503468	1.290849	0.094303	1.492914
5	2.983808	1.760050	0.431478	0.652455
6	0.072588	1.244559	0.542019	0.396821
7	1.128087	1.679889	2.076917	2.220048
8	2.011399	0.489728	0.683408	2.303356

- In the special case of working with time series data, if the Series is a TimeSeries (which it will be automatically if the index contains datetime objects), and the DataFrame index also contains dates, the broadcasting will be column-wise

In [61]:

```
df = DataFrame(np.random.randn(8, 3), index=date_range('1/1/2000', periods=8), columns=list('ABC'))
df
```

Out[61]:

	A	B	C
2000-01-01	3.080892	0.424692	1.645232
2000-01-02	1.084161	-0.525959	-0.152501
2000-01-03	0.042750	2.079533	-0.728150

	A	B	C
2000-01-04	-0.245890	-0.397901	-0.536219
2000-01-05	1.161138	0.270663	1.121778
2000-01-06	0.057607	0.913704	-0.088229
2000-01-07	-0.844213	-0.717212	-0.717335
2000-01-08	0.337880	-2.030252	0.360102

In [69]:

```
df = DataFrame(np.random.randn(6, 3), index=date_range('12/12/2018', periods=6), columns=list('ABC'
))
df
```

Out[69]:

	A	B	C
2018-12-12	-0.909984	0.208160	0.931912
2018-12-13	0.113421	0.971928	-0.685130
2018-12-14	-1.113824	-0.736116	-0.876918
2018-12-15	-0.545516	0.990318	1.228341
2018-12-16	0.961060	-0.399281	0.672322
2018-12-17	-2.292085	0.801723	-1.124244

In [70]:

```
type(df['A'])
```

Out[70]:

```
pandas.core.series.Series
```

In [71]:

```
df - df['A']
```

Out[71]:

	2018-12-12 00:00:00	2018-12-13 00:00:00	2018-12-14 00:00:00	2018-12-15 00:00:00	2018-12-16 00:00:00	2018-12-17 00:00:00	A	B	C
2018-12-12	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-12-13	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-12-14	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-12-15	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-12-16	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2018-12-17	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

```
    df - df['A']
    is now deprecated and will be removed in a future release. The preferred way to
replicate this behavior is
```

In [72]:

```
df.sub(df['A'], axis=0)
```

Out[72]:

Out[72]:

	A	B	C
2018-12-12	0.0	1.118144	1.841895
2018-12-13	0.0	0.858507	-0.798551
2018-12-14	0.0	0.377708	0.236907
2018-12-15	0.0	1.535834	1.773857
2018-12-16	0.0	-1.360342	-0.288739
2018-12-17	0.0	3.093808	1.167841

In [80]:

```
# Boolean expression
df1 = DataFrame({'a' : [1,0,1, 0, 1], 'b' : [0,1,1,0, 1] }, dtype=bool)
df1
```

Out[80]:

	a	b
0	True	False
1	False	True
2	True	True
3	False	False
4	True	True

In [82]:

```
df1 = DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=bool)
df1
```

Out[82]:

	a	b
0	True	False
1	False	True
2	True	True

In [86]:

```
df2 = DataFrame({'a' : [1, 1], 'b' : [0, 1],'c' : [0, 1] }, dtype=bool)
df2
```

Out[86]:

	a	b	c
0	True	False	False
1	True	True	True

In [91]:

```
df1 = pd.DataFrame({'a' : [1, 0,1,1], 'b' : [1,0, 1, 1] }, dtype=bool)
df2 = pd.DataFrame({'a' : [0, 1,0,1], 'b' : [1,1, 1, 0] }, dtype=bool)
#AND operation
df1&df2
```

Out[91]:

	a	b

	a	b
0	False	True
1	False	False
2	False	True
3	True	False

In [92]:

```
#OR operation
df1|df2
```

Out[92]:

	a	b
0	True	True
1	True	True
2	True	True
3	True	True

In [93]:

```
df1^df2
```

Out[93]:

	a	b
0	True	False
1	True	True
2	True	False
3	False	True

In [94]:

```
-df2
```

Out[94]:

	a	b
0	True	False
1	False	False
2	True	False
3	False	True

In [95]:

```
-df1
```

Out[95]:

	a	b
0	False	False
1	True	True
2	False	False
3	False	False

In [97]:

```
df1 = DataFrame({'a' : [1, 1], 'b' : [0, 1],'c' : [0, 1] }, dtype=bool)
df2 = DataFrame({'a' : [1, 0], 'b' : [1, 1],'c' : [1, 1] }, dtype=bool)
df1&df2
```

Out[97]:

	a	b	c
0	True	False	False
1	False	True	True

In [98]:

```
df1|df2
```

Out[98]:

	a	b	c
0	True	True	True
1	True	True	True

In [99]:

```
df2^df1
```

Out[99]:

	a	b	c
0	False	True	True
1	True	False	False

In [101]:

```
-df1
```

Out[101]:

	a	b	c
0	False	True	True
1	False	False	False

In [102]:

```
-df2
```

Out[102]:

	a	b	c
0	False	False	False
1	True	False	False

11.10 Transposing

- To transpose, access the T attribute (also the transpose function), similar to an ndarray
- in transpose we shal make rows as columns and columns as rows

In [110]:

```
df = pd.DataFrame(np.random.randn(15, 4), columns=['A', 'B', 'C', 'D'])
df[:5].T
```

Out[110]:

	0	1	2	3	4
A	0.422386	-1.675481	1.943668	0.205944	0.307267
B	1.852522	0.685087	0.456870	0.678399	-1.064333
C	0.162191	1.010860	-0.189490	-0.598375	0.903074
D	1.843371	-0.116040	-0.341397	0.528163	-0.204522

In [111]:

```
df[:4].T
```

Out[111]:

	0	1	2	3
A	0.422386	-1.675481	1.943668	0.205944
B	1.852522	0.685087	0.456870	0.678399
C	0.162191	1.010860	-0.189490	-0.598375
D	1.843371	-0.116040	-0.341397	0.528163

In [113]:

```
df[1:].T
```

Out[113]:

	1	2	3	4	5	6	7	8	9	10	11	12	13
A	-1.675481	1.943668	0.205944	0.307267	-2.184595	0.835598	-0.020251	0.108280	0.788030	-0.670179	-0.278994	1.543829	1.041719
B	0.685087	0.456870	0.678399	-1.064333	-0.756184	-0.004585	-0.114426	1.159189	-0.675931	0.852458	2.182998	-0.112381	1.225684
C	1.010860	-0.189490	-0.598375	0.903074	1.053421	-0.799341	1.463328	0.179332	0.170553	0.553672	-1.107186	0.625730	-0.566902
D	-0.116040	-0.341397	0.528163	-0.204522	-0.392968	-2.815089	0.299807	0.195018	-0.745756	-1.238504	0.026985	-0.161625	1.221987

In [122]:

```
dates = pd.date_range('20130101', periods=8)
dates
```

Out[122]:

```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06', '2013-01-07', '2013-01-08'],
              dtype='datetime64[ns]', freq='D')
```

In [123]:

```
df = pd.DataFrame(['a','s','f','a','d','r','v','y'], index=dates)
df
```

Out[123]:

	0
2013-01-01	a
2013-01-02	s

2013-01-03	f
2013-01-04	a
2013-01-05	d
2013-01-06	r
2013-01-07	v
2013-01-08	y

In [133]:

```
df = pd.DataFrame(np.random.randn(15, 4), columns=['A', 'B', 'C', 'D'])
```

- Elementwise NumPy ufuncs (log, exp, sqrt, ...) and various other NumPy functions can be used with no issues on DataFrame, assuming the data within are numeric:

In [137]:

```
np.exp(df)
```

Out[137]:

	A	B	C	D
0	2.877400	1.384591	1.633900	1.889546
1	0.666046	0.221397	0.817424	0.959038
2	0.655000	0.420773	0.318978	0.120194
3	0.219338	2.039273	0.934335	0.679687
4	0.224148	0.850646	0.593578	0.718229
5	0.690870	2.589355	1.250505	2.952235
6	0.640659	1.055815	0.486270	0.539043
7	0.677871	0.270040	0.458701	1.623157
8	0.179484	0.909542	0.834109	0.273583
9	0.677407	1.156914	0.167518	1.757071
10	1.101916	0.365337	1.747599	0.906319
11	1.630436	2.243547	0.484914	1.573548
12	1.229115	2.671397	0.871699	1.210844
13	1.044586	0.322221	1.670361	0.314857
14	0.483725	0.850738	0.235603	1.977192

In [142]:

```
# the dot method on DataFrame implements matrix multiplication
df.T.dot(df)
```

Out[142]:

	A	B	C	D
A	10.407808	1.290831	4.260428	4.066100
B	1.290831	10.256110	0.810230	4.301796
C	4.260428	0.810230	9.473570	0.491027
D	4.066100	4.301796	0.491027	10.992070

In [141]:

```
np.asarray(df)
```

Out[141]:

```
array([[ 1.0568871 ,  0.32540452,  0.49097003,  0.63633677],
       [-0.40639707, -1.50779963, -0.20159695, -0.04182406],
       [-0.4231193 , -0.8656619 , -1.14263239, -2.11864492],
       [-1.51714183,  0.71259341, -0.06792042, -0.38612244],
       [-1.4954468 , -0.16175914, -0.52158688, -0.33096626],
       [-0.369803  ,  0.95140869,  0.22354767,  1.08256266],
       [-0.44525737,  0.05431301, -0.72099086, -0.61795957],
       [-0.38879776, -1.3091851 , -0.77935765,  0.48437314],
       [-1.71766916, -0.09481371, -0.18139089, -1.29615139],
       [-0.38948353,  0.1457559 , -1.78666184,  0.56364834],
       [ 0.0970508 , -1.00693544,  0.55824305, -0.09836372],
       [ 0.48884761,  0.80805808, -0.7237841 ,  0.45333303],
       [ 0.20629424,  0.98260141, -0.13731127,  0.19131789],
       [ 0.04362095, -1.1325168 ,  0.51303991, -1.1556366 ],
       [-0.72623782, -0.16165154, -1.44560599,  0.68167768]])
```

In [147]:

```
# Similarly, the dot method on Series implements dot product:
s1 = Series(np.arange(5,10))
s1.dot(s1)
```

Out[147]:

255

In [146]:

```
s1 = Series(np.arange(12,120))
s1.dot(s1)
```

Out[146]:

568314

In [154]:

```
df = pd.DataFrame([123,123,4,123,12,53,25,2], index=['a','b','c','d','e','f','g','h'])
np.log(df)
```

Out[154]:

	0
a	4.812184
b	4.812184
c	1.386294
d	4.812184
e	2.484907
f	3.970292
g	3.218876
h	0.693147

In [151]:

```
df = pd.DataFrame([123,12,1,412], index=[1,2,3,4])
np.log(df)
```

Out[151]:

	0
1	4.812184
2	2.484907
3	0.000000

4 6.021023

- Creating a DataFrame by passing a dict of objects that can be converted to series-like.

In [156]:

```
df2 = pd.DataFrame({ 'A' : 1.,
                     'B' : pd.Timestamp('20130102'),
                     'C' : pd.Series(1,index=list(range(4)),dtype='float32'),
                     'D' : np.array([3] * 4,dtype='int32'),
                     'E' : pd.Categorical(["test","train","test","train"]),
                     'F' : 'foo' })
df2
```

Out[156]:

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

In [166]:

```
df2.dtypes
```

Out[166]:

```
A            object
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

In [165]:

```
df2 = pd.DataFrame({ 'A' : 'ASD',
                     'B' : pd.Timestamp('20191214'),
                     'C' : pd.Series(3,index=list(range(8)),dtype='float32'),
                     'D' : np.array([3] * 8,dtype='int32'),
                     'E' :
pd.Categorical(["Ali","Adeel","Hamza","Amir","Ashiq","Akmal","Asghar","Ahmar"]),
                     'F' : 'qwerrty' })
df2
```

Out[165]:

	A	B	C	D	E	F
0	ASD	2019-12-14	3.0	3	Ali	qwerrty
1	ASD	2019-12-14	3.0	3	Adeel	qwerrty
2	ASD	2019-12-14	3.0	3	Hamza	qwerrty
3	ASD	2019-12-14	3.0	3	Amir	qwerrty
4	ASD	2019-12-14	3.0	3	Ashiq	qwerrty
5	ASD	2019-12-14	3.0	3	Akmal	qwerrty
6	ASD	2019-12-14	3.0	3	Asghar	qwerrty
7	ASD	2019-12-14	3.0	3	Ahmar	qwerrty

In [167]:

```
df2.dtypes
```

Out[167]:

```
A            object
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

12. Data Viewing

- We can view data / display data in different ways
- See the top & bottom rows of the frame
- Selecting a single column
- Selecting via [], which slices the rows For getting a cross section using a label
- Selecting on a multi-axis by labe
- Showing label slicing, both endpoints are included
- Reduction in the dimensions of the returned object
- For getting a scalar value For getting fast access to a scalar
- Select via the position of the passed integers
- By integer slices, acting similar to numpy/python
- By lists of integer position locations, similar to the numpy/python style
- For slicing rows explicitly For slicing columns explicitly
- For getting a value explicitly For getting fast access to a scalar
- Using a single column's values to select data
- Selecting values from a DataFrame where a boolean condition is met
- Using the isin() method for filtering

In [169]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
data = {
    'Name':['Ali', 'Adnan', 'Akmal', 'Adeel','Asghar','Alam'],
    'Age':[28,34,29,42,23,42]
}
df = pd.DataFrame(data, index=['one','two','three','four','five','six'])
pd.DataFrame(df)
df.head(2)
```

Out[169]:

	Name	Age
one	Ali	28
two	Adnan	34

In [170]:

```
df.tail(2)
```

Out[170]:

	Name	Age
five	Asghar	23
six	Alam	42

In [171]:

```
df.index
```

Out[171]:

```
Index(['one', 'two', 'three', 'four', 'five', 'six'], dtype='object')
```

In [172]:

```
df.columns
```

Out[172]:

```
Index(['Name', 'Age'], dtype='object')
```

In [173]:

```
df.values
```

Out[173]:

```
array([['Ali', 28],
       ['Adnan', 34],
       ['Akmal', 29],
       ['Adeel', 42],
       ['Asghar', 23],
       ['Alam', 42]], dtype=object)
```

In [174]:

```
df.T
```

Out[174]:

	one	two	three	four	five	six
Name	Ali	Adnan	Akmal	Adeel	Asghar	Alam
Age	28	34	29	42	23	42

In [175]:

```
df.sort_index(axis=0, ascending=False)
```

Out[175]:

	Name	Age
two	Adnan	34
three	Akmal	29
six	Alam	42
one	Ali	28
four	Adeel	42
five	Asghar	23

In [176]:

```
df.sort_values(by='Age')
```

Out[176]:

	Name	Age
five	Asghar	23
one	Ali	28
three	Akmal	29
two	Adnan	34

	Name	Age
four	Adeel	42
six	Alam	42

In [177]:

```
df.sort_values(by='Name')
```

Out[177]:

	Name	Age
four	Adeel	42
two	Adnan	34
three	Akmal	29
six	Alam	42
one	Ali	28
five	Asghar	23

In [178]:

```
# Describe shows a quick statistic summary of your data
df.describe()
```

Out[178]:

	Age
count	6.000000
mean	33.000000
std	7.797435
min	23.000000
25%	28.250000
50%	31.500000
75%	40.000000
max	42.000000

In [179]:

```
#Selecting a single column, which yields a Series, equivalent to df.Age
df['Age']
```

Out[179]:

```
one      28
two      34
three    29
four     42
five     23
six      42
Name: Age, dtype: int64
```

In [180]:

```
#Selecting via [], which slices the rows. df[0:3]
df[2:4]
```

Out[180]:

	Name	Age
three	Akmal	29
four	Adeel	42

In [181]:

```
df['two':'five']
```

Out[181]:

	Name	Age
two	Adnan	34
three	Akmal	29
four	Adeel	42
five	Asghar	23

In [182]:

```
#Selecting on a multi-axis by label
df.loc[:,['Name','Age']]
```

Out[182]:

	Name	Age
one	Ali	28
two	Adnan	34
three	Akmal	29
four	Adeel	42
five	Asghar	23
six	Alam	42

In [183]:

```
#Selecting on a multi-axis by label
df.loc[:,['Name']]
```

Out[183]:

	Name
one	Ali
two	Adnan
three	Akmal
four	Adeel
five	Asghar
six	Alam

In [184]:

```
#Showing label slicing, both endpoints are included
df.loc['three':'five',['Age']]
```

Out[184]:

	Age
three	29
four	42
five	23

In [185]:

```
# Reduction in the dimensions of the returned object
df.loc['two',['Name','Age']]
```

Out[185]:

```
Name     Adnan
Age         34
Name: two, dtype: object
```

In [186]:

```
# Select via the position of the passed integers
df.iloc[4]
```

Out[186]:

```
Name     Asghar
Age          23
Name: five, dtype: object
```

In [187]:

```
# By integer slices, acting similar to numpy/python
df.iloc[2:4,0:2]
```

Out[187]:

	Name	Age
three	Akmal	29
four	Adeel	42

In [188]:

```
# For slicing rows explicitly
df.iloc[:,0:1]
```

Out[188]:

	Name
one	Ali
two	Adnan
three	Akmal
four	Adeel
five	Asghar
six	Alam

In [189]:

```
# For slicing rows explicitly
df.iloc[:,1:]
```

Out[189]:

	Age
one	28
two	34
three	29
four	42

	Age
five	23
six	42

In [190]:

```
# For getting a value explicitly
df.iloc[4,1]
```

Out[190]:

```
23
```

In [191]:

```
# For getting a value explicitly
df.iloc[4,0]
```

Out[191]:

```
'Asghar'
```

In [192]:

```
# Using a single column’s values to select data.
df[df.Age > 33]
```

Out[192]:

	Name	Age
two	Adnan	34
four	Adeel	42
six	Alam	42

In [193]:

```
# Selecting values from a DataFrame where a boolean condition is met.
df[df > 1]
```

Out[193]:

	Name	Age
one	Ali	28
two	Adnan	34
three	Akmal	29
four	Adeel	42
five	Asghar	23
six	Alam	42

In [194]:

```
# Using the isin() method for filtering:
df2 = df.copy()
df2['Age'] = [22,33,44,55,66,77]
df
```

Out[194]:

	Name	Age
one	Ali	28
two	Adnan	34
three	Akmal	29

	Name	Age
three	Akmal	29
four	Adeel	42
five	Asghar	23
six	Alam	42

In [195]:

```
df2
```

Out[195]:

	Name	Age
one	Ali	22
two	Adnan	33
three	Akmal	44
four	Adeel	55
five	Asghar	66
six	Alam	77

In [196]:

```
df2[df2['Age'].isin([22,44])]
```

Out[196]:

	Name	Age
one	Ali	22
three	Akmal	44

In [198]:

```
df2[df2['Name'].isin(['Akmal','Adeel','Asghar'])]
```

Out[198]:

	Name	Age
three	Akmal	44
four	Adeel	55
five	Asghar	66

----------THE END----------