

## FrameWork:

*Jupyter Notebook launched from Anaconda*

## URL:

*Please download anaconda from following link and launch Jupyter Notebook from there*

<https://www.anaconda.com/download/> (<https://www.anaconda.com/download/>).

## Helping Tutorial to install:

<https://www.youtube.com/watch?v=T8wK5loXkXg> (<https://www.youtube.com/watch?v=T8wK5loXkXg>).

## Table of Contents

## **1. Introduction**

## **2. Data Types**

2.1 Variables

2.2 Numbers

2.3 Strings

2.4 Lists

2.5 Dictionaries

2.6 Tuples

2.7 Sets

## **3. Comparison Operators**

## **4. If-Else Statements**

## **5. For & While Loop**

## **6. Functions**

## **7. Lambda Functions**

7.1 Map()

7.2 filter()

## **8. File I/O**

## **9. Pandas Library Introduction**

## **10. Series**

- 10.1 From ndarray
- 10.2 From dict
- 10.3 From Scalar value
- 10.4 Series is ndarray-like
- 10.5 Series is dict-like
- 10.6 Vectorized operations and label alignment with Series
- 10.7 Name Attribute

## 11. Data Frames

- 11.1 From dict of Series or dicts
- 11.2 From dict of ndarrays / lists
- 11.3 From a list of dicts
- 11.4 From a dict of tuples
- 11.5 Alternate Constructors

# 1. Introduction

- 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.

## 2. Data Types

Following data types are used in Python to store and access the data. Those are explained below.

## 2.1 Variables

- In Python, variables don't need explicit declaration to occupy memory space.
- The type of the variable will be decided after assigning the value to it.

Some examples are given as follows:

```
In [19]: counter = 100.00 #float type variable
        miles = 100 #integer type variable
        name = "Mehvish" #string type variable

        #print all values
        print (counter)
        print (miles)
        print (name)

100.0
100
Mehvish
```

## 2.2 Numbers

- 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).

Some examples are given below

```
In [20]: 2 #integer
```

```
Out[20]: 2
```

```
In [21]: 2+3 #integer
```

```
Out[21]: 5
```

```
In [22]: 2.3+5.5 #float
```

```
Out[22]: 7.8
```

```
In [23]: 2**2 #exponent
```

```
Out[23]: 4
```

## 2.3 Strings

- String data types are used to store words or combination of words having letters, numbers, special characters etc.
- It can be stored by enclosing within single quotes and double quotes also.
- Python doesn't support **char** data type. It will be as String of length one in Python.

```
In [24]: name = 'Mehvish'
         university = "COMSATS"

         print(name,university)
```

```
Mehvish COMSATS
```

## Access values in String

- Python doesn't support **Character** data type. These are treated as Strings of length one.
- Square brackets are used to access substrings.
- We can access a specific character or range of characters from string.

It would be cleared by using following examples.

```
In [25]: stringVariable = 'Hello World'

print("stringVariable[0]:", stringVariable[0])
print("stringVariable[1:5]:", stringVariable[1:5])

stringVariable[0]: H
stringVariable[1:5]: ello
```

## Updating String

- Value of the string can be updated by assigning a new value to it. New value will be replaced with older.
- New value must be related (having same data type) to previous value of the string.
- '+' operator will be used for this purpose like as follows:

```
In [26]: stringVariable = "Hello World"
stringVariable = stringVariable[:6] + "Python"

print("Updated string: ", stringVariable)

Updated string:  Hello Python
```

## Delete String

- To delete the value of the string, just delete its object (reference variable).

See following example:

```
In [27]: stringVariable = "Hello World"
         print (stringVariable)
         del stringVariable
         print (stringVariable)
```

Hello World

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-27-817d7dd3c7eb> in <module>()
      2 print (stringVariable)
      3 del stringVariable
----> 4 print (stringVariable)

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

## String special operators

Some special operators are used in Python to assist the user. These are given below.

- '+' operator is used for concatenation.

Example:

```
In [28]: variable = "Hello"
         print(variable+"Python")
```

HelloPython

- '\*' used of Repetition

```
In [29]: variable = "Hello"
         print(variable*3)
```

HelloHelloHello

- `[]` give the character of string at given index.

```
In [30]: variable = "Hello"  
print(variable[1])
```

e

- `[:]` is used to get a range of characters from string.

```
In [31]: variable = "Hello"  
print(variable[1:3])
```

el

- `in` returns true if given character exists in string, otherwise false.

```
In [32]: variable = "Hello"  
print('H' in variable)
```

True

- `in` returns true if given character doesn't exists in string, otherwise false.

```
In [33]: variable = "Hello"  
print('G' not in variable)
```

True

## String formatting operator



- One of the coolest feature is string formatting operator within print statement.

See following example:

```
In [34]: print ("My name is %s and age is %d kg!" % ("Zara",21))  
My name is Zara and age is 21 kg!
```

## 2.4 Lists

- It can be written as a list of comma-seperated values within square brackets.
- Multiple types of data can be stored in a List.
- Individual elements of a list can be changed (it can be read and write).
- List indices starts from 0 just like arrays.

```
In [35]: list1 = ['Mehvish','1880']  
print(list1)  
['Mehvish', '1880']
```

## Accessing values in Lists

- Square brackets are used to access the values of a list.
- We can access a specific values or range of values from list.

```
In [36]: list1 = ['physics', 'chemistry', 1997, 2000];  
list2 = [1, 2, 3, 4, 5, 6, 7 ];  
  
print ("list1[0]: ", list1[0])  
print ("list2[1:5]: ", list2[1:5])  
  
list1[0]: physics  
list2[1:5]: [2, 3, 4, 5]
```

## Updating Lists

- We can update single or multiple elements of a list by giving slice on left hand of the assignment operator.
- It can also be updated using `append()` function.

```
In [37]: list1 = ['physics', 'chemistry', 1997, 2000];  
list1[1] = "computer science"  
print(list1)  
  
list1.append('Computer Science')  
print(list1)  
  
['physics', 'computer science', 1997, 2000]  
['physics', 'computer science', 1997, 2000, 'Computer Science']
```

## Delete List element

- **del** statement is used to remove when you know the position or index of element to be deleted.
- **remove()** function can be used when you don't know the position of element to be removed.

```
In [38]: list1 = ['physics', 'chemistry', 1997, 2000];  
print (list1)  
  
del list1[2];  
  
print ("After deleting value at index 2 : ")  
print (list1)  
  
['physics', 'chemistry', 1997, 2000]  
After deleting value at index 2 :  
['physics', 'chemistry', 2000]
```

```
In [39]: list1 = ['physics', 'chemistry', 1997, 2000];  
print (list1)  
  
list1.remove('chemistry')  
  
print ("After deleting value at index 2 : ")  
print (list1)  
  
['physics', 'chemistry', 1997, 2000]  
After deleting value at index 2 :  
['physics', 1997, 2000]
```

## 2.5 Dictionaries

- It is same as array of objects having key and values in PHP.
- Each key is separated by colon (:) from its value.
- Each item is separated with comma (,).
- Empty dictionary can be written as {}.
- Keys are unique in dictionary but values may not.
- The values of the dictionary can be of any data type.
- The keys must be of an immutable data type such as strings, numbers, or tuples.

## Accessing value in Dictionary

- We can use the familiar square brackets along with the key to obtain its value. Following is a simple example

```
In [40]: dict = {'Name':"Mehvish", 'Depart': "BCS", 'Batch': 2014}
print (dict['Name'])
```

Mehvish

## Updating Dictionary

- We can update already existing value in dictionary.
- We can also add a new entry in dictionary.

See example

```
In [41]: dict = {'Name':"Mehvish", 'Depart': "BCS", 'Batch': 2014}
print (dict)
dict['Batch'] = "SP14"
print (dict)
```

```
dict['University'] = "COMSATS"
print(dict)
```

```
{'Name': 'Mehvish', 'Depart': 'BCS', 'Batch': 2014}
{'Name': 'Mehvish', 'Depart': 'BCS', 'Batch': 'SP14'}
{'Name': 'Mehvish', 'Depart': 'BCS', 'Batch': 'SP14', 'University': 'COMSATS'}
```

## Delete Dictionary Element

- We can delete individual element of dictionary and complete content of dictionary.
- **del** is used for individual element removal and **clear()** function is used to remove entire dictionary.

```
In [42]: dict = {'Name':"Mehvish", 'Depart': "BCS", 'Batch': 2014}  
print (dict)
```

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

```
{'Name': 'Mehvish', 'Depart': 'BCS', 'Batch': 2014}  
{'Name': 'Mehvish', 'Depart': 'BCS'}
```

```
In [43]: dict = {'Name':"Mehvish", 'Depart': "BCS", 'Batch': 2014}  
dict.clear()  
print (dict)
```

```
{}
```

## 2.6 Tuples

- It is same as list. The differences between tuples and lists are, the tuples cannot be changed.
- Each item is comma (,) separated.
- Empty Tuple is shown as ().
- To write a tuple containing a single value you have to include a comma (,) even though there is only one value. For Exampe: tup = (40,).
- It can also have multiple data type values.
- Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

## Accessing values in Tuples

- We can use the familiar square brackets along with the index to obtain its value. Following is a simple example

```
In [44]: tup = ('Math','98', 'C programming', '99')
print (tup)

print(tup[1])
print(tup[1:3])

('Math', '98', 'C programming', '99')
98
('98', 'C programming')
```

## Updating Tuples

- Tuples are immutable means we can't change it. It is read only.
- We are able to take portions of tuples to make a new tuple.

```
In [45]: tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print (tup3);

(12, 34.56, 'abc', 'xyz')
```

## Delete Tuple elements

- Removing individual element in Tuple is not possible because they can't be updated.
- **del** statment is used to remove entire Tuple.

```
In [46]: tup = ('physics', 'chemistry', 1997, 2000);
print (tup);
del tup;
print ("After deleting tup : ");
print (tup);
```

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
```

```
-----
NameError                                Traceback (most recent call last)
<ipython-input-46-750b57655605> in <module>()
      3 del tup;
      4 print ("After deleting tup : ");
----> 5 print (tup);

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

## 2.7 Sets

- A set is collection of unordered items.
- Every element is unique (No duplicates).
- Every element is immutable (can't be changed).
- However, the set itself is mutable. We can add or remove items from it.
- Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.
- Empty set will be written as {}.
- Each item in set will be comma (,) seperated.
- We can make a set from a list using **set()** function.
- Data type can be found using **type()** function.
- **add()** is used to add single value, **update()** is used for adding multiple values.
- **update()** function can take tupe, strings, list or other set as argument. In all cases, duplicates will be avoided.
- **discard()** and **remove()** are used to to delete particular item from set.
- **discard()** will not raise an error if item doesn't exists in set.
- **remove()** will raise an error if item doesn't exists in set.

```
In [47]: #list
list1 = [1,2,3,4,5]

print (type(list1))

my_set = set(list1)

print(my_set)

print (type(my_set))

# set of integers
my_set = {1,2,3}
print (my_set)

# set of mixed data types
my_set = {1,"Hello", 1.2,'C'}

# adding a single value
my_set.add('D')

#adding multiple values
my_set.update(list1)
print (my_set)

my_set.discard('G') # it will not raise an error
my_set.remove('G') # it will raise an error
```



```
<class 'list'>
{1, 2, 3, 4, 5}
<class 'set'>
{1, 2, 3}
{1, 1.2, 2, 3, 4, 5, 'D', 'C', 'Hello'}

-----
KeyError                                Traceback (most recent call last)
<ipython-input-47-0f2c0ed91f31> in <module>()
    25
    26 my_set.discard('G') # it will not raise an error
--> 27 my_set.remove('G') # it will raise an error
    28

KeyError: 'G'
```

### 3. Comparison Operators

- These are used to compare values (string or numbers) and return true/false according to situation.

```
In [48]: 1<3
```

```
Out[48]: True
```

```
In [49]: 15 <= 23
```

```
Out[49]: True
```

```
In [50]: 'Mehvish' == "Zeenat"
```

```
Out[50]: False
```

```
In [51]: "Mehvish" != "Mehvish"
```

```
Out[51]: False
```

```
In [52]: (1==1) or (5 > 2)
```

```
Out[52]: True
```

```
In [53]: (1 < 2) and (2 < 1)
```

```
Out[53]: False
```

## 4. If-Else Statements

- If-Else statements are used to execute a block of code depending on conditions. **If** block if condition is true otherwise **else** block. See following example:

```
In [54]: if 25 % 2:  
        print('Even')  
        else:  
        print('Odd')
```

```
Even
```

## 5. For and While Loop

- Python has **for** and **while** loop for iteration, used when we want to perform a specific a task repeatedly.

```
In [55]: #example of for loop  
fact = 1  
N = 5  
for i in range (1,N+1):  
    fact*=i  
  
print (fact)
```

```
120
```

```
In [56]: #example of while loop  
a = 0  
while a < 10:  
    a = a+1  
    print(a)
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

## Functions

- It is a block of organized and reusable code.
- It is used to perform a single, related action.
- It provides high modularity for your application.
- It has a high degree of code reusing.
- The syntax is:

```
def functionname( parameters ):
```

```
    "function_docstring"
```

```
    function_suite
```

```
    return [expression]
```

```
In [57]: # Function definition is here
def printme( str ):
    #This prints a passed string into this function
    print (str)
    return;
# Now you can call printme function
printme("I'm first call to user defined function!")
```

I'm first call to user defined function!

## 7. Lambda Functions

- The creation of anonymous functions at runtime, using a construct called "lambda".
- Lambda function doesn't include **return** statement, it always contains an expression which is returned.
- This piece of code shows the difference between a normal function definition ("f") and a lambda function ("g"):

```
In [58]: #Normal function
def f (x):
    return x**2
print (f(8))
```

64

```
In [59]: #Lambda Function
#Lambda expressions
times3 = lambda var:var*3
times3(10)
#Lambda expressions: another way to write a function in line
```

Out[59]: 30

### 7.1 Map()

- **Map()** function is used with two arguments. Just like: `r = map(func, seq)`
- The first argument `func` is the name of a function and the second a sequence (e.g. a list).
- `seq.map()` applies the function `func` to all the elements of the sequence `seq`. It returns a new list with the elements changed by `func`.

```
In [60]: sentence = 'It is raining cats and dogs'
         words = sentence.split()
         print (words)

         lengths = map(lambda word: len(word), words)
         list(lengths)

['It', 'is', 'raining', 'cats', 'and', 'dogs']

Out[60]: [2, 2, 7, 4, 3, 4]
```

## 7.2 Filter()

- The function **filter(function, list)** offers an elegant way to filter out all the elements of a list.
- 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 list `l`.
- Only if `f` returns `True` will the element of the list be included in the result list.

```
In [61]: fib = [0,1,1,2,3,5,8,13,21,34,55]
         result1 = filter(lambda x: x % 2, fib)
         list(result1)
```

```
Out[61]: [1, 1, 3, 5, 13, 21, 55]
```

```
In [62]: fib = [0,1,1,2,3,5,8,13,21,34,55]
         result2 = filter(lambda x: x % 2 == 0, fib)
         list (result2)
```

```
Out[62]: [0, 2, 8, 34]
```

## 8. File I/O

- In this section, we'll cover all basic I/O function(methods).

### Reading input from Keyboard

- For reading input from keyboard, ***raw\_input()*** method is used.
- It reads only one line from standard input and returns it as a string.

```
In [65]: from six.moves import input
         string = input("Enter your name: ");
         print(string)
```

```
Enter your name: Mehvish
Mehvish
```

### I/O from or to Text File

- In this scenario, we'll read and write to a text file.
  - **r** opens a file in read only mode.
  - **r+** opens a file read and write mode.
  - **w** opens a file in write mode only.
  - **a** opens a file in append mode
  - **a+** opens a file in append and read mode.

```
In [1]: # Open a file to read
fileOpen = open("file.txt", "r+")
str = fileOpen.read(); #to read specific content from start you can use read(12). It will read 12 characters
from the start of file
print (str)
# Close opened file
fileOpen.close()
```

Name: Mehvish Ashiq

Department: BSCS

```
In [2]: # Open a file to append
fileOpen = open("file.txt", "a+")
fileOpen.write(" Information Technology Lahore");
fileOpen.close()

# Open a file to read
fileOpen = open("file.txt", "r+")
string = fileOpen.read(); #to read specific content from start you can use read(12). It will read 12 characters
from the start of file
print (string)
# Close opened file
fileOpen.close()
```

Name: Mehvish Ashiq

Department: BSCS Information Technology Lahore

## File Position

- **tell()** method tells the current position within the file.
- **seek()** method changes the current file location.

```
In [3]: # Open a file
fo = open("file.txt", "r+")
str = fo.read(10);
print ("Read String is : \n", str)

# Check current position
position = fo.tell();
print ("Current file position : \n", position)

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10);
print ("Again read String is : \n", str)
# Close opened file
fo.close()
```

```
Read String is :
Name: Mehv
Current file position :
10
Again read String is :
Name: Mehv
```

```
In [4]: import os
# rename a file
os.rename("file.txt", "newfile.txt")
```

```
In [5]: #remove file
os.remove("newfile.txt")
```

## 9. Pandas Introduction



- Pandas is an open source library built on top of NumPy
- It allows for fast analysis and data cleaning and preparation
- It excels in performance and productivity
- It also has built-in visualization features
- It can work with data from a wide variety of sources

## 10. Series

- A series is very similar to NumPy array.
- Series is 1-D array labeled array capable of holding any type of data.
- 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.
- Following function is used to create a series:

```
s= pd.Series(data,index = index)
```

- In above function, **data** can be many different things:
  - A python dict
  - An ndarray
  - A scalar value (For example: 5)
- The passed **index** is a list of axis labels. So, this separates into a few cases depending on what data is:

### 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 [72]: import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

"""following a fucntion is called from panas to create a series.
data would be 5 random values and indexes are assigned a-e"""

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

```
Out[72]: a    -0.409934
b    -0.188738
c    -1.018550
d     0.950447
e    -0.786229
dtype: float64
```

```
In [73]: """Following function will print the index and its datatype"""
s.index
```

```
Out[73]: Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

```
In [74]: """If we don't assign the index then it will of length having values [0.....len(data-1)]"""
pd.Series(np.random.randn(5))
```

```
Out[74]: 0    0.165638
1    0.605373
2    1.473079
3   -0.497316
4    0.355552
dtype: float64
```

## 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 [75]: """ In following example, indexes are not given to it is constructed from the sorted keys of the dict"""  
d = {'a' : 0., 'b' : 1., 'c' : 2.} # a python dict  
pd.Series(d)
```

```
Out[75]: a    0.0  
        b    1.0  
        c    2.0  
        dtype: float64
```

```
In [76]: """ In following example, index are given, so the values in data corresponding in the index will be pulled out """  
pd.Series(d, index=['b', 'c', 'd', 'a'])
```

```
Out[76]: b    1.0  
        c    2.0  
        d    NaN  
        a    0.0  
        dtype: float64
```

## 10.3 From a scalar value

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

```
In [77]: """In following example, a scalar value is given as data so it will be repeated to match the length of index"""  
pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
```

```
Out[77]: a    5.0  
        b    5.0  
        c    5.0  
        d    5.0  
        e    5.0  
        dtype: float64
```

## 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 [78]: #we can access a value just like ndarray
```

```
#access single value  
s[0]
```

```
Out[78]: -0.4099336125957855
```

```
In [79]: #access range of values
```

```
s[:5]
```

```
Out[79]: a   -0.409934  
        b   -0.188738  
        c   -1.018550  
        d    0.950447  
        e   -0.786229  
        dtype: float64
```

```
In [80]: """ Following example will return a range of values in series whose value is greater than the median of serie  
s"""  
s[s > s.median()]
```

```
Out[80]: b   -0.188738  
        d    0.950447  
        dtype: float64
```

```
In [81]: """Following example is return the values in series with indexes. 4,3,1 are the positions of the indexes  
For example: the index at 4,3,1 are e,d,b respectively"""  
s[[4, 3, 1]]
```

```
Out[81]: e   -0.786229  
        d    0.950447  
        b   -0.188738  
        dtype: float64
```

```
In [82]: """ Following example returns the exponent values. just like e^a (here a is index and its respective data is placed here) """  
np.exp(s)
```

```
Out[82]: a    0.663694  
b    0.828003  
c    0.361118  
d    2.586866  
e    0.455559  
dtype: float64
```

```
In [83]: """Following example will get the data of given index"""  
s['a']
```

```
Out[83]: -0.4099336125957855
```

```
In [84]: """Following example will update the data of the given index"""  
s['e'] = 12.  
s # before updating e = 1.399281 but after updating e = 12.000000
```

```
Out[84]: a    -0.409934  
b    -0.188738  
c    -1.018550  
d     0.950447  
e    12.000000  
dtype: float64
```

```
In [85]: """ Following will return true if 'e' is in the values of index otherwise false """  
'e' in s
```

```
Out[85]: True
```

```
In [ ]: """If a label is not contained and you are trying to access its data, an exception is raised: """  
s['f']  
  
# This will create error
```

```
In [87]: """Using the get method, a missing label will return None or specified default"""  
s.get('f') #it will return none  
s.get('f', np.nan) #it will return default value
```

Out[87]: nan

## 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 [88]: """following will add the data of respective values of indexes. For example, in given output, it is calculate  
d as:  
a = s['a'] + s['a']  
b = s['b'] + s['b']  
c = s['c'] + s['c']  
d = s['d'] + s['d']  
e = s['e'] + s['e']  
  
"""  
s + s
```

```
Out[88]: a    -0.819867  
b    -0.377477  
c    -2.037099  
d     1.900894  
e    24.000000  
dtype: float64
```

```
In [89]: """following will multiply the data of each values of indexes, with 2. For example, in given output, it is calculated as:  
a = s['a'] *2  
b = s['b'] *2  
c = s['c'] *2  
d = s['d'] *2  
e = s['e'] *2"""  
  
s * 2
```

```
Out[89]: a    -0.819867  
b    -0.377477  
c    -2.037099  
d     1.900894  
e    24.000000  
dtype: float64
```

```
In [90]: s = pd.Series(np.random.randn(5), name='something')  
s
```

```
Out[90]: 0    -0.837043  
1     0.020726  
2     0.189074  
3    -0.847838  
4    -0.651682  
Name: something, dtype: float64
```

```
In [91]: s.name #print the name attribute of series
```

```
Out[91]: 'something'
```

```
In [92]: #rename the series name attribute and assign to s2 object. Note that s and s2 refer to different objects.
```

```
s2 = s.rename("different")  
s2.name
```

```
Out[92]: 'different'
```

## 11. Data Frames

- DataFrames are the workhorse of pandas and are directly inspired by the R programming language.
- Like Series, DataFrame accepts many different kinds of input:
  - Dict of 1D ndarrays, lists, dicts, or Series
  - 2-D numpy.ndarray
  - Structured or record ndarray
  - A Series
  - Another DataFrame
- 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 dicts

- 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 [93]: """ A dict is created """
d = {
    'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
    'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])
}

"""create a dataframe. row label will be the index of a series. As column labels are not given so it
will be sorted list of dict keys"""

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

Out[93]:

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

```
In [94]: """ a data frame will be constructed for given row labels """
pd.DataFrame(d, index=['d', 'b', 'a'])
```

Out[94]:

	one	two
d	NaN	4.0
b	2.0	2.0
a	1.0	1.0

```
In [95]: """following example shows a data frame when we give coloumn labels"""  
pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
```

Out[95]:

	two	three
d	4.0	NaN
b	2.0	NaN
a	1.0	NaN

```
In [96]: df.columns
```

Out[96]: Index(['one', 'two'], dtype='object')

## 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 [97]: """following examples shows that ndarray has same length"""
d = {
    'one' : [1., 2., 3., 4.],
    'two' : [4., 3., 2., 1.]
}

"""column labels are not given so the result will be range(n), where n is the array length"""
pd.DataFrame(d)
```

Out[97]:

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

```
In [98]: """If indexs are given then it would be same length as arrays"""
pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
```

Out[98]:

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

## 11.3 From a list of dicts

```
In [99]: """constructing data frame from a list of dicts"""
data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
pd.DataFrame(data2)
```

Out[99]:

	a	b	c
0	1	2	NaN
1	5	10	20.0

```
In [100]: """passing list of dicts as data and indexes (row labels)"""
pd.DataFrame(data2, index=['first', 'second'])
```

Out[100]:

	a	b	c
first	1	2	NaN
second	5	10	20.0

```
In [101]: """passing list of dicts as data and columns (columns labels)"""
pd.DataFrame(data2, columns=['a', 'b'])
```

Out[101]:

	a	b
0	1	2
1	5	10

## 11.4 From a dict of tuples

You can automatically create a multi-indexed frame by passing a tuples dictionary

```
In [102]: 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}})

#NaN shows missing data
```

Out[102]:

		a			b	
		a	b	c	a	b
A	B	4.0	1.0	5.0	8.0	10.0
	C	3.0	2.0	6.0	7.0	NaN
	D	NaN	NaN	NaN	NaN	9.0

## 11.5 Alternate Constructors

### **DataFrame.from\_dict**

- **DataFrame.from\_dict** 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.

### **DataFrame.from\_records**

- **DataFrame.from\_records** 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. For example:

```
In [103]: data = np.zeros((2,), dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10')])
data
```

```
Out[103]: array([(0, 0., b''), (0, 0., b'')],
                dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
```

```
In [104]: pd.DataFrame.from_records(data, index='C')
```

```
Out[104]:
```

	A	B
C		
b"	0	0.0
b"	0	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 [105]: pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])])
```

```
Out[105]:
```

	A	B
0	1	4
1	2	5
2	3	6

If you pass orient='index', the keys will be the row labels. But in this case you must also pass the desired column names:

```
In [106]: pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
                                orient='index', columns=['one', 'two', 'three'])
```

```
Out[106]:
```

	one	two	three
A	1	2	3
B	4	5	6

## 11.6 Column selection, addition, deletion

- DataFrame can be treated semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations.

```
In [107]: df['one'] # it is displaying data under coloumn 'one'
```

```
Out[107]: a    1.0
          b    2.0
          c    3.0
          d    NaN
          Name: one, dtype: float64
```

```
In [108]: df['three'] = df['one'] * df['two'] # assigning values to a colomn named 'three' after calculation
```

```
In [109]: df['flag'] = df['one'] > 2 #check if value at column 'one' is > 2 then assign True otherwise false
```

```
In [110]: df #print a complete data frame
```

```
Out[110]:
```

	one	two	three	flag
a	1.0	1.0	1.0	False
b	2.0	2.0	4.0	False
c	3.0	3.0	9.0	True
d	NaN	4.0	NaN	False

Columns can be deleted or popped like with a dict:

```
In [112]: del df['two'] #delete a coloumn 'two' from data frame
```

```
In [113]: three = df.pop('three') #pop a complete coloumn 'three' from dataframe
```

```
In [114]: df
```

```
Out[114]:
```

	one	flag
a	1.0	False
b	2.0	False
c	3.0	True
d	NaN	False

When inserting a scalar value, it will naturally be propagated to fill the column:

```
In [115]: df['foo'] = 'bar' #a coloumn 'foo' will be populated with 'bar'
```

```
In [116]: df
```

```
Out[116]:
```

	one	flag	foo
a	1.0	False	bar
b	2.0	False	bar
c	3.0	True	bar
d	NaN	False	bar

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame's index:

```
In [117]: """following example will take values from coloumn one until give range and will populate the new coloumn"""  
df['one_trunc'] = df['one'][:2]
```



In [118]: df

Out[118]:

	one	flag	foo	one_trunc
a	1.0	False	bar	1.0
b	2.0	False	bar	2.0
c	3.0	True	bar	NaN
d	NaN	False	bar	NaN

By default, columns get inserted at the end. The insert function is available to insert at a particular location in the columns:

```
In [119]: """following function has three arguments.
First argument: index where new column will be inserted.
Second argument: label or title of a new column
Third argument: it will create a column at specified position"""

df.insert(1, 'bar2', df['one'])
```

In [120]: df

Out[120]:

	one	bar2	flag	foo	one_trunc
a	1.0	1.0	False	bar	1.0
b	2.0	2.0	False	bar	2.0
c	3.0	3.0	True	bar	NaN
d	NaN	NaN	False	bar	NaN

## 11.7 Indexing / Selection

- Row selection, for example, returns a Series whose index is the columns of the DataFrame:

```
In [121]: df.loc['b'] #it will return the coloumn labels and values on row label 'b'
```

```
Out[121]: one          2  
bar2          2  
flag         False  
foo          bar  
one_trunc      2  
Name: b, dtype: object
```

```
In [122]: df.iloc[2] #it will return the values of those coloumns that is > than 2
```

```
Out[122]: one          3  
bar2          3  
flag          True  
foo          bar  
one_trunc     NaN  
Name: c, dtype: object
```

## 11.8 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 [123]: df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])
```

```
In [124]: df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
```

```
In [125]: df + df2 # add values of respective column labels
```

```
Out[125]:
```

	A	B	C	D
0	-0.155918	0.552013	-0.602292	NaN
1	1.599658	-1.959156	0.145379	NaN
2	2.190079	-0.066512	-0.862440	NaN
3	1.489033	0.639123	1.761369	NaN
4	-0.557114	-0.291224	-0.268751	NaN
5	-2.666360	0.832140	0.458249	NaN
6	-0.108468	0.794005	-0.843568	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	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. For example:

```
In [126]: df - df.iloc[0]
```

```
Out[126]:
```

	A	B	C	D
0	0.000000	0.000000	0.000000	0.000000
1	0.120963	-0.830042	2.472082	-0.679995
2	0.263346	-0.859860	1.463904	-0.674416
3	-0.129070	0.372500	3.231676	0.025416
4	0.920473	-0.165858	1.877756	-0.291604
5	0.458634	-0.122398	1.564088	1.948560
6	0.171784	0.114905	0.906762	-1.033237
7	-0.085082	-0.963476	2.602159	-0.316969
8	0.835161	-0.515849	2.682794	-0.405550
9	1.052372	-0.917148	1.701121	1.365964

In [127]: `df * 5 + 2`

Out[127]:

	A	B	C	D
0	0.200430	3.627299	-6.701878	-1.374810
1	0.805247	-0.522911	5.658535	-4.774786
2	1.517160	-0.672002	0.617642	-4.746888
3	-0.444921	5.489799	9.456503	-1.247728
4	4.802794	2.798009	2.686903	-2.832830
5	2.493598	3.015311	1.118563	8.367987
6	1.059353	4.201826	-2.168067	-6.540996
7	-0.224979	-1.190080	6.308918	-2.959657
8	4.376237	1.048055	6.712090	-3.402559
9	5.462289	-0.958443	1.803729	5.455008

In [128]: 1 / df

Out[128]:

	A	B	C	D
0	-2.778442	3.072576	-0.574589	-1.481565
1	-4.184967	-1.981837	1.366667	-0.738031
2	-10.355393	-1.871256	-3.617009	-0.741082
3	-2.045056	1.432747	0.670556	-1.539538
4	1.783934	6.265591	7.279048	-1.034590
5	10.129696	4.924599	-5.672558	0.785177
6	-5.315488	2.270843	-1.199597	-0.585412
7	-2.247212	-1.567359	1.160384	-1.008134
8	2.104167	-5.252404	1.061100	-0.925487
9	1.444131	-1.690078	-25.474980	1.447175

```
In [129]: df ** 4
```

```
Out[129]:
```

	A	B	C	D
0	0.016780	0.011220	9.174278	0.207547
1	0.003260	0.064823	0.286648	3.370561
2	0.000087	0.081558	0.005843	3.315385
3	0.057171	0.237313	4.946076	0.178008
4	0.098738	0.000649	0.000356	0.872823
5	0.000095	0.001700	0.000966	2.631048
6	0.001253	0.037606	0.482902	8.514402
7	0.039212	0.165701	0.551560	0.968114
8	0.051013	0.001314	0.788814	1.363069
9	0.229918	0.122567	0.000002	0.227990

Boolean operators work as well:

```
In [130]: df1 = pd.DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=bool)
```

```
In [131]: df2 = pd.DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)
```

```
In [132]: pd.DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)
```

```
Out[132]:
```

	a	b
0	False	True
1	True	True
2	True	False

```
In [133]: df1 & df2 #and logical operator
```

```
Out[133]:
```

	a	b
0	False	False
1	False	True
2	True	False

```
In [134]: df1 | df2 # or operator
```

```
Out[134]:
```

	a	b
0	True	True
1	True	True
2	True	True

```
In [135]: ~df1
```

```
Out[135]:
```

	a	b
0	False	True
1	True	False
2	False	False

## 11.9 Transposing

- To transpose, access the T attribute (also the transpose function), similar to an ndarray



```
In [136]: # only show the first 5 rows
df[:5].T
```

Out[136]:

	0	1	2	3	4
A	-0.359914	-0.238951	-0.096568	-0.488984	0.560559
B	0.325460	-0.504582	-0.534400	0.697960	0.159602
C	-1.740376	0.731707	-0.276472	1.491301	0.137381
D	-0.674962	-1.354957	-1.349378	-0.649546	-0.966566

Creating a DataFrame by passing a numpy array, with a datetime index and labeled columns:

```
In [137]: dates = pd.date_range('20130101', periods=6)
```

```
In [138]: dates
```

```
Out[138]: DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
                        '2013-01-05', '2013-01-06'],
                        dtype='datetime64[ns]', freq='D')
```

```
In [139]: df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
```

In [140]: df

Out[140]:

	A	B	C	D
<b>2013-01-01</b>	-0.119099	-1.220438	-1.533635	-1.024571
<b>2013-01-02</b>	0.793731	-0.469725	1.116814	-0.235097
<b>2013-01-03</b>	2.527459	-1.344347	-0.043718	0.136468
<b>2013-01-04</b>	0.388668	-1.613518	-1.713179	0.035402
<b>2013-01-05</b>	1.988020	-0.068843	0.948234	0.136082
<b>2013-01-06</b>	1.724599	-0.356748	-0.178683	0.779853

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

```
In [141]: 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' })
```

In [142]: df2

Out[142]:

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

```
In [143]: #Having specific dtypes  
df2.dtypes
```

```
Out[143]: A          float64  
         B    datetime64[ns]  
         C          float32  
         D          int32  
         E          category  
         F          object  
dtype: object
```

## 12. Viewing Data

- 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 label
- 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 [144]: df.head() #display first 5 records
```

```
Out[144]:
```

	A	B	C	D
<b>2013-01-01</b>	-0.119099	-1.220438	-1.533635	-1.024571
<b>2013-01-02</b>	0.793731	-0.469725	1.116814	-0.235097
<b>2013-01-03</b>	2.527459	-1.344347	-0.043718	0.136468
<b>2013-01-04</b>	0.388668	-1.613518	-1.713179	0.035402
<b>2013-01-05</b>	1.988020	-0.068843	0.948234	0.136082

```
In [145]: df.tail(3) #display last 3 records
```

```
Out[145]:
```

	A	B	C	D
<b>2013-01-04</b>	0.388668	-1.613518	-1.713179	0.035402
<b>2013-01-05</b>	1.988020	-0.068843	0.948234	0.136082
<b>2013-01-06</b>	1.724599	-0.356748	-0.178683	0.779853

```
In [146]: df.index #display indexes
```

```
Out[146]: DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',  
                        '2013-01-05', '2013-01-06'],  
                        dtype='datetime64[ns]', freq='D')
```

```
In [147]: df.columns #display coloumns
```

```
Out[147]: Index(['A', 'B', 'C', 'D'], dtype='object')
```

In [148]: `df.values # print values`

```
Out[148]: array([[ -0.11909882, -1.22043765, -1.53363511, -1.02457067],
 [  0.79373055, -0.4697249 ,  1.11681407, -0.23509747],
 [  2.52745909, -1.34434745, -0.04371833,  0.1364679 ],
 [  0.38866789, -1.61351844, -1.71317937,  0.03540218],
 [  1.98802045, -0.06884328,  0.9482335 ,  0.13608198],
 [  1.72459927, -0.35674845, -0.17868323,  0.7798529 ]])
```

In [149]: `# Transposing your data`  
`df.T`

Out[149]:

	<b>2013-01-01 00:00:00</b>	<b>2013-01-02 00:00:00</b>	<b>2013-01-03 00:00:00</b>	<b>2013-01-04 00:00:00</b>	<b>2013-01-05 00:00:00</b>	<b>2013-01-06 00:00:00</b>
<b>A</b>	-0.119099	0.793731	2.527459	0.388668	1.988020	1.724599
<b>B</b>	-1.220438	-0.469725	-1.344347	-1.613518	-0.068843	-0.356748
<b>C</b>	-1.533635	1.116814	-0.043718	-1.713179	0.948234	-0.178683
<b>D</b>	-1.024571	-0.235097	0.136468	0.035402	0.136082	0.779853

In [150]: `#Sorting by an axis`  
`df.sort_index(axis=1, ascending=False)`

Out[150]:

	<b>D</b>	<b>C</b>	<b>B</b>	<b>A</b>
<b>2013-01-01</b>	-1.024571	-1.533635	-1.220438	-0.119099
<b>2013-01-02</b>	-0.235097	1.116814	-0.469725	0.793731
<b>2013-01-03</b>	0.136468	-0.043718	-1.344347	2.527459
<b>2013-01-04</b>	0.035402	-1.713179	-1.613518	0.388668
<b>2013-01-05</b>	0.136082	0.948234	-0.068843	1.988020
<b>2013-01-06</b>	0.779853	-0.178683	-0.356748	1.724599

```
In [151]: #Sorting by values
df.sort_values(by='B')
```

Out[151]:

	A	B	C	D
<b>2013-01-04</b>	0.388668	-1.613518	-1.713179	0.035402
<b>2013-01-03</b>	2.527459	-1.344347	-0.043718	0.136468
<b>2013-01-01</b>	-0.119099	-1.220438	-1.533635	-1.024571
<b>2013-01-02</b>	0.793731	-0.469725	1.116814	-0.235097
<b>2013-01-06</b>	1.724599	-0.356748	-0.178683	0.779853
<b>2013-01-05</b>	1.988020	-0.068843	0.948234	0.136082

```
In [152]: # Describe shows a quick statistic summary of your data
df.describe()
```

Out[152]:

	A	B	C	D
<b>count</b>	6.000000	6.000000	6.000000	6.000000
<b>mean</b>	1.217230	-0.845603	-0.234028	-0.028644
<b>std</b>	1.021752	0.626508	1.194878	0.591043
<b>min</b>	-0.119099	-1.613518	-1.713179	-1.024571
<b>25%</b>	0.489934	-1.313370	-1.194897	-0.167473
<b>50%</b>	1.259165	-0.845081	-0.111201	0.085742
<b>75%</b>	1.922165	-0.384993	0.700246	0.136371
<b>max</b>	2.527459	-0.068843	1.116814	0.779853

```
In [153]: #Selecting a single column, which yields a Series, equivalent to df.A  
df['A']
```

```
Out[153]: 2013-01-01    -0.119099  
          2013-01-02     0.793731  
          2013-01-03     2.527459  
          2013-01-04     0.388668  
          2013-01-05     1.988020  
          2013-01-06     1.724599  
          Freq: D, Name: A, dtype: float64
```

```
In [154]: #Selecting via [], which slices the rows.  
df[0:3]
```

```
Out[154]:
```

	A	B	C	D
<b>2013-01-01</b>	-0.119099	-1.220438	-1.533635	-1.024571
<b>2013-01-02</b>	0.793731	-0.469725	1.116814	-0.235097
<b>2013-01-03</b>	2.527459	-1.344347	-0.043718	0.136468

```
In [155]: df['20130102':'20130104']
```

```
Out[155]:
```

	A	B	C	D
<b>2013-01-02</b>	0.793731	-0.469725	1.116814	-0.235097
<b>2013-01-03</b>	2.527459	-1.344347	-0.043718	0.136468
<b>2013-01-04</b>	0.388668	-1.613518	-1.713179	0.035402

```
In [156]: #Selecting on a multi-axis by label  
df.loc[:,['A','B']]
```

Out[156]:

	A	B
2013-01-01	-0.119099	-1.220438
2013-01-02	0.793731	-0.469725
2013-01-03	2.527459	-1.344347
2013-01-04	0.388668	-1.613518
2013-01-05	1.988020	-0.068843
2013-01-06	1.724599	-0.356748

```
In [157]: #Showing label slicing, both endpoints are included  
df.loc['20130102':'20130104',['A','B']]
```

Out[157]:

	A	B
2013-01-02	0.793731	-0.469725
2013-01-03	2.527459	-1.344347
2013-01-04	0.388668	-1.613518

```
In [158]: # Reduction in the dimensions of the returned object  
df.loc['20130102',['A','B']]
```

Out[158]: A     0.793731  
          B    -0.469725  
          Name: 2013-01-02 00:00:00, dtype: float64

```
In [159]: # For getting a scalar value  
df.loc[dates[0], 'A']
```

Out[159]: -0.1190988226942807



```
In [160]: # For getting fast access to a scalar  
df.at[dates[0], 'A']
```

```
Out[160]: -0.1190988226942807
```

```
In [161]: # Select via the position of the passed integers  
df.iloc[3]
```

```
Out[161]: A    0.388668  
         B   -1.613518  
         C   -1.713179  
         D    0.035402  
         Name: 2013-01-04 00:00:00, dtype: float64
```

```
In [162]: # By integer slices, acting similar to numpy/python  
df.iloc[3:5, 0:2]
```

```
Out[162]:
```

	A	B
2013-01-04	0.388668	-1.613518
2013-01-05	1.988020	-0.068843

```
In [163]: # By lists of integer position locations, similar to the numpy/python style  
df.iloc[[1, 2, 4], [0, 2]]
```

```
Out[163]:
```

	A	C
2013-01-02	0.793731	1.116814
2013-01-03	2.527459	-0.043718
2013-01-05	1.988020	0.948234

```
In [164]: # For slicing rows explicitly  
df.iloc[:,1:3]
```

Out[164]:

	B	C
2013-01-01	-1.220438	-1.533635
2013-01-02	-0.469725	1.116814
2013-01-03	-1.344347	-0.043718
2013-01-04	-1.613518	-1.713179
2013-01-05	-0.068843	0.948234
2013-01-06	-0.356748	-0.178683

```
In [165]: # For getting a value explicitly  
df.iloc[1,1]
```

Out[165]: -0.4697249010369114

```
In [166]: # Using a single column's values to select data.  
df[df.A > 0]
```

Out[166]:

	A	B	C	D
2013-01-02	0.793731	-0.469725	1.116814	-0.235097
2013-01-03	2.527459	-1.344347	-0.043718	0.136468
2013-01-04	0.388668	-1.613518	-1.713179	0.035402
2013-01-05	1.988020	-0.068843	0.948234	0.136082
2013-01-06	1.724599	-0.356748	-0.178683	0.779853

```
In [167]: # Selecting values from a DataFrame where a boolean condition is met.
df[df > 0]
```

Out[167]:

	A	B	C	D
<b>2013-01-01</b>	NaN	NaN	NaN	NaN
<b>2013-01-02</b>	0.793731	NaN	1.116814	NaN
<b>2013-01-03</b>	2.527459	NaN	NaN	0.136468
<b>2013-01-04</b>	0.388668	NaN	NaN	0.035402
<b>2013-01-05</b>	1.988020	NaN	0.948234	0.136082
<b>2013-01-06</b>	1.724599	NaN	NaN	0.779853

```
In [168]: # Using the isin() method for filtering:
df2 = df.copy()
```

```
In [169]: df2['E'] = ['one', 'one', 'two', 'three', 'four', 'three']
```

```
In [170]: df2
```

Out[170]:

	A	B	C	D	E
<b>2013-01-01</b>	-0.119099	-1.220438	-1.533635	-1.024571	one
<b>2013-01-02</b>	0.793731	-0.469725	1.116814	-0.235097	one
<b>2013-01-03</b>	2.527459	-1.344347	-0.043718	0.136468	two
<b>2013-01-04</b>	0.388668	-1.613518	-1.713179	0.035402	three
<b>2013-01-05</b>	1.988020	-0.068843	0.948234	0.136082	four
<b>2013-01-06</b>	1.724599	-0.356748	-0.178683	0.779853	three

```
In [171]: df2[df2['E'].isin(['two', 'four'])]
```

```
Out[171]:
```

	A	B	C	D	E
<b>2013-01-03</b>	2.527459	-1.344347	-0.043718	0.136468	two
<b>2013-01-05</b>	1.988020	-0.068843	0.948234	0.136082	four