Prof. Dhanashree S.
Expt. No. 7

Python Pandas - Date Functionality

Extending the Time series, Date functionalities play major role in financial data analysis. While working with Date data, we will frequently come across the following −

- Generating sequence of dates
- Convert the date series to different frequencies

Create a Range of Dates

Using the **date.range()** function by specifying the periods and the frequency, we can create the date series. By default, the frequency of range is Days.

```
import pandas as pd

print pd.date_range('1/1/2011', periods=5)
```

Its **output** is as follows −

```
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-
01-04', '2011-01-05'],
   dtype='datetime64[ns]', freq='D')
```

Change the Date Frequency

```
import pandas as pd

print pd.date_range('1/1/2011', periods=5,freq='M')
```

Its **output** is as follows −

```
DatetimeIndex(['2011-01-31', '2011-02-28', '2011-03-31', '2011-
04-30', '2011-05-31'],
   dtype='datetime64[ns]', freq='M')
```

bdate_range

bdate_range() stands for business date ranges. Unlike date_range(), it excludes Saturday and Sunday.

```
import pandas as pd

print pd.date_range('1/1/2011', periods=5)
```

Its **output** is as follows −

```
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-
01-04', '2011-01-05'],
   dtype='datetime64[ns]', freq='D')
```

Observe, after 3rd March, the date jumps to 6th march excluding 4th and 5th. Just check your calendar for the days.

Convenience functions like **date_range** and **bdate_range** utilize a variety of frequency aliases. The default frequency for date_range is a calendar day while the default for bdate_range is a business day.

```
import pandas as pd
start = pd.datetime(2011, 1, 1)
end = pd.datetime(2011, 1, 5)

print pd.date_range(start, end)
```

Its **output** is as follows −

```
DatetimeIndex(['2011-01-01', '2011-01-02', '2011-01-03', '2011-
01-04', '2011-01-05'],
   dtype='datetime64[ns]', freq='D')
```

Offset Aliases

A number of string aliases are given to useful common time series frequencies. We will refer to these aliases as offset aliases.

Alias	Description	Alias	Description
B	business day frequency	BQS	business quarter start frequency
D	calendar day frequency	A	annual(Year) end frequency
W	weekly frequency	BA	business year end frequency
M	month end frequency	BAS	business year start frequency
SM	semi-month end frequency	BH	business hour frequency
BM	business month end frequency	H	hourly frequency
MS	month start frequency	T, min	minutely frequency

SMS	SMS semi month start frequency	S	secondly frequency
BMS	business month start frequency	L, ms	milliseconds
Q	quarter end frequency	U, us	microseconds
BQ	business quarter end frequency	N	nanoseconds
QS	quarter start frequency		

Python Pandas - Timedelta

Timedeltas are differences in times, expressed in difference units, for example, days, hours, minutes, seconds. They can be both positive and negative.

We can create Timedelta objects using various arguments as shown below −

String

By passing a string literal, we can create a timedelta object.

```
import pandas as pd

print pd.Timedelta('2 days 2 hours 15 minutes 30 seconds')
```

Its **output** is as follows −

```
2 days 02:15:30
```

Integer

By passing an integer value with the unit, an argument creates a Timedelta object.

```
import pandas as pd

print pd.Timedelta(6,unit='h')
```

Its **output** is as follows −

```
0 days 06:00:00
```

Data Offsets

Data offsets such as - weeks, days, hours, minutes, seconds, milliseconds, microseconds, nanoseconds can also be used in construction.

```
import pandas as pd
```

```
print pd.Timedelta(days=2)
```

Its **output** is as follows −

```
2 days 00:00:00
```

to_timedelta()

Using the top-level **pd.to_timedelta**, you can convert a scalar, array, list, or series from a recognized timedelta format/ value into a Timedelta type. It will construct Series if the input is a Series, a scalar if the input is scalar-like, otherwise will output a **TimedeltaIndex**.

```
import pandas as pd

print pd.Timedelta(days=2)
```

Its **output** is as follows −

```
2 days 00:00:00
```

Operations

You can operate on Series/ DataFrames and construct **timedelta64[ns]** Series through subtraction operations on **datetime64[ns]** Series, or Timestamps.

Let us now create a DataFrame with Timedelta and datetime objects and perform some arithmetic operations on it −

```
import pandas as pd

s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))
td = pd.Series([ pd.Timedelta(days=i) for i in range(3) ])
df = pd.DataFrame(dict(A = s, B = td))

print df
```

Its **output** is as follows −

```
            A       B
0  2012-01-01 0 days
1  2012-01-02 1 days
2  2012-01-03 2 days
```

Addition Operations

```
import pandas as pd

s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))
td = pd.Series([ pd.Timedelta(days=i) for i in range(3) ])
df = pd.DataFrame(dict(A = s, B = td))
df['C']=df['A']+df['B']

print df
```

Its **output** is as follows −

```
           A       B          C
```

```
0 2012-01-01 0 days 2012-01-01
1 2012-01-02 1 days 2012-01-03
2 2012-01-03 2 days 2012-01-05
```

Subtraction Operation

```
import pandas as pd

s = pd.Series(pd.date_range('2012-1-1', periods=3, freq='D'))
td = pd.Series([ pd.Timedelta(days=i) for i in range(3) ])
df = pd.DataFrame(dict(A = s, B = td))
df['C']=df['A']+df['B']
df['D']=df['C']+df['B']

print df
```

Its **output** is as follows −

```
           A      B          C          D
0 2012-01-01 0 days 2012-01-01 2012-01-01
1 2012-01-02 1 days 2012-01-03 2012-01-04
2 2012-01-03 2 days 2012-01-05 2012-01-07
```

Python Pandas - Categorical Data

Often in real-time, data includes the text columns, which are repetitive. Features like gender, country, and codes are always repetitive. These are the examples for categorical data.

Categorical variables can take on only a limited, and usually fixed number of possible values. Besides the fixed length, categorical data might have an order but cannot perform numerical operation. Categorical are a Pandas data type.

The categorical data type is useful in the following cases −

- A string variable consisting of only a few different values. Converting such a string variable to a categorical variable will save some memory.
- The lexical order of a variable is not the same as the logical order ("one", "two", "three"). By converting to a categorical and specifying an order on the categories, sorting and min/max will use the logical order instead of the lexical order.
- As a signal to other python libraries that this column should be treated as a categorical variable (e.g. to use suitable statistical methods or plot types).

Object Creation

Categorical object can be created in multiple ways. The different ways have been described below −

category

By specifying the dtype as "category" in pandas object creation.

```
import pandas as pd
```

```
s = pd.Series(["a","b","c","a"], dtype="category")
print s
```

Its **output** is as follows −

```
0   a
1   b
2   c
3   a
dtype: category
Categories (3, object): [a, b, c]
```

The number of elements passed to the series object is four, but the categories are only three. Observe the same in the output Categories.

pd.Categorical

Using the standard pandas Categorical constructor, we can create a category object.

```
pandas.Categorical(values, categories, ordered)
```

Let's take an example −

```
import pandas as pd

cat = pd.Categorical(['a', 'b', 'c', 'a', 'b', 'c'])
print cat
```

Its **output** is as follows −

```
[a, b, c, a, b, c]
Categories (3, object): [a, b, c]
```

Let's have another example −

```
import pandas as pd

cat = cat=pd.Categorical(['a','b','c','a','b','c','d'], ['c',
'b', 'a'])
print cat
```

Its **output** is as follows −

```
[a, b, c, a, b, c, NaN]
Categories (3, object): [c, b, a]
```

Here, the second argument signifies the categories. Thus, any value which is not present in the categories will be treated as **NaN**.

Now, take a look at the following example −

```
import pandas as pd

cat = cat=pd.Categorical(['a','b','c','a','b','c','d'], ['c',
'b', 'a'],ordered=True)
print cat
```

Its **output** is as follows –

```
[a, b, c, a, b, c, NaN]
Categories (3, object): [c < b < a]
```

Logically, the order means that, **a** is greater than **b** and **b** is greater than **c**.

Description

Using the **.describe()** command on the categorical data, we get similar output to a **Series** or **DataFrame** of the **type** string.

```
import pandas as pd
import numpy as np

cat = pd.Categorical(["a", "c", "c", np.nan], categories=["b",
"a", "c"])
df = pd.DataFrame({"cat":cat, "s":["a", "c", "c", np.nan]})

print df.describe()
print df["cat"].describe()
```

Its **output** is as follows –

```
       cat s
count    3 3
unique   2 2
top      c c
freq     2 2
count     3
unique    2
top       c
freq      2
Name: cat, dtype: object
```

Get the Properties of the Category

obj.cat.categories command is used to get the **categories of the object**.

```
import pandas as pd
import numpy as np

s = pd.Categorical(["a", "c", "c", np.nan], categories=["b", "a",
"c"])
print s.categories
```

Its **output** is as follows –

```
Index([u'b', u'a', u'c'], dtype='object')
```

obj.ordered command is used to get the order of the object.

```
import pandas as pd
import numpy as np

cat = pd.Categorical(["a", "c", "c", np.nan], categories=["b",
"a", "c"])
print cat.ordered
```

Its **output** is as follows −

```
False
```

The function returned **false** because we haven't specified any order.

Renaming Categories

Renaming categories is done by assigning new values to the **series.cat.categories**series.cat.categories property.

```
import pandas as pd

s = pd.Series(["a","b","c","a"], dtype="category")
s.cat.categories = ["Group %s" % g for g in s.cat.categories]
print s.cat.categories
```

Its **output** is as follows −

```
Index([u'Group a', u'Group b', u'Group c'], dtype='object')
```

Initial categories **[a,b,c]** are updated by the **s.cat.categories** property of the object.

Appending New Categories

Using the Categorical.add.categories() method, new categories can be appended.

```
import pandas as pd

s = pd.Series(["a","b","c","a"], dtype="category")
s = s.cat.add_categories([4])
print s.cat.categories
```

Its **output** is as follows −

```
Index([u'a', u'b', u'c', 4], dtype='object')
```

Removing Categories

Using the **Categorical.remove_categories()** method, unwanted categories can be removed.

```
import pandas as pd

s = pd.Series(["a","b","c","a"], dtype="category")
print ("Original object:")
print s

print ("After removal:")
print s.cat.remove_categories("a")
```

Its **output** is as follows −

```
Original object:
0   a
1   b
2   c
3   a
dtype: category
```

```
Categories (3, object): [a, b, c]

After removal:
0   NaN
1   b
2   c
3   NaN
dtype: category
Categories (2, object): [b, c]
```

Comparison of Categorical Data

Comparing categorical data with other objects is possible in three cases −

- comparing equality (== and !=) to a list-like object (list, Series, array, ...) of the same length as the categorical data.
- all comparisons (==, !=, >, >=, <, and <=) of categorical data to another categorical Series, when ordered==True and the categories are the same.
- all comparisons of a categorical data to a scalar.

Take a look at the following example −

```
import pandas as pd

cat = pd.Series([1,2,3]).astype("category", categories=[1,2,3],
ordered=True)
cat1 = pd.Series([2,2,2]).astype("category", categories=[1,2,3],
ordered=True)

print cat>cat1
```

Its **output** is as follows −

```
0   False
1   False
2   True
dtype: bool
```

Python Pandas - Visualization

Basic Plotting: plot

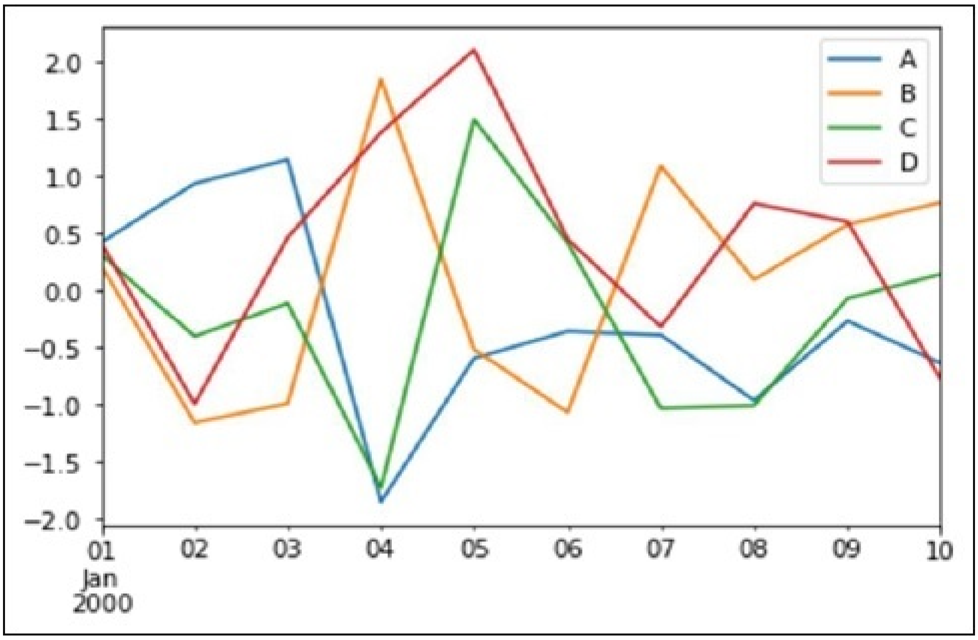
This functionality on Series and DataFrame is just a simple wrapper around the **matplotlib libraries plot()** method.

```
import pandas as pd
import numpy as np

df =
pd.DataFrame(np.random.randn(10,4),index=pd.date_range('1/1/2000'
,
   periods=10), columns=list('ABCD'))

df.plot()
```

Its **output** is as follows −



If the index consists of dates, it calls **gct().autofmt_xdate()** to format the x-axis as shown in the above illustration.

We can plot one column versus another using the **x** and **y** keywords.

Plotting methods allow a handful of plot styles other than the default line plot. These methods can be provided as the kind keyword argument to **plot()**. These include −

- bar or barh for bar plots
- hist for histogram
- box for boxplot
- 'area' for area plots
- 'scatter' for scatter plots

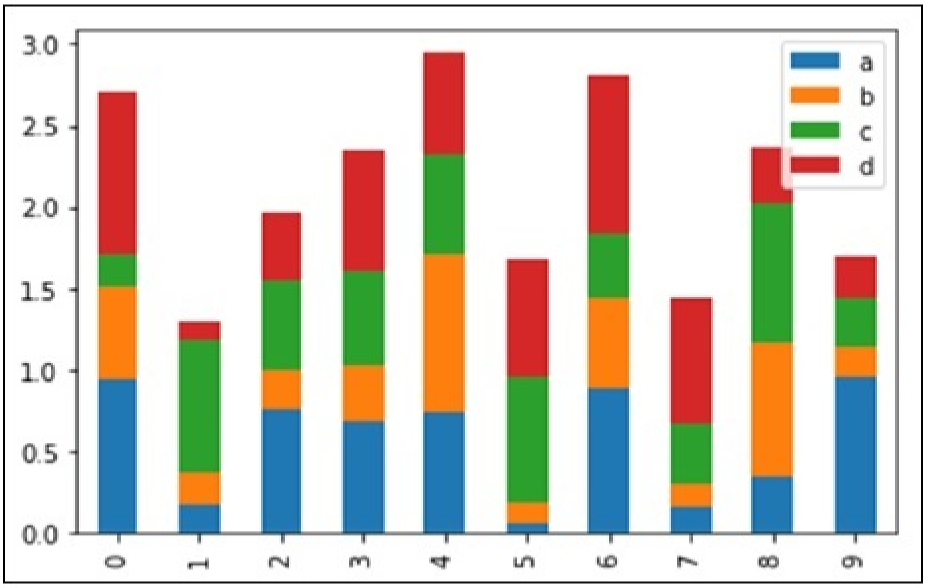
Bar Plot

Let us now see what a Bar Plot is by creating one. A bar plot can be created in the following way −

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.rand(10,4),columns=['a','b','c','d')
df.plot.bar()
```

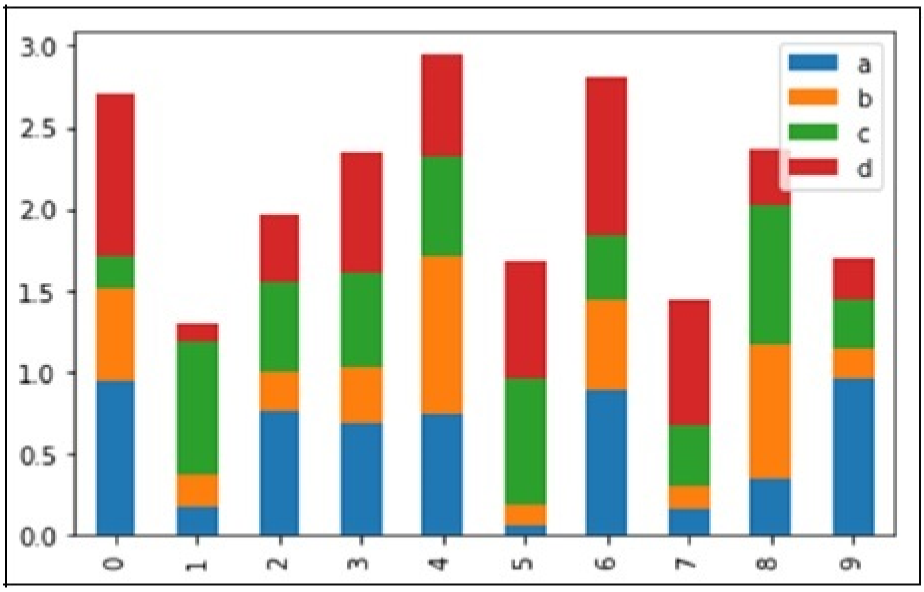
Its **output** is as follows −



To produce a stacked bar plot, **pass stacked=True** −

```
import pandas as pd
df = pd.DataFrame(np.random.rand(10,4),columns=['a','b','c','d')
df.plot.bar(stacked=True)
```

Its **output** is as follows −



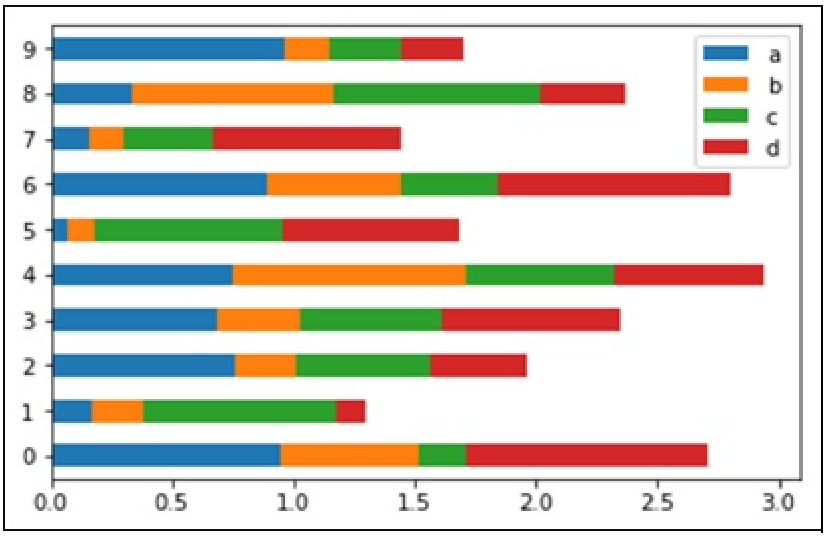
To get horizontal bar plots, use the **barh** method −

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.rand(10,4),columns=['a','b','c','d')

df.plot.barh(stacked=True)
```

Its **output** is as follows −



Histograms

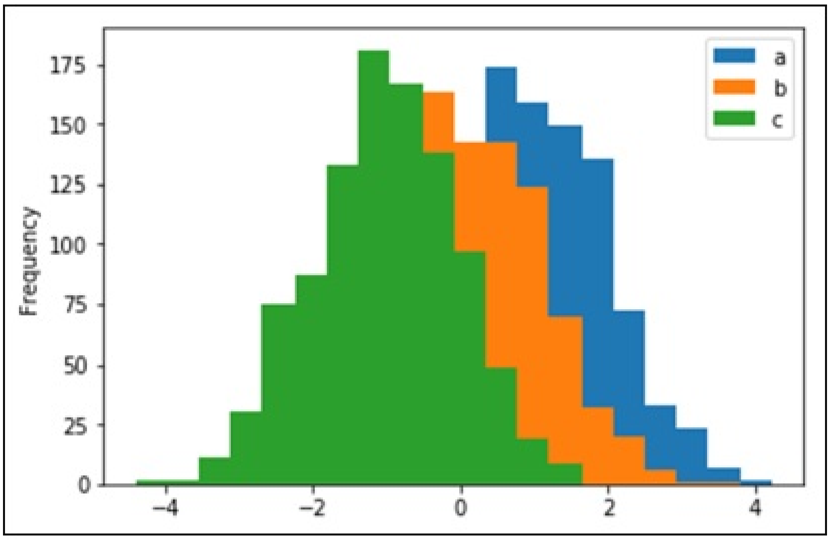
Histograms can be plotted using the **plot.hist()** method. We can specify number of bins.

```
import pandas as pd
import numpy as np

df =
pd.DataFrame({'a':np.random.randn(1000)+1,'b':np.random.randn(100
0),'c':
np.random.randn(1000) - 1}, columns=['a', 'b', 'c'])

df.plot.hist(bins=20)
```

Its **output** is as follows −



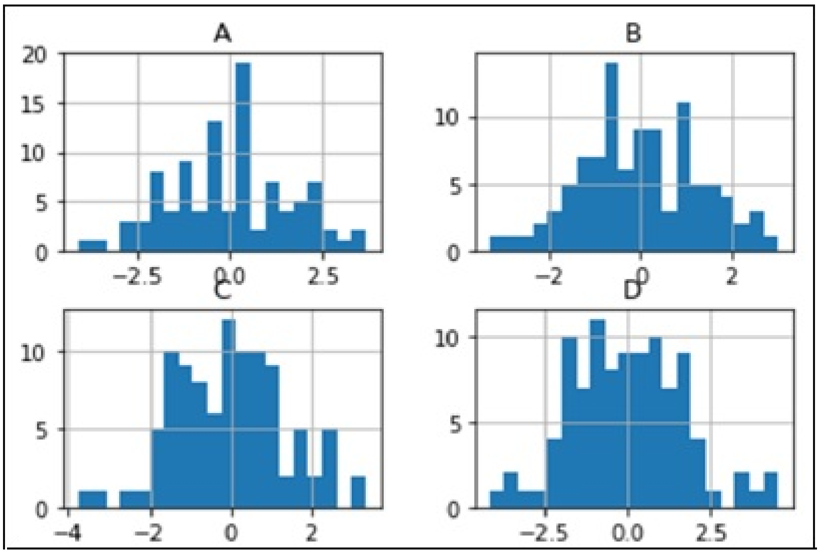
To plot different histograms for each column, use the following code −

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

```
df=pd.DataFrame({'a':np.random.randn(1000)+1,'b':np.random.randn(
1000),'c':
np.random.randn(1000) - 1}, columns=['a', 'b', 'c'])

df.diff.hist(bins=20)
```

Its **output** is as follows −



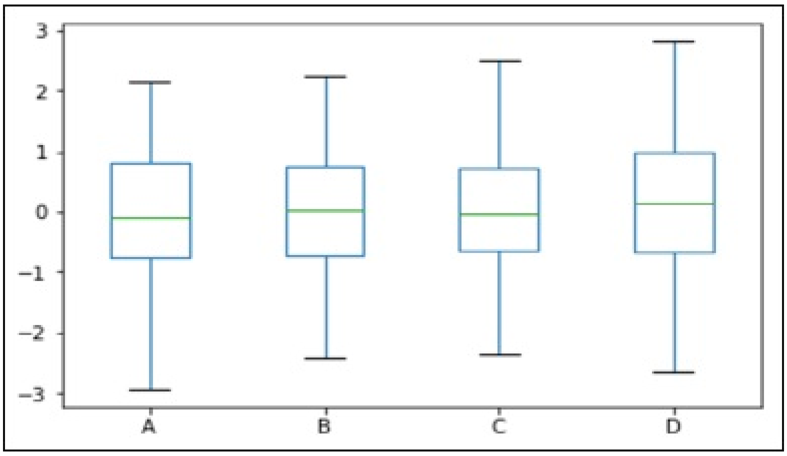
Box Plots

Boxplot can be drawn calling **Series.box.plot()** and **DataFrame.box.plot()**, or **DataFrame.boxplot()** to visualize the distribution of values within each column.

For instance, here is a boxplot representing five trials of 10 observations of a uniform random variable on [0,1).

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.rand(10, 5), columns=['A', 'B', 'C',
'D', 'E'])
df.plot.box()
```

Its **output** is as follows −



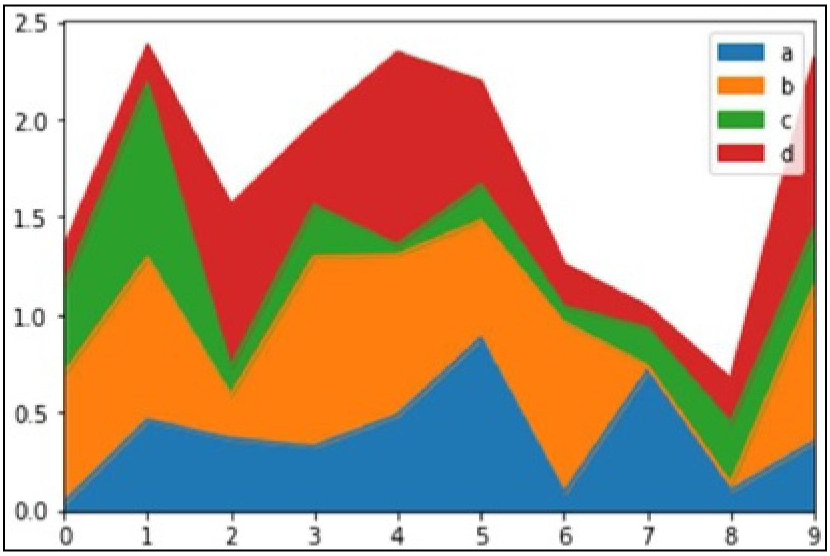
Area Plot

Area plot can be created using the **Series.plot.area()** or the **DataFrame.plot.area()** methods.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.rand(10, 4), columns=['a', 'b', 'c',
'd'])
df.plot.area()
```

Its **output** is as follows −

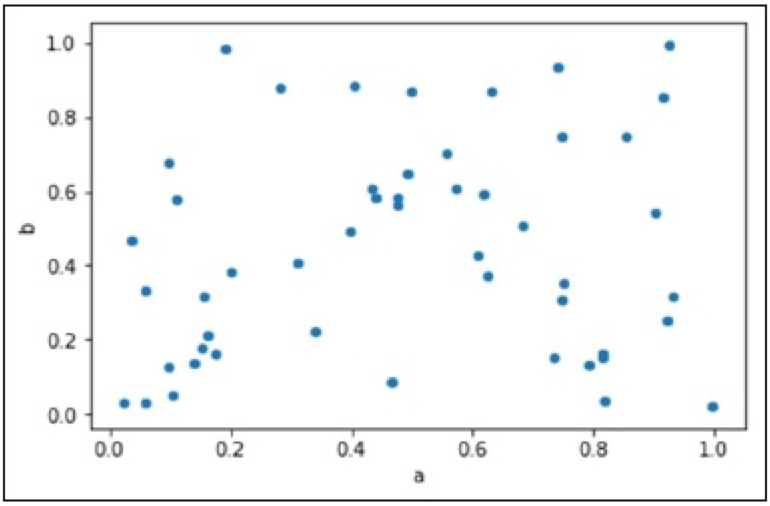


Scatter Plot

Scatter plot can be created using the **DataFrame.plot.scatter()** methods.

```
import pandas as pd
import numpy as np
df = pd.DataFrame(np.random.rand(50, 4), columns=['a', 'b', 'c',
'd'])
df.plot.scatter(x='a', y='b')
```

Its **output** is as follows −



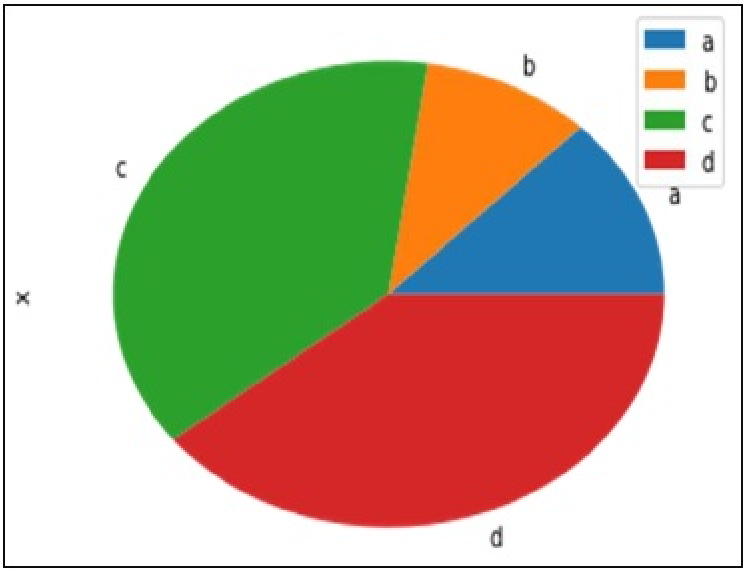
Pie Chart

Pie chart can be created using the **DataFrame.plot.pie()** method.

```
import pandas as pd
import numpy as np

df = pd.DataFrame(3 * np.random.rand(4), index=['a', 'b', 'c',
'd'], columns=['x'])
df.plot.pie(subplots=True)
```

Its **output** is as follows −



Python Pandas - IO Tools

The **Pandas I/O API** is a set of top level reader functions accessed like **pd.read_csv()** that generally return a Pandas object.

The two workhorse functions for reading text files (or the flat files) are **read_csv()** and **read_table()**. They both use the same parsing code to intelligently convert tabular data into a **DataFrame** object −

```
pandas.read_csv(filepath_or_buffer, sep=',', delimiter=None,
header='infer',
names=None, index_col=None, usecols=None
pandas.read_csv(filepath_or_buffer, sep='\t', delimiter=None,
header='infer',
names=None, index_col=None, usecols=None
```

Here is how the **csv** file data looks like −

```
S.No,Name,Age,City,Salary
1,Tom,28,Toronto,20000
2,Lee,32,HongKong,3000
3,Steven,43,Bay Area,8300
4,Ram,38,Hyderabad,3900
```

Save this data as **temp.csv** and conduct operations on it.


```
S.No,Name,Age,City,Salary
1,Tom,28,Toronto,20000
2,Lee,32,HongKong,3000
3,Steven,43,Bay Area,8300
4,Ram,38,Hyderabad,3900
```

Save this data as **temp.csv** and conduct operations on it.

read.csv

read.csv reads data from the csv files and creates a DataFrame object.

```
import pandas as pd

df=pd.read_csv("temp.csv")
print df
```

Its **output** is as follows −

```
   S.No     Name   Age       City   Salary
0     1      Tom    28    Toronto    20000
1     2      Lee    32   HongKong     3000
2     3   Steven    43   Bay Area     8300
3     4      Ram    38  Hyderabad     3900
```

custom index

This specifies a column in the csv file to customize the index using **index_col.**

```
import pandas as pd

df=pd.read_csv("temp.csv",index_col=['S.No'])
print df
```

Its **output** is as follows −

```
S.No    Name    Age       City    Salary
1        Tom     28    Toronto     20000
```

```
2        Lee     32   HongKong      3000
3     Steven     43   Bay Area      8300
4        Ram     38  Hyderabad      3900
```

Converters

dtype of the columns can be passed as a dict.

```
import pandas as pd

df = pd.read_csv("temp.csv", dtype={'Salary': np.float64})
print df.dtypes
```

Its **output** is as follows −

```
S.No        int64
Name       object
Age         int64
City       object
Salary    float64
dtype: object
```

By default, the **dtype** of the Salary column is **int**, but the result shows it as **float** because we have explicitly casted the type.

Thus, the data looks like float −

```
  S.No    Name   Age       City    Salary
0   1      Tom   28     Toronto   20000.0
1   2      Lee   32    HongKong    3000.0
2   3   Steven   43    Bay Area    8300.0
3   4      Ram   38   Hyderabad    3900.0
```

header_names

Specify the names of the header using the names argument.

```
import pandas as pd

df=pd.read_csv("temp.csv", names=['a', 'b', 'c','d','e'])
print df
```

Its **output** is as follows −

```
       a        b     c           d        e
0   S.No     Name   Age        City   Salary
1      1      Tom   28      Toronto    20000
2      2      Lee   32     HongKong     3000
3      3   Steven   43     Bay Area     8300
4      4      Ram   38    Hyderabad     3900
```

Observe, the header names are appended with the custom names, but the header in the file has not been eliminated. Now, we use the header argument to remove that.

If the header is in a row other than the first, pass the row number to header. This will skip the preceding rows.

```
import pandas as pd
```

```
df=pd.read_csv("temp.csv",names=['a','b','c','d','e'],header=0)
print df
```

Its **output** is as follows −

```
      a         b    c          d        e
0  S.No      Name  Age       City   Salary
1     1       Tom   28    Toronto    20000
2     2       Lee   32   HongKong     3000
3     3    Steven   43   Bay Area     8300
4     4       Ram   38  Hyderabad     3900
```

skiprows

skiprows skips the number of rows specified.

```
import pandas as pd

df=pd.read_csv("temp.csv", skiprows=2)
print df
```

Its **output** is as follows −

```
    2      Lee   32    HongKong   3000
0   3   Steven   43    Bay Area   8300
1   4      Ram   38   Hyderabad   3900
```

Python Pandas - Sparse Data

Sparse objects are “compressed” when any data matching a specific value (NaN / missing value, though any value can be chosen) is omitted. A special SparseIndex object tracks where data has been “sparsified”. This will make much more sense in an example. All of the standard Pandas data structures apply the **to_sparse** method −

```
import pandas as pd
import numpy as np

ts = pd.Series(np.random.randn(10))
ts[2:-2] = np.nan
sts = ts.to_sparse()
print sts
```

Its **output** is as follows −

```
0   -0.810497
1   -1.419954
2         NaN
3         NaN
4         NaN
5         NaN
6         NaN
```

```
7          NaN
8     0.439240
9    -1.095910
dtype: float64
BlockIndex
Block locations: array([0, 8], dtype=int32)
Block lengths: array([2, 2], dtype=int32)
```

The sparse objects exist for memory efficiency reasons.

Let us now assume you had a large NA DataFrame and execute the following code −

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(10000, 4))
df.ix[:9998] = np.nan
sdf = df.to_sparse()

print sdf.density
```

Its **output** is as follows −

```
0.0001
```

Any sparse object can be converted back to the standard dense form by calling **to_dense** −

```
import pandas as pd
import numpy as np
ts = pd.Series(np.random.randn(10))
ts[2:-2] = np.nan
sts = ts.to_sparse()
print sts.to_dense()
```

Its **output** is as follows −

```
0   -0.810497
1   -1.419954
2          NaN
3          NaN
4          NaN
5          NaN
6          NaN
7          NaN
8     0.439240
9    -1.095910
dtype: float64
```

Sparse Dtypes

Sparse data should have the same dtype as its dense representation. Currently, **float64**, **int64** and **booldtypes** are supported. Depending on the original **dtype, fill_value default** changes −

- **float64** − np.nan

- **int64** − 0
- **bool** − False

Let us execute the following code to understand the same −

```
import pandas as pd
import numpy as np

s = pd.Series([1, np.nan, np.nan])
print s

s.to_sparse()
print s
```

Its **output** is as follows −

```
0    1.0
1    NaN
2    NaN
dtype: float64

0    1.0
1    NaN
2    NaN
dtype: float64
```

Python Pandas - Caveats & Gotchas

Caveats means warning and gotcha means an unseen problem.

Using If/Truth Statement with Pandas

Pandas follows the numpy convention of raising an error when you try to convert something to a **bool**. This happens in an **if** or **when** using the Boolean operations, and, **or**, or **not**. It is not clear what the result should be. Should it be True because it is not zerolength? False because there are False values? It is unclear, so instead, Pandas raises a **ValueError** −

```
import pandas as pd

if pd.Series([False, True, False]):
   print 'I am True'
```

Its **output** is as follows −

```
ValueError: The truth value of a Series is ambiguous.
Use a.empty, a.bool() a.item(),a.any() or a.all().
```

In **if** condition, it is unclear what to do with it. The error is suggestive of whether to use a **None** or **any of those**.

```
import pandas as pd

if pd.Series([False, True, False]).any():
   print("I am any")
```

Its **output** is as follows −

```
I am any
```

To evaluate single-element pandas objects in a Boolean context, use the method **.bool()** −

```
import pandas as pd

print pd.Series([True]).bool()
```

Its **output** is as follows −

```
True
```

Bitwise Boolean

Bitwise Boolean operators like == and **!**= will return a Boolean series, which is almost always what is required anyways.

```
import pandas as pd

s = pd.Series(range(5))
print s==4
```

Its **output** is as follows −

```
0 False
1 False
2 False
3 False
4 True
dtype: bool
```

isin Operation

This returns a Boolean series showing whether each element in the Series is exactly contained in the passed sequence of values.

```
import pandas as pd

s = pd.Series(list('abc'))
s = s.isin(['a', 'c', 'e'])
print s
```

Its **output** is as follows −

```
0 True
1 False
2 True
dtype: bool
```

Reindexing vs ix Gotcha

Many users will find themselves using the **ix indexing capabilities** as a concise means of selecting data from a Pandas object −

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(6, 4), columns=['one', 'two',
'three',
'four'],index=list('abcdef'))

print df
print df.ix[['b', 'c', 'e']]
```

Its **output** is as follows −

```
         one        two      three       four
a  -1.582025   1.335773   0.961417  -1.272084
b   1.461512   0.111372  -0.072225   0.553058
c  -1.240671   0.762185   1.511936  -0.630920
d  -2.380648  -0.029981   0.196489   0.531714
e   1.846746   0.148149   0.275398  -0.244559
f  -1.842662  -0.933195   2.303949   0.677641

         one        two      three       four
b   1.461512   0.111372  -0.072225   0.553058
c  -1.240671   0.762185   1.511936  -0.630920
e   1.846746   0.148149   0.275398  -0.244559
```

This is, of course, completely equivalent in this case to using the **reindex** method −

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(6, 4), columns=['one', 'two',
'three',
'four'],index=list('abcdef'))

print df
print df.reindex(['b', 'c', 'e'])
```

Its **output** is as follows −

```
         one        two      three       four
a   1.639081   1.369838   0.261287  -1.662003
b  -0.173359   0.242447  -0.494384   0.346882
c  -0.106411   0.623568   0.282401  -0.916361
d  -1.078791  -0.612607  -0.897289  -1.146893
e   0.465215   1.552873  -1.841959   0.329404
f   0.966022  -0.190077   1.324247   0.678064

         one        two      three       four
b  -0.173359   0.242447  -0.494384   0.346882
c  -0.106411   0.623568   0.282401  -0.916361
e   0.465215   1.552873  -1.841959   0.329404
```

Some might conclude that **ix** and **reindex** are 100% equivalent based on this. This is true except in the case of integer indexing. For example, the above operation can alternatively be expressed as −

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(6, 4), columns=['one', 'two',
'three',
'four'],index=list('abcdef'))

print df
print df.ix[[1, 2, 4]]
print df.reindex([1, 2, 4])
```

Its **output** is as follows −

```
         one        two      three       four
a  -1.015695  -0.553847   1.106235  -0.784460
b  -0.527398  -0.518198  -0.710546  -0.512036
c  -0.842803  -1.050374   0.787146   0.205147
d  -1.238016  -0.749554  -0.547470  -0.029045
e  -0.056788   1.063999  -0.767220   0.212476
f   1.139714   0.036159   0.201912   0.710119

         one        two      three       four
b  -0.527398  -0.518198  -0.710546  -0.512036
c  -0.842803  -1.050374   0.787146   0.205147
e  -0.056788   1.063999  -0.767220   0.212476

    one  two  three  four
1   NaN  NaN    NaN   NaN
2   NaN  NaN    NaN   NaN
4   NaN  NaN    NaN   NaN
```

It is important to remember that **reindex is strict label indexing only**. This can lead to some potentially surprising results in pathological cases where an index contains, say, both integers and strings.

Python Pandas - Comparison with SQL

Since many potential Pandas users have some familiarity with SQL, this page is meant to provide some examples of how various SQL operations can be performed using pandas.

```
import pandas as pd

url = 'https://raw.github.com/pandasdev/
pandas/master/pandas/tests/data/tips.csv'
```

```
tips=pd.read_csv(url)
print tips.head()
```

Its **output** is as follows −

```
    total_bill   tip     sex  smoker  day    time  size
0        16.99  1.01  Female      No  Sun  Dinner     2
1        10.34  1.66    Male      No  Sun  Dinner     3
2        21.01  3.50    Male      No  Sun  Dinner     3
3        23.68  3.31    Male      No  Sun  Dinner     2
4        24.59  3.61  Female      No  Sun  Dinner     4
```

SELECT

In SQL, selection is done using a comma-separated list of columns that you select (or a * to select all columns) −

```
SELECT total_bill, tip, smoker, time
FROM tips
LIMIT 5;
```

With Pandas, column selection is done by passing a list of column names to your DataFrame −

```
tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
```

Let's check the full program −

```
import pandas as pd

url = 'https://raw.github.com/pandasdev/
pandas/master/pandas/tests/data/tips.csv'

tips=pd.read_csv(url)
print tips[['total_bill', 'tip', 'smoker', 'time']].head(5)
```

Its **output** is as follows −

```
   total_bill   tip  smoker    time
0       16.99  1.01      No  Dinner
1       10.34  1.66      No  Dinner
2       21.01  3.50      No  Dinner
3       23.68  3.31      No  Dinner
4       24.59  3.61      No  Dinner
```

Calling the DataFrame without the list of column names will display all columns (akin to SQL's *).

WHERE

Filtering in SQL is done via a WHERE clause.

```
  SELECT * FROM tips WHERE time = 'Dinner' LIMIT 5;
```

DataFrames can be filtered in multiple ways; the most intuitive of which is using Boolean indexing.

```
  tips[tips['time'] == 'Dinner'].head(5)
```

Let's check the full program −

```
import pandas as pd

url = 'https://raw.github.com/pandasdev/
pandas/master/pandas/tests/data/tips.csv'

tips=pd.read_csv(url)
print tips[tips['time'] == 'Dinner'].head(5)
```

Its **output** is as follows −

```
   total_bill   tip      sex  smoker  day    time  size
0       16.99  1.01   Female      No  Sun  Dinner     2
1       10.34  1.66     Male      No  Sun  Dinner     3
2       21.01  3.50     Male      No  Sun  Dinner     3
3       23.68  3.31     Male      No  Sun  Dinner     2
4       24.59  3.61   Female      No  Sun  Dinner     4
```

The above statement passes a Series of True/False objects to the DataFrame, returning all rows with True.

GroupBy

This operation fetches the count of records in each group throughout a dataset. For instance, a query fetching us the number of tips left by sex −

```
SELECT sex, count(*)
FROM tips
GROUP BY sex;
```

The Pandas equivalent would be −

```
tips.groupby('sex').size()
```

Let's check the full program −

```
import pandas as pd

url = 'https://raw.github.com/pandasdev/
pandas/master/pandas/tests/data/tips.csv'

tips=pd.read_csv(url)
print tips.groupby('sex').size()
```

Its **output** is as follows −

```
sex
Female   87
Male    157
dtype: int64
```

Top N rows

SQL returns the **top n rows** using **LIMIT** −

```
SELECT * FROM tips
LIMIT 5 ;
```

The Pandas equivalent would be −

```
tips.head(5)
```

Let’s check the full example −

```
import pandas as pd

url = 'https://raw.github.com/pandas-
dev/pandas/master/pandas/tests/data/tips.csv'

tips=pd.read_csv(url)
tips = tips[['smoker', 'day', 'time']].head(5)
print tips
```

Its **output** is as follows −

```
   smoker   day     time
0      No   Sun   Dinner
1      No   Sun   Dinner
2      No   Sun   Dinner
3      No   Sun   Dinner
4      No   Sun   Dinner
```

These are the few basic operations we compared are, which we learnt, in the previous chapters of the Pandas Library.