Part 1

Series

Data Type Name - Series

There are some differences worth noting between ndarrays and Series objects. First of all, elements in NumPy arrays are accessed by their integer position, starting with zero for the first element. A pandas Series Object is more flexible as you can use define your own labeled index to index and access elements of an array. You can also use letters instead of numbers, or number an array in descending order instead of ascending order. Second, aligning data from different Series and matching labels with Series objects is more efficient than using ndarrays, for example dealing with missing values. If there are no matching labels during alignment, pandas returns NaN (not any number) so that the operation does not fail.

Source: “Learning pandas”, Michael Heyd (Packt Publishing).

Let us explore the same:

In [ ]:
```
import numpy as np
import pandas as pd
```

Creating a Series using Pandas

You could convert a list,numpy array, or dictionary to a Series in the following manner

In [ ]:
```
labels = ['w','x','y','z']
list = [10,20,30,40]
array = np.array([10,20,30,40])
dict = {'w':10,'x':20,'y':30,'z':40}
```

Using Lists

In [ ]:
```
pd.Series(data=list)
```

In [ ]:
```
pd.Series(data=list,index=labels)
```

In [ ]:
```
pd.Series(list,labels)
```

Using NumPy Arrays to create Series

In [ ]:
```
pd.Series(array)
```

In [ ]:
```
pd.Series(array,labels)
```

Using Dictionary to create series

In [ ]:
```
pd.Series(dict)
```

Using an Index

We shall now see how to index in a Series using the following examples of 2 series

In [ ]:
```
sports1 = pd.Series([1,2,3,4],index = ['Cricket', 'Football','Basketball', 'Golf'])
```

In [ ]:
```
sports1
```

In [ ]:
```
sports2 = pd.Series([1,2,5,4],index = ['Cricket', 'Football','Baseball', 'Golf'])
```

In [ ]:
```
sports2
```

In [ ]:
```
sports1['Cricket']
```

Operations are then also done based off of index:

In [ ]:
```
sports1 + sports2
```

Part 2

DataFrames

DataFrames concept in python is similar to that of R programming language. DataFrame is a collection of Series combined together to share the same index positions.

In [ ]:
```
from numpy.random import randn
np.random.seed(1)
```

In [ ]:
```
dataframe = pd.DataFrame(randn(10,5),index='A B C D E F G H I J'.split(),columns='Score1 Score2 Score3 Score4 Score5'.split())
```

In [ ]:
```
dataframe
```

Selection and Indexing

Ways in which we can grab data from a DataFrame

In [ ]:
```
dataframe['Score3']
```

In [ ]:
```
# Pass a list of column names in any order necessary
dataframe[['Score2','Score1']]
```

DataFrame Columns are nothing but a Series each

In [ ]:
```
type(dataframe['Score1'])
```

Adding a new column to the DataFrame

In [ ]:
```
dataframe['Score6'] = dataframe['Score1'] + dataframe['Score2']
```

In [ ]:
```
dataframe
```

Removing Columns from DataFrame

In [ ]:
```
dataframe.drop('Score6',axis=1)              # Use axis=0 for dropping rows an
d axis=1 for dropping columns
```

In [ ]:
```
# column is not dropped unless inplace input is TRUE
dataframe
```

In [ ]:
```
dataframe.drop('Score6',axis=1,inplace=True)
```

In [ ]:
```
dataframe
```

Dropping rows using axis=0

In [ ]:
```
dataframe.drop('A',axis=0)      # Row will also be dropped only if inplace=TRU
E is given as input
```

Selecting Rows

In [ ]:
```
dataframe.loc['F']
```

Or select based off of index position instead of label - use iloc instead of loc function

In [ ]:
```
dataframe.iloc[2]
```

Selecting subset of rows and columns using loc function

In [ ]:
```
dataframe.loc['A','Score1']
```

In [ ]:
```
dataframe.loc[['A','B'],['Score1','Score2']]
```

Conditional Selection

Similar to NumPy, we can make conditional selections using Brackets

In [ ]:
```
dataframe
```

In [ ]:
```
dataframe>0.5
```

In [ ]:
```
dataframe[dataframe>0.5]
```

In [ ]:
```
dataframe[dataframe['Score1']>0.5]
```

In [ ]:
```
dataframe[dataframe['Score1']>0.5]['Score2']
```

In [ ]:
```
dataframe[dataframe['Score1']>0.5][['Score2','Score3']]
```

For multiple conditions you can use | and & with parenthesis

In [ ]:
```
dataframe[(dataframe['Score1']>0.5) & (dataframe['Score2'] > 0)]
```

More Index Details

Some more features of indexing includes

- resetting the index
- setting a different value
- index hierarchy

In [ ]:
```
dataframe
```

In [ ]:
```
# Reset to default index value instead of A to J
dataframe.reset_index()
```

In [ ]:
```
# Setting new index value
newindex = 'IND JP CAN GE IT PL FY IU RT IP'.split()
```

In [ ]:
```
dataframe['Countries'] = newindex
```

In [ ]:
```
dataframe
```

In [ ]:
```
dataframe.set_index('Countries')
```

In [ ]:
```
# Once again, ensure that you input inplace=TRUE
dataframe
```

In [ ]:
```
dataframe.set_index('Countries',inplace=True)
```

In [ ]:
```
dataframe
```

Part 3

Missing Data

Methods to deal with missing data in Pandas

In [ ]:
```
dataframe = pd.DataFrame({'Cricket':[1,2,np.nan,4,6,7,2,np.nan],
                          'Baseball':[5,np.nan,np.nan,5,7,2,4,5],
                          'Tennis':[1,2,3,4,5,6,7,8]})
```

In [ ]:
```
dataframe
```

In [ ]:
```
dataframe.dropna()
```

In [ ]:
```
dataframe.dropna(axis=1)       # Use axis=1 for dropping columns with nan valu
es
```

In [ ]:
```
dataframe.dropna(thresh=2)
```

In [ ]:
```
dataframe.fillna(value=0)
```

In [ ]:
```
dataframe['Baseball'].fillna(value=dataframe['Baseball'].mean())
```

Part 4

Groupby

The groupby method is used to group rows together and perform aggregate functions

In [ ]:
```
# Create dataframe as given below
dat = {'CustID':['1001','1001','1002','1002','1003','1003'],
       'CustName':['UIPat','DatRob','Goog','Chrysler','Ford','GM'],
       'Profitinlakhs':[2005,3245,1245,8765,5463,3547]}
```

In [ ]:
```
dataframe = pd.DataFrame(dat)
```

In [ ]:
```
dataframe
```

We can now use the .groupby() method to group rows together based on a column name. For example let's group based on CustID. This will create a DataFrameGroupBy object:

In [ ]:
```
dataframe.groupby('CustID')
```

This object can be saved as a variable

In [ ]:
```
CustID_grouped = dataframe.groupby("CustID")
```

Now we can aggregate using the variable

In [ ]:
```
CustID_grouped.mean()
```

Or we can call the groupby function for each aggregation

In [ ]:
```
dataframe.groupby('CustID').mean()
```

Some more examples

In [ ]:
```
CustID_grouped.std()
```

In [ ]:
```
CustID_grouped.min()
```

In [ ]:
```
CustID_grouped.max()
```

In [ ]:
```
CustID_grouped.count()
```

In [ ]:
```
CustID_grouped.describe()
```

In [ ]:
```
CustID_grouped.describe().transpose()
```

In [ ]:
```
CustID_grouped.describe().transpose()['1001']
```

Part 5

Merging, Joining, and Concatenating

There are 3 important ways of combining DataFrames together:

- Merging
- Joining
- Concatenating

Example DataFrames

In [ ]:
```
dafa1 = pd.DataFrame({'CustID': ['101', '102', '103', '104'],
                      'Sales': [13456, 45321, 54385, 53212],
                      'Priority': ['CAT0', 'CAT1', 'CAT2', 'CAT3'],
                      'Prime': ['yes', 'no', 'no', 'yes']},
                     index=[0, 1, 2, 3])
```

In [ ]:
```
dafa2 = pd.DataFrame({'CustID': ['101', '103', '104', '105'],
                      'Sales': [13456, 54385, 53212, 4534],
                      'Payback': ['CAT4', 'CAT5', 'CAT6', 'CAT7'],
                      'Imp': ['yes', 'no', 'no', 'no']},
                       index=[4, 5, 6, 7])
```

In [ ]:
```
dafa3 = pd.DataFrame({'CustID': ['101', '104', '105', '106'],
                      'Sales': [13456, 53212, 4534, 3241],
                      'Pol': ['CAT8', 'CAT9', 'CAT10', 'CAT11'],
                      'Level': ['yes', 'no', 'no', 'yes']},
                     index=[8, 9, 10, 11])
```

In [ ]:
```
dafa1
```

In [ ]:
```
dafa2
```

In [ ]:
```
dafa3
```

Concatenation

Concatenation joins DataFrames basically either by rows or colums(axis=0 or 1).

We also need to ensure dimension sizes of dataframes are the same

In [ ]:
```
pd.concat([dafa1,dafa2])# by rows concatenation
```

In [ ]:
```
pd.concat([dafa1,dafa2,dafa3],axis=1)#by column concatenation
```

Example DataFrames

In [ ]:
```
Table1 = pd.DataFrame({'CustID': ['1001', '1002', '1003', '1004'],
                       'Q1': ['101', '102', '103', '104'],
                       'Q2': ['201', '202', '203', '204']})

Table2 = pd.DataFrame({'CustID': ['1001', '1006', '1003', '1004'],
                           'Q3': ['301', '302', '303', '304'],
                           'Q4': ['401', '402', '403', '404']})
```

In [ ]:
```
Table1
```

In [ ]:
```
Table2
```

Merging

Just like SQL tables, merge function in python allows us to merge dataframes

In [ ]:
```
pd.merge(dafa1,dafa2,how='outer',on='CustID')
```

Joining

Join can be used to combine columns of 2 dataframes that have different index values into a signle dataframe

The one difference between merge and join is that, merge uses common columns to combine two dataframes, whereas join uses the row index to join two dataframes

In [ ]:
```
daf3 = pd.DataFrame({'Q1': ['101', '102', '103'],
                     'Q2': ['201', '202', '203']},
                     index=['I0', 'I1', 'I2'])

dafa4 = pd.DataFrame({'Q3': ['301', '302', '303'],
                    'Q4': ['401', '402', '403']},
                     index=['I0', 'I2', 'I3'])
```

In [ ]:
```
Table1.join(Table2)
```

In [ ]:
```
Table1.join(Table2, how='outer')
```

Part 6

Operations

Let us discuss some useful Operations using Pandas

In [ ]:
```
dataframe = pd.DataFrame({'custID':[1,2,3,4],'SaleType':['big','small','mediu
m','big'],'SalesCode':['121','131','141','151']})
dataframe.head()
```

Info on Unique Values

In [ ]:
```
dataframe['SaleType'].unique()
```

In [ ]:
```
dataframe['SaleType'].nunique()
```

In [ ]:
```
dataframe['SaleType'].value_counts()
```

Selecting Data

In [ ]:
```
#Select from DataFrame using criteria from multiple columns
newdataframe = dataframe[(dataframe['custID']!=3) & (dataframe['SaleType']=='b
ig')]
```

In [ ]:
```
newdataframe
```

Applying Functions

In [ ]:
```
def profit(a):
    return a*4
```

In [ ]:
```
dataframe['custID'].apply(profit)
```

In [ ]:
```
dataframe['SaleType'].apply(len)
```

In [ ]:
```
dataframe['custID'].sum()
```

Permanently Removing a Column

In [ ]:
```
del dataframe['custID']
```

In [ ]:
```
dataframe
```

Get column and index names:

In [ ]:
```
dataframe.columns
```

In [ ]:
```
dataframe.index
```

Sorting and Ordering a DataFrame:

In [ ]:
```
dataframe
```

In [ ]:
```
dataframe.sort_values(by='SaleType') #inplace=False by default
```

Find Null Values or Check for Null Values

In [ ]:
```
dataframe.isnull()
```

In [ ]:
```
# Drop rows with NaN Values
dataframe.dropna()
```

Filling in NaN values with something else:

In [ ]:
```
import numpy as np
```

In [ ]:
```
dataframe = pd.DataFrame({'Sale1':[5,np.nan,10,np.nan],
                          'Sale2':[np.nan,121,np.nan,141],
                          'Sale3':['XUI','VYU','NMA','IUY']})
dataframe.head()
```

In [ ]:
```
dataframe.fillna('Not nan')
```

Part 7

Data Input and Output

Reading DataFrames from external sources using pd.read functions

CSV

CSV Input

In [ ]:
```
dataframe = pd.read_csv('pandas-train.csv')
```

CSV Output

In [ ]:
```
dataframe.to_csv('train2.csv',index=False)    #If index=FALSE then csv does no
t store index values
```

Excel

Using Pandas, one can read excel files, however it can only import data. It does not fetch formulae or any formatting/images/macros and having such things in excel files can crash the python function to crash and not execute successfully.

Excel Input

In [ ]:
```
pd.read_excel('pandas-Consumer.xlsx',sheet_name='Data1')
```

Excel Output

In [ ]:
```
dataframe.to_excel('Consumer2.xlsx',sheet_name='Sheet1')
```

End of Pandas Section