Introduction to Pandas

Topics to be covered are:

- Introduction to Pandas objects
- Data Indexing & Selection
- Operating on Data in Pandas
- Handling missing data
- Hierarchical Indexing
- Vectorized String Operations
- Visualization with Matplotlib

Python Pandas - Introduction

Pandas is an open-source Python Library providing high-performance data manipulation and analysis tool using its powerful data structures. The name Pandas is derived from the word Panel Data – an Econometrics from Multidimensional data.

In 2008, developer Wes McKinney started developing pandas when in need of high performance, flexible tool for analysis of data.

Prior to Pandas, Python was majorly used for data munging and preparation. It had very little contribution towards data analysis. Pandas solved this problem. Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — load, prepare, manipulate, model, and analyze.

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

Key Features of Pandas

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

Introduction to Pandas objects

At the very basic level, Pandas objects can be thought of as enhanced versions of NumPy structured arrays in which the rows and columns are identified with labels rather than simple integer indices.

The three fundamental Pandas data structures:

- Series
- DataFrame
- Index.

We will start our code sessions with the standard NumPy and Pandas imports:

In [2]:

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

The Pandas Series Object

A Pandas Series is a one-dimensional array of indexed data.

pd.Series(data, index=index)

where index is an optional argument, and data can be one of many entities.

For example, data can be a list or NumPy array, in which case index defaults to an integer sequence:

In [10]:

```
# Create a numpy array
np_array = np.array([0.25, 0.5, 0.75, 1.0])

# Create a pandas series object
data = pd.Series(np_array)
data
```

Out[10]:

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64
```

Attributes of Series object

index

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

dtype

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

In [43]:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print(data)
print("\n")
# Print the attributes of Series object data
print(data.index)
print("\n")
print(data.dtype)
```

```
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64


RangeIndex(start=0, stop=4, step=1)


float64
```

Explicit indexing

From what we've seen so far, it may look like the Series object is basically interchangeable with a one-dimensional NumPy array. The essential difference is the presence of the index: while the Numpy Array has an implicitly defined integer index used to access the values, the Pandas Series has an explicitly defined index associated with the values.

This explicit index definition gives the Series object additional capabilities. For example, the index need not be an integer, but can consist of values of any desired type. For example, if we wish, we can use strings as an index:

In [6]:

```
# implicit indexing
data = pd.Series([0.25, 0.5, 0.75, 1.0])
print("implicit indexing")
print(data)
print("\n")

# explicit indexing
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
print("explicit indexing")
data
```

```
implicit indexing
0    0.25
1    0.50
2    0.75
3    1.00
dtype: float64


explicit indexing
```

Out[6]:

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

We can even use non-contiguous or non-sequential indices:

In [7]:

```
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=[2, 5, 3, 7])
data
```

Out[7]:

```
2    0.25
5    0.50
3    0.75
7    1.00
dtype: float64
```

Series as specialized dictionary

In this way, you can think of a Pandas Series a bit like a specialization of a Python dictionary. A dictionary is a structure that maps arbitrary keys to a set of arbitrary values, and a Series is a structure which maps typed keys to a set of typed values. This typing is important: just as the type-specific compiled code behind a NumPy array makes it more efficient than a Python list for certain operations, the type information of a Pandas Series makes it much more efficient than Python dictionaries for certain operations.

The Series-as-dictionary analogy can be made even more clear by constructing a Series object directly from a Python dictionary:

In [8]:

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

Out[8]:

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

The Pandas Series Object

The next fundamental structure in Pandas is the DataFrame. Like the Series object discussed in the previous section, the DataFrame can be thought of either as a generalization of a NumPy array, or as a specialization of a Python dictionary. We'll now take a look at each of these perspectives.

In [24]:

```
# create a pandas series object population_dist
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)
population
```

Out[24]:

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

In [28]:

```
# create a pandas data frame simple_df using the above pandas series object population_
dist
simple_df = pd.DataFrame(population)
simple_df
```

Out[28]:

	0
California	38332521
Texas	26448193
New York	19651127
Florida	19552860
Illinois	12882135

In [29]:

```
# create a pandas data frame simple_df using the above pandas series object population_
dist
# name the column as 'Polpulation'
simple_df = pd.DataFrame(population,columns=['Population'])
simple_df
```

Out[29]:

	Population
California	38332521
Texas	26448193
New York	19651127
Florida	19552860
Illinois	12882135

In [31]:

```
# create a pandas series object population_dist
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
population = pd.Series(population_dict)

# print the pandas series object population_dist
population
```

Out[31]:

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64
```

In [33]:

```
# create a pandas series object area_dist
area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
             'Florida': 170312, 'Illinois': 149995}
area = pd.Series(area_dict)

# create a pandas series object area_dist
area
```

Out[33]:

```
California    423967
Texas         695662
New York      141297
Florida       170312
Illinois      149995
dtype: int64
```

Now that we have this along with the population Series from before, we can use a dictionary to construct a single two-dimensional object containing this information:

In [38]:

```
states = pd.DataFrame([population, area],columns=['population','area'])
states
```

Out[38]:

	population	area
0	NaN	NaN
1	NaN	NaN

In [25]:

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
```

Out[25]:

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

Given a two-dimensional array of data, we can create a DataFrame with any specified column and index names. If omitted, an integer index will be used for each:

In [46]:

```
# From a two-dimensional NumPy array

# create a 2-d numpy array
data = np.random.rand(3, 2)

# create a pandas data frame my_df from 2-d numpy array data

my_df = pd.DataFrame(data,
             columns=['foo', 'bar'],
             index=['a', 'b', 'c'])

# print the data frame my_df
my_df
```

Out[46]:

	foo	bar
a	0.888558	0.793189
b	0.368391	0.464119
c	0.390875	0.344930

Attributes of DataFrame object

index

For the row labels, the Index to be used for the resulting frame is Optional Default np.arrange(n) if no index is passed.

columns

For column labels, the optional default syntax is - np.arrange(n). This is only true if no index is passed.

dtype

Data type of each column.

In [42]:

```
states = pd.DataFrame({'population': population,
                       'area': area})
states
print(states)
print("\n")
# Print the attributes of Series object data
print(states.index)
print("\n")
print(states.columns)
print("\n")
print(states.population.dtype)
print("\n")
print(states.area.dtype)
```

```
            population    area
California    38332521  423967
Texas         26448193  695662
New York      19651127  141297
Florida       19552860  170312
Illinois      12882135  149995


Index(['California', 'Texas', 'New York', 'Florida', 'Illinois'], dtype='o
bject')


Index(['population', 'area'], dtype='object')


int64


int64
```

The Pandas Index Object

We have seen here that both the Series and DataFrame objects contain an explicit index that lets you reference and modify data.

This Index object is an interesting structure in itself, and it can be thought of either as an immutable array or as an ordered set (technically a multi-set, as Index objects may contain repeated values). Those views have some interesting consequences in the operations available on Index objects. As a simple example, let's construct an Index from a list of integers:

In [47]:

```
ind = pd.Index([2, 3, 5, 7, 11])
ind
```

Out[47]:

```
Int64Index([2, 3, 5, 7, 11], dtype='int64')
```

Attributes of Pandas Index Object

In [48]:

```
# create a pandas index object
ind = pd.Index([2, 3, 5, 7, 11])
ind

# print the index object
print("size of given index is")
print(ind.size)
print("\n")
print("shape of given index is")
print(ind.shape)
print("\n")
print("No of dimensions of given index is")
print(ind.ndim)
print("\n")
print("datatype of given index is")
print(ind.dtype)
```

```
size of given index is
5


shape of given index is
(5,)


No of dimensions of given index is
1


datatype of given index is
int64
```

In [49]:

```
# index objects are immutable
ind[1] = 0
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-49-abacefeb0579> in <module>()
      1 # index objects are immutable
----> 2 ind[1] = 0

~\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in __setitem__(s
elf, key, value)
   2063
   2064     def __setitem__(self, key, value):
-> 2065         raise TypeError("Index does not support mutable operation
s")
   2066
   2067     def __getitem__(self, key):

TypeError: Index does not support mutable operations
```

Pandas objects are designed to facilitate operations such as joins across datasets, which depend on many aspects of set arithmetic. The Index object follows many of the conventions used by Python's built-in set data structure, so that unions, intersections, differences, and other combinations can be computed in a familiar way:

In [50]:

```
indA = pd.Index([1, 3, 5, 7, 9])
indB = pd.Index([2, 3, 5, 7, 11])
```

In [51]:

```
indA & indB  # intersection
```

Out[51]:

```
Int64Index([3, 5, 7], dtype='int64')
```

In [52]:

```
indA | indB  # union
```

Out[52]:

```
Int64Index([1, 2, 3, 5, 7, 9, 11], dtype='int64')
```

Data Indexing & Selection

In the previous module, we looked in detail at methods and tools to access, set, and modify values in NumPy arrays. These included

indexing (e.g., arr[2, 1]),

slicing (e.g., arr[:, 1:5]),

masking (e.g., arr[arr > 0]),

fancy indexing (e.g., arr[0, [1, 5]]),

and combinations thereof (e.g., arr[:, [1, 5]]).

Here we'll look at similar means of accessing and modifying values in Pandas Series and DataFrame objects. If you have used the NumPy patterns, the corresponding patterns in Pandas will feel very familiar, though there are a few quirks to be aware of.

We'll start with the simple case of the one-dimensional Series object, and then move on to the more complicated two-dimesnional DataFrame object.

Data Selection in Series

As we saw in the previous section, a Series object acts in many ways like a one-dimensional NumPy array, and in many ways like a standard Python dictionary. If we keep these two overlapping analogies in mind, it will help us to understand the patterns of data indexing and selection in these arrays.

Series as dictionary

Like a dictionary, the Series object provides a mapping from a collection of keys to a collection of values:

In [53]:

```
import pandas as pd
data = pd.Series([0.25, 0.5, 0.75, 1.0],
                 index=['a', 'b', 'c', 'd'])
data
```

Out[53]:

```
a    0.25
b    0.50
c    0.75
d    1.00
dtype: float64
```

In [54]:

```
data['b']
```

Out[54]:

```
0.5
```

In [56]:

```
data[0]
```

Out[56]:

```
0.25
```

Series objects can even be modified with a dictionary-like syntax. Just as you can extend a dictionary by assigning to a new key, you can extend a Series by assigning to a new index value:

In [58]:

```
# adding new value to a Series object
data['e'] = 1.25
data
```

Out[58]:

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64
```

Series as one-dimensional array

A Series builds on this dictionary-like interface and provides array-style item selection via the same basic mechanisms as NumPy arrays – that is, slices, masking, and fancy indexing. Examples of these are as follows:

In [62]:

```
# print data
print(data)
print("\n")

# slicing by explicit index
print(data['a':'c'])
```

```
a    0.25
b    0.50
c    0.75
d    1.00
e    1.25
dtype: float64


a    0.25
b    0.50
c    0.75
dtype: float64
```

In [63]:

```
# masking
print(data[(data > 0.3) & (data < 0.8)])
```

```
b    0.50
c    0.75
dtype: float64
```

In [64]:

```
# fancy indexing
data[['a', 'e']]
```

Out[64]:

```
a    0.25
e    1.25
dtype: float64
```

Among these, slicing may be the source of the most confusion. Notice that when slicing with an explicit index (i.e., data['a':'c']), the final index is included in the slice, while when slicing with an implicit index (i.e., data[0:2]), the final index is excluded from the slice.

Indexers: loc, iloc, and ix

These slicing and indexing conventions can be a source of confusion. For example, if your Series has an explicit integer index, an indexing operation such as data[1] will use the explicit indices, while a slicing operation like data[1:3] will use the implicit Python-style index.

In [65]:

```
data = pd.Series(['a', 'b', 'c'], index=[1, 2, 3])
data
```

Out[65]:

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

In [66]:

```
# explicit index when indexing
data[1]
```

Out[66]:

```
'a'
```

In [67]:

```
# explicit index when indexing
data[2]
```

Out[67]:

```
'b'
```

In [71]:

```
# implicit index when slicing
data[1:3]
```

Out[71]:

```
2    b
3    c
dtype: object
```

Because of this potential confusion in the case of integer indexes, Pandas provides some special indexer attributes that explicitly expose certain indexing schemes. These are not functional methods, but attributes that expose a particular slicing interface to the data in the Series.

First, the **loc** attribute allows indexing and slicing that always references the **explicit index**:

In [72]:

```
# print data
data
```

Out[72]:

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

In [73]:

```
# loc references explicit index in indexing
data.loc[1]
```

Out[73]:

```
'a'
```

In [74]:

```
# loc references explicit index in indexing
data.loc[2]
```

Out[74]:

```
'b'
```

In [75]:

```
# loc references explicit index in slicing also
data.loc[1:3]
```

Out[75]:

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

The **iloc** attribute allows indexing and slicing that always references the **implicit Python-style index**:

In [78]:

```
# Print data
data
```

Out[78]:

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

In [76]:

```
# iloc references implicit index in indexing
data.iloc[1]
```

Out[76]:

```
'b'
```

In [77]:

```
# iloc references implicit index in indexing
data.iloc[2]
```

Out[77]:

```
'c'
```

In [79]:

```
# iloc references implicit index in slicing
data.iloc[1:3]
```

Out[79]:

```
2    b
3    c
dtype: object
```