Data Handling using Pandas

At the end of this chapter, you will be able to

- Create and use various data structures in Pandas
- Perform various operations on Pandas data structures
- Import/Export data from/to different sources

What is Pandas?

- PANDAS - PANel DAta
- Built on top of NumPy
- Used for data analysis and pre-processing
- Two Data Structures
  1. Series
  2. DataFrame
- Pandas Data structures can have different data types
- Preferred when the data is in tabular format
- Supports integration with different types of data sorces such as excel files, csv files, sql tables, json files etc.

Installation

- Installing Pandas fron PyPI via pip on commannd line

  pip install pandas

- Installing Pandas with Anaconda

  conda install pandas

Loading Pandas

- Need to import the pandas to load and use it

In [1]:
```
# Import pandas
import pandas as pd
```

Pandas Data Structures

1. Series

2. DataFrame

Pandas Series

- One-dimensional array-like object contains a sequence of data values of any type.
- Index - Label associated with a value in Series. (Index starts with 0)
- Example series: Employees

```
Index     Value

  0        Ram
  1        Sam
  2        Raj
  3        Rani
```

Creation of a Series

- pandas.Series(data): Returns a pandas Series object with values specified by given iterable or scalar value.

Pandas Series from a list

In [2]:
```
sl1 = pd.Series([10, 20, 30, 40])
print(sl1)
print(type(sl1))
```

```
0    10
1    20
2    30
3    40
dtype: int64
<class 'pandas.core.series.Series'>
```

In [3]:
```
sl2 = pd.Series(["Raju", 45, 2345.78])
print(sl2)
print(type(sl2))
```

```
0       Raju
1         45
2    2345.78
dtype: object
<class 'pandas.core.series.Series'>
```

Pandas Series from a Tuple

In [4]:
```
st1 = pd.Series((100.0, 200.0, 300.0))
print(st1)
print(type(st1))
```

```
0    100.0
1    200.0
2    300.0
dtype: float64
<class 'pandas.core.series.Series'>
```

Pandas Series from a dictionary

In [5]:
```
sd1 = pd.Series({"name":"ramu", "age":41, "Dept":"Inter"})
print(sd1)
```

```
name     ramu
age        41
Dept    Inter
dtype: object
```

Pandas Series from a ndarray

In [6]:
```
import numpy as np
arr1 = np.arange(1, 10, 2)
print(arr1)
print(type(arr1))
print()
sn1 = pd.Series(arr1)
print(sn1)
print(type(sn1))
```

```
[1 3 5 7 9]
<class 'numpy.ndarray'>

0    1
1    3
2    5
3    7
4    9
dtype: int32
<class 'pandas.core.series.Series'>
```

In [7]:
```
ser1 = pd.Series(range(1, 11))
print(ser1)
```

```
0     1
1     2
2     3
3     4
4     5
5     6
6     7
7     8
8     9
9    10
dtype: int64
```

Accessing Series Elements

1. Indexing: To access individial elements

2. Slicing: To access a part of the Series

1. Indexing

Two Types of Indexing

1. Positional Indexing: Values can be accessed by their position. (Starts from 0)

2. Labelled Indexing: Values can be accessed using any user-defined labels.

In [8]:
```
# Positional Indexing
si1 = pd.Series([5, 10, 15, 20])
print(si1)
print(si1[0])
si1[0] = 0
print(si1)
print(si1[1])
```

```
0     5
1    10
2    15
3    20
dtype: int64
5
0     0
1    10
2    15
3    20
dtype: int64
10
```

In [9]:
```
# Labelled Indexing (We can assign labels as a list with index attribute)
si2 = pd.Series([2000, 4500, 6700], index = ["Ramu", "Raju", "Rani"])
print(si2)
print(si2["Raju"])
print(si2[1])
si2["Rani"] = 10000
print(si2["Rani"])
```

```
Ramu    2000
Raju    4500
Rani    6700
dtype: int64
4500
4500
10000
```

2. Slicing

- Usage: pandas.Series[start_index:end_index:step]
- -> Default start_index is 0
- -> Default step is 1
- -> end_index is excluded

In [10]:
```
data1 = pd.Series(range(10, 101, 10))
print(data1)
```

```
0     10
1     20
2     30
3     40
4     50
5     60
6     70
7     80
8     90
9    100
dtype: int64
```

In [11]:
```
print(data1[:4]) # start index is 0, step is 1 and end_index is 4 (excluded)

                 # indexes -> 0, 1, 2, 3
```

```
0    10
1    20
2    30
3    40
dtype: int64
```

In [12]:
```
print(data1[: : 2]) # step 2 -> 0, 2, 4, 6, 8
```

```
0    10
2    30
4    50
6    70
8    90
dtype: int64
```

In [13]:
```
print(data1[1:3]) # start_index = 1, end_index = 3(excluded, step = 1 -> indexes = 1, 2
```

```
1    20
2    30
dtype: int64
```

In [14]:
```
data2 = pd.Series([1000, 2000, 3000, 4000, 5000])
print(data2)
```

```
0    1000
1    2000
2    3000
3    4000
4    5000
dtype: int64
```

In [15]:
```
data21 = data2[2:4] # start_index = 2, end_index = 4 (excluded), step = 1 -> 2, 3
print(data21)
```

```
2    3000
3    4000
dtype: int64
```

In [16]:
```
data22 = data2[: : -1] # indexes = 4, 3, 2, 1, 0
print(data22)
```

```
4    5000
3    4000
2    3000
1    2000
0    1000
dtype: int64
```

Series Attributes

1. values: Array of all values in the series

2. index: Object with all indexes as its values

3. dtype: Data type of the Series's elements

4. name: Name of the Series object

5. size: Number of values in the Series object

6. empty: True if Series is empty

In [17]:
```
sa = pd.Series([100, 200, 300, 400, 500])
print(sa)
```

```
0    100
1    200
2    300
3    400
4    500
dtype: int64
```

In [18]:
```
print(sa.values)
```

```
[100 200 300 400 500]
```

In [19]:
```
print(sa.index)
sa.index = ['a', 'b', 'c', 'd', 'e']
print(sa)
print(sa.index)
```

```
RangeIndex(start=0, stop=5, step=1)
a    100
b    200
c    300
d    400
e    500
dtype: int64
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

In [20]:
```
print(sa.dtype)
```

```
int64
```

In [21]:
```
print(sa.name)
sa.name = 'My values'
print(sa.name)
print(sa)
```

```
None
My values
a    100
b    200
c    300
d    400
e    500
Name: My values, dtype: int64
```

In [22]:
```
print(sa.size)
```

```
5
```

In [23]:
```
print(sa.empty)

sempty = pd.Series()
print(sempty.empty)
```

```
False
True

<ipython-input-23-af451130acea>:3: DeprecationWarning: The default dtype for empty Series will be 'object' instead of 'float64'
in a future version. Specify a dtype explicitly to silence this warning.
  sempty = pd.Series()
```

Series Methods

head() and tail() Methods

Series.head(number_of_elements): Displays specified number of Series elements from the first.

Series.tail(number_of_elements): Displays specified number of Series elements from the last.

Note: The default number of elements to be displayed is 5

In [24]:
```
ht = pd.Series(range(10, 1001, 10))
print(ht)
```

```
0       10
1       20
2       30
3       40
4       50
      ...
95     960
96     970
97     980
98     990
99    1000
Length: 100, dtype: int64
```

In [25]:
```
print("First five elements are: ")

print(ht.head())
```

```
First five elements are:
0    10
1    20
2    30
3    40
4    50
dtype: int64
```

In [26]:
```
print("First 6 elements are: ")
print(ht.head(6))
```

```
First 6 elements are:
0    10
1    20
2    30
3    40
4    50
5    60
dtype: int64
```

In [27]:
```
print("Last 5 elements are: ")
print(ht.tail())
```

```
Last 5 elements are:
95     960
96     970
97     980
98     990
99    1000
dtype: int64
```

In [28]:
```
print("Last 10 elements are: ")
print(ht.tail(10))
```

```
Last 10 elements are:
90     910
91     920
92     930
93     940
94     950
95     960
96     970
97     980
98     990
99    1000
dtype: int64
```

Mathematical Operations on Series objects

- All basic operations(addition, subtraction, multiplication, division etc.) can be performed.
- Operations are performed on index matiching and missing values are filled with NaN by default.

In [29]:
```
# Create two series: s1 from a list [10, 20, -30] with indices ['a', 'b', 'c'] & s1 from a list [40, -10, 60] with indices ['k',
s1 = pd.Series([10, 20, -30], index = ["a", "b", "c"])
print(s1)

s2 = pd.Series([40, -10, 60], index = ["k", "c", "a"])
print(s2)
```

```
a    10
b    20
c   -30
dtype: int64
k    40
c   -10
a    60
dtype: int64
```

Label	s1	s2	s1 + s2	s1 – s2	s1 * s2	s1 / s2
a	10	60	70	-50	600	0.16
b	20	NaN	NaN	NaN	NaN	NaN
c	-30	-10	-40	-20	300	3
k	NaN	40	NaN	NaN	NaN	NaN

In [30]:
```
print("Addition: ")
s3 = s1 + s2
print(s3)
```

```
Addition:
a    70.0
b     NaN
c   -40.0
k     NaN
dtype: float64
```

In [31]:
```
print("Subtraction: ")
s4 = s1 - s2
print(s4)
```

```
Subtraction:
a   -50.0
b     NaN
c   -20.0
k     NaN
dtype: float64
```

In [32]:
```
print("Multiplication: ")
s5 = s1 * s2
print(s5)
```

```
Multiplication:
a    600.0
b      NaN
c    300.0
k      NaN
dtype: float64
```

In [33]:
```
print("Division: ")
s6 = s1 / s2
print(s6)
```

```
Division:
a    0.166667
b         NaN
c    3.000000
k         NaN
dtype: float64
```

Methods for Mathematical Operations

new_Series = SeriesA.add(SeriesB, fill_value = value)

new_Series = SeriesA.sub(SeriesB, fill_value = value)

new_Series = SeriesA.mul(SeriesB, fill_value = value)

new_Series = SeriesA.div(SeriesB, fill_value = value)

- Note: fill_value is optional argument to fill NaNs with required value

In [34]:
```
sa1 = s1.add(s2, fill_value = 0)
print(sa1)
```

```
a    70.0
b    20.0
c   -40.0
k    40.0
dtype: float64
```

In [35]:
```
sa2 = s1.sub(s2, fill_value = 0)
print(sa2)
```

```
a   -50.0
b    20.0
c   -20.0
k   -40.0
dtype: float64
```

In [36]:
```
sa3 = s1.mul(s2, fill_value = 1)
print(sa3)
```

```
a    600.0
b     20.0
c    300.0
k     40.0
dtype: float64
```

In [37]:
```
sa4 = s1.div(s2, fill_value = 0)
print(sa4)
```

```
a    0.166667
b         inf
c    3.000000
k    0.000000
dtype: float64
```

In [ ]:
```
```