**Objective**: ***Pandas*** is an important Python package that is built over NumPy to deal with the relational data in more efficient way. It supports three types of objects: ***Series***, ***DataFrame*** and ***Index***. The series is one-dimensional data while the data frame is for two-dimensional data. This work sheet introduces Pandas and focuses on series object of Pandas. Subsequent work sheets will introduce other factures of the Pandas package.

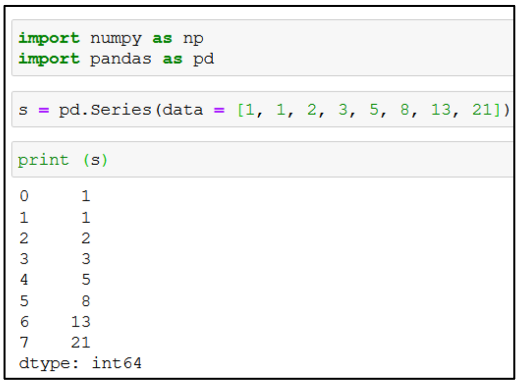
- Several data structures like Lists, Sets, Tuples, Dictionaries and NumPy arrays are already reviewed. NumPy arrays are significantly faster and the package provides several inbuilt functions to perform computations over data.
- To discuss few features of Pandas let us say there is a relational data table as shown below:

| Employee ID | Name | Age | Gender | Title |
|---|---|---|---|---|
| ABC123 | Alice | 29 | Female | Business Analyst |
| PQR987 | Bob | 30 | Male | Lead Engineer |
| XYZ456 | Cathy | - | Female | - |

- Pandas provide a direct way to attach labels (column names or attribute names) to data which is not possible with NumPy. For example, the fourth column can be named as 'gender' when this table is prepared taking the help of Pandas.
- There are few fields in the table with missing entries (shown as -). This is a very common issue in Data Science. When data is collected either the attribute value is not available or it is corrupted. Many times these missing values are represented as *-, ** or *NaN*, *None* etc. Pandas provide few pre-built methods to fill these missing values.
- It may be a requirement to group the data for the title Business Analyst in the table above and report their average age. NumPy does not provide a way to group the data.
- The data table is organized per employee basis. There are other views possible on the same table like the data for all female employees or all lead engineers. In these cases, the same table data will be presented in a differently organized view. This is called Pivoting. Pandas provide a way to explore the data through pivoting.

## Series Objects

- Let us first create a Pandas' series using a Python list data structure using the function ***Series()***. The function takes many arguments. The first one is ***data***. A list of few Fibonacci elements is provided as data.



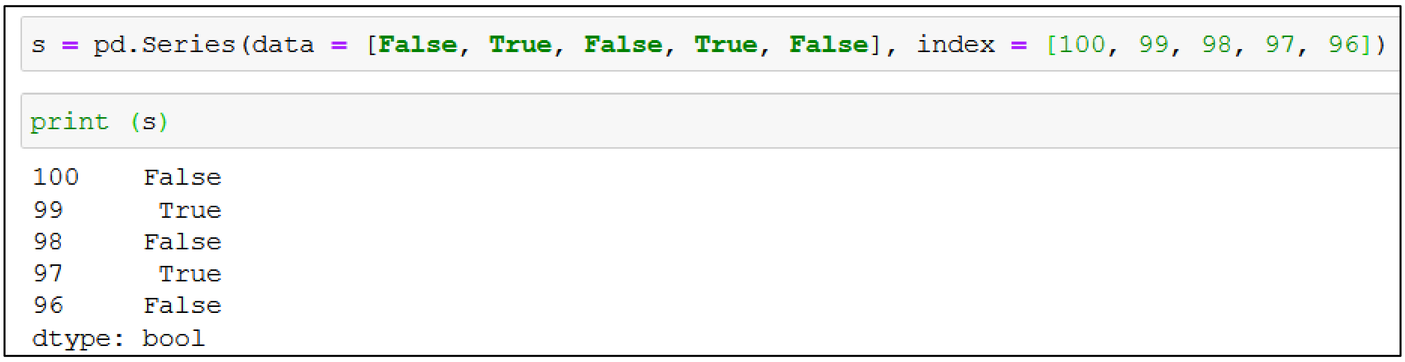
```
import numpy as np
import pandas as pd
```

```
s = pd.Series(data = [1, 1, 2, 3, 5, 8, 13, 21])
```

```
print (s)
```

```
0     1
1     1
2     2
3     3
4     5
5     8
6    13
7    21
dtype: int64
```

- When the series is printed, it prints two columns. The second column is the elements of the list as provided and the first column is ***auto-created indices*** of the list elements from 0 to 7 for 8 elements. Each elements is of type int64.
- Some other user defined indices can also be provided using the second argument ***index*** as shown below:

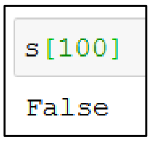


```
s = pd.Series(data = [False, True, False, True, False], index = [100, 99, 98, 97, 96])
```

```
print (s)
```

```
100     False
99       True
98      False
97       True
96      False
dtype: bool
```

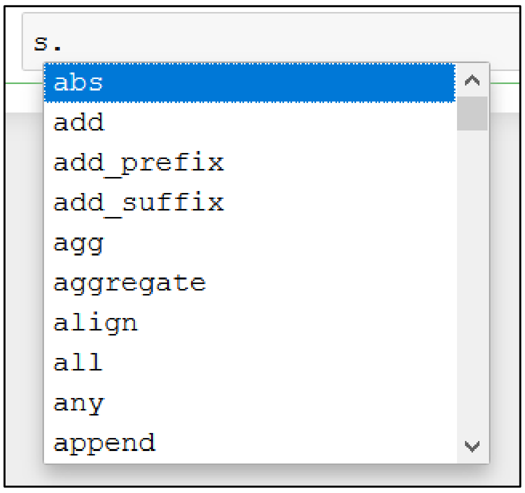
- Notice in the above example that the input argument data is a list of boolean values, index is user defined list of indices which is shown in the first column. Individual values can be accessed using the indices.



```
s[100]
```

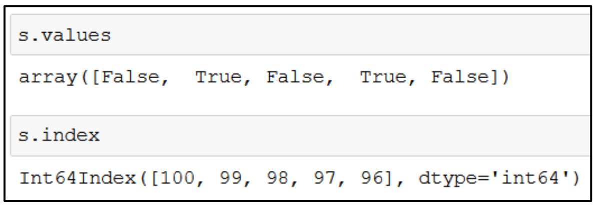
```
False
```

- There are several fields and functions available with the created Series object. It can be seen typing the series name, dot(.) and then pressing the TAB.



```
s.
```

- Two such fields are ***values*** and ***index***.



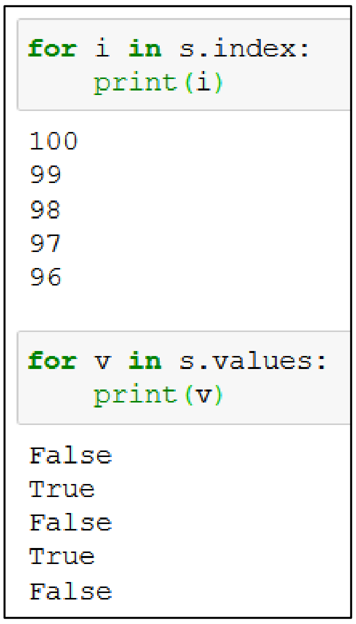
```
s.values
```

```
array([False,  True, False,  True, False])
```

```
s.index
```

```
Int64Index([100, 99, 98, 97, 96], dtype='int64')
```

- Values and index are iterable and can be individually used to print the data and indices as shown below:



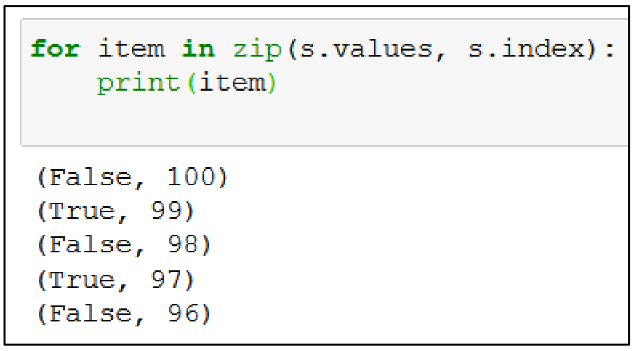
```
for i in s.index:
    print(i)
```

```
100
99
98
97
96
```

```
for v in s.values:
    print(v)
```

```
False
True
False
True
False
```

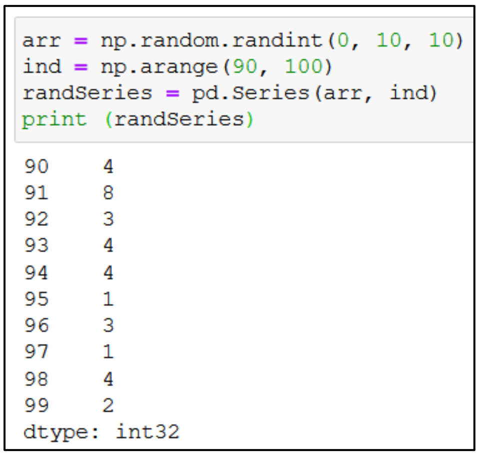
- This is not specific to Pandas but there is a function ***zip ()*** that takes arguments and prepares tuples. For example values and index can be zipped into tuples using this function.



```
for item in zip(s.values, s.index):
    print(item)
```

```
(False, 100)
(True, 99)
(False, 98)
(True, 97)
(False, 96)
```

- The Series object can also be created using NumPy arrays. In the example below 10 random integers are taken as values and their indices are from 90 to 99. Both values and indices are NumPy arrays.



```
arr = np.random.randint(0, 10, 10)
ind = np.arange(90, 100)
randSeries = pd.Series(arr, ind)
print (randSeries)
```

```
90    4
91    8
92    3
93    4
94    4
95    1
96    3
97    1
98    4
99    2
dtype: int32
```

- The Series object can also be created using dictionaries. For example:

```
dict = {}                          # empty dictionary
dict['mass'] = 5.97E24
dict['diamater'] = 12742
dict['day'] = 24
earth = pd.Series(dict)
print(earth)
```

```
mass        5.970000e+24
diamater    1.274200e+04
day         2.400000e+01
dtype: float64
```

- When a Series object is already created using dictionaries, the fields can be filtered providing few indices of interest. For example:

```
newVal = pd.Series(dict, index = ['mass', 'diameter'])
newVal
```

```
mass        5.970000e+24
diameter    1.274200e+04
dtype: float64
```

- Pandas also provide two properties for the objects ***loc*** and ***iloc***. On one side where loc returns the label based index, iloc returns the location based index. Essentially iloc starts counting from 0. The example below elaborates the idea:

```
newSeries = pd.Series(data = [11, 22, 33, 44], index = [1, 2, 3, 4])
print(newSeries.loc[1])
print(newSeries.iloc[1])
```

```
11
22
```

## Exercise:

1. If possible, create a Pandas Series object that looks like the following:

```
mine       104
yours      105
ours       209
dtype: int64
```

2. What is the output of the following?

```
aSeries = pd.Series([3.14, 6.28, 9.42])
for i in aSeries.index:
    print (i)
```

3. Mass of earth is 5.97x10$^{24}$ kg, diameter is 12742 km and the day length is 24 hours. Prepare a Pandas Series object with appropriate index names and values for these three attributes. Ignore units.
4. In the above exercise, if the object name is earth and one of its index for mass is mass, can the mass value will be printed as ***earth ['mass']*** and ***earth.mass***?
5. In the loc and iloc example, the following is being attempted ***print(newSeries.loc[1])***. What will be the output?