Series

A Series is very similar to a NumPy array (in fact it is built on top of the NumPy array object). What differentiates the NumPy array from a Series, is that:

Series is a **one-dimensional labeled array** capable of holding `any data type` (integers, strings, floating-point numbers, Python objects, etc.).

A Series can have axis labels, meaning it can be indexed by a label, instead of just a number location.

Importing the required libraries

pd is the abbreviation for pandas used by the data science community.

```
In [140]: import numpy as np
          import pandas as pd
```

1. Creating a Series

pd.Series()

You can create a series by calling pd.Series()

- **Syntax:** pd.Series(data= None, index=None, dtype= None, name=None, copy=False, fastpath=False)

  `data`: data can be many different things:

  - a Python dict
  - an ndarray
  - a scalar value (like 5)

  `index`: index is a list of axis labels.

1.1 Using List

Note: If no index is given, an index with the values (0,..., len(data) - 1) is generated.

```
In [106]: # Create a pandas series using list
          my_list = [10,20,30]
          pd.Series(data=my_list)

Out[106]: 0    10
          1    20
          2    30
          dtype: int64
```

```
In [107]: # A pandas series with custom index
          my_list = [10,20,30]
          pd.Series(data=my_list, index=[1,2,3])  # index must be the same length as data
          pd.Series(data=my_list,index=labels)

Out[107]: 1    10
          2    20
          3    30
          dtype: int64
```

```
In [152]: # If we do not pass data and index with keyword arguments,
          # the first argument will be data and the second will be index.
          pd.Series(labels,my_list)

Out[152]: 10    a
          20    b
          30    c
          dtype: object
```

1.2 Using NumPy Array

```
In [153]: # Create a pandas series using numpy array
          arr = np.array([10,20,30])

          pd.Series(arr)

Out[153]: 0    10
          1    20
          2    30
          dtype: int32
```

```
In [154]: # Example
          pd.Series(arr,labels)

Out[154]: a    10
          b    20
          c    30
          dtype: int32
```

1.3 Using Dictionary

Note: All the keys in the dictionary will become the indices of the Series object.

```
In [155]: # Create a pandas series using dictionary
          d = {'a':10,'b':20,'c':30}
          pd.Series(d)

Out[155]: a    10
          b    20
          c    30
          dtype: int64
```

- If an index is passed, the values in data corresponding to the labels in the index will be pulled out and the remaining index values will be NaN.
- NaN (not a number) is the standard missing data marker used in pandas.

```
In [156]: pd.Series(d,index=["a","b","c","d","e"])

Out[156]: a    10.0
          b    20.0
          c    30.0
          d     NaN
          e     NaN
          dtype: float64
```

1.4 From scalar value

- If data is a scalar value, an index must be provided. The value will be repeated to match the length of index.

```
In [157]: pd.Series(5.0,index=["a", "b", "c", "d", "e"])

Out[157]: a    5.0
          b    5.0
          c    5.0
          d    5.0
          e    5.0
          dtype: float64
```

A pandas Series can hold a variety of object types. Even functions can be a part of the Series

```
In [158]: pd.Series([sum,print,len])

Out[158]: 0    <built-in function sum>
          1    <built-in function print>
          2    <built-in function len>
          dtype: object
```

2) Selecting Data

2.1 Using Index

The key to using a Series is understanding its index. Pandas makes use of these index names or numbers by allowing for fast look-ups of information (works like a hash table).

```
In [159]: ser1 = pd.Series([1,2,3,4],index = ["USA", "Germany","USSR", "Japan"])
          print(ser1)

          USA        1
          Germany    2
          USSR       3
          Japan      4
          dtype: int64
```

Use indexing notation `[]` for quick and easy access

```
In [160]: # We can use pandas' explicit index (U.I.) as well as our own label to access the value.
          print(ser1["USA"])
          print(ser1[0])

          1
          1
```

```
In [161]: print(ser1[-1])  # negative indexing

          4
```

2.2 Slicing

```
In [162]: ser1 = pd.Series([1,2,3,4],index = ["USA", "Germany","USSR", "Japan"])
          print(ser1)

          USA        1
          Germany    2
          USSR       3
          Japan      4
          dtype: int64
```

```
In [163]: ser1["USA":"USSR"]  # "start:" the start and the stop are included with custom indexing
```

```
Out[163]: USA        1
          Germany    2
          USSR       3
          dtype: int64
```

```
In [164]: ser1[0:2]  # but with default indexing start(included) and stop(excluded)
```

```
Out[164]: USA        1
          Germany    2
          dtype: int64
```

2.3 fancy indexing

```
In [165]: ser1 = pd.Series([1,2,3,4],index = ["USA", "Germany","USSR", "Japan"])
          print(ser1)

          USA        1
          Germany    2
          USSR       3
          Japan      4
          dtype: int64
```

```
In [166]: ser1[0,3] # we can pass multiple indexes in a list
```

```
Out[166]: USA      1
          Japan    4
          dtype: int64
```

```
In [167]: ser1["USA","Japan"]
```

```
Out[167]: USA      1
          Japan    4
          dtype: int64
```

Using Boolean Mask

```
In [168]: # mask
          ser1[ser1%2==0]  # Condition: even values
                           # It generates a series of Boolean values.
```

```
Out[168]: USA       False
          Germany    True
          USSR       False
          Japan      True
          dtype: bool
```

```
In [169]: extract_from_ser1 = ser1[mask]
          print(extract_from_ser1)  # Even values only

          Germany    2
          Japan      4
          dtype: int64
```

3. Assign a value

```
In [170]: ser1 = pd.Series([1,2,3,4],index = ["USA", "Germany","USSR", "Japan"])
          print(ser1)

          USA        1
          Germany    2
          USSR       3
          Japan      4
          dtype: int64
```

```
In [171]: ser1[0]=232
          print(ser1)

          USA        232
          Germany      2
          USSR         3
          Japan        4
          dtype: int64
```

```
In [172]: ser1["Japan"]=345
          print(ser1)

          USA        232
          Germany      2
          USSR         3
          Japan      345
          dtype: int64
```

4. Series operations

You can perform arithmetic operations like addition, subtraction, division and multiplication on two Series objects.

The operations are performed only on the matching indexes.

For all non-matching indexes, NaN (Not a Number) will be returned.

```
In [173]: ser1 = pd.Series([1,2,3,4],index = ["USA", "Germany","USSR", "Japan"])

          ser2 = pd.Series([1,2,3,4],index = ["USA", "Germany","Italy", "Japan"])
```

```
In [174]: # addition of two series
          ser1 + ser2
```

```
Out[174]: Germany   4.0
          Italy     NaN
          Japan     8.0
          USA       2.0
          USSR      NaN
          dtype: float64
```

```
In [175]: ser1*ser2
```

```
Out[175]: Germany    4.0
          Italy      NaN
          Japan     16.0
          USA        1.0
          USSR       NaN
          dtype: float64
```

```
In [176]: # An arithmetic operation on the array with a scalar value

          ser1*5 # This code multiplies each element of the array by 5
```

```
Out[176]: USA       5
          Germany   10
          USSR      15
          Japan     20
          dtype: int64
```

```
In [177]: # Membership operator (checking index)
          "USA" in ser1
```

```
Out[177]: True
```

```
In [178]: # relationship operator
          ser1>2
```

```
Out[178]: USA       False
          Germany   False
          USSR       True
          Japan      True
          dtype: bool
```

```
In [179]: # min()
          ser1.min()
```

```
Out[179]: 1
```

```
In [180]: # max()
          ser1.max()
```

```
Out[180]: 4
```

```
In [181]: # sum()
          ser1.sum() # It will give the sum of all the values.
```

```
Out[181]: 10
```

```
In [182]: # mean()
          ser1.mean()
```

```
Out[182]: 2.5
```

```
In [183]: # count
          ser1.count()
```

```
Out[183]: 4
```

4.1 Series Attributes

```
In [184]: ser1 = pd.Series([1,2,3,4],index = ["USA", "Germany","USSR", "Japan"])

          ser1
```

```
Out[184]: USA       1
          Germany   2
          USSR      3
          Japan     4
          dtype: int64
```

```
In [185]: # size
          ser1.size
```

```
Out[185]: 4
```

```
In [186]: # shape
          ser1.shape
```

```
Out[186]: (4,)
```

```
In [187]: # dtype
          ser1.dtype
```

```
Out[187]: dtype('int64')
```

```
In [188]: # index
          ser1.index
```

```
Out[188]: Index(['USA', 'Germany', 'USSR', 'Japan'], dtype='object')
```

```
In [189]: # values
          ser1.values
```

```
Out[189]: array([1, 2, 3, 4], dtype=int64)
```

4.2 Series methods

```
In [190]: ser = pd.Series([33,22,11,44,22,55,66,77,33,55,22,22,88])
          ser
```

```
Out[190]: 0    33
          1    22
          2    22
          3    11
          4    44
          5    22
          6    55
          7    66
          8    77
          9    33
          10   55
          11   22
          12   22
          13   88
          dtype: int64
```

value_counts()

Return a Series containing counts of unique values.

- **Syntax:** Series.value_counts(normalize= False, sort= True, ascending= False, bins= None, dropna= True)

```
In [191]: ser.value_counts()
```

```
Out[191]: 22    5
          33    2
          55    2
          11    1
          44    1
          66    1
          77    1
          88    1
          dtype: int64
```

sort_values()

Sort by the values.

Sort a Series in ascending or descending order by some criterion.

```
In [192]: ser.sort_values()  # Default is ascending
```

```
Out[192]: 3     11
          1     22
          2     22
          5     22
          11    22
          12    22
          0     33
          8     33
          9     55
          10    55
          4     44
          6     66
          7     77
          13    88
          dtype: int64
```

```
In [193]: ser.sort_values(ascending=False)  # descending to ascending
```

```
Out[193]: 13    88
          7     77
          6     66
          4     55
          10    55
          9     44
          0     33
          8     33
          1     22
          2     22
          5     22
          11    22
          12    22
          3     11
          dtype: int64
```

Let's stop for now and move on to DataFrames, which will expand on the concept of Series!

You can read the documentation for further details: click the link below. https://pandas.pydata.org/docs/reference/api/pandas.Series.html

Great Job!