Topics to be covered

- Introduction
- Pandas Data Structures
- Series CRUD operations
- Series Using List, Dictionary & Ndarray
- Series Indexing
- Series Methods

“pandas is an open source, BSD-licensed library providing highperformance, easy-to-use data structures and data analysis tools for the Python programming language.”
-pandas.pydata.org

What's Pandas for?

- **This tool is essentially your data’s home.**

- **Through pandas, we use Pandas for data cleaning, transforming, and analyzing it.**

With Pandas you do things like

- **Load or open file like Excel,process tabular data, load CSV or JSON files, and more.**

- **Calculate statistics and answer questions about the data, like**

    - **What's the average, median, max, or min of each column?**
    - **Does column A correlate with column B?**
    - **What does the distribution of data in column C look like?**
- **Clean the data by doing things like removing missing values and filtering rows or columns by some criteria**
- **Visualize the data with help from Matplotlib. Plot bars, lines, histograms, bubbles, and more.**
- **Store the cleaned, transformed data back into a CSV, other file or database**

Importing Pandas Package

In [1]:

```
import pandas as pd
```

Check Version of Pandas

In [11]:

```
pd.__version__
```

Out[11]:

```
'0.23.4'
```

Introduction to pandas Data Structures

ONE OF THE KEYS TO UNDERSTANDING PANDAS IS TO UNDERSTAND THE DATA model.

At the core of pandas are three data structures:

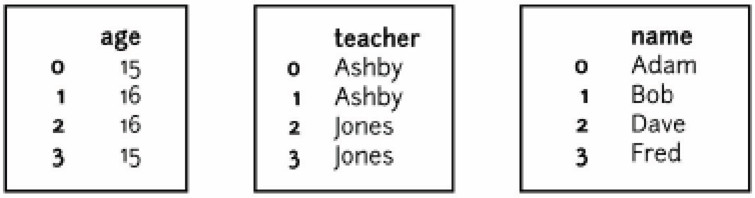
Different dimensions of pandas data structures

Data Structure	Dimensionality	Spreadsheet Analog
`Series`	1D	Column
`DataFrame`	2D	Single Sheet
`Panel`	3D	Multiple Sheets

The most widely used data structures are the Series and the DataFrame that deal with array data and tabular data respectively.

Series

A Series is similar to a single column of data.



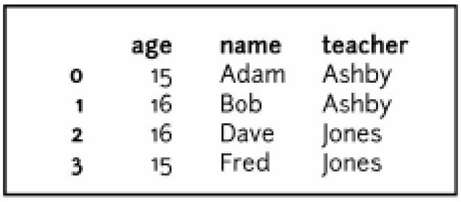
	age
0	15
1	16
2	16
3	15

	teacher
0	Ashby
1	Ashby
2	Jones
3	Jones

	name
0	Adam
1	Bob
2	Dave
3	Fred

DataFrame

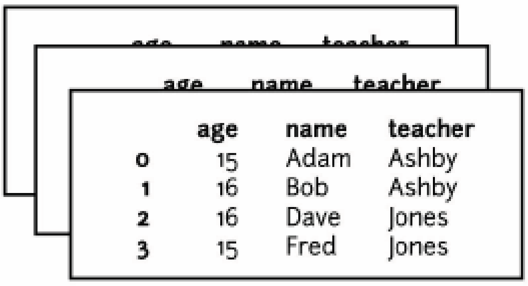
A DataFrame is similar to a sheet with rows and columns.



	age	name	teacher
0	15	Adam	Ashby
1	16	Bob	Ashby
2	16	Dave	Jones
3	15	Fred	Jones

Panel

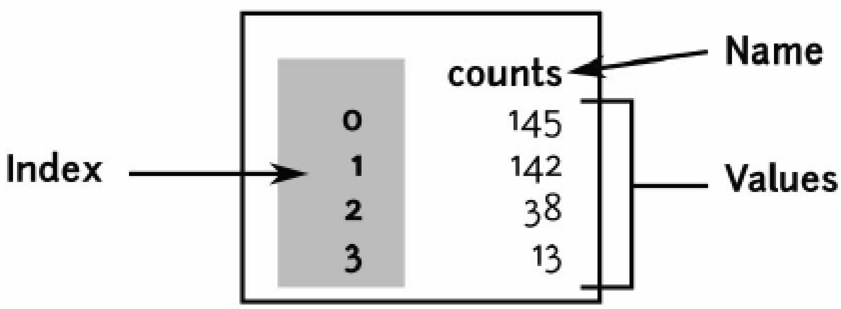
A Panel is a group of sheets



	age	name	teacher
0	15	Adam	Ashby
1	16	Bob	Ashby
2	16	Dave	Jones
3	15	Fred	Jones

Pandas Series

A SERIES IS USED TO MODEL ONE DIMENSIONAL DATA, SIMILAR TO A LIST IN Python.



- **The left most column is the index column which contains entries for the index. It is also known as axis. The value of index - 0,1,2,3.....**
- **The rightmost column is the output contains the values of the series**
- **Counts - is the name of column**

Create a Series object

In [12]:

```
s = pd.Series([145,142,38,13],name='counts')
```

In [7]:

```
s
```

Out[7]:

```
2002    33
2002    34
2003    40
2010    41
2001    36
Name: data, dtype: int64
```

General values of a Series can hold strings, floats, booleans, or arbitrary Python objects.

In [14]:

```
# You can inspect the index of a series, as it is an attribute of the object
s.index
```

Out[14]:

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

In [15]:

```
# String as index
s = pd.Series([145,142,38,13],index=['John','Bob','Rohan','Jolly'],name='counts')
s
```

Out[15]:

```
John     145
Bob      142
Rohan     38
Jolly     13
Name: counts, dtype: int64
```

Note that the dtype that we see when we print a Series is the type of the values, not of the index

In [16]:

```
# You can inspect the index of a series, as it is an object type
s.index
```

Out[16]:

```
Index(['John', 'Bob', 'Rohan', 'Jolly'], dtype='object')
```

Note -

- If you have numeric data, you wouldn't want it to be stored as a Python object, but rather as an int64 or float64, which allow you to do vectorized numeric operations.
- If you have time data and it says it has the object type, you probably have strings for the dates.

The NaN value

When pandas determines that a series holds numeric values, but it cannot find a number to represent an entry, it will use NaN.

This value stands for Not A Number, and is usually ignored in arithmetic operations.

In [17]:

```
# Example

nan_ex = pd.Series([2,None,3,None] )
nan_ex
```

Out[17]:

```
0    2.0
1    NaN
2    3.0
3    NaN
dtype: float64
```

Load Data from csv file into series

In [13]:

```
s = pd.Series.from_csv('stores.csv')
s
```

Out[13]:

```
Chennai         2235.4
Apparel          915.4
Electronincs     600.8
Super Market     719.2
Delhi           1777.7
Apparel          745.0
Electronincs     306.7
Super Market     726.0
Kolkata         1612.4
Apparel            NaN
Electronincs     521.0
Super Market     566.4
Mumbai          1757.6
Apparel          700.7
Electronincs       NaN
Super Market     266.9
Grand Total     7383.1
dtype: float64
```

In [14]:

```
type(s)
```

Out[14]:

```
pandas.core.series.Series
```

In [15]:

```
s.index
```

Out[15]:

```
Index(['Chennai', 'Apparel', 'Electronincs', 'Super Market', 'Delhi',
       'Apparel', 'Electronincs', 'Super Market', 'Kolkata', 'Apparel',
       'Electronincs', 'Super Market', 'Mumbai', 'Apparel', 'Electronincs',
       'Super Market', 'Grand Total'],
      dtype='object')
```

In [ ]:

```

```

NOTE -

One thing to note is that the type of this series is float64, not int64!

This is because the only numeric column that supports NaN is the float column.

In [18]:

```
# Count number of value in series
# pandas ignores NaN

nan_ex.count()
```

Out[18]:

2

Note -

If you load data from a CSV file, an empty value will become NaN.

Series CRUD

THE PANDAS SERIES DATA STRUCTURE PROVIDES SUPPORT FOR THE BASIC CRUD operations—

```
* create
* read
* update
* delete.
```

Creation

In [19]:

```
# Already done
# Example of duplicate index
data=[33,34,40,41,36]
year = [2002,2002,2003,2010,2001]

s =pd.Series(data,index=year,name='data')
s
```

Out[19]:

```
2002    33
2002    34
2003    40
2010    41
2001    36
Name: data, dtype: int64
```

In [20]:

```
s.index
```

Out[20]:

```
Int64Index([2002, 2002, 2003, 2010, 2001], dtype='int64')
```

Reading

To read or select the data from a series, one can simply use an index operation in combination with the index entry

In [21]:

```
# Normally this returns a scalar value
s[2010]
```

Out[21]:

```
41
```

In [22]:

```
# In the case where index entries repeat, the result will be another Series object:
s[2002]
```

Out[22]:

```
2002    33
2002    34
Name: data, dtype: int64
```

In [23]:

```
type(s[2002])
```

Out[23]:

```
pandas.core.series.Series
```

Iterating over a series

- iteration occurs over the values of a series.
- membership is against the index items.

In [24]:

```
for item in s:
    print(item)
```

```
33
34
40
41
36
```

Membership "in" operator

In [25]:

```
# 33 in s gives false because membership is checking against index
33 in s
```

Out[25]:

```
False
```

In [26]:

```
# here 2002 is index
2002 in s
```

Out[26]:

```
True
```

In [27]:

```
# iterate over the tuple both the index and value
for item in s.iteritems():
    print(item)
```

```
(2002, 33)
(2002, 34)
(2003, 40)
(2010, 41)
(2001, 36)
```

Updating

To update a value for a given index label, the standard index assignment operation works and performs the update in-place

In [28]:

```
s
```

Out[28]:

```
2002    33
2002    34
2003    40
2010    41
2001    36
Name: data, dtype: int64
```

In [29]:

```
# Update value at index 2003
s[2003] = 1000
```

In [30]:

```
s
```

Out[30]:

```
2002      33
2002      34
2003    1000
2010      41
2001      36
Name: data, dtype: int64
```

In [31]:

```
# adding new index and value
s[2020] = 20
```

In [32]:

```
s
```

Out[32]:

```
2002      33
2002      34
2003    1000
2010      41
2001      36
2020      20
Name: data, dtype: int64
```

Note - what happens when we try to update an index that has duplicate entries.

In [33]:

```
# update s[2002]
s[2002]=100
```

In [34]:

```
s
```

Out[34]:

```
2002     100
2002     100
2003    1000
2010      41
2001      36
2020      20
Name: data, dtype: int64
```

Deletion

In [35]:

```
del s[2002]
```

```
C:\Anaconda3\lib\site-packages\pandas\core\internals.py:389: FutureWarning:
in the future insert will treat boolean arrays and array-likes as boolean in
dex instead of casting it to integer
  self.values = np.delete(self.values, loc, 0)
C:\Anaconda3\lib\site-packages\pandas\core\internals.py:390: FutureWarning:
in the future insert will treat boolean arrays and array-likes as boolean in
dex instead of casting it to integer
  self.mgr_locs = self.mgr_locs.delete(loc)
C:\Anaconda3\lib\site-packages\pandas\core\indexes\base.py:4353: FutureWarni
ng: in the future insert will treat boolean arrays and array-likes as boolea
n index instead of casting it to integer
  return self._shallow_copy(np.delete(self._data, loc))
```

In [36]:

```
s
```

Out[36]:

```
2003    1000
2010      41
2001      36
2020      20
Name: data, dtype: int64
```

Series Using List, Dictionary & Ndarray

In [2]:

```
# Using List
ser1 = pd.Series([1.5, 2.5, 3, 4.5, 5.0, 6])
ser1
```

Out[2]:

```
0    1.5
1    2.5
2    3.0
3    4.5
4    5.0
5    6.0
dtype: float64
```

In [3]:

```
# Creating Series using dictionary
ser1 = pd.Series({"India": "New Delhi","Japan": "Tokyo","UK": "London"})
ser1
```

Out[3]:

```
India    New Delhi
Japan        Tokyo
UK          London
dtype: object
```

In [6]:

```
# Using ndarray
import numpy as np
ser1 = pd.Series(np.arange(100,300,20.5))
ser1
```

Out[6]:

```
0    100.0
1    120.5
2    141.0
3    161.5
4    182.0
5    202.5
6    223.0
7    243.5
8    264.0
9    284.5
dtype: float64
```

Series Indexing

Just like numpy arrays, a Series object can be both indexed and sliced along the axis.

Indexing pulls out either a scalar or multiple values

In [2]:

```
data=[33,34,40,41,36]
year = [2002,2002,2003,2010,2001]

s =pd.Series(data,index=year,name='data')
```

In [3]:

```
# to check index is unique or not
s.index.is_unique
```

Out[3]:

```
False
```

In [4]:

```
s
```

Out[4]:

```
2002    33
2002    34
2003    40
2010    41
2001    36
Name: data, dtype: int64
```

In [5]:

```
# index by position is not working normaly like in python collection
s[0]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-5-19e14ec38e98> in <module>
      1 # index by position is not working normaly like in python collection
----> 2 s[0]

C:\Anaconda3\lib\site-packages\pandas\core\series.py in __getitem__(self, ke
y)
    765         key = com._apply_if_callable(key, self)
    766         try:
--> 767             result = self.index.get_value(self, key)
    768 
    769             if not is_scalar(result):

C:\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_value(sel
f, series, key)
   3116         try:
   3117             return self._engine.get_value(s, k,
-> 3118                                           tz=getattr(series.dtype,
 'tz', None))
   3119         except KeyError as e1:
   3120             if len(self) > 0 and self.inferred_type in ['integer',
'boolean']:

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine._get_loc_duplicates
()

pandas\_libs\index_class_helper.pxi in pandas._libs.index.Int64Engine._maybe
_get_bool_indexer()

KeyError: 0
```

.iloc and .loc

These two attributes allow label-based and position-based indexing respectively

In [40]:

```
# example
s.iloc[0]
```

Out[40]:

```
33
```

In [41]:

```
s.iloc[-1]
```

Out[41]:

```
36
```

When we perform an index operation on the .iloc attribute, it does lookup based on index position. pandas will raise an IndexError if there is no index at that location

In [42]:

```
s.iloc[9]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-42-745ffdd4e301> in <module>
----> 1 s.iloc[9]

C:\Anaconda3\lib\site-packages\pandas\core\indexing.py in __getitem__(self,
 key)
   1476 
   1477             maybe_callable = com._apply_if_callable(key, self.obj)
-> 1478             return self._getitem_axis(maybe_callable, axis=axis)
   1479 
   1480     def _is_scalar_access(self, key):

C:\Anaconda3\lib\site-packages\pandas\core\indexing.py in _getitem_axis(sel
f, key, axis)
   2100 
   2101             # validate the location
-> 2102             self._validate_integer(key, axis)
   2103 
   2104             return self._get_loc(key, axis=axis)

C:\Anaconda3\lib\site-packages\pandas\core\indexing.py in _validate_integer
(self, key, axis)
   2007         l = len(ax)
   2008         if key >= l or key < -l:
-> 2009             raise IndexError("single positional indexer is out-of-bo
unds")
   2010 
   2011     def _getitem_tuple(self, tup):

IndexError: single positional indexer is out-of-bounds
```

In [43]:

```
# Slicing multiple items  we pass list of index
print(s)
s.iloc[[2,4]]
```

```
2002    33
2002    34
2003    40
2010    41
2001    36
Name: data, dtype: int64
```

Out[43]:

```
2003    40
2001    36
Name: data, dtype: int64
```

In [44]:

```
# Pulling out multiple item in given range of index
s.iloc[2:5]
```

Out[44]:

```
2003    40
2010    41
2001    36
Name: data, dtype: int64
```

.loc is supposed to be based on the index labels and not the positions

In [45]:

```
# Example
s.loc[2001]
```

Out[45]:

```
36
```

In [46]:

```
s.loc[[2002,2003]]
```

Out[46]:

```
2002    33
2002    34
2003    40
Name: data, dtype: int64
```

.at and .iat

The .at and .iat index accessors are analogous to .loc and .iloc. The difference being that they will return a numpy.ndarray when pulling out a duplicate value, whereas .loc and .iloc return a Series

In [47]:

```
s.at[2001]
```

Out[47]:

```
36
```

In [48]:

```
type(s.at[2001])
```

Out[48]:

```
numpy.int64
```

In [49]:

```
s.at[2002]
```

Out[49]:

```
array([33, 34], dtype=int64)
```

In [50]:

```
# More Example on indexing

s.iloc[-3:]
```

Out[50]:

```
2003    40
2010    41
2001    36
Name: data, dtype: int64
```

In [51]:

```
s.iloc[::-2]
```

Out[51]:

```
2001    36
2003    40
2002    33
Name: data, dtype: int64
```

In [52]:

```
s.iloc[::2]
```

Out[52]:

```
2002    33
2003    40
2001    36
Name: data, dtype: int64
```

In [53]:

```
s.iloc[::-1]
```

Out[53]:

```
2001    36
2010    41
2003    40
2002    34
2002    33
Name: data, dtype: int64
```

SLICE	RESULT
`0:1`	First item
`:1`	First item (start default is `0`)
`:-2`	Take from the start up to the second to last item
`::2`	Take from start to the end skipping every other item

Boolean Arrays

A slice using the result of a boolean operation is called a boolean array.

In [54]:

```
s>35
```

Out[54]:

```
2002    False
2002    False
2003     True
2010     True
2001     True
Name: data, dtype: bool
```

In [55]:

```
s[s>35]
```

Out[55]:

```
2003    40
2010    41
2001    36
Name: data, dtype: int64
```

Taking a series and applying an operation to each value of the series is known as broadcasting.

The > operation is broadcasted, or applied, to every entry in the series. And the result is a new series with the result of each of those operations

Multiple boolean operations

Operation	Example
And	ser[a & b]
Or	ser[a | b]
Not	ser[~a]

In [56]:

```
# Example don't forget to write inside paranethesis when you write inline masks
s[(s>35) & (s<41)]
```

Out[56]:

```
2003    40
2001    36
Name: data, dtype: int64
```

Series Methods

A series object has many attributes and methods that are useful for data analysis.

.iteritems method

The .iteritems method returns a sequence of index, value tuples

In [57]:

```
# Lets apply iteration over pandas Series
import numpy as np
s = pd.Series(np.arange(1,10,2))
s
```

Out[57]:

```
0    1
1    3
2    5
3    7
4    9
dtype: int32
```

In [58]:

```
# Iteration over a series iterates over the values
for value in s:
    print(value)
```

```
1
3
5
7
9
```

In [59]:

```
# Iteration over the index and value pairs
for item in s.iteritems():
    print(item)
```

```
(0, 1)
(1, 3)
(2, 5)
(3, 7)
(4, 9)
```

In [60]:

```
# Iteration over the index and value pairs
for idx,value in s.iteritems():
    print(idx,value)
```

```
0 1
1 3
2 5
3 7
4 9
```

.key method

A .keys method is provided as a shortcut to the index as well:

In [61]:

```
#Example-1
for idx in s.keys():
    print(idx,"=>", s[idx])
```

```
0 => 1
1 => 3
2 => 5
3 => 7
4 => 9
```

In [62]:

```
#Example-1=2
s1 = pd.Series([22,34,45,56],index=['Rahul','Bob','Joy','Tom'],name='Age')
for idx in s1.keys():
    print(idx,"=>", s1[idx])
```

```
Rahul => 22
Bob => 34
Joy => 45
Tom => 56
```

Overloaded operations

The table below lists overloaded operations for a Series object.

OPERATION	RESULT
`+`	Adds scalar (or series with matching index values) returns `Series`
`-`	Subtracts scalar (or series with matching index values) returns `Series`
`/`	Divides scalar (or series with matching index values) returns `Series`
`//`	"Floor" Divides scalar (or series with matching index values) returns `Series`
`*`	Multiplies scalar (or series with matching index values) returns `Series`
`%`	Modulus scalar (or series with matching index values) returns `Series`
`==, !=`	Equality scalar (or series with matching index values) returns `Series`
`>, <`	Greater/less than scalar (or series with matching index values) returns `Series`
`>=, <=`	Greater/less than or equal scalar (or series with matching index values) returns `Series`
`^`	Binary XOR returns `Series`
`|`	Binary OR returns `Series`
`&`	Binary AND returns `Series`

In [63]:

```
s = pd.Series([10,20,30,40,50],index=['r1','r2','r3','r4','r5'])
```

In [64]:

```
s
```

Out[64]:

```
r1    10
r2    20
r3    30
r4    40
r5    50
dtype: int64
```

In [65]:

```
s+2
```

Out[65]:

```
r1    12
r2    22
r3    32
r4    42
r5    52
dtype: int64
```

In [66]:

```
s+s['r1']
```

Out[66]:

```
r1    20
r2    30
r3    40
r4    50
r5    60
dtype: int64
```

Addition with two series objects adds only those items whose index occurs in both series, otherwise it inserts a NaN for index values found only in one of the series.

In [8]:

```
# Addition of two series

s1 = pd.Series([12,13,14,15])
s2 = pd.Series([1,2,3,4,5,6])
s1+s2
```

Out[8]:

```
0    13.0
1    15.0
2    17.0
3    19.0
4     NaN
5     NaN
dtype: float64
```

In [68]:

```
# subtraction
s1-s2
```

Out[68]:

```
0    11.0
1    11.0
2    11.0
3    11.0
4     NaN
5     NaN
dtype: float64
```

In [69]:

```
# Mutiplication
s1*s2
```

Out[69]:

```
0    12.0
1    26.0
2    42.0
3    60.0
4     NaN
5     NaN
dtype: float64
```

In [70]:

```
#division
s1/s2
```

Out[70]:

```
0    12.000000
1     6.500000
2     4.666667
3     3.750000
4          NaN
5          NaN
dtype: float64
```

In [71]:

```
s1//s2
```

Out[71]:

```
0    12.0
1     6.0
2     4.0
3     3.0
4     NaN
5     NaN
dtype: float64
```

In [72]:

```
s1//2
```

Out[72]:

```
0    6
1    6
2    7
3    7
dtype: int64
```

In [73]:

```
s1==s1
```

Out[73]:

```
0    True
1    True
2    True
3    True
dtype: bool
```

Reset Index

- The reset_index() function is used to generate a new DataFrame or Series with the index reset.
- This is useful when the index needs to be treated as a column, or when the index is meaningless and needs to be reset to the default before another operation.

```
Series.reset_index(self, level=None, drop=False, name=None, inplace=False)
```

In [11]:

```
# Create a series

s = pd.Series([2, 3, 4, 5], name='f1', index=pd.Index(['p', 'q', 'r', 's'], name='idx'))
s
```

Out[11]:

```
idx
p    2
q    3
r    4
s    5
Name: f1, dtype: int64
```

In [12]:

```
# Generate a DataFrame with default index.
s.reset_index()
```

Out[12]:

	idx	f1
0	p	2
1	q	3
2	r	4
3	s	5

In [76]:

```
# To specify the name of the new column use name.
s.reset_index(name="values")
```

Out[76]:

	idx	values
0	p	2
1	q	3
2	r	4
3	s	5

In [77]:

```
# To generate a new Series with the default set drop to True.
s.reset_index(drop=True)
```

Out[77]:

```
0    2
1    3
2    4
3    5
Name: f1, dtype: int64
```

In [78]:

```
# To update the Series in place,
#without generating a new one set inplace to True.
s.reset_index(inplace=True, drop=True)
s
```

Out[78]:

```
0    2
1    3
2    4
3    5
Name: f1, dtype: int64
```

Counts

the .count method returns the number of non-null items.

In [79]:

```
s = pd.Series([100,23.4,100,None,44],name='value')
s
```

Out[79]:

```
0    100.0
1     23.4
2    100.0
3      NaN
4     44.0
Name: value, dtype: float64
```

In [80]:

```
s.count()
```

Out[80]:

```
4
```

.value_counts

The .value_counts method returns a series indexed by the values found in the series.

In [81]:

```
s.value_counts()
```

Out[81]:

```
100.0    2
23.4     1
44.0     1
Name: value, dtype: int64
```

To get the unique values or the count of non-NaN items use the .unique and .nunique methods respectively.

In [82]:

```
s.unique()
```

Out[82]:

```
array([100. ,  23.4,   nan,  44. ])
```

In [83]:

```
s.nunique()
```

Out[83]:

```
3
```

To drop duplicate values use the .drop_duplicates method.

In [84]:

```
s.drop_duplicates()
```

Out[84]:

```
0    100.0
1     23.4
3      NaN
4     44.0
Name: value, dtype: float64
```

To drop duplicate index entries

In [85]:

```
s = pd.Series([390., 350., 30., 20.],index=['Falcon', 'Falcon', 'Parrot', 'Parrot'], name="
```

In [86]:

```
s
```

Out[86]:

```
Falcon    390.0
Falcon    350.0
Parrot     30.0
Parrot     20.0
Name: Max Speed, dtype: float64
```

In [87]:

```
s.groupby(s.index).mean()
```

Out[87]:

```
Falcon    370.0
Parrot     25.0
Name: Max Speed, dtype: float64
```

In [88]:

```
s.groupby(s.index).first()
```

Out[88]:

```
Falcon    390.0
Parrot     30.0
Name: Max Speed, dtype: float64
```

In [89]:

```
s.groupby(s.index).last()
```

Out[89]:

```
Falcon    350.0
Parrot     20.0
Name: Max Speed, dtype: float64
```

Statistics

There are many basic statistical measures in a series object's methods

sum() , mean(), describe()

In [90]:

```
import numpy as np
s = pd.Series(np.arange(100,300,10),name='values')
s[21]=10000
s[22]=np.nan
print(s)
print("Total values ",s.count())
```

```
0       100.0
1       110.0
2       120.0
3       130.0
4       140.0
5       150.0
6       160.0
7       170.0
8       180.0
9       190.0
10      200.0
11      210.0
12      220.0
13      230.0
14      240.0
15      250.0
16      260.0
17      270.0
18      280.0
19      290.0
21    10000.0
22        NaN
Name: values, dtype: float64
Total values  21
```

In [91]:

```
# sum
s.sum()
```

Out[91]:

```
13900.0
```

In [92]:

```
# Calculating the mean (the “expected value” or average)
s.mean()
```

Out[92]:

```
661.9047619047619
```

In [93]:

```
# Calculating the median
# the “middle” value at 50% that separates the lower values from the upper
#values
s.median()
```

Out[93]:

```
200.0
```

Note -

both of these methods (mean and median) ignore NaN (unless skipna is set to False)

But in practice if you do not ignore NaN, the result is nan.

In [94]:

```
s.mean(skipna=False)
```

Out[94]:

```
nan
```

In [95]:

```
s.median(skipna=False)
```

Out[95]:

```
nan
```

.describe method presents a good number of summary statistics and returns the result as a series.

It includes the count of values, their mean, standard deviation, minimum and maximum values, and the 25%, 50%, and 75% quantiles.

In [96]:

```
s.describe()
```

Out[96]:

```
count       21.000000
mean       661.904762
std       2140.403278
min        100.000000
25%        150.000000
50%        200.000000
75%        250.000000
max      10000.000000
Name: values, dtype: float64
```

Video about Quantiles

https://www.youtube.com/watch?v=IFKQLDmRK0Y (https://www.youtube.com/watch?v=IFKQLDmRK0Y)

series also has methods to find the minimum and maximum for the values, .min and .max.

In [97]:

```
s.min()
```

Out[97]:

```
100.0
```

In [98]:

```
s.max()
```

Out[98]:

```
10000.0
```

Dealing with None

In [102]:

```
# Fillna()
s.fillna(s.mean())
```

Out[102]:

```
0       100.000000
1       110.000000
2       120.000000
3       130.000000
4       140.000000
5       150.000000
6       160.000000
7       170.000000
8       180.000000
9       190.000000
10      200.000000
11      210.000000
12      220.000000
13      230.000000
14      240.000000
15      250.000000
16      260.000000
17      270.000000
18      280.000000
19      290.000000
21    10000.000000
22      661.904762
Name: values, dtype: float64
```

In [103]:

```
#dropna()
s.dropna()
```

Out[103]:

```
0       100.0
1       110.0
2       120.0
3       130.0
4       140.0
5       150.0
6       160.0
7       170.0
8       180.0
9       190.0
10      200.0
11      210.0
12      220.0
13      230.0
14      240.0
15      250.0
16      260.0
17      270.0
18      280.0
19      290.0
21    10000.0
Name: values, dtype: float64
```

In [106]:

```
# isnull()
s.isnull()
```

Out[106]:

```
0     False
1     False
2     False
3     False
4     False
5     False
6     False
7     False
8     False
9     False
10    False
11    False
12    False
13    False
14    False
15    False
16    False
17    False
18    False
19    False
21    False
22     True
Name: values, dtype: bool
```

Sorting

    .sort_values()

In [108]:

```
s = pd.Series([23,3,21,4,34,5],index=['a1','a2','a3','a4','a5','a6'])
s
```

Out[108]:

```
a1    23
a2     3
a3    21
a4     4
a5    34
a6     5
dtype: int64
```

In [109]:

```
s.sort_values()
```

Out[109]:

```
a2     3
a4     4
a6     5
a3    21
a1    23
a5    34
dtype: int64
```

In [113]:

```
s.sort_values(ascending=False)
```

Out[113]:

```
a5    34
a1    23
a3    21
a6     5
a4     4
a2     3
dtype: int64
```

In [116]:

```
s.sort_values(ascending=False , inplace=True)
s
```

Out[116]:

```
a5    34
a1    23
a3    21
a6     5
a4     4
a2     3
dtype: int64
```

.sort_index()

In [117]:

```
s.sort_index()
```

Out[117]:

```
a1    23
a2     3
a3    21
a4     4
a5    34
a6     5
dtype: int64
```

Applying Function to every item in series

The .map method applies a function to every item in the series.

In [135]:

```
s = pd.Series(['123','200 $','300 $'],name='price',index=['printer','mobile','laptop'])
s
```

Out[135]:

```
printer      123
mobile     200 $
laptop     300 $
Name: price, dtype: object
```

In [136]:

```
def remove_d(x):
    if x.endswith('$'):
        return int(x[:-2])
    else:
        return x
```

In [143]:

```
s.map(remove_d)
```

Out[143]:

```
printer    123
mobile     200
laptop     300
Name: price, dtype: object
```

In [144]:

```
# the . map function also accept a series.
# Any value of the series that matches the passed in index value
#will be updated to the corresponding value

x = pd.Series([1,2,3], index=['one', 'two', 'three'])
print(x)
y = pd.Series(['foo', 'bar', 'baz'], index=[1,2,3])
print(y)
```

```
one      1
two      2
three    3
dtype: int64
1    foo
2    bar
3    baz
dtype: object
```

In [145]:

```
x.map(y)
```

Out[145]:

```
one      foo
two      bar
three    baz
dtype: object
```

String operations

A series that has string data can be manipulated by vectorized string operations.

Though it is possible to accomplish these same operations via the .map method.

Typically, built-in methods will be faster because they are vectorized and often implemented in Cython.

To invoke the string operations, simply invoke them on the .str attribute of the series

In [146]:

```
names = pd.Series(['George', 'John', 'Paul'])
names
```

Out[146]:

```
0    George
1      John
2      Paul
dtype: object
```

In [147]:

```
names.str.lower()
```

Out[147]:

```
0    george
1      john
2      paul
dtype: object
```

The following vectorized string methods are available

METHOD	RESULT
cat	Concatenate list of strings onto items
center	Centers strings to width
contains	Boolean for whether pattern matches
count	Count pattern occurs in string
decode	Decode a codec encoding
encode	Encode a codec encoding
endswith	Boolean if strings end with item
findall	Find pattern in string
get	Attribute access on items
join	Join items with separator
len	Return length of items
lower	Lowercase the items
lstrip	Remove whitespace on left of items
match	Find groups in items from the pattern
pad	Pad the items
repeat	Repeat the string a certain number of times
replace	Replace a pattern with a new value
rstrip	Remove whitespace on the right of items
slice	Pull out slices from strings

`split`	Split items by pattern
`startswith`	Boolean if strings starts with item
`strip`	Remove whitespace from the items
`title`	Titlecase the items
`upper`	Uppercase the items

In [ ]:

```

```