Data Structures

Series

Series: It is a one-dimensional array-like structure used to represent a dataset and can be visualized as **a single column dataset**. It supports multiple data types, such as Integer, string, float.

Series can be created in multiple ways with the help of data elements which, if defined properly, act as data input to create a series. Therefore, data input can be an ndarray, dict, scalar, or a list. Let's take a look at each one in detail.

Now, let's see how we can create a series.

In [1]:

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

List

This basic Python data structure which can act as an input to create Pandas series. List can hold a range of values of multiple data types. So, if a dataset appears as list, use list as input to create series.

In [2]:

```
print (list('abcdef'))
```

```
['a', 'b', 'c', 'd', 'e', 'f']
```

In [3]:

```
# Pass list as an argument

first_series = pd.Series(list('abcdef'))
print (first_series)
```

```
0    a
1    b
2    c
3    d
4    e
5    f
dtype: object
```

Shows index, data value and data type

We have not created index for data but notice that data alignment is done automatically.

ndarray

An ndarray can be used as an input to create Pandas series. The use of ndarray is recommended wherever the dataset is number-centric and requires complex numerical computing.

In [4]:

```
# ndarray for countries
np_countries = np.array(['Algeria','Angola','Argentina','Australia','Austria','Bahamas'
,'Bangladesh','Belarus','Belgium',
                      'Bhutan','Brazil','Bulgaria','Cambodia','Cameroon','Chile','Chin
a','Colombia','Cyprus','Denmark'])
print (np_countries)
```

```
['Algeria' 'Angola' 'Argentina' 'Australia' 'Austria' 'Bahamas'
 'Bangladesh' 'Belarus' 'Belgium' 'Bhutan' 'Brazil' 'Bulgaria' 'Cambodia'
 'Cameroon' 'Chile' 'China' 'Colombia' 'Cyprus' 'Denmark']
```

In [5]:

```
# Pass ndarray as an argument

s_countries = pd.Series(np_countries)
print (s_countries)
```

```
0        Algeria
1         Angola
2      Argentina
3      Australia
4        Austria
5        Bahamas
6     Bangladesh
7        Belarus
8        Belgium
9         Bhutan
10        Brazil
11      Bulgaria
12      Cambodia
13      Cameroon
14         Chile
15         China
16      Colombia
17        Cyprus
18       Denmark
dtype: object
```

dict

A Pandas series can also be created using dictionary and it is very efficient when it comes to indexing or reindexing a dataset for data wrangling purposes. dict works in a key-value fashion, so use it whenever the dataset is structured as key-value pair.

In [6]:

```
dictionary = {"A" : 20, "B" : 35, 'C': 100}
print (dictionary)
```

```
{'A': 20, 'B': 35, 'C': 100}
```

In [7]:

```
# Pass dictionary as an argument

series = pd.Series(dictionary)
print(series)
```

```
A     20
B     35
C    100
dtype: int64
```

Input values in pd.Series

In [8]:

```
series_1 = pd.Series([100, 200, 300, 400, 500], index = ['A', "B", 'C', 'D', "E"])
print(series_1)
```

```
A    100
B    200
C    300
D    400
E    500
dtype: int64
```

In [13]:

```
series_1 = pd.Series([100, 200, 300, 400, 500], index = ['A', 'B', 'C', 'D', "E", 'F'])
print(series_1)
```

```
---------------------------------------------------------------------------
-
ValueError                                Traceback (most recent call las
t)
<ipython-input-13-3bed3d983fb4> in <module>
----> 1 series_1 = pd.Series([100, 200, 300, 400, 500], index = ['A', 'B',
'C', 'D', "E", 'F'])
      2 print(series_1)

/usr/local/lib/python3.7/site-packages/pandas/core/series.py in __init__(s
elf, data, index, dtype, name, copy, fastpath)
    290                     if len(index) != len(data):
    291                         raise ValueError(
--> 292                             f"Length of passed values is {len(dat
a)}, "
    293                             f"index implies {len(index)}."
    294                         )

ValueError: Length of passed values is 5, index implies 6.
```

np.int32 ---- Integer (-2147483648 to 2147483647) # np.int64 ---- Integer (-9223372036854775808 to 9223372036854775807)

In [9]:

```
# Series for countries and their gdp

country_gdp = pd.Series([2255.225482,629.9553062,11601.63022,25306.82494,27266.40335,19
466.99052,588.3691778,2890.345675,
                            24733.62696,1445.760002,4803.398244,2618.876037,590.4521124,
665.7982328,7122.938458,2639.54156,
                            3362.4656,15378.16704,30860.12808],
                              index = ['Algeria','Angola','Argentina','Australia','Austr
ia','Bahamas','Bangladesh','Belarus',
                                       'Belgium','Bhutan','Brazil','Bulgaria','Cambodia'
,'Cameroon','Chile','China','Colombia',
                                       'Cyprus','Denmark'])
print (country_gdp)
```

```
Algeria        2255.225482
Angola          629.955306
Argentina     11601.630220
Australia     25306.824940
Austria       27266.403350
Bahamas       19466.990520
Bangladesh      588.369178
Belarus        2890.345675
Belgium       24733.626960
Bhutan         1445.760002
Brazil         4803.398244
Bulgaria       2618.876037
Cambodia        590.452112
Cameroon        665.798233
Chile          7122.938458
China          2639.541560
Colombia       3362.465600
Cyprus        15378.167040
Denmark       30860.128080
dtype: float64
```

Scalar

Scalar data is another way to create Series. It is a stand-alone quantity and works with both vector and scalar datasets that can be used accordingly.

In [10]:

```
scalar_series = pd.Series(100, index = ['A', "B", 'C', 'D', "E"])
print(scalar_series)
```

```
A    100
B    100
C    100
D    100
E    100
dtype: int64
```

Accessing Elements in Series

In [14]:

```
# countries and their gdp

country_gdp = pd.Series([2255.225482,629.9553062,11601.63022,25306.82494,27266.40335,19
466.99052,588.3691778,2890.345675,
                           24733.62696,1445.760002,4803.398244,2618.876037,590.4521124,
665.7982328,7122.938458,2639.54156,
                           3362.4656,15378.16704,30860.12808],
                             index = ['Algeria','Angola','Argentina','Australia','Austr
ia','Bahamas','Bangladesh','Belarus',
                                      'Belgium','Bhutan','Brazil','Bulgaria','Cambodia'
,'Cameroon','Chile','China','Colombia',
                                      'Cyprus','Denmark'])
print (country_gdp)
```

```
Algeria        2255.225482
Angola          629.955306
Argentina     11601.630220
Australia     25306.824940
Austria       27266.403350
Bahamas       19466.990520
Bangladesh      588.369178
Belarus        2890.345675
Belgium       24733.626960
Bhutan         1445.760002
Brazil         4803.398244
Bulgaria       2618.876037
Cambodia        590.452112
Cameroon        665.798233
Chile          7122.938458
China          2639.541560
Colombia       3362.465600
Cyprus        15378.167040
Denmark       30860.128080
dtype: float64
```

In [15]:

```
country_gdp[0:5]
```

Out[15]:

```
Algeria       2255.225482
Angola         629.955306
Argentina    11601.630220
Australia    25306.824940
Austria      27266.403350
dtype: float64
```

In [16]:

```
# pass the country name in the argument and it will return gdp per capita for the
# country. This method is used to access elements through index values.

country_gdp['Bulgaria':'Denmark']
```

Out[16]:

```
Bulgaria     2618.876037
Cambodia      590.452112
Cameroon      665.798233
Chile        7122.938458
China        2639.541560
Colombia     3362.465600
Cyprus      15378.167040
Denmark     30860.128080
dtype: float64
```

In [17]:

```
country_gdp['Bulgaria': 5]
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-17-49fb5e315ed2> in <module>
----> 1 country_gdp['Bulgaria': 5]

/usr/local/lib/python3.7/site-packages/pandas/core/series.py in __getitem__(self, key)
    908             key = check_bool_indexer(self.index, key)
    909 
--> 910         return self._get_with(key)
    911 
    912     def _get_with(self, key):

/usr/local/lib/python3.7/site-packages/pandas/core/series.py in _get_with(self, key)
    913         # other: fancy integer or otherwise
    914         if isinstance(key, slice):
--> 915             return self._slice(key)
    916         elif isinstance(key, ABCDataFrame):
    917             raise TypeError(

/usr/local/lib/python3.7/site-packages/pandas/core/series.py in _slice(self, slobj, axis, kind)
    863 
    864     def _slice(self, slobj: slice, axis: int = 0, kind=None):
--> 865         slobj = self.index._convert_slice_indexer(slobj, kind=kind or "getitem")
    866         return self._get_values(slobj)
    867 

/usr/local/lib/python3.7/site-packages/pandas/core/indexes/base.py in _convert_slice_indexer(self, key, kind)
   2960             indexer = key
   2961         else:
-> 2962             indexer = self.slice_indexer(start, stop, step, kind=kind)
   2963 
   2964         return indexer

/usr/local/lib/python3.7/site-packages/pandas/core/indexes/base.py in slice_indexer(self, start, end, step, kind)
   4710         slice(1, 3)
   4711         """
-> 4712         start_slice, end_slice = self.slice_locs(start, end, step=step, kind=kind)
   4713 
   4714         # return a slice

/usr/local/lib/python3.7/site-packages/pandas/core/indexes/base.py in slice_locs(self, start, end, step, kind)
   4929         end_slice = None
   4930         if end is not None:
-> 4931             end_slice = self.get_slice_bound(end, "right", kind)
   4932         if end_slice is None:
   4933             end_slice = len(self)

/usr/local/lib/python3.7/site-packages/pandas/core/indexes/base.py in get_slice_bound(self, label, side, kind)
   4835         # For datetime indices label may be a string that has to b
```

```
e converted
   4836         # to datetime boundary according to its resolution.
-> 4837         label = self._maybe_cast_slice_bound(label, side, kind)
   4838 
   4839         # we need to look up the label

/usr/local/lib/python3.7/site-packages/pandas/core/indexes/base.py in _may
be_cast_slice_bound(self, label, side, kind)
   4787         # this is rejected (generally .loc gets you here)
   4788         elif is_integer(label):
-> 4789             self._invalid_indexer("slice", label)
   4790 
   4791         return label

/usr/local/lib/python3.7/site-packages/pandas/core/indexes/base.py in _inv
alid_indexer(self, form, key)
   3074         """
   3075         raise TypeError(
-> 3076             f"cannot do {form} indexing on {type(self)} with these
"
   3077             f"indexers [{key}] of {type(key)}"
   3078         )

TypeError: cannot do slice indexing on <class 'pandas.core.indexes.base.In
dex'> with these indexers [5] of <class 'int'>
```

We want to check with countries have gdp value greater than 3000?

In [18]:

```
country_gdp > 3000
```

Out[18]:

```
Algeria       False
Angola        False
Argentina      True
Australia      True
Austria        True
Bahamas        True
Bangladesh    False
Belarus       False
Belgium        True
Bhutan        False
Brazil         True
Bulgaria      False
Cambodia      False
Cameroon      False
Chile          True
China         False
Colombia       True
Cyprus         True
Denmark        True
dtype: bool
```

Print the name of the country and the gdp where ever the gdp > 3000

In [19]:

```
country_gdp[country_gdp > 3000]
```

Out[19]:

```
Argentina    11601.630220
Australia    25306.824940
Austria      27266.403350
Bahamas      19466.990520
Belgium      24733.626960
Brazil        4803.398244
Chile         7122.938458
Colombia      3362.465600
Cyprus       15378.167040
Denmark      30860.128080
dtype: float64
```

Print the name of the country and the gdp where ever the gdp >= 5000

In [20]:

```
country_gdp[country_gdp >= 5000]
```

Out[20]:

```
Argentina    11601.630220
Australia    25306.824940
Austria      27266.403350
Bahamas      19466.990520
Belgium      24733.626960
Chile         7122.938458
Cyprus       15378.167040
Denmark      30860.128080
dtype: float64
```

Vectorized operations

Vectorized operations show you how you can add two or more series. The vector operations are essentially performed by the index positions of data elements.

The first example shows how the two series, 'first_vector_series' and 'second_vector_series' are added and this is done at index level.

In [21]:

```
first_vector_series = pd.Series([1,2,3,4], index = ['a','b','c','d'])
second_vector_series = pd.Series([10,20,30,40], index = ['a','b','c','d'])

print (first_vector_series)
print ()
print (second_vector_series)
```

```
a    1
b    2
c    3
d    4
dtype: int64

a    10
b    20
c    30
d    40
dtype: int64
```

In [22]:

```
print (first_vector_series + second_vector_series)
```

```
a    11
b    22
c    33
d    44
dtype: int64
```

Let's **shuffle indices** and see what happens. For the second vector series, we change the values of indices a, d, b, and c. Thus, when we add the two vector series, we get a different output as the data element is bound to the index position.

In [23]:

```
first_vector_series = pd.Series([1,2,3,4], index = ['a','b','c','d'])
second_vector_series = pd.Series([10,20,30,40], index = ['c','a','d','b'])

print (first_vector_series)
print ()
print (second_vector_series)
```

```
a    1
b    2
c    3
d    4
dtype: int64

c    10
a    20
d    30
b    40
dtype: int64
```

In [24]:

```
print (first_vector_series + second_vector_series)
```

```
a    21
b    42
c    13
d    34
dtype: int64
```

In [25]:

```
first_vector_series = pd.Series([1,2,3,4], index = ['a','b','c','d'])
second_vector_series = pd.Series([10.0,20,30,40], index = ['a','b','e','f'])

print (first_vector_series)
print ()
print (second_vector_series)
```

```
a    1
b    2
c    3
d    4
dtype: int64

a    10.0
b    20.0
e    30.0
f    40.0
dtype: float64
```

In [26]:

```
print (first_vector_series + second_vector_series)
```

```
a    11.0
b    22.0
c     NaN
d     NaN
e     NaN
f     NaN
dtype: float64
```

Where ever the indices don't match, it will not add and would hold NOT A NUMBER or NaN

Dataframes

DataFrame is another core feature of the Pandas data structure.

DataFrame is a two-dimensional labeled data structure with columns of potentially different data types.

A DataFrame looks like a spreadsheet with a row-columnar structure or a SQL data table with rows and columns.

There can be several inputs to the DataFrame and we'll go through them in detail. Let's have a quick overview of the data inputs:

dict

A Pandas DataFrame can also be created using dictionary of list. It is very efficient when it comes to indexing or reindexing a dataset for data wrangling purposes.

In this example, we will create a dataset related to Summer Olympics.

First, import the Pandas library. Then, declare a dict 'Olympic_data_list' and pass the indices 'HostCity', 'No. of Participating Countries', and 'Year' with its data elements as arguments.

As you can observe, it is a tabular representation of data with rows and columns. Next, pass this list to the DataFrame method 'pd.DataFrame' to create a basic DataFrame.

Note that data alignment is automatically taken care here. When we call this DataFrame 'df_Olympic_data', the output displays all the rows with its corresponding indices.

In [27]:

```
olympic_data = {'HostCity':['London', 'Beijing', 'Athens', 'Sydney', 'Atlanta'], 'Year'
: [2012, 2008, 2004, 2000, 1996],
                    'No. of Participating Countries': [205, 205, 201, 200, 197]}
print (type(olympic_data))
print ()
print (olympic_data)
```

```
<class 'dict'>

{'HostCity': ['London', 'Beijing', 'Athens', 'Sydney', 'Atlanta'], 'Year':
[2012, 2008, 2004, 2000, 1996], 'No. of Participating Countries': [205, 20
5, 201, 200, 197]}
```

In [29]:

```
df_olympic_data = pd.DataFrame(olympic_data)
print (df_olympic_data)
print ()
display (df_olympic_data)
print ()
df_olympic_data
```

```
  HostCity  Year  No. of Participating Countries
0   London  2012                             205
1  Beijing  2008                             205
2   Athens  2004                             201
3   Sydney  2000                             200
4  Atlanta  1996                             197
```

	HostCity	Year	No. of Participating Countries
0	London	2012	205
1	Beijing	2008	205
2	Athens	2004	201
3	Sydney	2000	200
4	Atlanta	1996	197

Out[29]:

	HostCity	Year	No. of Participating Countries
0	London	2012	205
1	Beijing	2008	205
2	Athens	2004	201
3	Sydney	2000	200
4	Atlanta	1996	197

Series

Series can also be an input to a DataFrame.

Let’s learn how to create DataFrame from series.

Let’s create two series first. The first series, ‘olympic_series_participation’, is for the number of countries participating for the given year. The second series, ‘olympic_series_country’, is for the cities which held the Olympics that year. Now, create a DataFrame ‘df_olympic_series’ and pass both the series as dicts in it. You can also assign column names in the DataFrame and manipulate the dataset as shown in this example.

In [30]:

```
olympic_series_participation = pd.Series([205,205,201,200,197], index = [2012,2008,2004
,2000,1996])
olympic_series_countries = pd.Series(['London', 'Beijing', 'Athens', 'Sydney', 'Atlant
a'], index = [2012,2008,2004,2000,1996])
```

In [31]:

```
print (olympic_series_participation)
print ()
print (olympic_series_countries)
```

```
2012    205
2008    205
2004    201
2000    200
1996    197
dtype: int64

2012     London
2008    Beijing
2004     Athens
2000     Sydney
1996    Atlanta
dtype: object
```

In [34]:

```
dict_Series = {'No. of Participating Countries': olympic_series_participation,
                                  'HostCity': olympic_series_countries} # dictionary
print (dict_Series)
```

```
{'No. of Participating Countries': 2012    205
2008    205
2004    201
2000    200
1996    197
dtype: int64, 'HostCity': 2012     London
2008    Beijing
2004     Athens
2000     Sydney
1996    Atlanta
dtype: object}
```

In [32]:

```
df_olympic_series = pd.DataFrame({'No. of Participating Countries': olympic_series_part
icipation,
                                  'HostCity': olympic_series_countries})
display (df_olympic_series)
```

	No. of Participating Countries	HostCity
2012	205	London
2008	205	Beijing
2004	201	Athens
2000	200	Sydney
1996	197	Atlanta

ndarray

An ndarray can be used as an input to creating Pandas DataFrame. The use of ndarray is recommended wherever the dataset is number centric and when instances require complex numerical computing.

In [33]:

```
# Create an ndarrays with years

np_array = np.array([2012,2008,2004,2006]) # array

dict_ndarray = {'year':np_array} # dictionary
print (dict_ndarray)
```

```
{'year': array([2012, 2008, 2004, 2006])}
```

In [35]:

```
# Create a df with the ndarray dict

df_ndarray = pd.DataFrame(dict_ndarray)

display (df_ndarray)
```

	year
0	2012
1	2008
2	2004
3	2006

Accessing column in a dataframe

In [36]:

```
display (df_olympic_data)
```

	HostCity	Year	No. of Participating Countries
0	London	2012	205
1	Beijing	2008	205
2	Athens	2004	201
3	Sydney	2000	200
4	Atlanta	1996	197

In [37]:

```
display (df_olympic_data.HostCity)
```

```
0     London
1    Beijing
2     Athens
3     Sydney
4    Atlanta
Name: HostCity, dtype: object
```

In [38]:

```
display (df_olympic_data[['HostCity', "Year"]]) # used for accessing multiple columns
```

	HostCity	Year
0	London	2012
1	Beijing	2008
2	Athens	2004
3	Sydney	2000
4	Atlanta	1996

In [42]:

```
display (df_olympic_data[['HostCity']])
```

	HostCity
0	London
1	Beijing
2	Athens
3	Sydney
4	Atlanta

In [43]:

```
display (df_olympic_data.No. of Participating Countries)
```

```
  File "<ipython-input-43-be50ea07044e>", line 1
    display (df_olympic_data.No. of Participating Countries)
                                                  ^
SyntaxError: invalid syntax
```

In [44]:

```
display (df_olympic_data[['No. of Participating Countries']]) # used for accessing colu
mns with spaces in the name
```

	No. of Participating Countries
0	205
1	205
2	201
3	200
4	197

Data Operation with Statistical Functions

In [45]:

```
df_test_scores = pd.DataFrame({'Test1': [95,84,73,88,82,61], 'Test2': [74,85,82,73,77,7
9]},
                              index = ['Jack','Lewis','Patrick','Rich','Kelly','Paula'
])

display (df_test_scores)
```

	Test1	Test2
Jack	95	74
Lewis	84	85
Patrick	73	82
Rich	88	73
Kelly	82	77
Paula	61	79

In [46]:

```
print (df_test_scores.max()) # default axis = 0; column wise ans
```

```
Test1    95
Test2    85
dtype: int64
```

In [47]:

```
print (df_test_scores.mean())
```

```
Test1    80.500000
Test2    78.333333
dtype: float64
```

In [48]:

```
print (df_test_scores.median())
```

```
Test1    83.0
Test2    78.0
dtype: float64
```

In [49]:

```
print (df_test_scores.std())
```

```
Test1    11.979149
Test2     4.633213
dtype: float64
```

Who has the highest score and what is the highest score in Test1 ?

In [51]:

```
print (df_test_scores.Test1)
print ()
print (df_test_scores.Test1.max())
print ()
print (df_test_scores.Test1 == df_test_scores.Test1.max())
```

```
Jack       95
Lewis      84
Patrick    73
Rich       88
Kelly      82
Paula      61
Name: Test1, dtype: int64

95

Jack        True
Lewis      False
Patrick    False
Rich       False
Kelly      False
Paula      False
Name: Test1, dtype: bool
```

In [54]:

```
print (df_test_scores.Test1[df_test_scores.Test1 == df_test_scores.Test1.max()])
```

```
Jack    95
Name: Test1, dtype: int64
```

Creating a new column

In [55]:

```
df_test_scores.Total_Scores = df_test_scores.Test1 + df_test_scores.Test2
display (df_test_scores)
```

```
/usr/local/lib/python3.7/site-packages/ipykernel_launcher.py:1: UserWarnin
g: Pandas doesn't allow columns to be created via a new attribute name - s
ee https://pandas.pydata.org/pandas-docs/stable/indexing.html#attribute-ac
cess
  """Entry point for launching an IPython kernel.
```

	Test1	Test2
Jack	95	74
Lewis	84	85
Patrick	73	82
Rich	88	73
Kelly	82	77
Paula	61	79

In [56]:

```
df_test_scores[['Total_Scores']] = df_test_scores.Test1 + df_test_scores.Test2
display (df_test_scores)
```

```
---------------------------------------------------------------------------
-
KeyError                                  Traceback (most recent call las
t)
<ipython-input-56-b971b93ada34> in <module>
----> 1 df_test_scores[['Total_Scores']] = df_test_scores.Test1 + df_test_
scores.Test2
      2 display (df_test_scores)

/usr/local/lib/python3.7/site-packages/pandas/core/frame.py in __setitem__
(self, key, value)
   2933             self._setitem_frame(key, value)
   2934         elif isinstance(key, (Series, np.ndarray, list, Index)):
-> 2935             self._setitem_array(key, value)
   2936         else:
   2937             # set column

/usr/local/lib/python3.7/site-packages/pandas/core/frame.py in _setitem_ar
ray(self, key, value)
   2964             else:
   2965                 indexer = self.loc._get_listlike_indexer(
-> 2966                     key, axis=1, raise_missing=False
   2967                 )[1]
   2968                 self._check_setitem_copy()

/usr/local/lib/python3.7/site-packages/pandas/core/indexing.py in _get_lis
tlike_indexer(self, key, axis, raise_missing)
   1551
   1552         self._validate_read_indexer(
-> 1553             keyarr, indexer, o._get_axis_number(axis), raise_missi
ng=raise_missing
   1554         )
   1555         return keyarr, indexer

/usr/local/lib/python3.7/site-packages/pandas/core/indexing.py in _validat
e_read_indexer(self, key, indexer, axis, raise_missing)
   1638             if missing == len(indexer):
   1639                 axis_name = self.obj._get_axis_name(axis)
-> 1640                 raise KeyError(f"None of [{key}] are in the [{axis
_name}]")
   1641
   1642             # We (temporarily) allow for some missing keys with .l
oc, except in

KeyError: "None of [Index(['Total_Scores'], dtype='object')] are in the [c
olumns]"
```

In [57]:

```
df_test_scores['Total_Scores'] = df_test_scores.Test1 + df_test_scores.Test2
display (df_test_scores)
```

	Test1	Test2	Total_Scores
Jack	95	74	169
Lewis	84	85	169
Patrick	73	82	155
Rich	88	73	161
Kelly	82	77	159
Paula	61	79	140

How will we find the mean score for each student?

In [58]:

```
df_test_scores[['Test1','Test2']].mean(axis=1)
```

Out[58]:

```
Jack       84.5
Lewis      84.5
Patrick    77.5
Rich       80.5
Kelly      79.5
Paula      70.0
dtype: float64
```

In [60]:

```
df_test_scores[["Total_Scores"]]/2
```

Out[60]:

	Total_Scores
Jack	84.5
Lewis	84.5
Patrick	77.5
Rich	80.5
Kelly	79.5
Paula	70.0

In [61]:

```
df_test_scores['Avg_Scores'] = df_test_scores['Total_Scores']/2
display (df_test_scores)
```

	Test1	Test2	Total_Scores	Avg_Scores
Jack	95	74	169	84.5
Lewis	84	85	169	84.5
Patrick	73	82	155	77.5
Rich	88	73	161	80.5
Kelly	82	77	159	79.5
Paula	61	79	140	70.0

In [ ]:

```
```