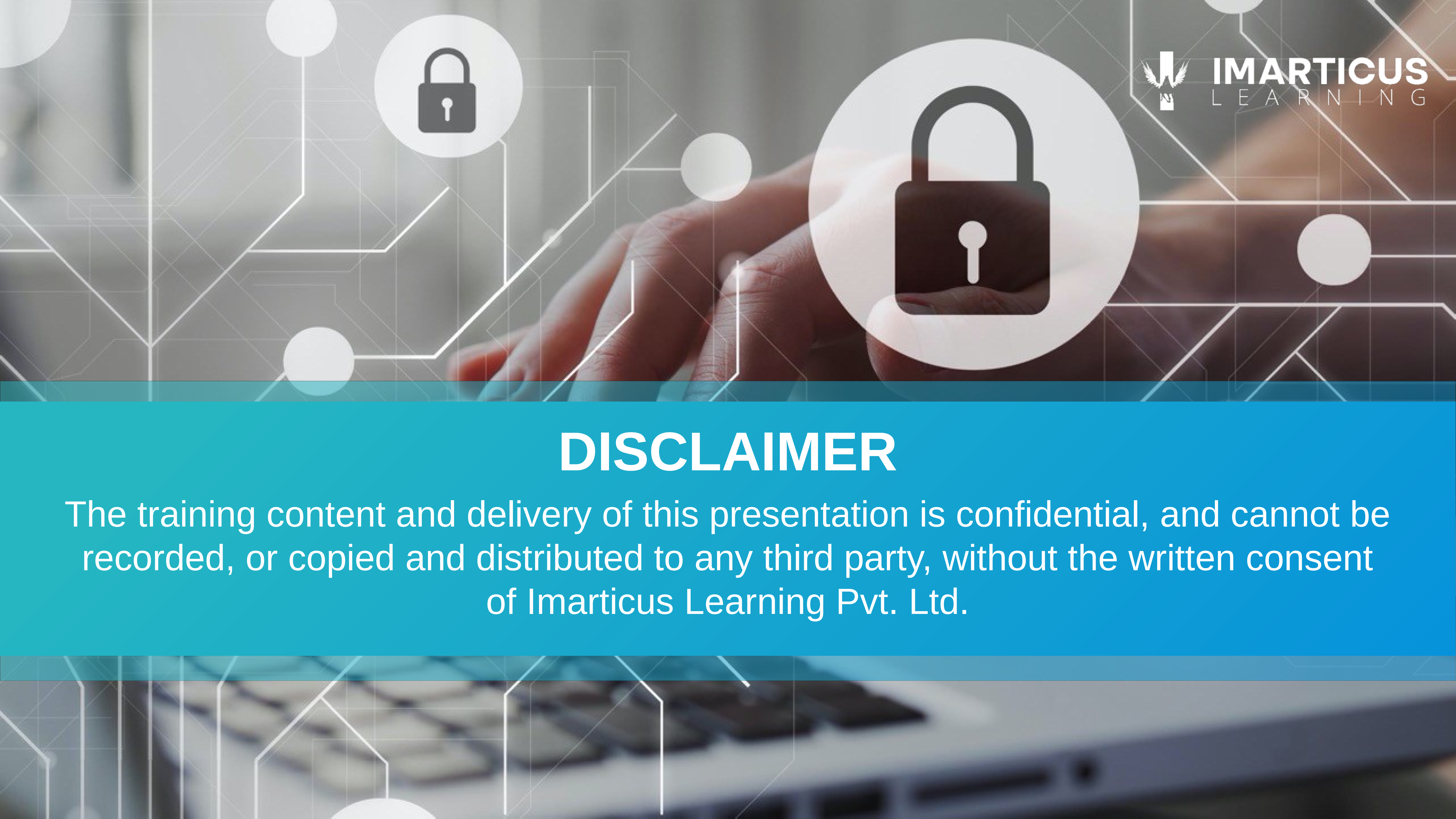
Python Programming

Pandas



IMARTICUS LEARNING

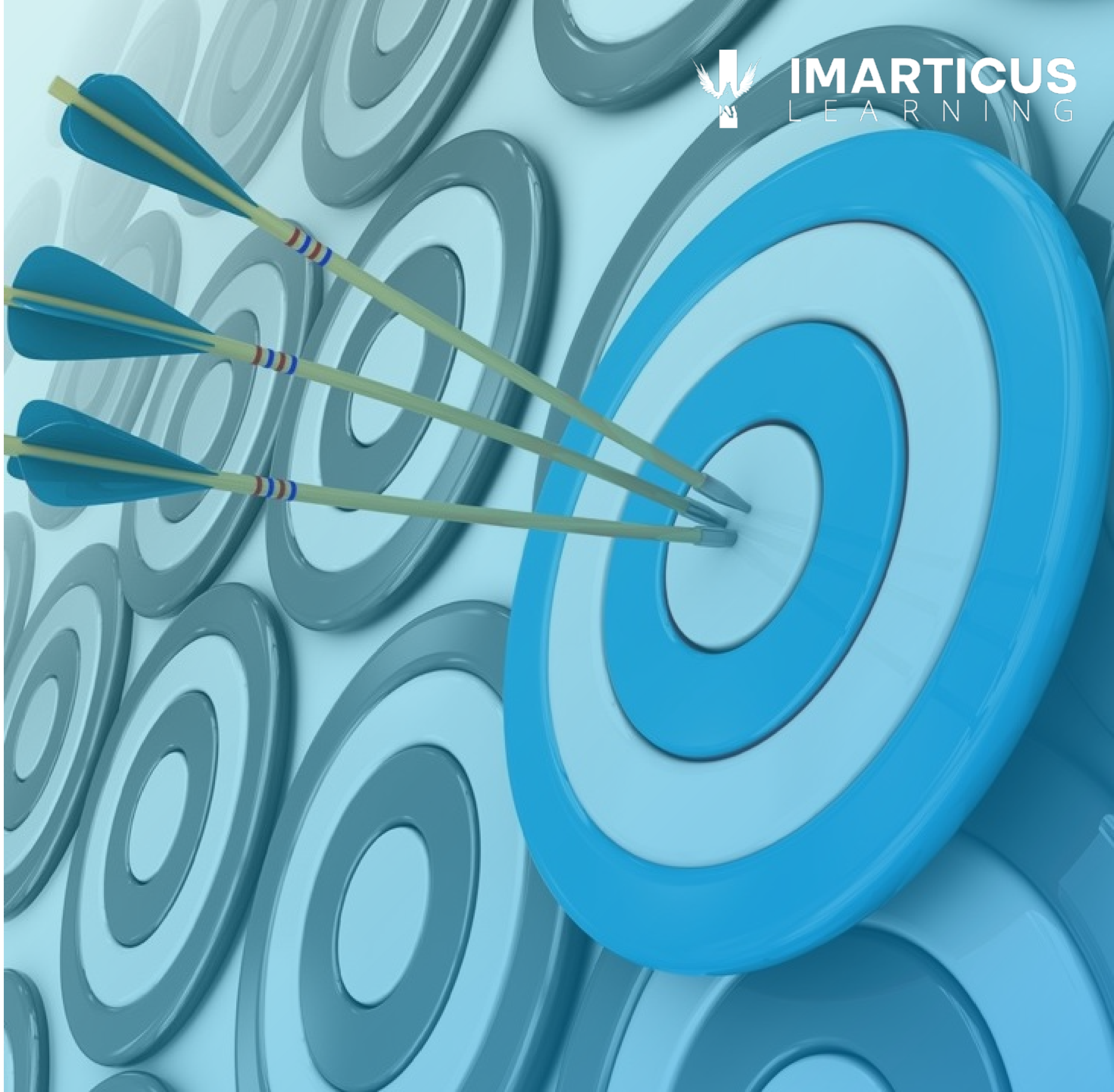
DISCLAIMER

The training content and delivery of this presentation is confidential, and cannot be recorded, or copied and distributed to any third party, without the written consent of Imarticus Learning Pvt. Ltd.

LEARNING OBJECTIVES

At the end of this session, you will be able to:

- Introduction to Pandas
- Pandas Series
- Creating Pandas Series
- Accessing Series Elements
- Filtering a Series
- Arithmetic Operations
- Series Ranking and Sorting
- Checking Null Values
- Concatenate a Series



Introduction to Pandas

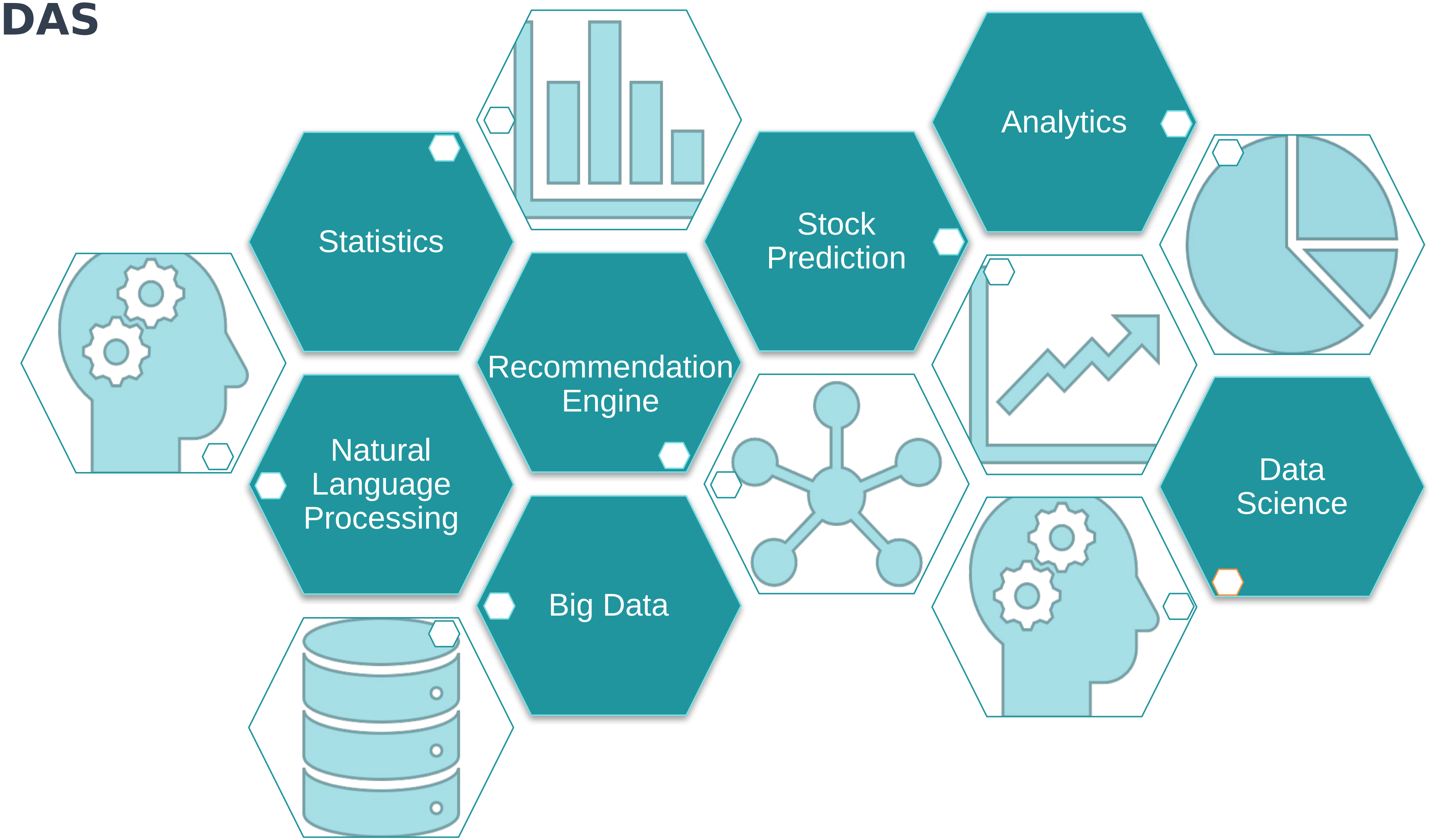
INTRODUCTION TO PANDAS

Pandas is an open source library in Python

It is useful in data manipulation and analysis

It provides fast, flexible, and expressive data structures designed to make working with structured (tabular, multidimensional, potentially heterogeneous) and time series data

INTRODUCTION TO PANDAS



Did You Know?

Pandas library is built on top of NumPy library providing high performance, easy to use data structures and data analysis tools for the python programming language

PANDAS vs NUMPY

Pandas

- It provides some powerful objects like DataFrames and Series
- More streamlined handling of tabular data, and rich time series functionality
- Easy handling of missing data, data alignment, groupby, merge, and join methods

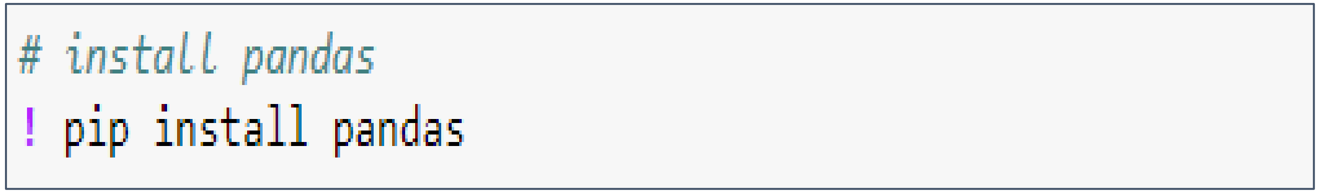
NumPy

- It provides us with a powerful object known as an Array
- Supports fast mathematical computation on arrays and matrices
- A wide range of mathematical array operations

INSTALLATION

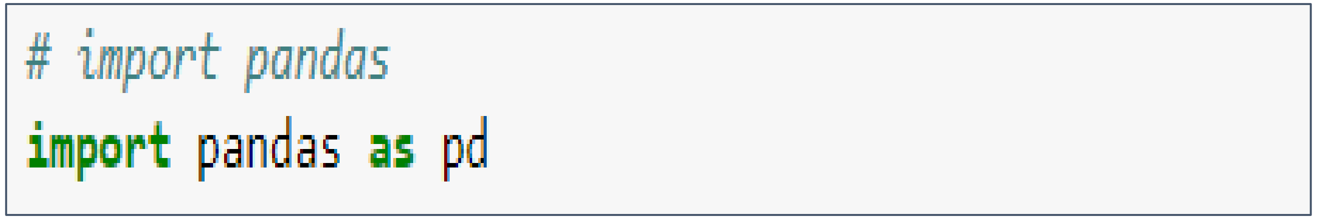
Installing Pandas

Use the following command to install Pandas using Jupyter Notebook



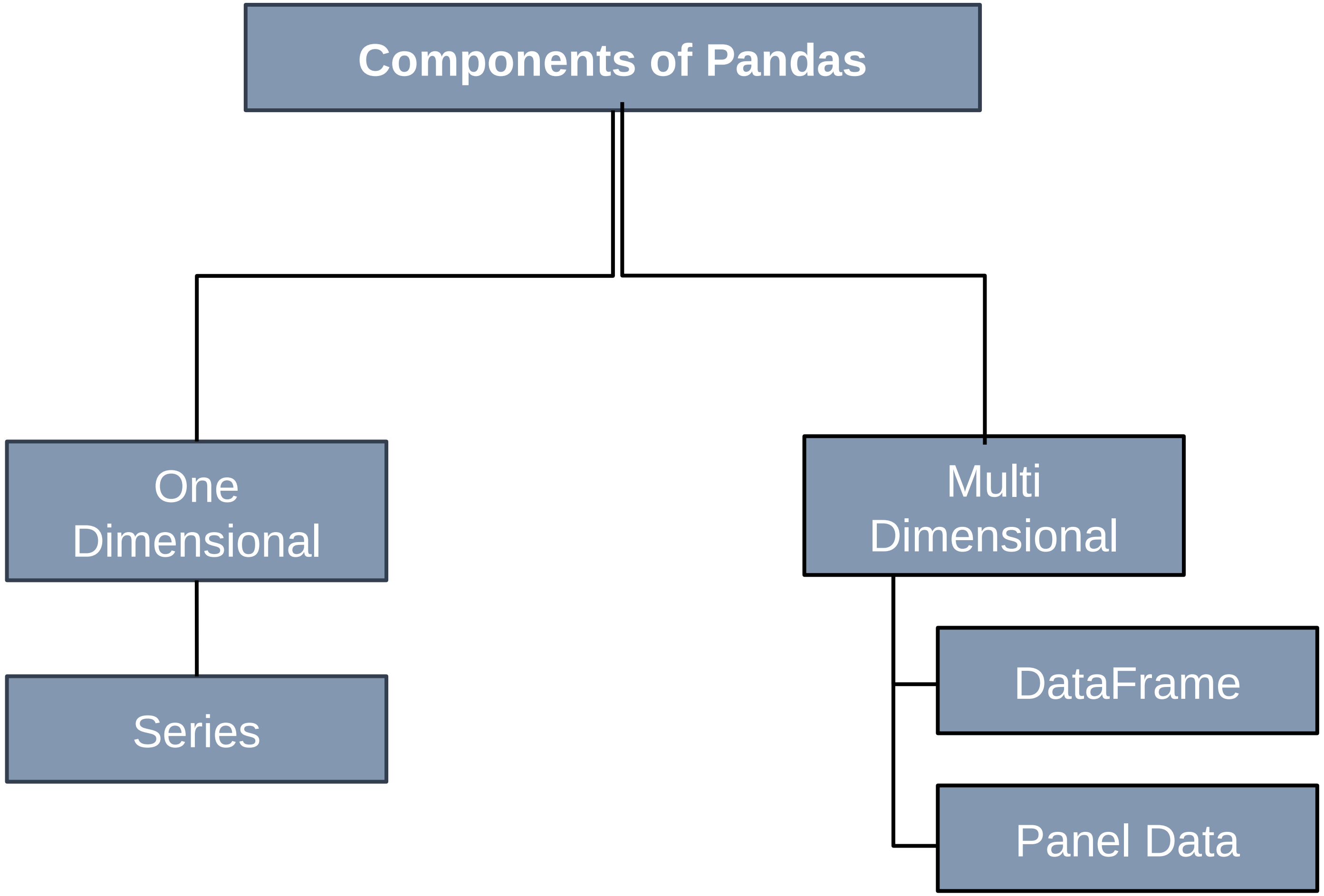
```
# install pandas
! pip install pandas
```

Importing pandas as alias ‘pd’ is a common practice



```
# import pandas
import pandas as pd
```

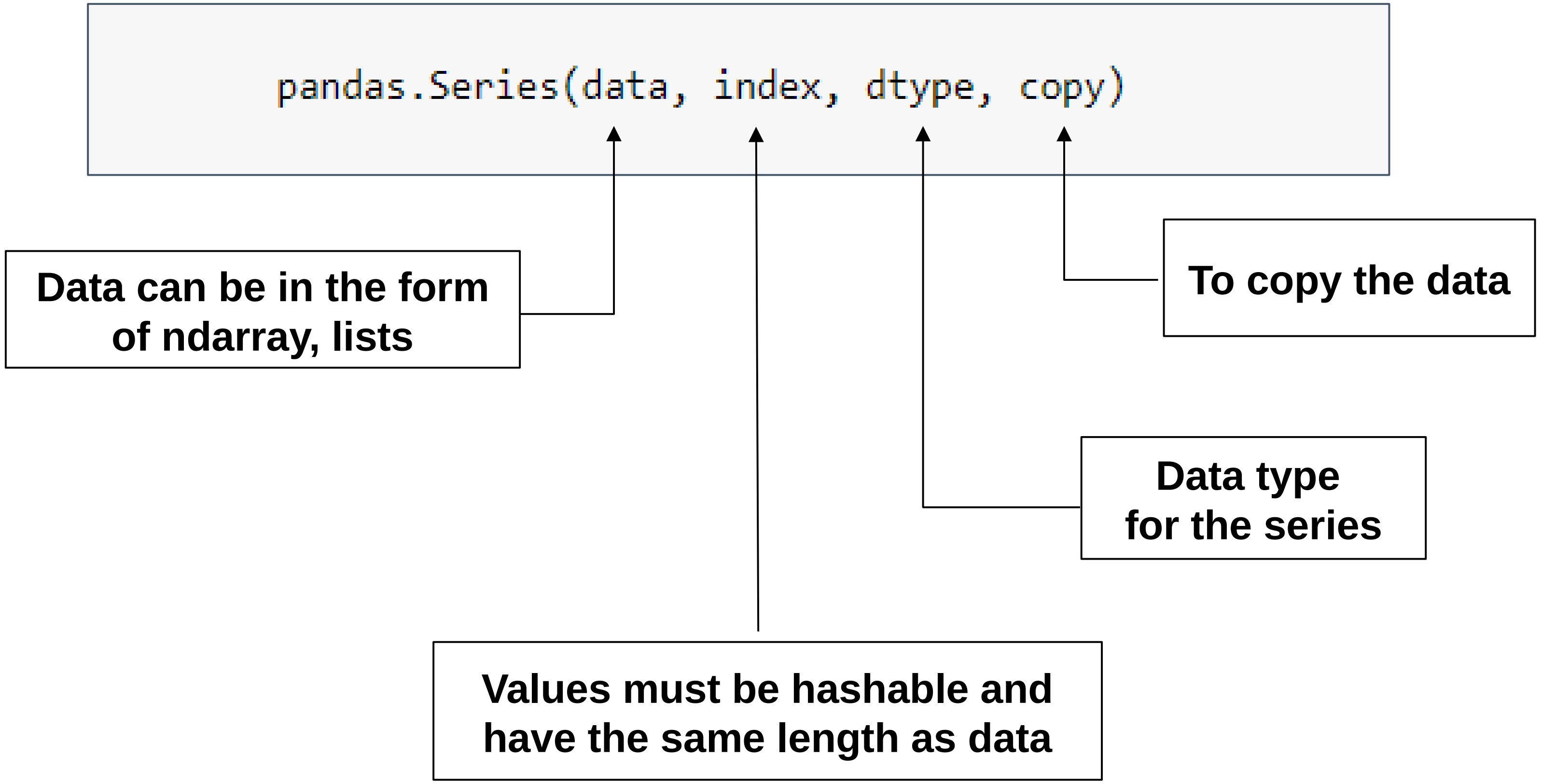
PANDAS COMPONENTS



Pandas Series

PANDAS SERIES

A Pandas Series is a one-dimensional array of indexed data



```
pandas.Series(data, index, dtype, copy)
```

Data can be in the form of ndarray, lists

To copy the data

Data type for the series

Values must be hashable and have the same length as data

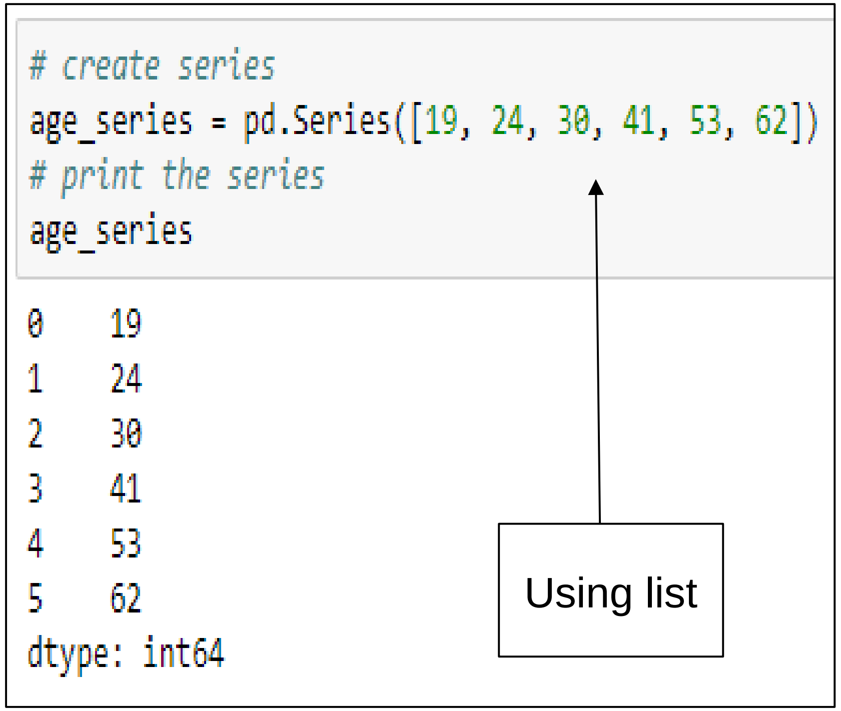
Pandas Series can be thought of as a column in the excel sheet

Creating Pandas Series

CREATING PANDAS SERIES

A pandas series can be created using:

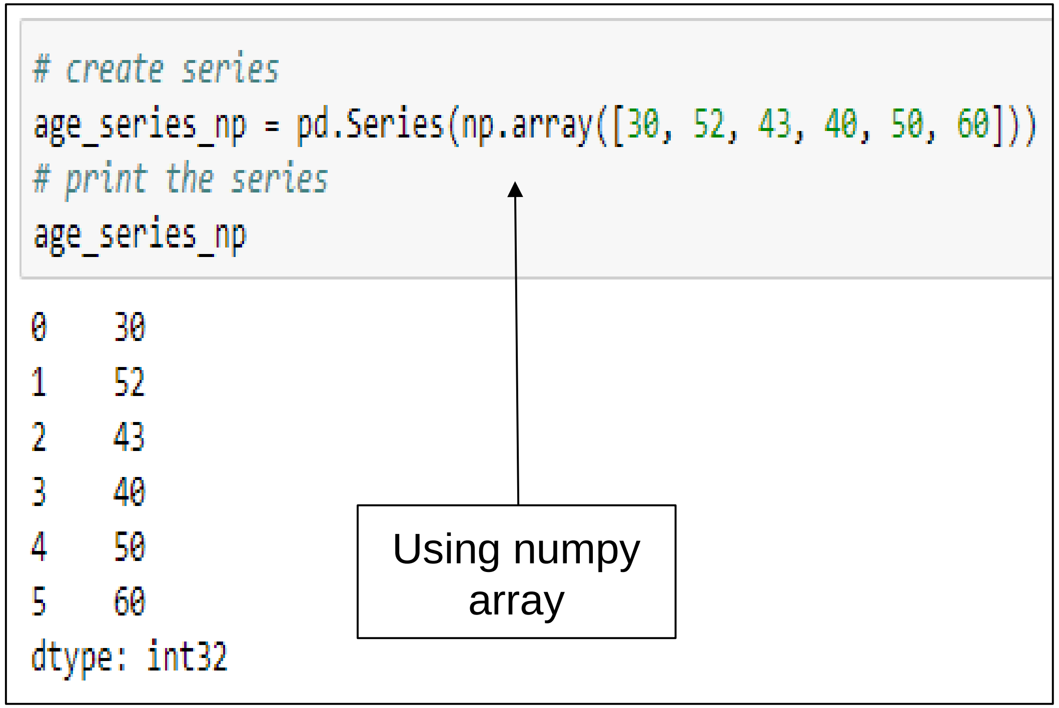
- A Python List
- Numpy Array



```
# create series
age_series = pd.Series([19, 24, 30, 41, 53, 62])
# print the series
age_series
```

```
0    19
1    24
2    30
3    41
4    53
5    62
dtype: int64
```

Using list



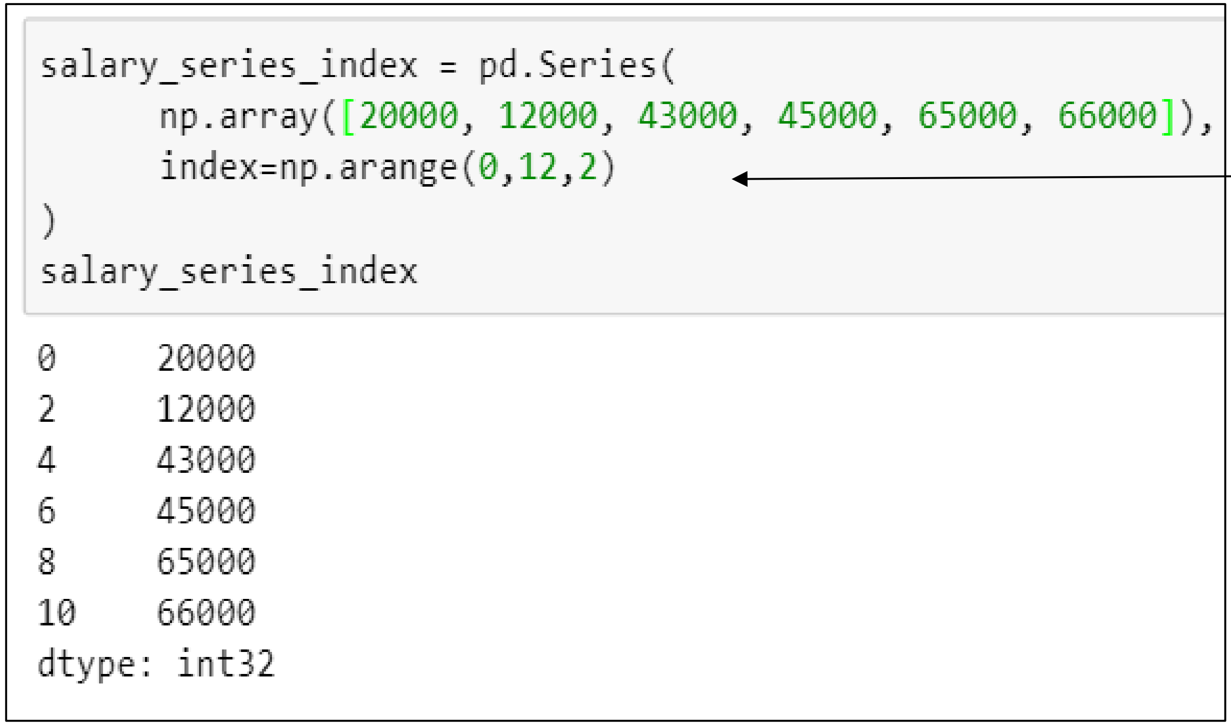
```
# create series
age_series_np = pd.Series(np.array([30, 52, 43, 40, 50, 60]))
# print the series
age_series_np
```

```
0    30
1    52
2    43
3    40
4    50
5    60
dtype: int32
```

Using numpy array

SETTING INDEX TO SERIES

- We can set specific numeric values as index while creating a series



```
salary_series_index = pd.Series(
      np.array([20000, 12000, 43000, 45000, 65000, 66000]),
      index=np.arange(0,12,2)
)
salary_series_index
```

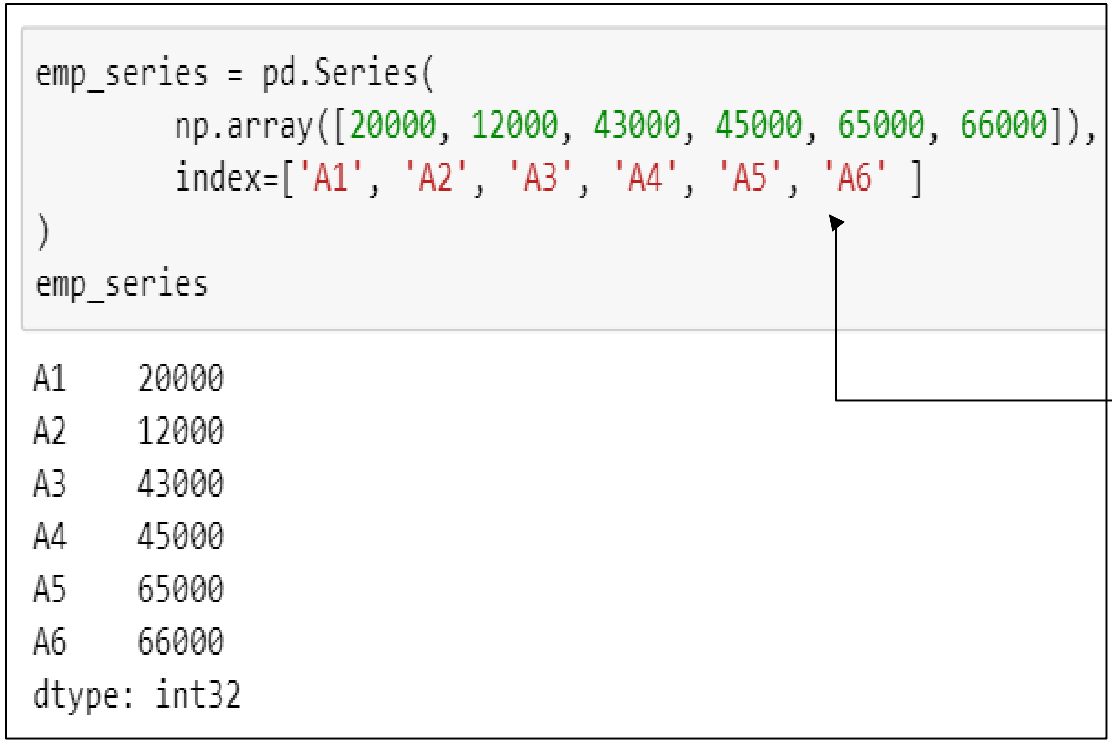
```
0      20000
2      12000
4      43000
6      45000
8      65000
10     66000
dtype: int32
```

Pass index parameter

By default, index ranges from 0 to (n-1) for series of length ‘n’

SETTING INDEX TO SERIES

- We can also specify the strings as index values



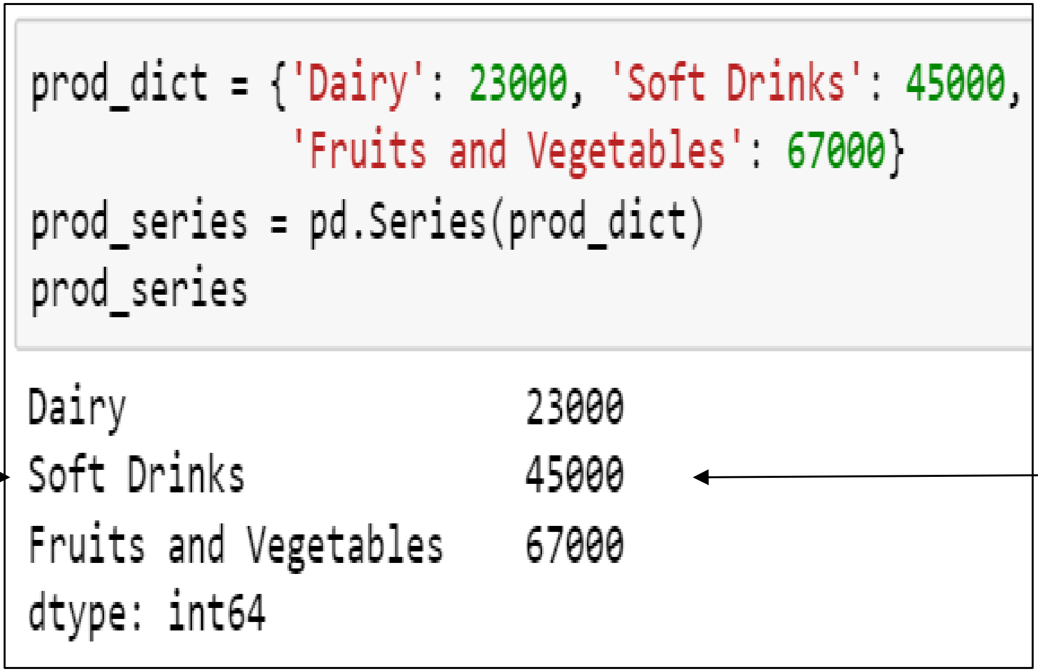
```
emp_series = pd.Series(
        np.array([20000, 12000, 43000, 45000, 65000, 66000]),
        index=['A1', 'A2', 'A3', 'A4', 'A5', 'A6' ]
)
emp_series
```

```
A1    20000
A2    12000
A3    43000
A4    45000
A5    65000
A6    66000
dtype: int32
```

String as a row index

CREATING SERIES FROM DICTIONARY

Keys as index



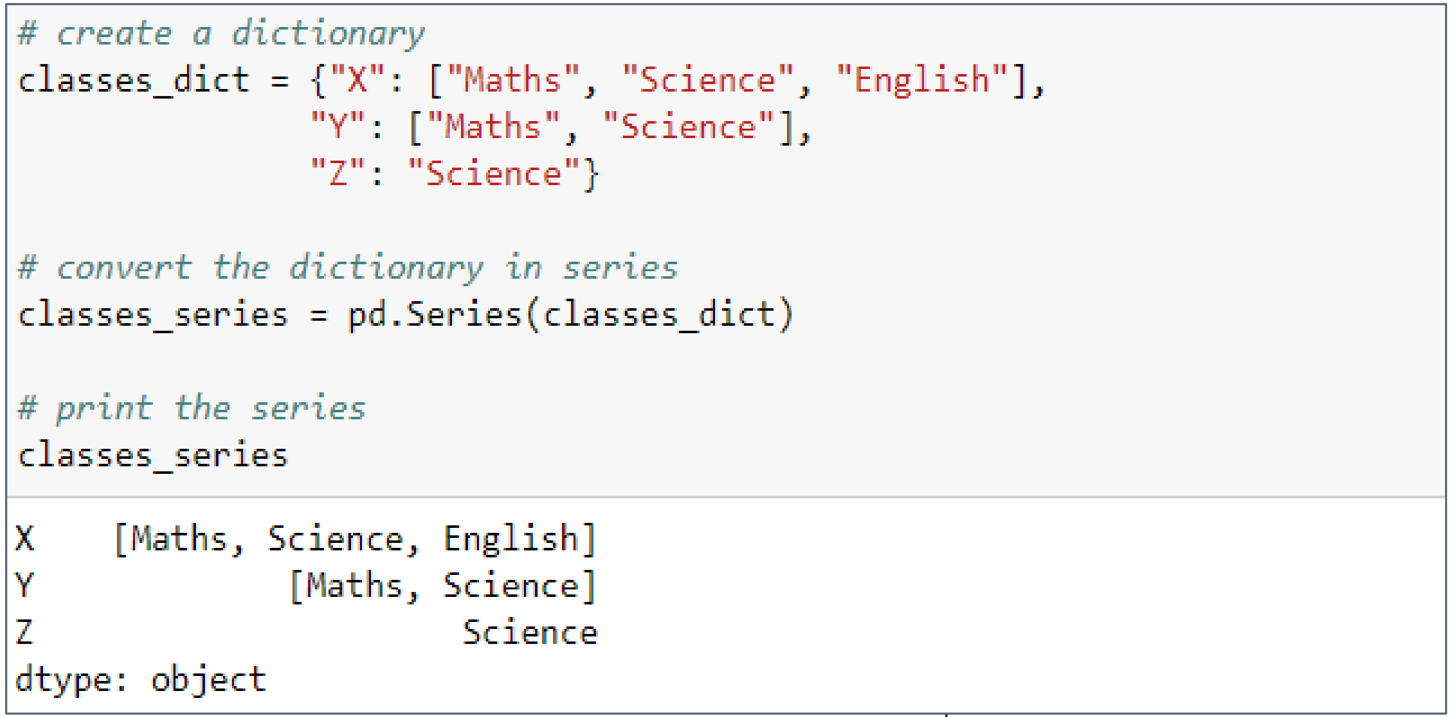
```
prod_dict = {'Dairy': 23000, 'Soft Drinks': 45000,
             'Fruits and Vegetables': 67000}
prod_series = pd.Series(prod_dict)
prod_series
```

```
Dairy                    23000
Soft Drinks              45000
Fruits and Vegetables    67000
dtype: int64
```

Values as row values

The key becomes the row index while the value is the row value at that row index

ACCESSING SERIES INDEX AND VALUES



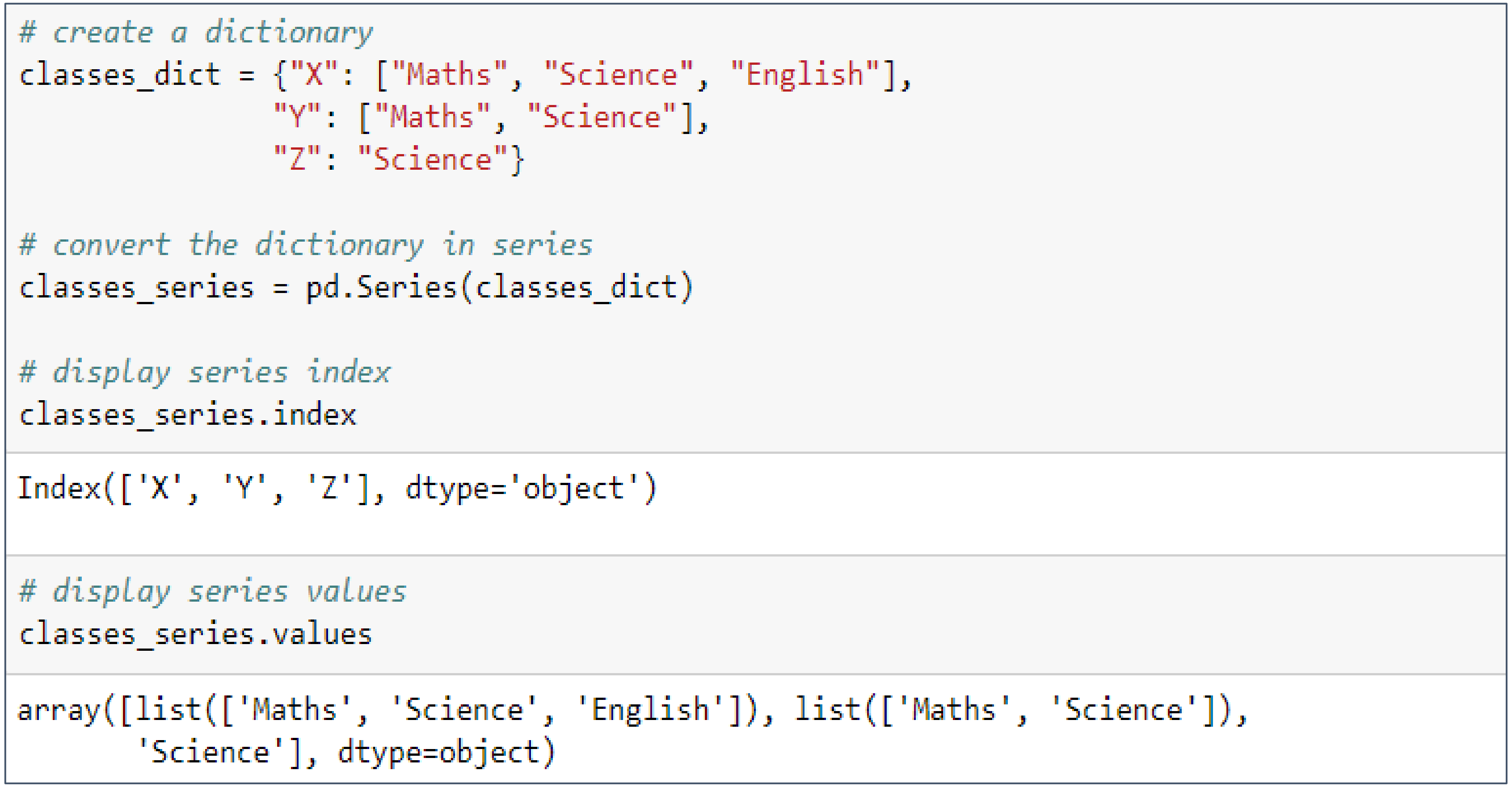
```
# create a dictionary
classes_dict = {"X": ["Maths", "Science", "English"],
               "Y": ["Maths", "Science"],
               "Z": "Science"}

# convert the dictionary in series
classes_series = pd.Series(classes_dict)

# print the series
classes_series
```

```
X    [Maths, Science, English]
Y             [Maths, Science]
Z                      Science
dtype: object
```

If you have multiple values for a single key, those multiple values will take up a single row

ACCESSING SERIES INDEX AND VALUES

- To display the index names and values of the series use **index** and **values attributes** respectively



```
# create a dictionary
classes_dict = {"X": ["Maths", "Science", "English"],
               "Y": ["Maths", "Science"],
               "Z": "Science"}

# convert the dictionary in series
classes_series = pd.Series(classes_dict)

# display series index
classes_series.index
```

```
Index(['X', 'Y', 'Z'], dtype='object')
```

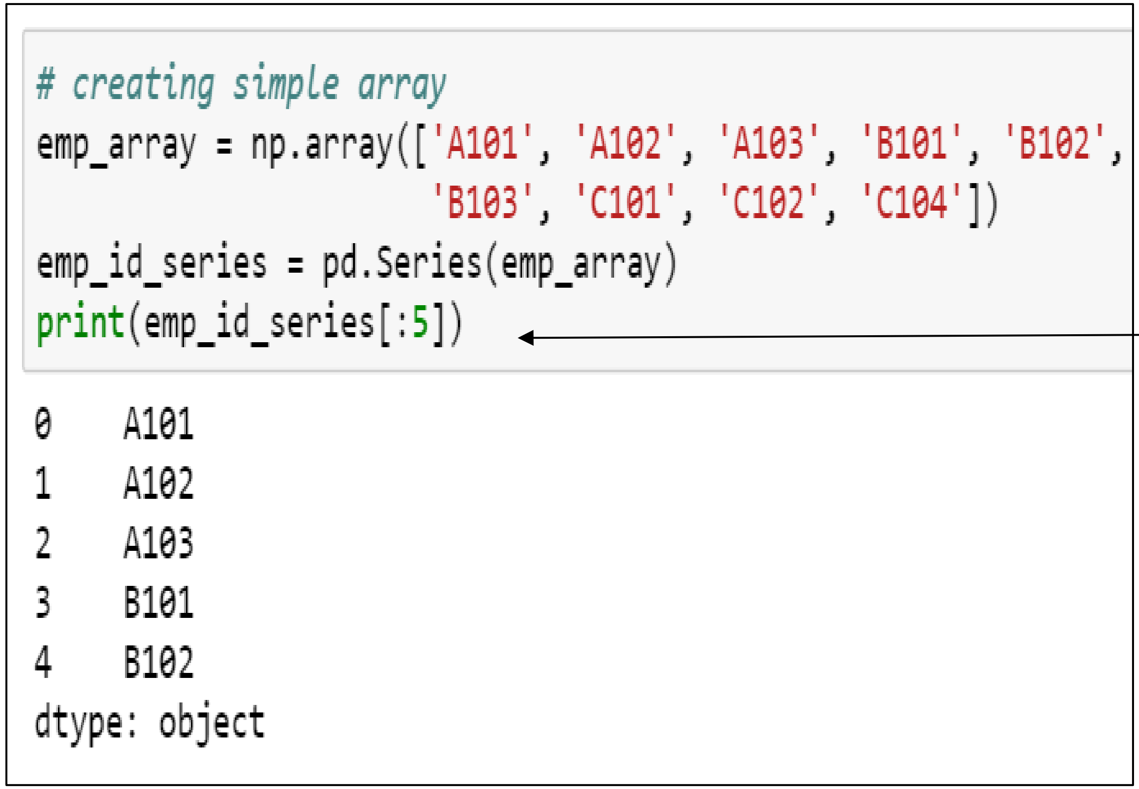
```
# display series values
classes_series.values
```

```
array([list(['Maths', 'Science', 'English']), list(['Maths', 'Science']),
       'Science'], dtype=object)
```

Accessing Series Elements

ACCESSING SERIES ELEMENTS

- Access the element in a series using the index operator **‘[]’**

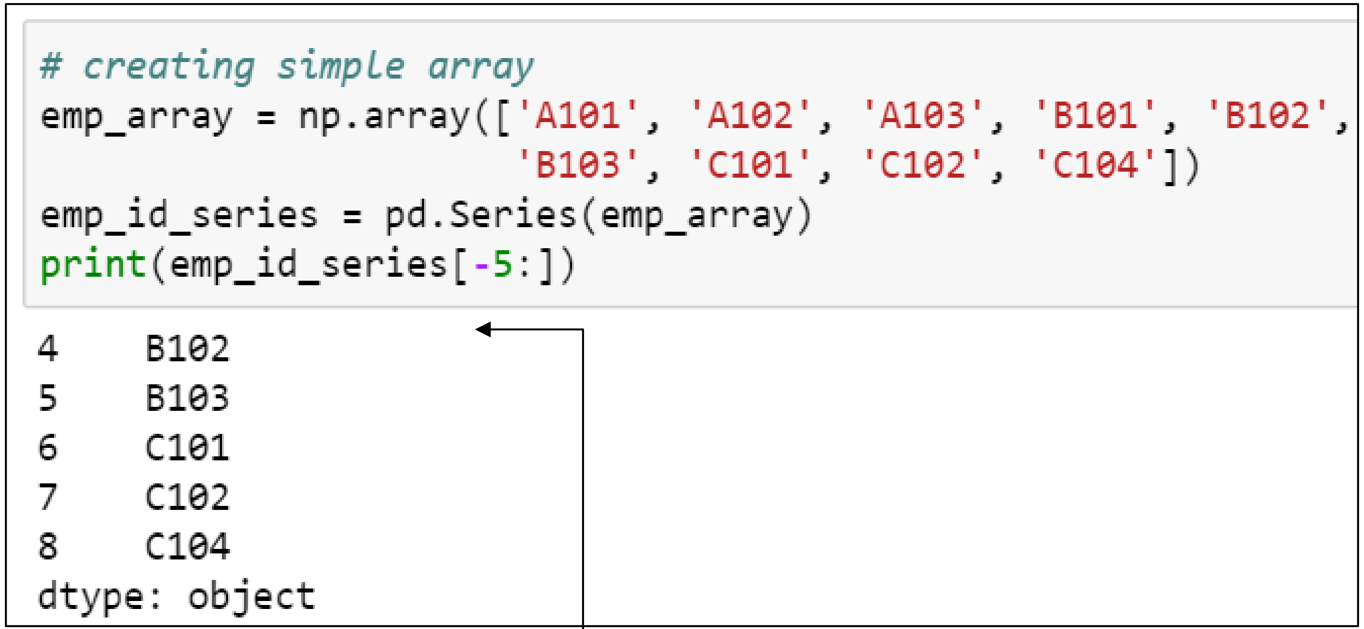


```
# creating simple array
emp_array = np.array(['A101', 'A102', 'A103', 'B101', 'B102',
                      'B103', 'C101', 'C102', 'C104'])
emp_id_series = pd.Series(emp_array)
print(emp_id_series[:5])
```

```
0    A101
1    A102
2    A103
3    B101
4    B102
dtype: object
```

Retrieve first five elements

ACCESSING SERIES ELEMENTS

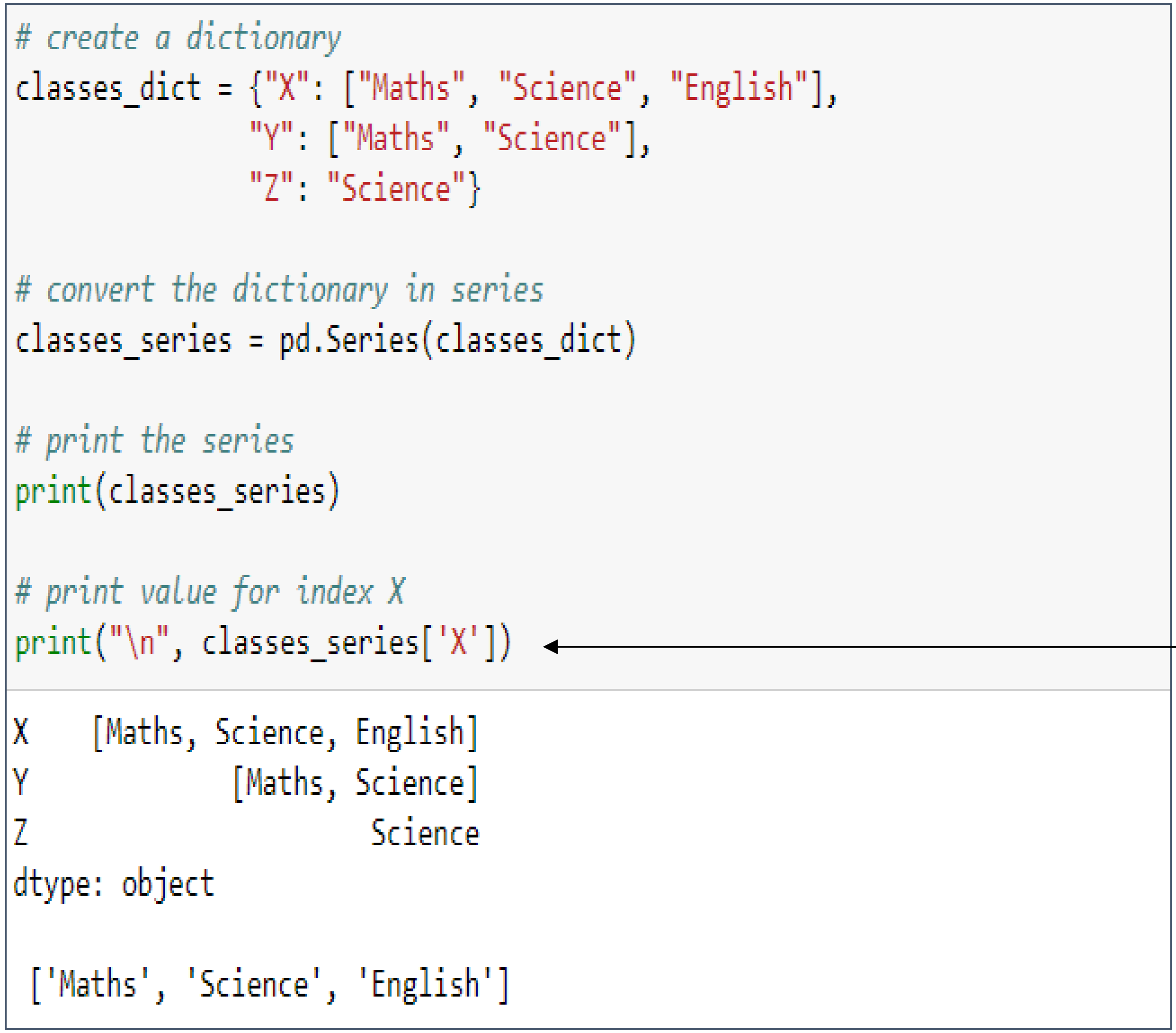


```
# creating simple array
emp_array = np.array(['A101', 'A102', 'A103', 'B101', 'B102',
                      'B103', 'C101', 'C102', 'C104'])
emp_id_series = pd.Series(emp_array)
print(emp_id_series[-5:])
```

```
4    B102
5    B103
6    C101
7    C102
8    C104
dtype: object
```

Retrieve last five elements

ACCESSING SERIES ELEMENTS

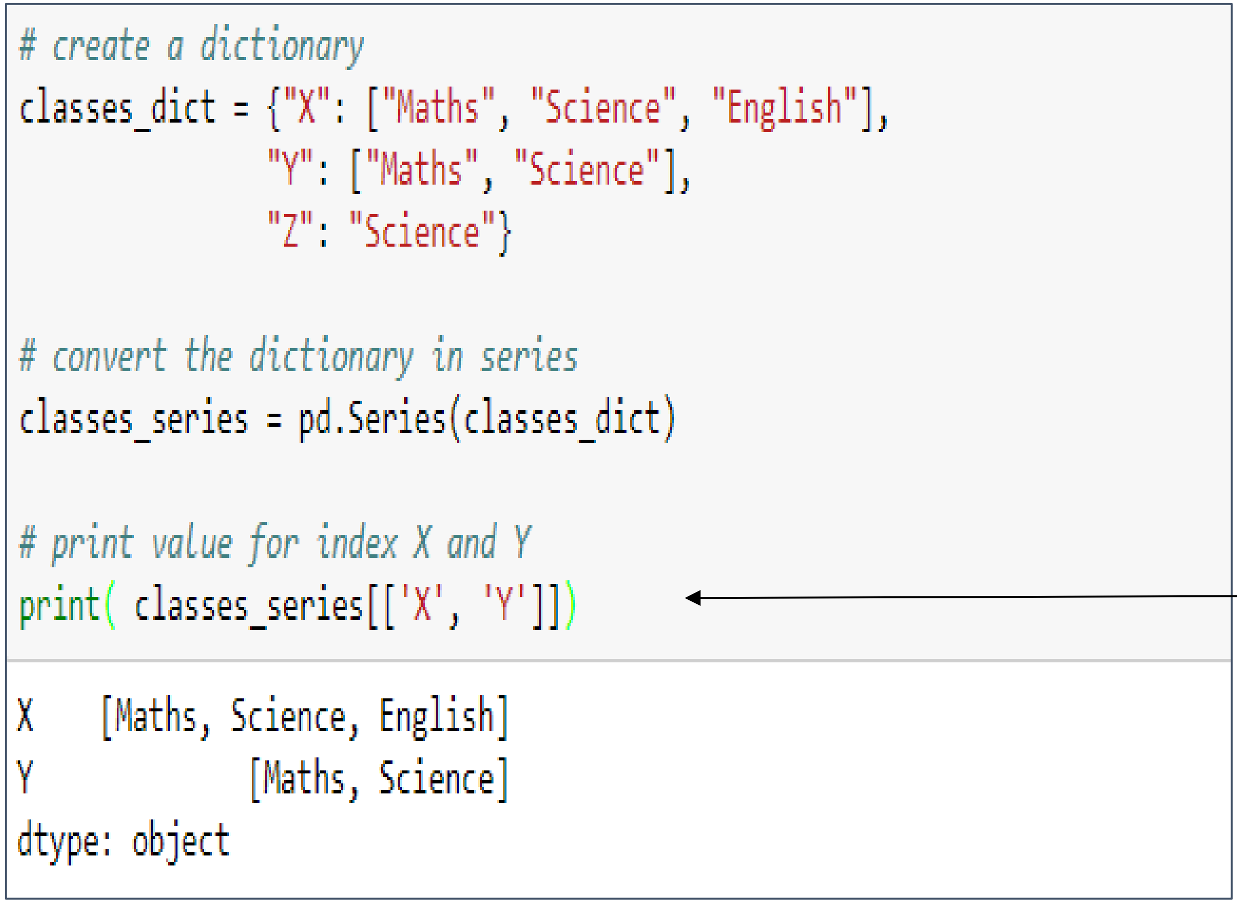


```python
# create a dictionary
classes_dict = {"X": ["Maths", "Science", "English"],
                "Y": ["Maths", "Science"],
                "Z": "Science"}

# convert the dictionary in series
classes_series = pd.Series(classes_dict)

# print the series
print(classes_series)

# print value for index X
print("\n", classes_series['X'])
```

```
X    [Maths, Science, English]
Y             [Maths, Science]
Z                      Science
dtype: object

 ['Maths', 'Science', 'English']
```

Use index to access the element

ACCESSING SERIES ELEMENTS



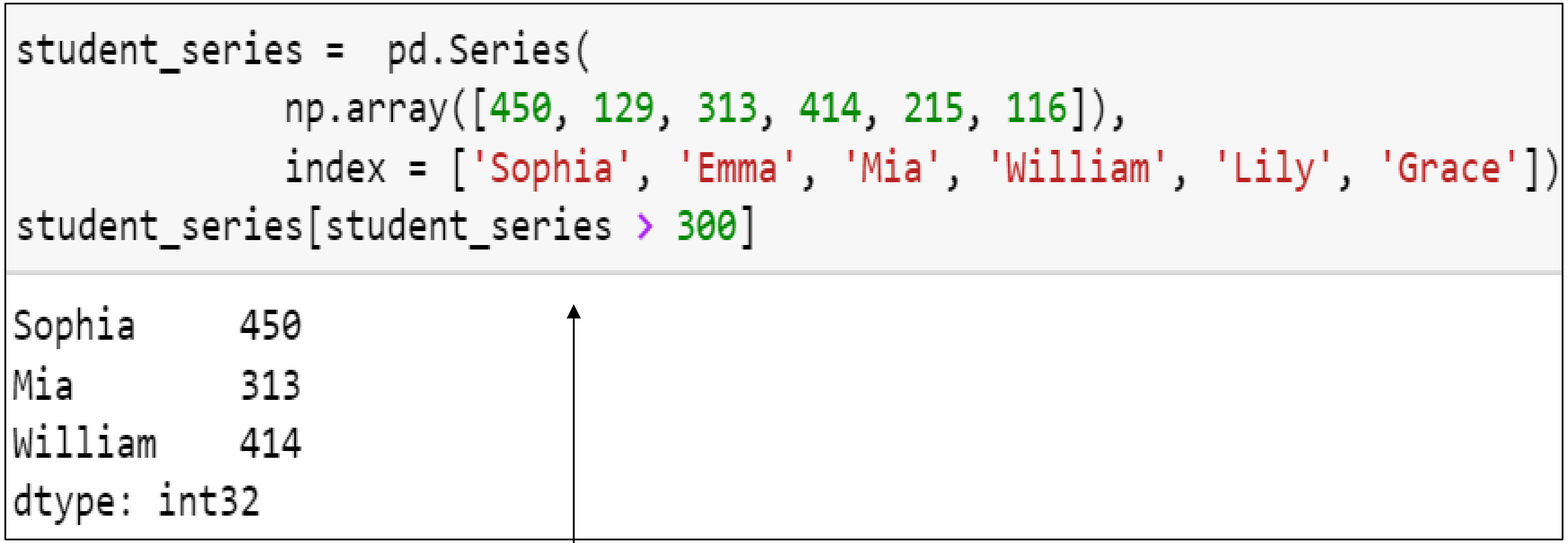
```
# create a dictionary
classes_dict = {"X": ["Maths", "Science", "English"],
                "Y": ["Maths", "Science"],
                "Z": "Science"}

# convert the dictionary in series
classes_series = pd.Series(classes_dict)

# print value for index X and Y
print( classes_series[['X', 'Y']])
```

```
X    [Maths, Science, English]
Y             [Maths, Science]
dtype: object
```

Retrieve multiple elements using a list of indices

Filtering a Series

FILTER THE VALUES



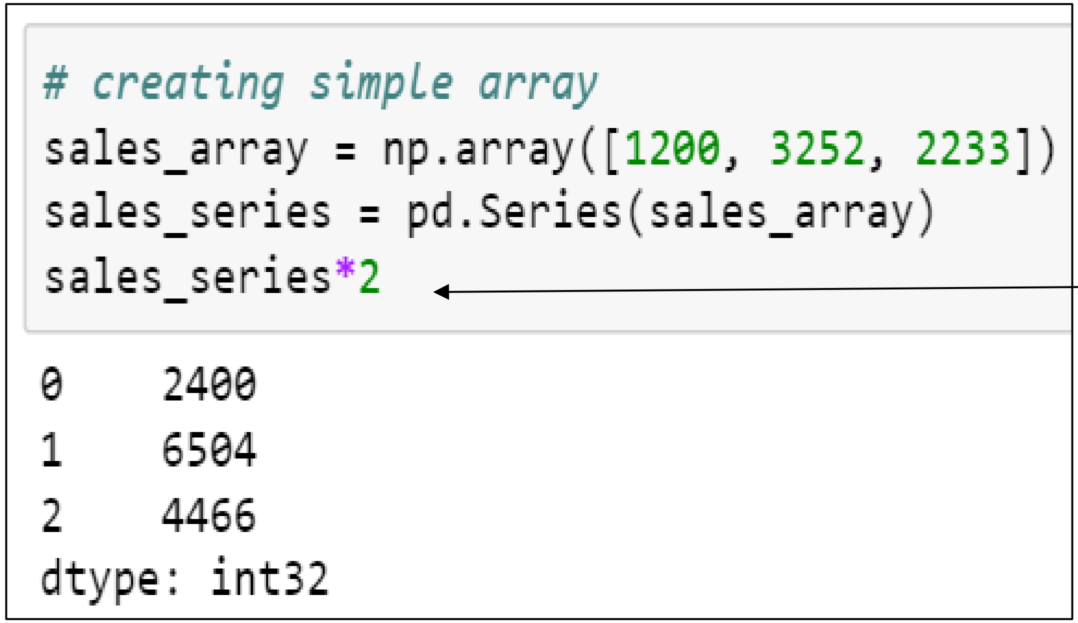
```
student_series =  pd.Series(
             np.array([450, 129, 313, 414, 215, 116]),
             index = ['Sophia', 'Emma', 'Mia', 'William', 'Lily', 'Grace'])
student_series[student_series > 300]
```

```
Sophia     450
Mia        313
William    414
dtype: int32
```

Filter all the values that are greater than 300

Arithmetic Operations

SCALAR – SERIES MULTIPLICATION



```
# creating simple array
sales_array = np.array([1200, 3252, 2233])
sales_series = pd.Series(sales_array)
sales_series*2
```

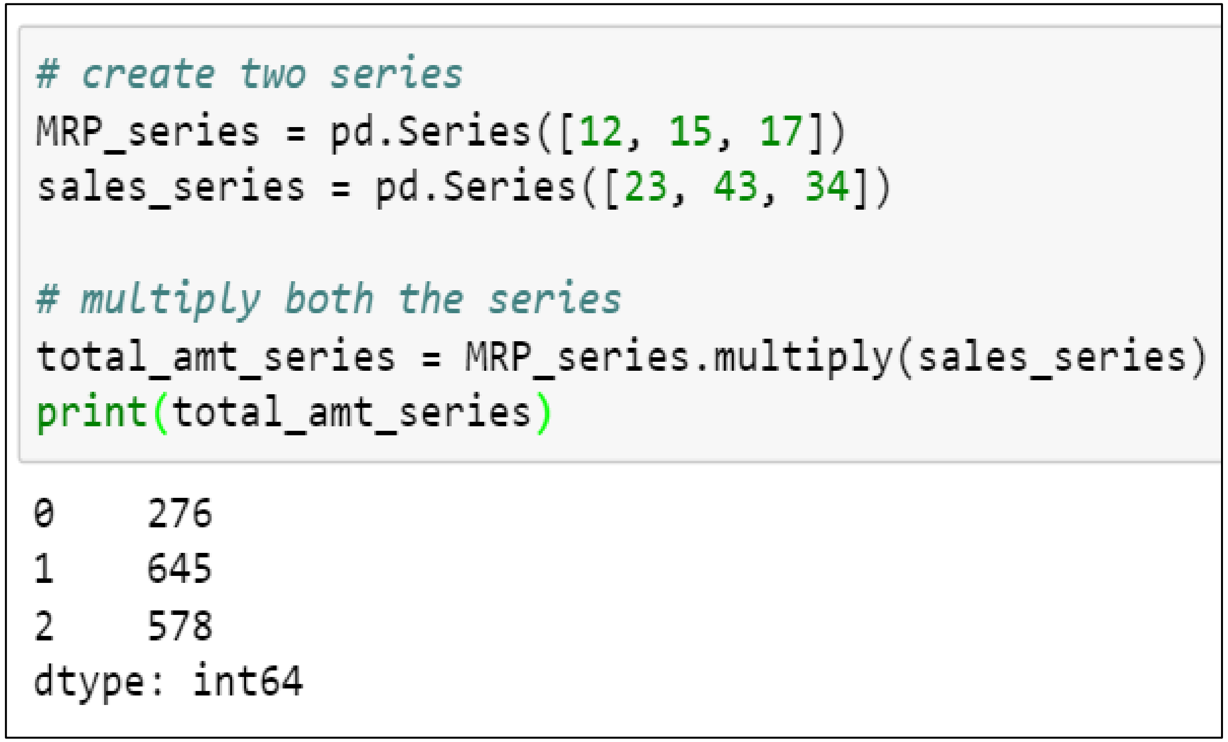
```
0    2400
1    6504
2    4466
dtype: int32
```

Use ‘*’ operator to perform multiplication

One can also use the multiply() method to perform the multiplication operation

SCALAR - SERIES MULTIPLICATION

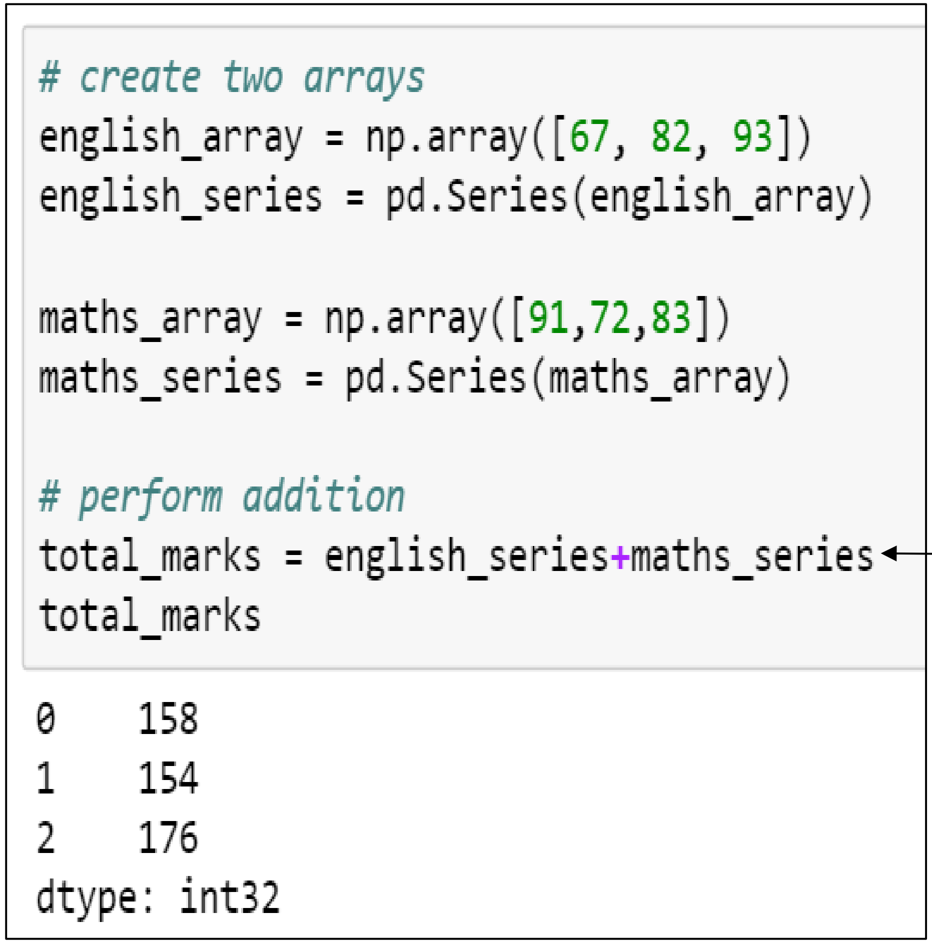
- The multiply() method returns the element-wise multiplication of the two series



```
# create two series
MRP_series = pd.Series([12, 15, 17])
sales_series = pd.Series([23, 43, 34])

# multiply both the series
total_amt_series = MRP_series.multiply(sales_series)
print(total_amt_series)
```

```
0    276
1    645
2    578
dtype: int64
```

ADDITION OF TWO SERIES



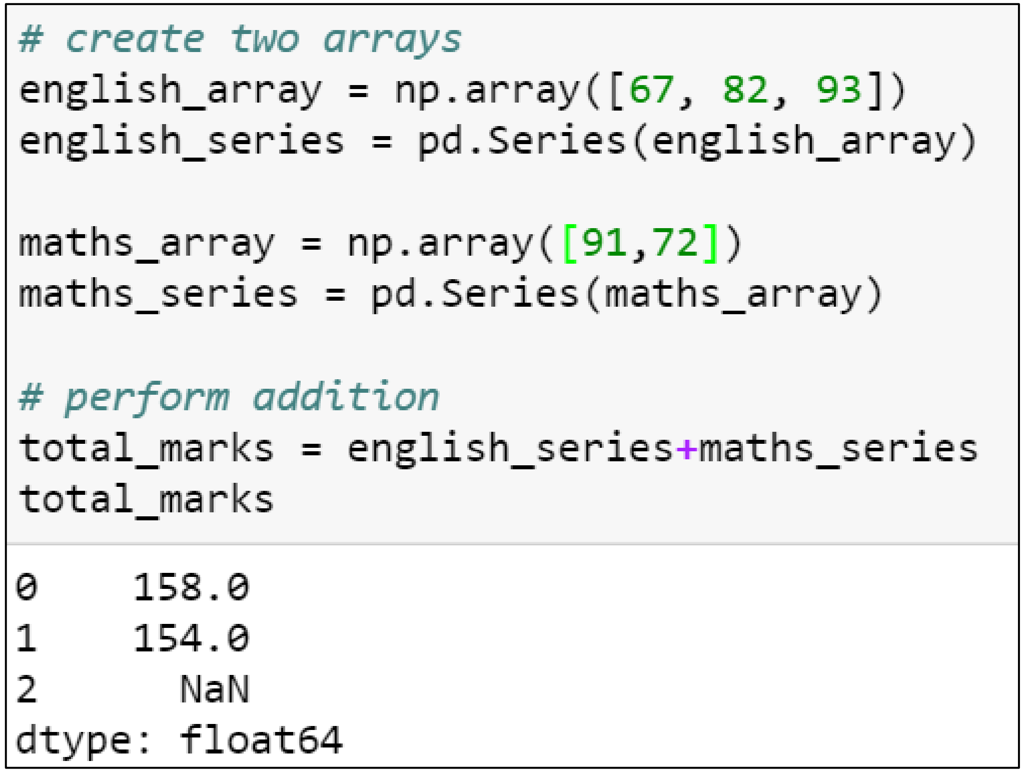
```
# create two arrays
english_array = np.array([67, 82, 93])
english_series = pd.Series(english_array)

maths_array = np.array([91,72,83])
maths_series = pd.Series(maths_array)

# perform addition
total_marks = english_series+maths_series
total_marks
```

```
0    158
1    154
2    176
dtype: int32
```

Use ‘+’ operator to perform addition

ADDITION OF TWO SERIES

- If the length of the two series are different, then the addition of such series shows the null values (NaN) for the indexes where the values are missing in one of the series

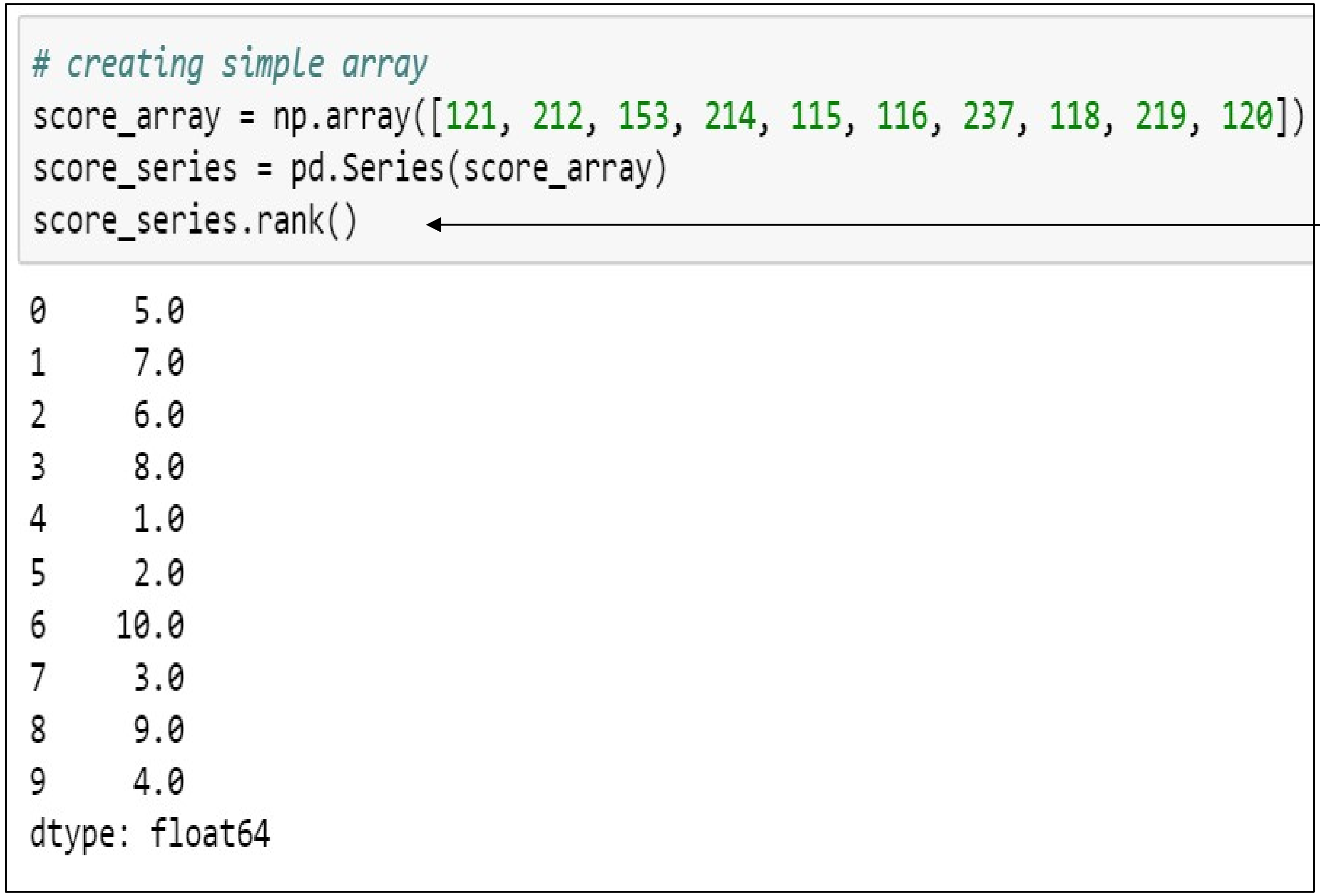


```
# create two arrays
english_array = np.array([67, 82, 93])
english_series = pd.Series(english_array)

maths_array = np.array([91,72])
maths_series = pd.Series(maths_array)

# perform addition
total_marks = english_series+maths_series
total_marks
```

```
0    158.0
1    154.0
2      NaN
dtype: float64
```

Series Ranking and Sorting

RANKING SERIES



```
# creating simple array
score_array = np.array([121, 212, 153, 214, 115, 116, 237, 118, 219, 120])
score_series = pd.Series(score_array)
score_series.rank()
```

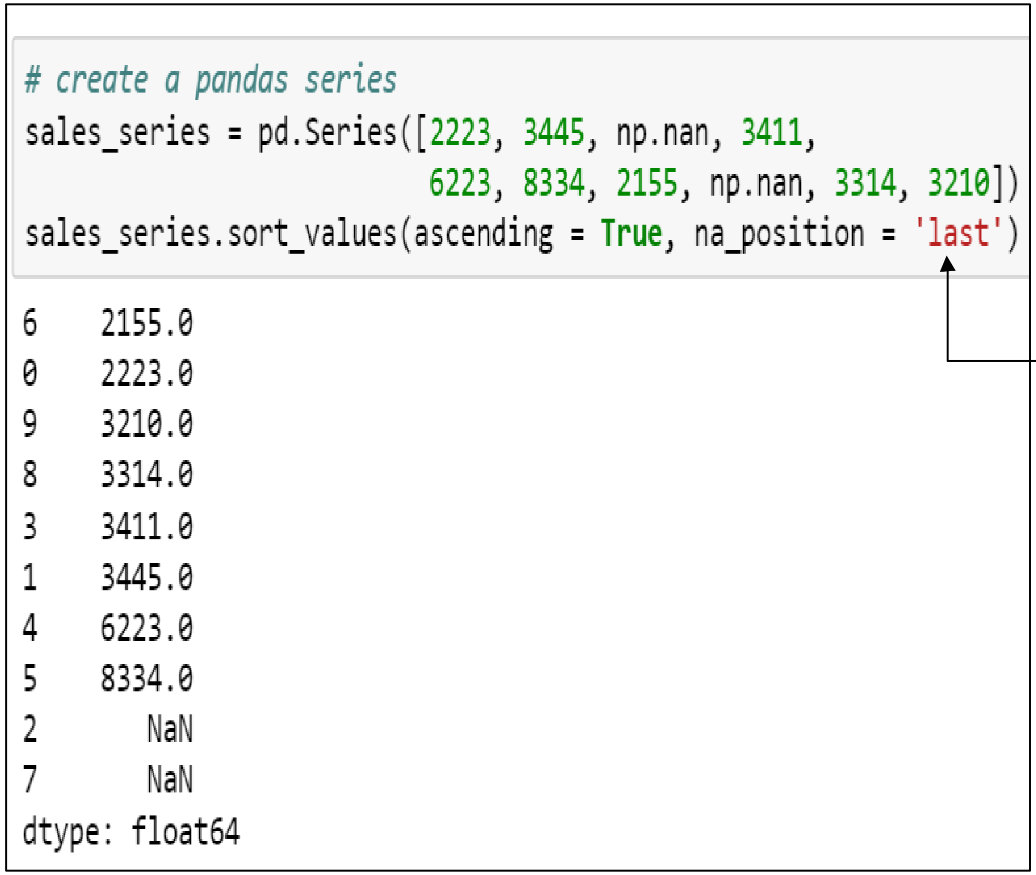
```
0     5.0
1     7.0
2     6.0
3     8.0
4     1.0
5     2.0
6    10.0
7     3.0
8     9.0
9     4.0
dtype: float64
```

Returns the rank of the underlying data

The rank() method, by default, returns the ranking in ascending order

SORTING SERIES

- The sort_values() method sorts the series by values in the series

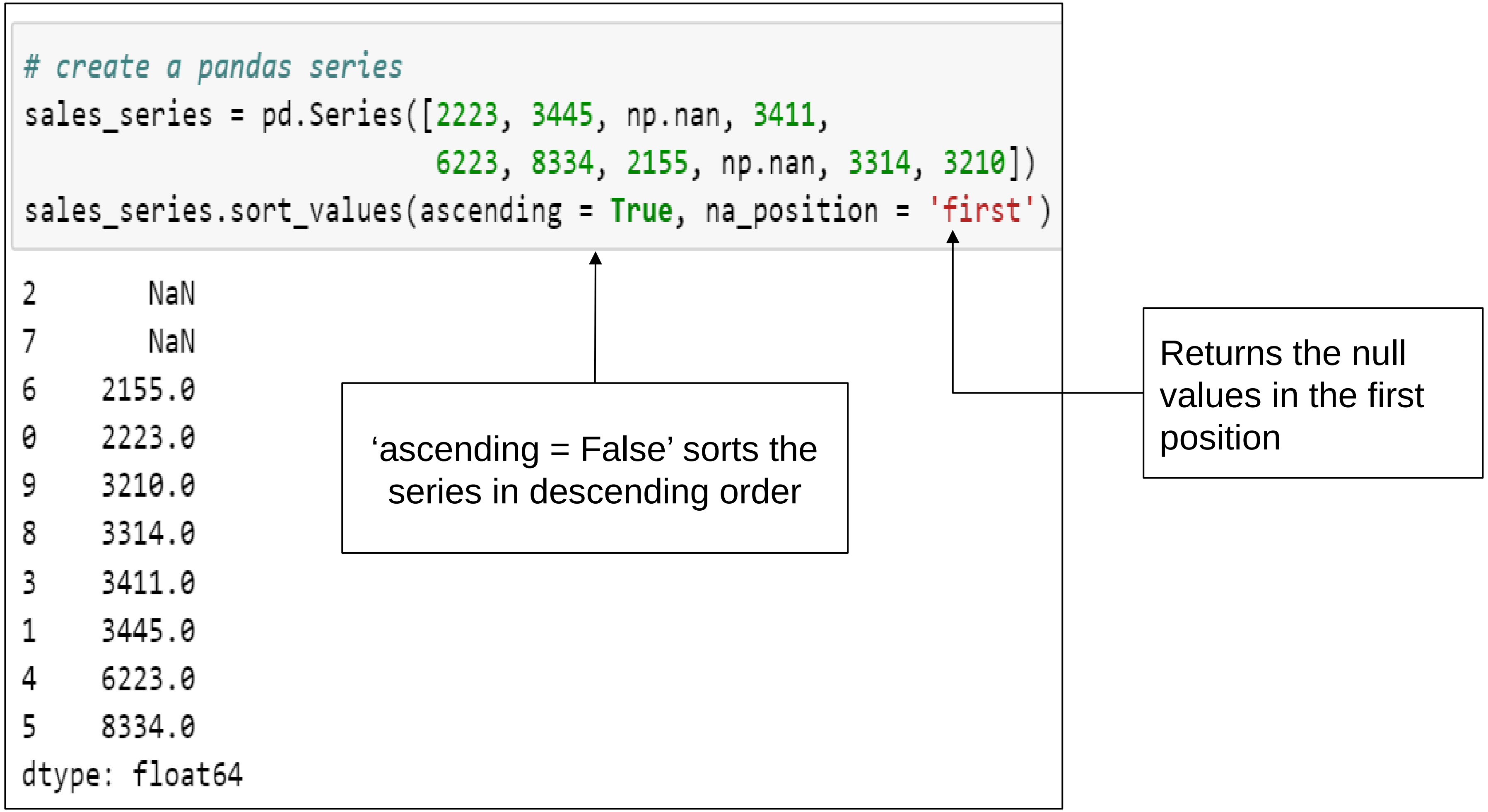


```
# create a pandas series
sales_series = pd.Series([2223, 3445, np.nan, 3411,
                          6223, 8334, 2155, np.nan, 3314, 3210])
sales_series.sort_values(ascending = True, na_position = 'last')
```

```
6    2155.0
0    2223.0
9    3210.0
8    3314.0
3    3411.0
1    3445.0
4    6223.0
5    8334.0
2       NaN
7       NaN
dtype: float64
```

Returns the null values in the last position

SORTING SERIES



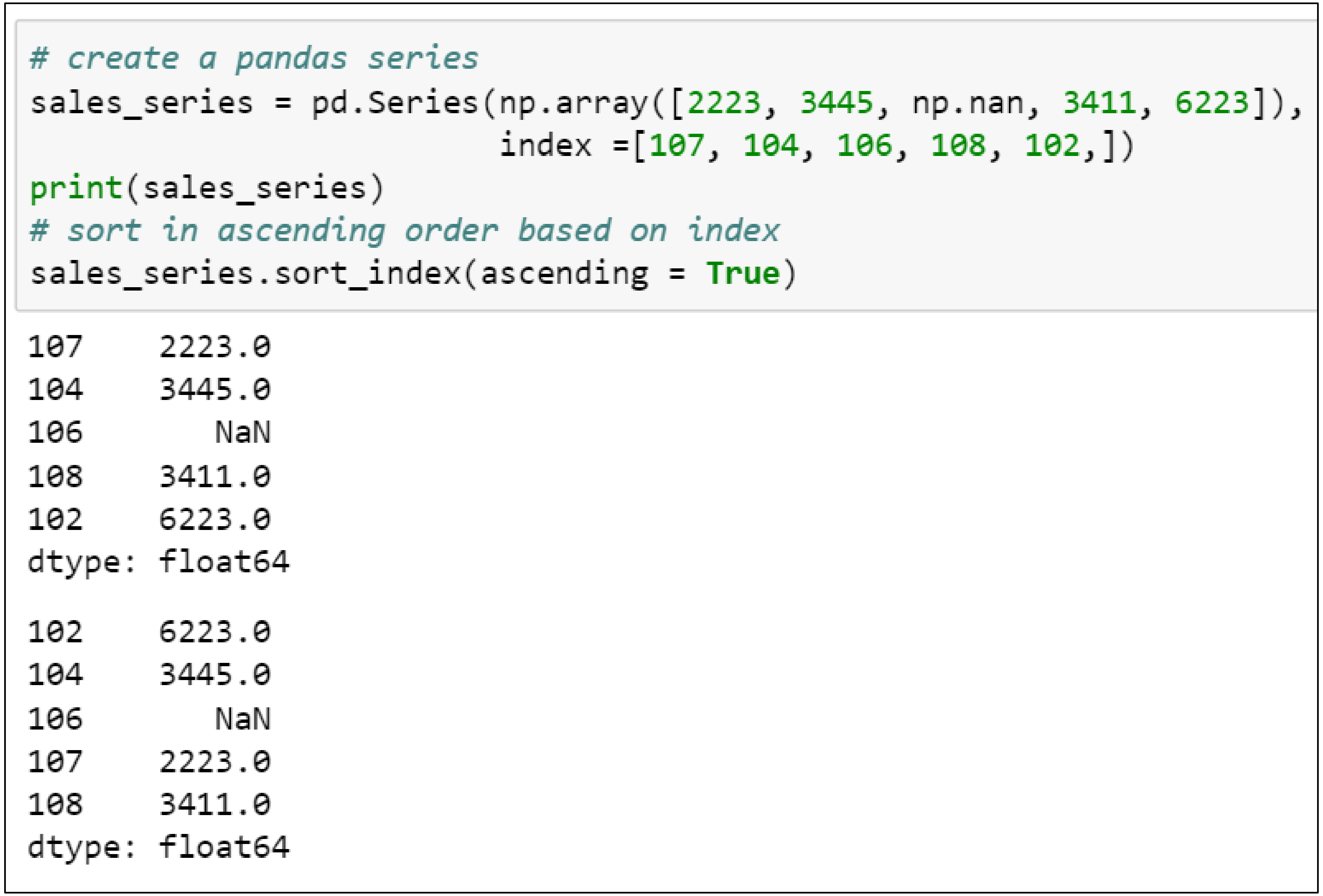
```
# create a pandas series
sales_series = pd.Series([2223, 3445, np.nan, 3411,
                          6223, 8334, 2155, np.nan, 3314, 3210])
sales_series.sort_values(ascending = True, na_position = 'first')
```

```
2       NaN
7       NaN
6    2155.0
0    2223.0
9    3210.0
8    3314.0
3    3411.0
1    3445.0
4    6223.0
5    8334.0
dtype: float64
```

‘ascending = False’ sorts the series in descending order

Returns the null values in the first position

SORTING SERIES



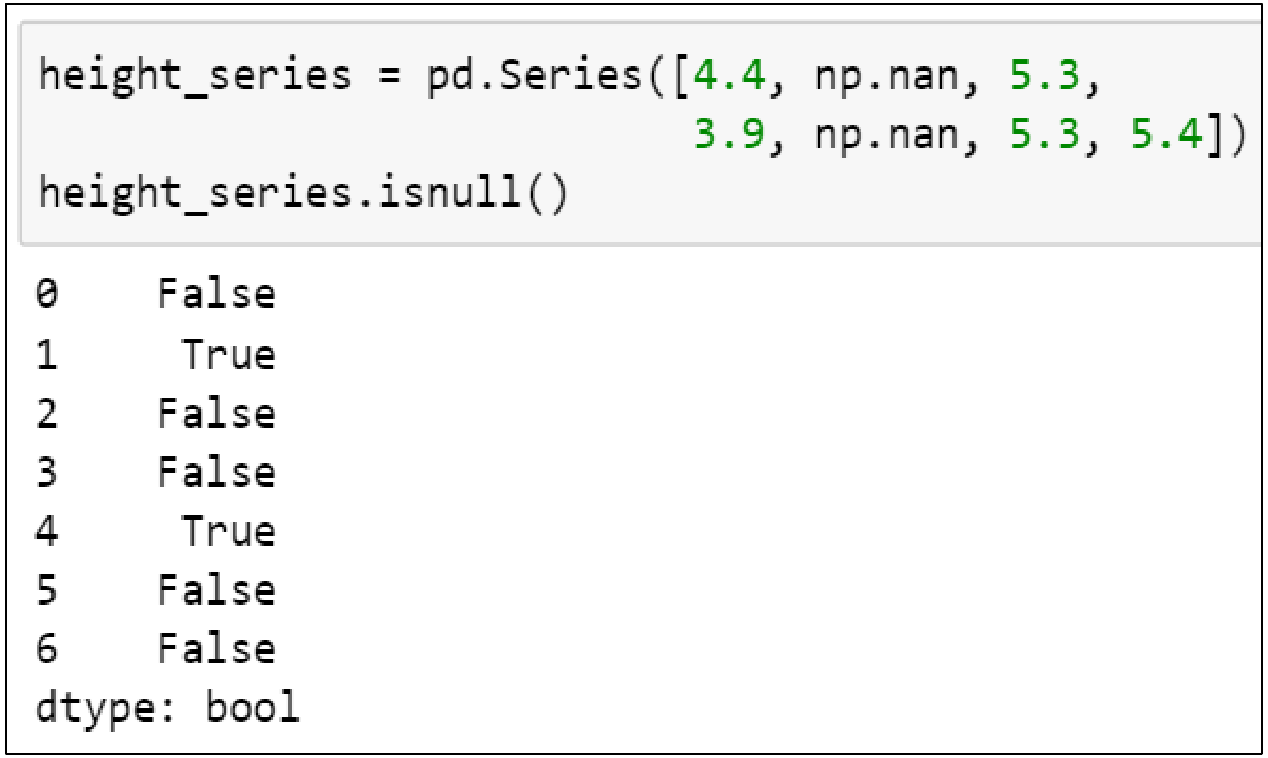
```
# create a pandas series
sales_series = pd.Series(np.array([2223, 3445, np.nan, 3411, 6223]),
                         index =[107, 104, 106, 108, 102,])
print(sales_series)
# sort in ascending order based on index
sales_series.sort_index(ascending = True)
```

```
107    2223.0
104    3445.0
106       NaN
108    3411.0
102    6223.0
dtype: float64

102    6223.0
104    3445.0
106       NaN
107    2223.0
108    3411.0
dtype: float64
```

Checking Null Values

CHECKING NULL VALUES

- The isnull() method returns the boolean output indicating the presence of null values
- ‘True’ value indicates that the corresponding value is null



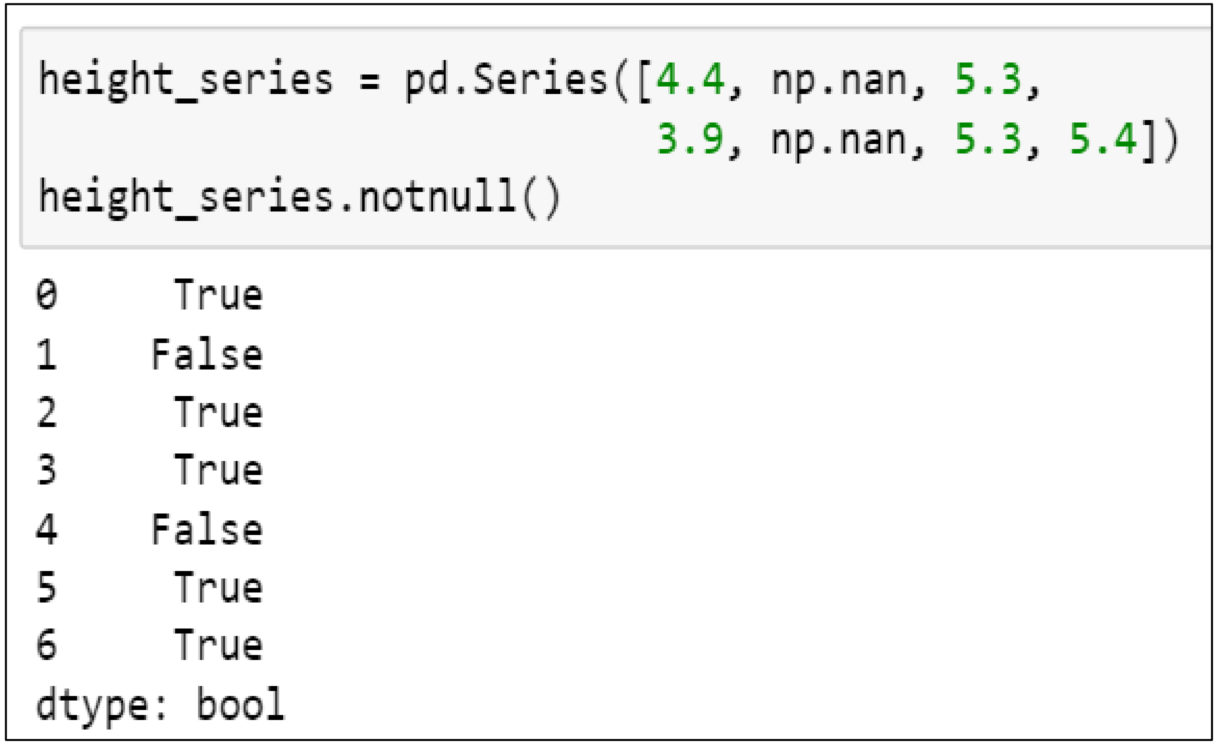
```
height_series = pd.Series([4.4, np.nan, 5.3,
                           3.9, np.nan, 5.3, 5.4])
height_series.isnull()
```

```
0    False
1     True
2    False
3    False
4     True
5    False
6    False
dtype: bool
```

CHECKING NULL VALUES

- The notnull() method returns the boolean output indicating the presence of non-null values

- ‘False’ in the output indicates that the corresponding value is null

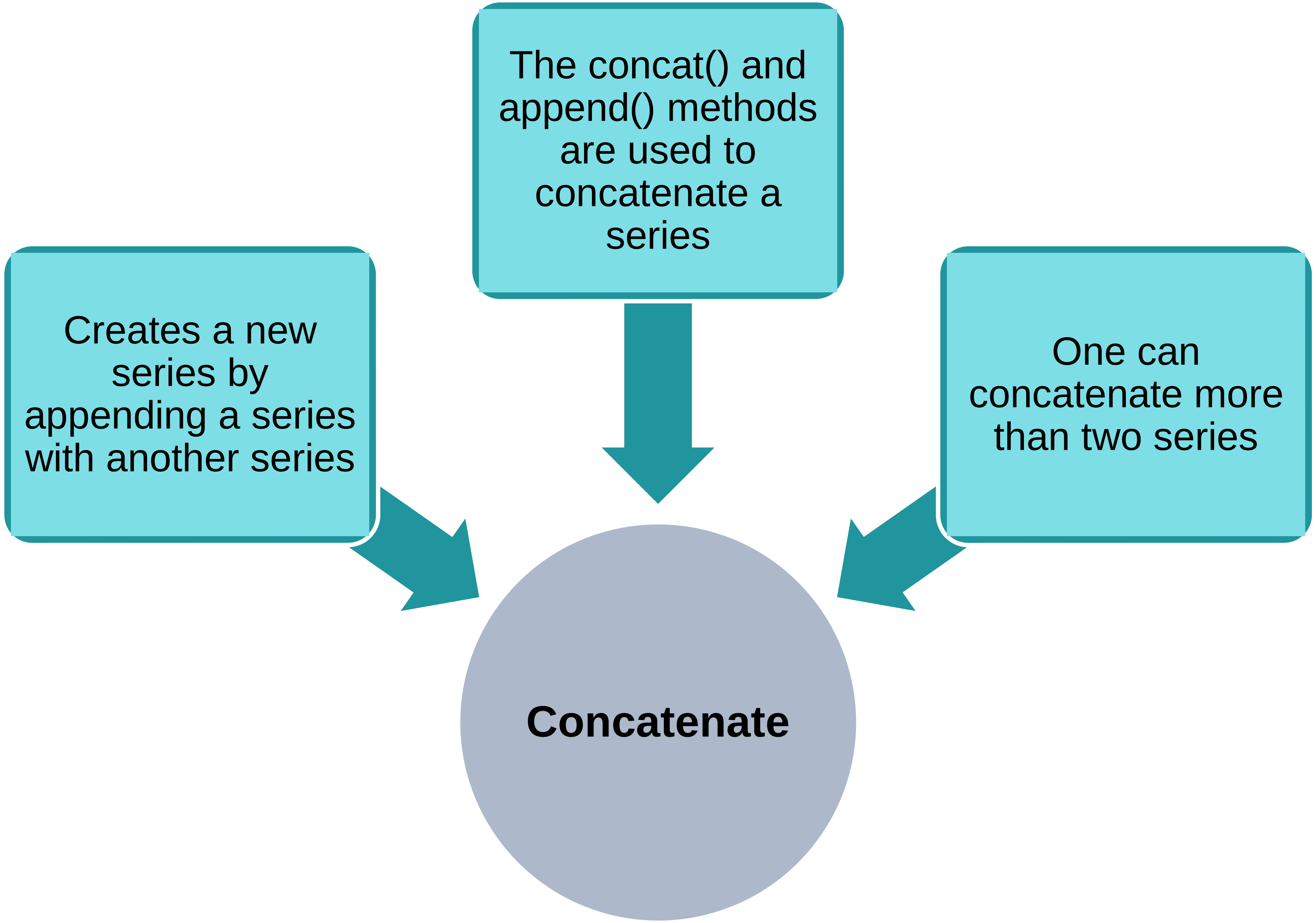


```
height_series = pd.Series([4.4, np.nan, 5.3,
                           3.9, np.nan, 5.3, 5.4])
height_series.notnull()
```

```
0     True
1    False
2     True
3     True
4    False
5     True
6     True
dtype: bool
```

Concatenate a Series

CONCATENATE A SERIES



The concat() and append() methods are used to concatenate a series

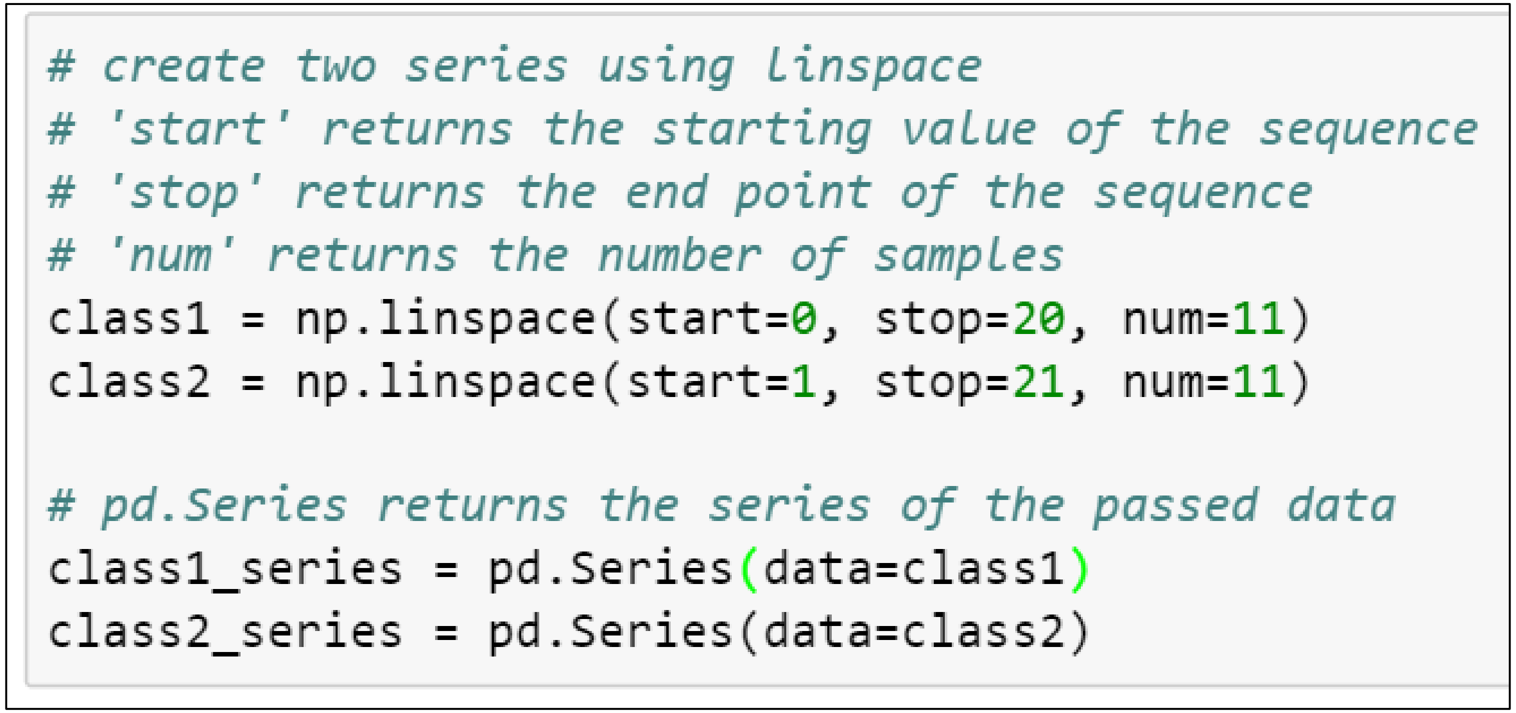
Creates a new series by appending a series with another series

One can concatenate more than two series

Concatenate

CREATE A SERIES

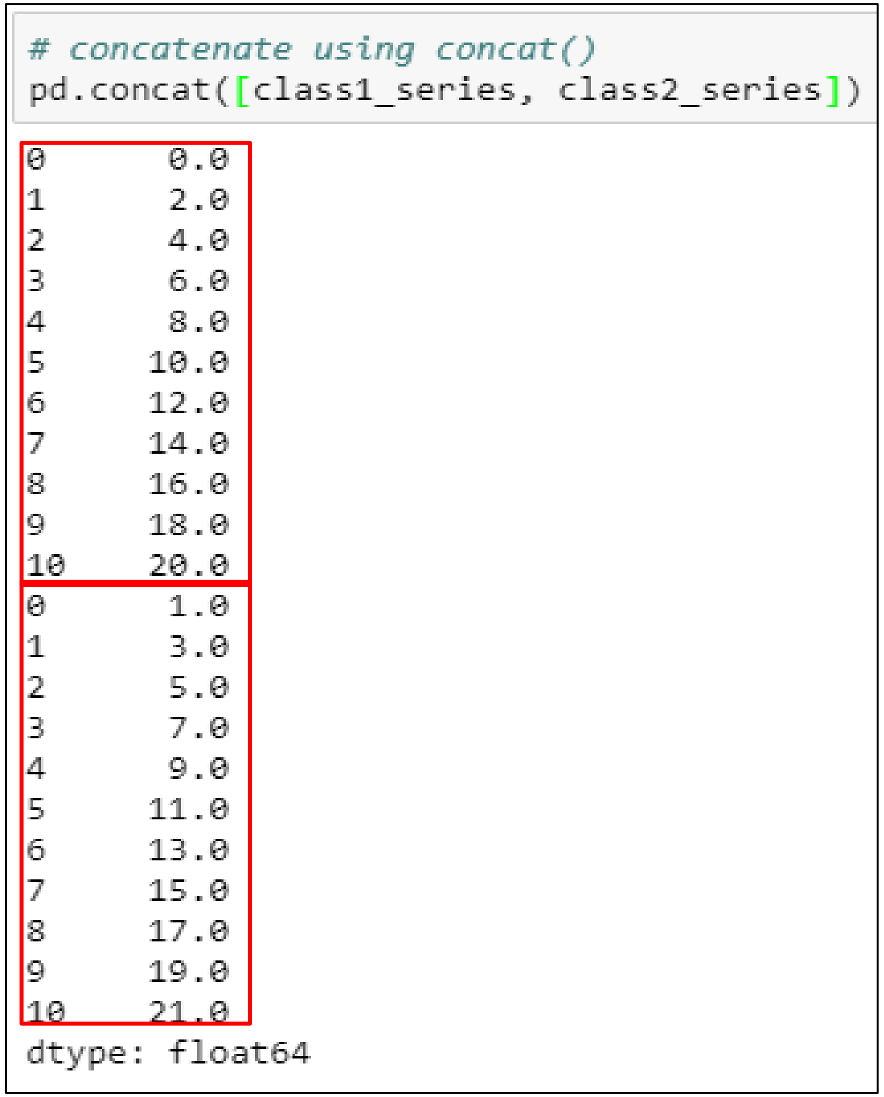
- Create python series as shown below:



```
# create two series using linspace
# 'start' returns the starting value of the sequence
# 'stop' returns the end point of the sequence
# 'num' returns the number of samples
class1 = np.linspace(start=0, stop=20, num=11)
class2 = np.linspace(start=1, stop=21, num=11)

# pd.Series returns the series of the passed data
class1_series = pd.Series(data=class1)
class2_series = pd.Series(data=class2)
```

CONCATENATE A SERIES

- Creates a new series by appending a series with another series

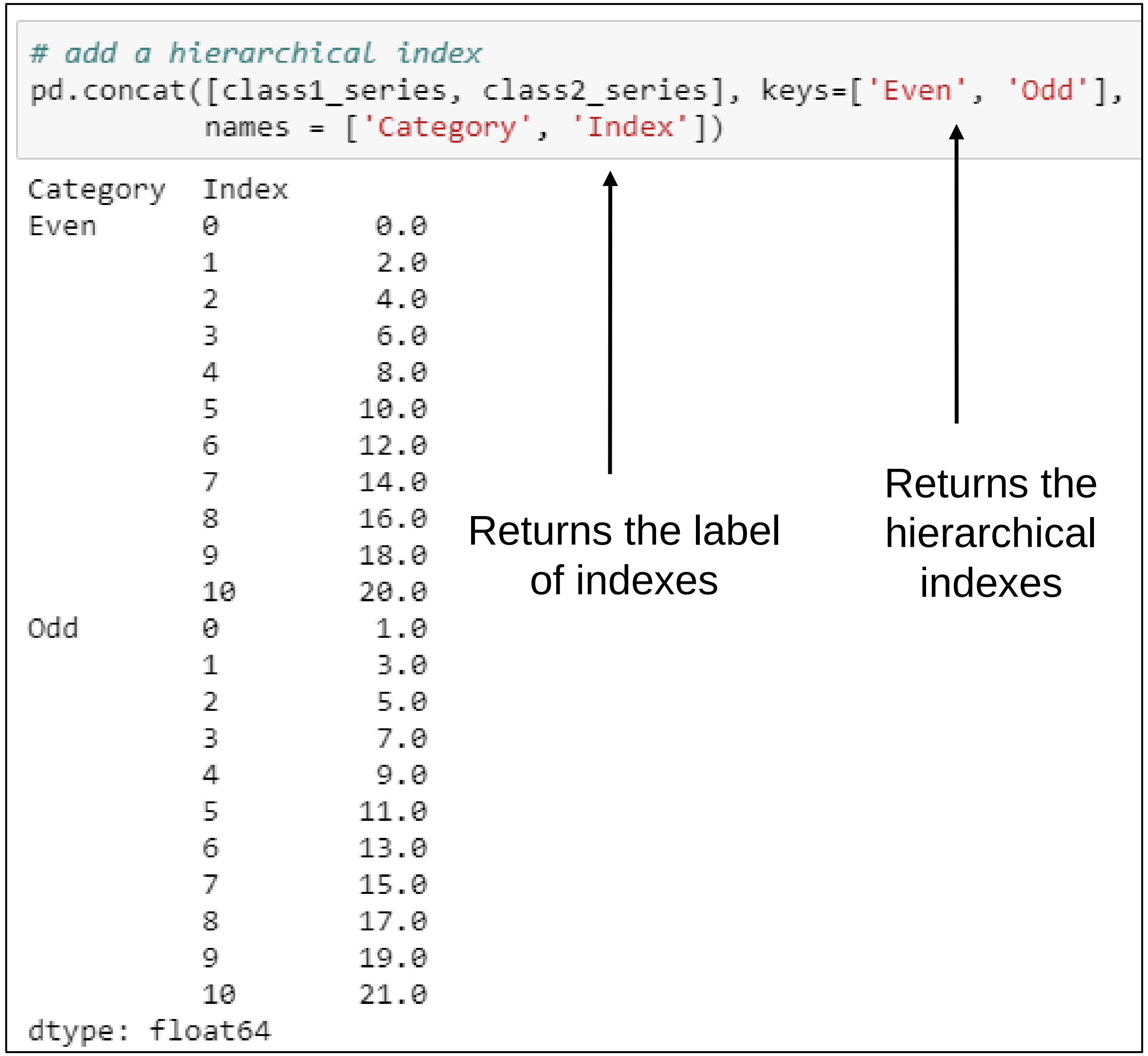


```
# concatenate using concat()
pd.concat([class1_series, class2_series])
```

```
0      0.0
1      2.0
2      4.0
3      6.0
4      8.0
5     10.0
6     12.0
7     14.0
8     16.0
9     18.0
10    20.0
0      1.0
1      3.0
2      5.0
3      7.0
4      9.0
5     11.0
6     13.0
7     15.0
8     17.0
9     19.0
10    21.0
dtype: float64
```

ADD HIERARCHICAL INDEX AND LABEL THE INDEX

- Add the hierarchical indexes and labels while concatenating two series



```
# add a hierarchical index
pd.concat([class1_series, class2_series], keys=['Even', 'Odd'],
          names = ['Category', 'Index'])
```

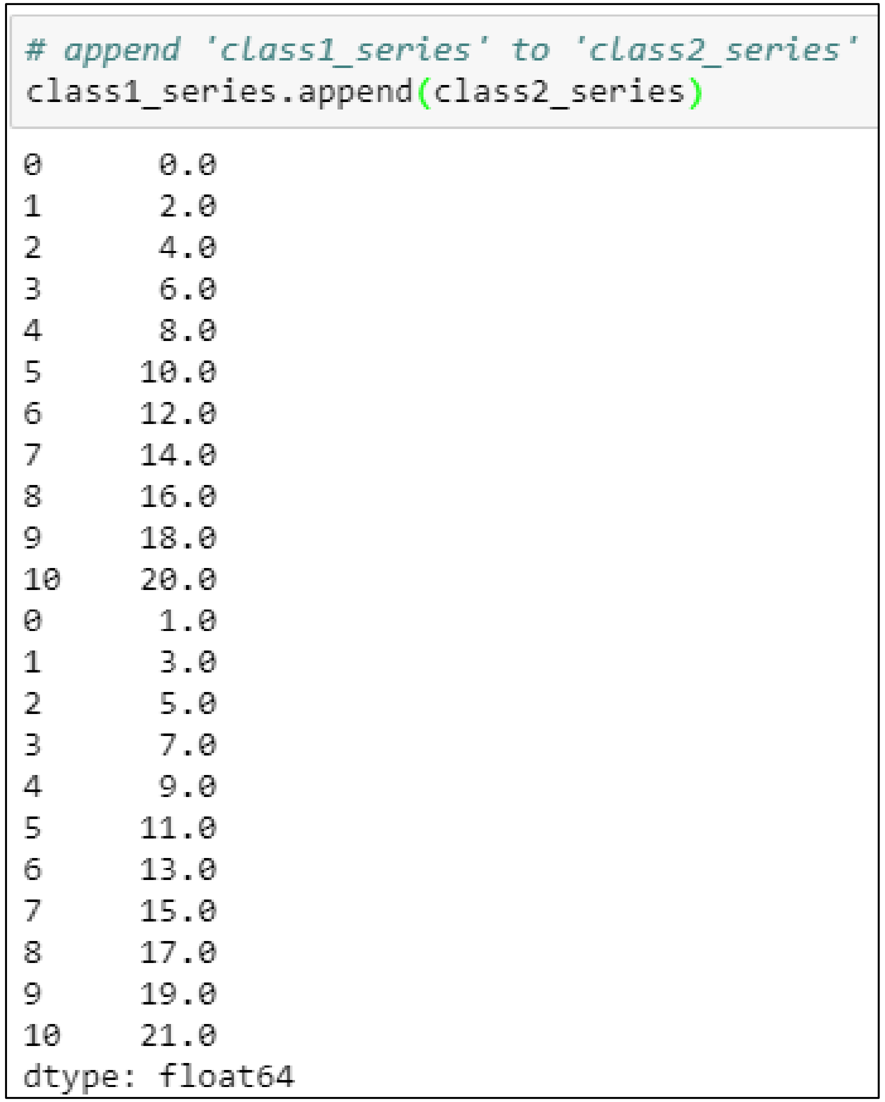
```
Category  Index
Even      0         0.0
          1         2.0
          2         4.0
          3         6.0
          4         8.0
          5        10.0
          6        12.0
          7        14.0
          8        16.0
          9        18.0
          10       20.0
Odd       0         1.0
          1         3.0
          2         5.0
          3         7.0
          4         9.0
          5        11.0
          6        13.0
          7        15.0
          8        17.0
          9        19.0
          10       21.0
dtype: float64
```

Returns the label of indexes

Returns the hierarchical indexes

CONCATENATE A SERIES

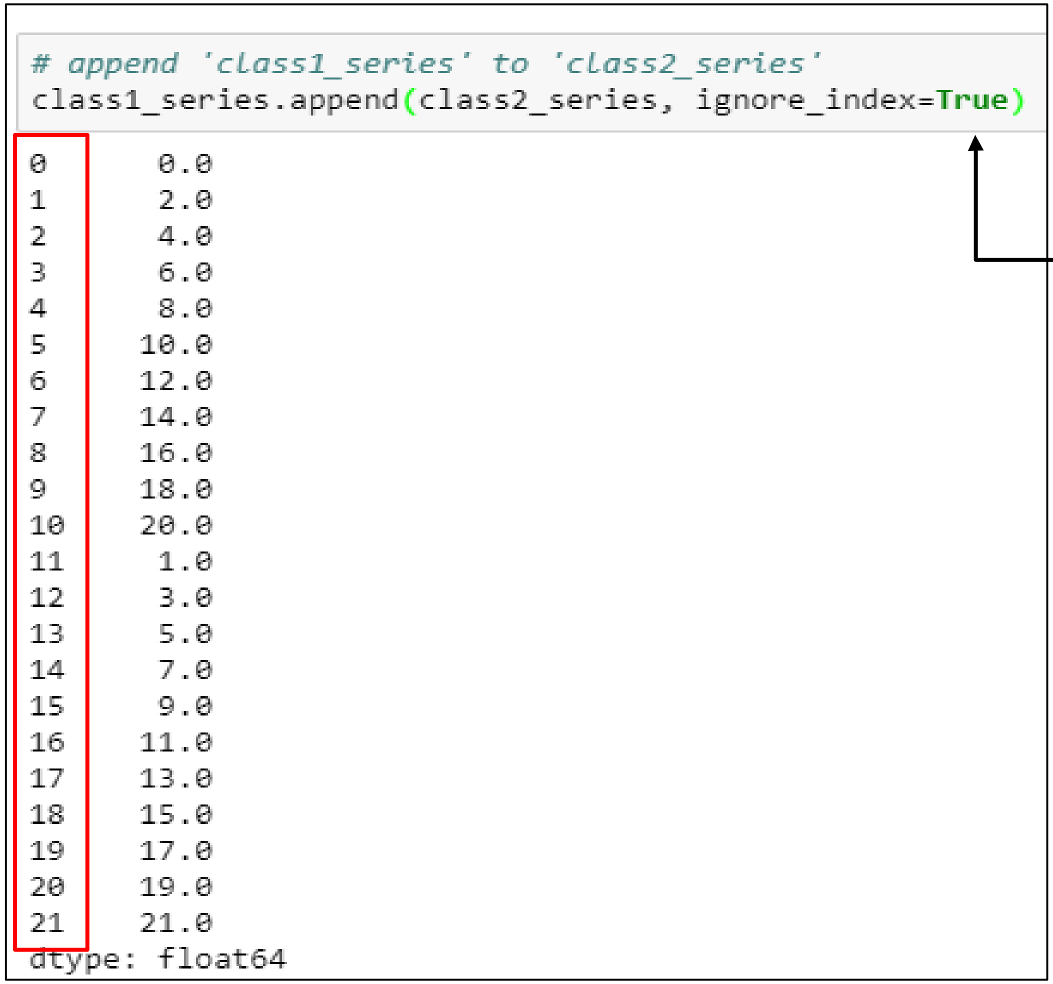
- The append() method is used append a series with another
- Here, we append the ‘class1_series’ to ‘class2_series’
- Appended indexes are same as the original series



```
# append 'class1_series' to 'class2_series'
class1_series.append(class2_series)
```

```
0      0.0
1      2.0
2      4.0
3      6.0
4      8.0
5     10.0
6     12.0
7     14.0
8     16.0
9     18.0
10    20.0
0      1.0
1      3.0
2      5.0
3      7.0
4      9.0
5     11.0
6     13.0
7     15.0
8     17.0
9     19.0
10    21.0
dtype: float64
```

CONCATENATE A SERIES



```
# append 'class1_series' to 'class2_series'
class1_series.append(class2_series, ignore_index=True)
```

```
0      0.0
1      2.0
2      4.0
3      6.0
4      8.0
5     10.0
6     12.0
7     14.0
8     16.0
9     18.0
10    20.0
11     1.0
12     3.0
13     5.0
14     7.0
15     9.0
16    11.0
17    13.0
18    15.0
19    17.0
20    19.0
21    21.0
dtype: float64
```

Ignores the index labels of original series