# Let's get Started

## Introduction

The Pandas Interview Questions and Answers revolve around the tool's features, data structures, and functions in **Python interviews**. It is widely used in **data science** and **machine learning** projects, as well as in industries such as finance, healthcare, and marketing. Pandas provides a wide range of functionalities, including data loading, cleaning, filtering, transforming, merging, grouping, and aggregating.

For those seeking a **career in data science** or related fields, it's important to have a good understanding of Pandas and their applications. Therefore, it's common for job interviews in these fields to include questions about Pandas. These questions can range from basic to advanced and cover various topics, such as data structures, indexing, merging and joining, groupby operations, and time series analysis.

Whether you are a beginner or an experienced Python programmer, this article will help you prepare for your next Pandas-related job interview. In this article, we will explore some commonly asked **Pandas Interview Questions and Answers** which are divided into the following sections:



## Pandas Basic Interview Questions

### 1. What is Pandas in Python?

Pandas is an open-source Python package that is most commonly used for data science, data analysis, and machine learning tasks. It is built on top of another library named **Numpy**. It provides various data structures and operations for manipulating numerical data and time series and is very efficient in performing various functions like data visualization, data manipulation, data analysis, etc.

## 2. Mention the different types of Data Structures in Pandas?

Pandas have three different types of data structures. It is due to these simple and flexible data structures that it is fast and efficient.

- **Series** - It is a one-dimensional array-like structure with homogeneous data which means data of different data types cannot be a part of the same series. It can hold any data type such as **integers**, **floats**, and **strings** and its values are mutable i.e. it can be changed but the size of the series is immutable i.e. it cannot be changed.
- **DataFrame** - It is a two-dimensional array-like structure with heterogeneous data. It can contain data of different data types and the data is aligned in a tabular manner. Both size and values of DataFrame are mutable.
- **Panel** - The Pandas have a third type of data structure known as Panel, which is a 3D data structure capable of storing heterogeneous data but it isn't that widely used.

## 3. What are the significant features of the pandas Library?

Pandas library is known for its efficient data analysis and state-of-the-art data visualization.
The key features of the panda's library are as follows:

- Fast and efficient DataFrame object with default and customized indexing.
- High-performance merging and joining of data.
- Data alignment and integrated handling of missing data.
- Label-based slicing, indexing, and subsetting of large data sets.
- Reshaping and pivoting of data sets.
- Tools for loading data into in-memory data objects from different file formats.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- Time Series functionality.

## 4. Define Series in Pandas?

It is a one-dimensional array-like structure with homogeneous data which means data of different data types cannot be a part of the same series. It can hold any data type such as **integers**, **floats**, and **strings** and its values are mutable i.e. it can be changed but the size of the series is immutable i.e. it cannot be changed. By using a 'series' method, we can easily convert the list, tuple, and dictionary into a series. A Series cannot contain multiple columns.

## 5. Define DataFrame in Pandas?

It is a two-dimensional array-like structure with heterogeneous data. It can contain data of different data types and the data is aligned in a tabular manner i.e. in rows and columns and the indexes with respect to these are called row index and column index respectively. Both size and values of DataFrame are mutable. The columns can be heterogeneous types like int and bool. It can also be defined as a dictionary of Series.

The **syntax** for creating a dataframe:

```
import pandas as pd
dataframe = pd.DataFrame( data, index, columns, dtype)
```

Here:

- **data** - It represents various forms like series, map, ndarray, lists, dict, etc.
- **index** - It is an optional argument that represents an index to row labels.
- **columns** - Optional argument for column labels.
- **Dtype** - It represents the data type of each column. It is an optional parameter.

## 6. What are the different ways in which a series can be created?

There are different ways of creating a series in Pandas.

- **Creating an empty Series:** The simplest series that can be created is an empty series. The `Series()` function of Pandas is used to create a series of any kind.

**Code Example 1:**

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

# Creating empty Series
ser = pd.Series()

print(ser)
```

**Output:**

```
Series([], dtype: float64)
```

- **Creating a series from an array:** Pandas is built on top of the Numpy library. In order to create a series from the NumPy array, we have to import the NumPy module and have to use `numpy.array()` the function.

**Code Example 2:**

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

# import numpy as np
import numpy as np

# simple array
data = np.array(['s', 'c', 'a', 'l', 'a','r'])

ser = pd.Series(data)
print(ser)
```

**Output:**

```
0    s
1    c
2    a
3    l
4    a
5    r
dtype: object
```

- **Creating a series from the array with an index:** In order to create a series by exclusively providing an index instead of the default value we need to provide a list of elements to the index parameter with the same number of elements as given in the array.

**Code Example 3:**

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

# import numpy as np
import numpy as np

# simple array
data = np.array(['s', 'c', 'a', 'l', 'a','r'])

# providing an index
ser = pd.Series(data, index=[10, 11, 12, 13, 14,15])
print(ser)
```

**Output:**

```
10    s
11    c
12    a
13    l
14    a
15    r
dtype: object
```

- **Creating a series from Lists:** In order to create a series from a list, the first step is to create a list, and then we need to create a series from the given list.

**Code Example 4:**

```
import pandas as pd

# a simple list
list = ['s', 'c', 'a', 'l', 'a','r']

# create series form a list
ser = pd.Series(list)
print(ser)
```

**Output:**

```
0    s
1    c
2    a
3    l
4    a
5    r
dtype: object
```

- **Creating a series from Dictionary:** In order to create a series from the dictionary, the first step is to create a dictionary, and only then we can create a series using. The dictionary keys serve as indexes for the Series.

**Code Example 5:**

```
import pandas as pd

# a simple dictionary
dict = {'A': 101,
        'B': 202,
        'C': 303}

# create series from dictionary
ser = pd.Series(dict)

print(ser)
```

**Output:**

```
A    101
B    202
C    303
dtype: int64
```

- **Creating a series from Scalar value:** In order to create a series from scalar value, an index must be provided. The value repeats itself to fit the length of the series or index given in general.

**Code Example 6:**

```
import pandas as pd

import numpy as np

# giving a scalar value with index
ser = pd.Series(10, index=[0, 1, 2, 3, 4, 5])

print(ser)
```

**Output:**

```
0    10
1    10
2    10
3    10
4    10
5    10
dtype: int64
```

## 7. What are the different ways in which a dataframe can be created?

- **Creating an empty dataframe:** A basic DataFrame, which can be created is an Empty Dataframe. An Empty Dataframe is created just by calling a `pandas.DataFrame()` constructor.

**Code Example :**

```
# Importing Pandas to create DataFrame

import pandas as pd

# Creating Empty DataFrame and Storing it in variable df
df = pd.DataFrame()

# Printing Empty DataFrame
print(df)
```

**Output:**

```
Empty DataFrame
Columns: []
Index: []
```

- **Creating a dataframe using List:** DataFrame can be created using a single list or by using a list of lists.

**Code Example :**

```
# Import pandas library
import pandas as pd

# initialize list elements
data = [110,202,303,404,550,650]

# Create the pandas DataFrame with the column name provided explicitly
df = pd.DataFrame(data, columns=['Amounts'])

# print dataframe.
print(df)
```

## Output:

```
    Amounts
0   110
1   202
2   303
3   404
4   550
5   650
```

## Code Example :

```
# Import pandas library
import pandas as pd

# initialize list of lists
data = [['mark', 20], ['zack', 16], ['ron', 24]]

# Create the pandas DataFrame
df = pd.DataFrame(data, columns=['Name', 'Age'])

# print dataframe.
print(df)
```

## Output:

```
    Name    Age
0   mark    20
1   zack    16
2   ron 24
```

- **Creating DataFrame from dict of ndarray/lists:** To create a DataFrame from dict of narray/list there are a few conditions to be met.
- First, all the arrays must be of the same length.
- Second, if the index is passed then the length index should be equal to the length of arrays.
- Third, if no index is passed, then by default, the index will be in the range( `n` ) where `n` is the length of the array.

**Code Example :**

```
# Python code demonstrates creating
# DataFrame from dict narray / lists
# By default addresses.

import pandas as pd

# initialize data of lists.
data = {'Name': ['Max', 'Lara', 'Koke', 'muller'],
        'Age': [10, 31, 91, 48]}

# Create DataFrame
df = pd.DataFrame(data)

# Print the output.
print(df)
```

**Output:**

```
     Name    Age
0   Max 10
1   Lara    31
2   Koke    91
3   muller  48
```

- **Create pandas dataframe from lists using a dictionary:** Creating pandas DataFrame from lists using a dictionary can be achieved in multiple ways. We can create pandas DataFrame from lists using a dictionary by using `pandas.DataFrame()` .

**Code Example :**

```
# Python code demonstrates how to create
# Pandas DataFrame by lists of dicts.
import pandas as pd

# Initialize data to lists.
data = [{'aa': 1, 'bs': 2, 'cd': 3},
        {'aa': 10, 'bs': 20, 'cd': 30}]

# Creates DataFrame.
df = pd.DataFrame(data)

# Print the data
print(df)
```

**Output:**

```
    aa  bs  cd
0   1   2   3
1   10  20  30
```

- **Creating dataframe from series:** In order to create a dataframe using series the argument to be passed in a `DataFrame()` function has to be a Series.

**Code Example:**

```
# Python code demonstrates creating
# Pandas Dataframe from series.

import pandas as pd

# Initialize data to series.
d = pd.Series([10, 20, 30, 40])
# creates Dataframe.
df = pd.DataFrame(d)

# print the data.
print(df)
```

**Output:**

```
    0
0   10
1   20
2   30
3   40
```

- **Creating DataFrame from Dictionary of series:** To create a DataFrame from Dict of series, a dictionary needs to be passed as an argument to form a DataFrame. The resultant index is the union of all the series of passed indexed.

**Code Example :**

```
# Python code demonstrate creating
# Pandas Dataframe from Dicts of series.

import pandas as pd

# Initialize data to Dicts of series.
d = {'one': pd.Series([10, 20, 30, 40],
                      index=['a', 'b', 'c', 'd']),
     'two': pd.Series([10, 20, 30, 40],
                      index=['a', 'b', 'c', 'd'])}

# creates Dataframe.
df = pd.DataFrame(d)

# print the data.
print(df)
```

**Output:**

```
    one two
a   10  10
b   20  20
c   30  30
d   40  40
```

## 8. How can we create a copy of the series in Pandas?

We can create a copy of the series by using the following syntax:

```
Series.copy(deep=True)
```

The default value for the deep parameter is set to **True**.

When the value of `deep=True`, the creation of a new object with a copy of the calling object's data and indices takes place. Modifications to the data or indices of the copy will not be reflected in the original object whereas when the value of `deep=False`, the creation of a new object will take place without copying the calling object's data or index i.e. only the references to the data and index will be copied. Any changes made to the data of the original object will be reflected in the shallow copy and vice versa.

## 9. Explain Categorical data in Pandas?

Categorical data is a discrete set of values for a particular outcome and has a fixed range. Also, the data in the category need not be numerical, it can be textual in nature. Examples are gender, social class, blood type, country affiliation, observation time, etc. There is no hard and fast rule for how many values a categorical value should have. One should apply one's domain knowledge to make that determination on the data sets.

## 10. Explain Reindexing in pandas along with its parameters?

Reindexing as the name suggests is used to alter the rows and columns in a DataFrame. It is also defined as the process of conforming a dataframe to a new index with optional filling logic. For missing values in a dataframe, the `reindex()` method assigns `NA/NaN` as the value. A new object is returned unless a new index is produced that is equivalent to the current one. The copy value is set to **False**. This is also used for changing the index of rows and columns in the dataframe.

## 11. What is NumPy?

NumPy is one of the most widely used, versatile, simple, open-source, python-based, general-purpose packages that is used for processing arrays. NumPy is an abbreviation for **NUM**erical **PY**thon. Due to its highly optimized tools, it provides high-performance and powerful `N` -dimensional array processing capabilities that are explicitly designed to handle complex arrays. It is most commonly used in performing scientific computations and various broadcasting functions because of its popularity, powerful performance, and flexibility to perform various operations.

## 12. Give a brief description of time series in Panda?

A time series is an organized collection of data that depicts the evolution of a quantity through time. Pandas have a wide range of capabilities and tools for working with time-series data in all fields.

Supported by pandas:

- Analyzing time-series data from a variety of sources and formats.
- Create time and date sequences with preset frequencies.
- Date and time manipulation and conversion with timezone information.
- A time series is resampled or converted to a specific frequency.
- Calculating dates and times using absolute or relative time increments is one way to.

## 13. Explain MultiIndexing in Pandas.

Multiple indexing is defined as essential indexing because it deals with data analysis and manipulation, especially for working with higher dimensional data. It also enables us to store and manipulate data with an arbitrary number of dimensions in lower-dimensional data structures like Series and DataFrame.

## 14. How can we convert Series to DataFrame?

The conversion of Series to DataFrame is quite a simple process. All we need to do is to use the `to_frame()` function.

**Syntax:**

```
Series.to_frame(name=None)
```

**Parameters:**

- **name:** It accepts data objects as input. It is an optional parameter. The value of the name parameter will be equal to the name of the Series if it has any.
- **Return Type:** It returns the DataFrame after converting it from Series.

## 15. How can we convert DataFrame to Numpy Array?

In order to convert DataFrame to a Numpy array we need to use `DataFrame.to_numpy()` method.

**Syntax:**

```
DataFrame.to_numpy(dtype=None, copy=False, na_value=_NoDefault.no_default)
```

**Parameters:**

- **dtype:** It accepts string or numpy.dtype. It is an optional parameter.
- **copy:** It accepts a boolean value whose default is set to **False**.
- **na_value:** It is an optional parameter. It specifies the value to use for missing values. The data type will depend on the data type of the column in the dataframe.

## 16. How can we convert DataFrame to an excel file?

In order to convert DataFrame to an excel file we need to use the `to_excel()` function. There are various parameters to be considered. But initially, all you need is to mention the DataFrame name and the name of the excel sheet.

> **Note:** To write a single object to an Excel file, you need to provide the target file name. However, if you want to write to multiple sheets, you must create an ExcelWriter object that specifies the target file name and the sheet that needs to be written. Alternatively, you can specify a unique sheet name to write multiple sheets to the same Excel file.

**Syntax:**

```
data.to_excel( excel_writer, sheet_name='Sheet1', \*\*kwargs )
```

**Parameters:**

- **excel_writer:** It accepts a string or ExcelWriter object. It specifies the path of the file to be written or an existing ExcelWriter object.
- **sheet_name:** It accepts a string value. The default value is set to 'Sheet1'. It specifies the name of the sheet that will contain the DataFrame.
- **columns:** It accepts a sequence or list of strings as input. It is an optional parameter that specifies the columns that need to be written.
- **index:** It accepts a boolean value whose default is set to True. It specifies the rows/index to be written.
- **index_label:** It accepts string or sequence of string values. It is an optional parameter. It specifies the column label for index column(s) if required. If nothing is specified, and the header and index are set to True, then the index names are used. A sequence value should be given only if the DataFrame uses MultiIndexing.

## 17. What is TimeDelta?

Timedeltas are differences in times, expressed in different units, e.g. days, hours, minutes, and seconds. They can be both positive and negative.

## 18. Explain Pandas Timedelta.seconds Property

`Timedelta.seconds` in pandas is used to return the number of seconds. Its implementation is simpler than it sounds. We do not need any special parameters and the return type is in the form of seconds.

**Code Example :**

```
#importing necessary libraries
import pandas as pd
import numpy as np

# Create the Timedelta object
td = pd.Timedelta('5 days 09:08:03.000000312')

# Print the Timedelta object
print(td)
# Print the Timedelta object in seconds format
print(td.seconds)
```

**Output:-**

```
5 days 09:08:03.000000312
32883
```

# Pandas Interview Questions for Experienced

## 19. Is iterating over a Pandas Dataframe a good practice? If not what are the important conditions to keep in mind before iterating?

Ideally, iterating over pandas DataFrames is **definitely not the best practice** and one should only consider doing so when it is absolutely necessary and no other function is applicable. The iteration process through DataFrames is very inefficient. Pandas provide a lot of functions using which an operation can be executed without iterating through the dataframe. There are certain conditions that need to be checked before

Before attempting to iterate through pandas objects, we must first ensure that none of the below-stated conditions aligns with our use case:

- **Applying a function to rows:** A common use case of iteration is when it comes to applying a function to every row, which is designed to work only one row at a time and cannot be applied on the full DataFrame or Series. In such cases, it's always recommended to use `apply()` method instead of iterating through the pandas object.
- **Iterative manipulations:** In case we need to perform iterative manipulations and at the same time performance is a major area of concern, then we have alternatives like numba and cython.
- **Printing a DataFrame:** If we want to print out a DataFrame then instead of iterating through the whole DataFrame we can simply use `DataFrame.to_string()` method in order to render the DataFrame to a console-friendly tabular output.
- **Vectorisation over iteration:** It is always preferred to choose vectorization over iteration as pandas come with a rich set of built-in methods whose performance is highly optimized and super efficient.

## 20. How would you iterate over rows in a DataFrame in Pandas?

Although it is not a good practice to iterate over rows in Pandas if there is no other alternative we do so using either `iterrows()` or `itertuples()` built-in methods.

- **pandas.DataFrame.iterrows():** This method is used to iterate over DataFrame rows as (index, Series) pairs. There is only one drawback for this method it does not preserve the dtypes across rows due to the fact that it converts each row into a Series. If you need to preserve the dtypes of the pandas object, then one should use `itertuples()` method instead.

**Code Example :**

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

# Define a dictionary containing students data
data = {'Name': ['Sneha', 'Shreya',
                 'Sabhya', 'Riya'],
        'Age': [22, 18, 10, 19],
        'Stream': ['Computer', 'Commerce',
                   'Arts', 'Mechanical'],
        'Percentage': [89, 93, 97, 73]}

# Convert the dictionary into DataFrame
df = pd.DataFrame(data, columns=['Name', 'Age',
                                 'Stream', 'Percentage'])

print("Given Dataframe :\n", df)

print("\nIterating over rows using iterrows() method :\n")

# iterate through each row and select
# 'Name' and 'Age' columns respectively.
for index, row in df.iterrows():
    print(row["Name"], row["Age"])
```

**Output:-**

```
Given Dataframe :
      Name  Age      Stream  Percentage
0   Sneha   22    Computer          89
1  Shreya   18    Commerce          93
2  Sabhya   10        Arts          97
3    Riya   19  Mechanical          73

Iterating over rows using iterrows() method :

Sneha 22
Shreya 18
Sabhya 10
Riya 19
```

- **pandas.DataFrame.itertuples():** This method is used to iterate over DataFrame rows as **namedtuples**. Also, `itertuples()` are faster than compared to `iterrows()` .

**Code Example :**

```
print("\nIterating over rows using itertuples() method :\n")

# iterate through each row and select
# 'Name' and 'Percentage' column respectively.
for row in df.itertuples(index=True, name='Pandas'):
    print(getattr(row, "Name"), getattr(row, "Percentage"))
```

**Output:-**

```
Iterating over rows using itertuples() method :

Sneha 89
Shreya 93
Sabhya 97
Riya 73
```

## 21. List some statistical functions in Python Pandas?

Some of the major statistical functions in Python Pandas are:

- **sum()** – It returns the sum of the values.
- **min()** – It returns the minimum value.
- **max()** – It returns the maximum value.
- **abs()** – It returns the absolute value.
- **mean()** – It returns the mean which is the average of the values.
- **std()** – It returns the standard deviation of the numerical columns.
- **prod()** – It returns the product of the values.

## 22. How to Read Text Files with Pandas?

There are multiple ways in which we read a text file using Pandas.

- **Using read_csv():** CSV is a comma-separated file i.e. any text file that uses commas as a delimiter to separate the record values for each field. Therefore, in order to load data from a text file we use `pandas.read_csv()` method.
- **Using read_table():** This function is very much like the `read_csv()` function, the major difference being that in read_table the delimiter value is ' `\t` ' and not a comma which is the default value for `read_csv()` . We will read data with the read_table function making the separator equal to a single space(' ').
- **Using read_fwf():** It stands for fixed-width lines. This function is used to load DataFrames from files. Another very interesting feature is that it supports optionally iterating or breaking the file into chunks. Since the columns in the text file were separated with a fixed width, this `read_fwf()` read the contents effectively into separate columns.

## 23. How are iloc() and loc() different?

- **DataFrame.iloc():** It is a method used to retrieve data from a Data frame, and it is an integer position-based locator (from `0` to `length-1` of the axis), but may also be used with a boolean array and this is the major difference factor between `iloc()` and `loc()` . It takes input as **integers**, arrays of integers, an object, boolean arrays, and functions.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12], 'Marks': [85,

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)
print(df.iloc[[0, 2]])
```

**Output:-**

```
    Name  Age  Marks
0    Kate   10     85
2  Sheila   12     91
```

- **DataFrame.loc():** It gets rows or columns with particular labels as input. It takes input as a single label, a list of arrays, and objects with labels. It does not work with boolean arrays or values.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12], 'Marks': [85,

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)

print(df.loc[(df.Name=='Kate')])
```

**Output:-**

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   14     77
2  Sheila   12     91
   Name  Age  Marks
0  Kate   10     85
```

## 24. How will you sort a DataFrame?

The function used for sorting in pandas is called `DataFrame.sort_values()`. It is used to sort a DataFrame by its column or row values. The function comes with a lot of parameters, but the most important ones to consider for sort are:

- **by:** It is used to specify the column/row(s) which are used to determine the sorted order. It is an optional parameter.
- **axis:** It specifies whether the sorting is to be performed for a row or column and the value is `0` and `1` respectively.
- **ascending:** It specifies whether to sort the dataframe in ascending or descending order. The default value is set to ascending. If the value is set as **ascending=False** it will sort in descending order.

## 25. How would you convert continuous values into discrete values in Pandas?

Depending on the problem, continuous values can be discretized using the `cut()` or `qcut()` function:

- **cut()** It bins the data based on values. We use it when we need to segment and sort data values into bins that are evenly spaced. `cut()` will choose the bins to be evenly spaced based on the values themselves and not the frequency of those values. For example, cut could convert ages to groups of age ranges.
- **qcut()** bins the data based on sample quantiles. We use it when we want to have the same number of records in each bin or simply study the data by quantiles. For example, if in a data we have `30` records, and we want to compute the quintiles, `qcut()` will divide the data such that we have `6` records in each bin.

## 26. What is the difference between join() and merge() in Pandas?

Both join and merge functions are used to combine two dataframes. The major difference is that the join method combines two dataframes on the basis of their indexes whereas the merge method is more flexible and allows us to specify columns along with the index to combine the two dataframes.

These are the main differences between `df.join()` and `df.merge()` :

- **lookup on right table:** When performing a lookup on the right table, the `join()` method will always use the index of `df2` to perform the join operation. However, if you use the `merge()` method, you can choose to join based on one or more columns of `df2` by default, or even the index of `df2` if you specify the `right_index=True` parameter.
- **lookup on left table:** When performing a lookup on the left table, `df1.join(df2)` method will use the index of `df1` by default, while `df1.merge(df2)` method will use the column(s) of `df1` for the join operation. However, you can override this behavior by specifying the `on=key_or_keys` parameter in `df1.join(df2)` or by setting the `left_index=True` parameter in `df1.merge(df2)` .
- **left vs inner join:** By default, the `df1.join(df2)` method performs a left join (retains all rows of `df1` ), while the `df1.merge(df2)` method performs an inner join (returns only the matching rows of `df1` and `df2` ).

## 27. What is the difference(s) between merge() and concat() in Pandas?

Both concat and merge functions are used to combine dataframes. There are three major key differences between these two functions.

- **The Way of Combining: concat()** function concatenates dataframes along rows or columns. It is nothing but stacking up of multiple dataframes whereas `merge()` combines dataframes based on values in shared columns thus it is more flexible compared to `concat()` as the combination can happen based on the given condition.
- **Axis parameter:** `concat()` function has axis parameter. Since `merge()` function combines dataframes on the basis of shared columns side by side it does not really need an axis parameter. The value of the axis parameter decides in what direction will the concatenation happen. For it to happen row-wise the value of the axis parameter will be '0' and for it to happen side-by-side it will be '1'. The default value is 1.
- **Join vs How:** Join is a parameter of `concat()` function and how is a parameter of `merge()` function. Join can take two values outer and inner whereas how can take four values inner, outer, left, and right.

## 28. What's the difference between interpolate() and fillna() in Pandas?

- **fillna():** It fills the NaN values with a given number with which you want to substitute. It gives you the option to fill according to the index of rows of a `pd.DataFrame` or on the name of the columns in the form of a python dict.

**Code Example :**

```
import pandas as pd
import numpy as np

#Creating a dataframe
df = pd.DataFrame({"Value_1":[None, 14, 35, None, 1,12, 74, 65, None, 1],
                   "Value_2":[None, 24, 54, 3, None,None, 2, 54, 3, None],
                   "Value_3":[20, 16, None, 3, 8,None, 2, 54, 3, None],
                   "Value_4":[None, 2, 54, 3, None,14, 3, None, None, 6]})

print(df)

#Fill the missing value with average values in column Value1.
df['Value_1'].fillna(int(df13['Value_1'].mean()), inplace = True)
df.head()
```

**Output:**

```
   Value_1  Value_2  Value_3  Value_4
0      NaN      NaN     20.0      NaN
1     14.0     24.0     16.0      2.0
2     35.0     54.0      NaN     54.0
3      NaN      3.0      3.0      3.0
4      1.0      NaN      8.0      NaN
5     12.0      NaN      NaN     14.0
6     74.0      2.0      2.0      3.0
7     65.0     54.0     54.0      NaN
8      NaN      3.0      3.0      NaN
9      1.0      NaN      NaN      6.0

# after using fillna method.

    Value_1 Value_2 Value_3 Value_4
0   28  <NA>    20  <NA>
1   14  24  16  2
2   35  54  15  54
3   28  3   3   3
4   1   <NA>    8   <NA>
```

- **interpolate():** It gives you the flexibility to fill the missing values with many kinds of interpolations between the values like **linear**, **time**, etc.

**Code Example :**

```
import pandas as pd, numpy as np
df = pd.Series([1, np.nan, np.nan, 3])
print(df.interpolate())
```

**Output:-**

```
0    1.000000
1    1.666667
2    2.333333
3    3.000000
dtype: float64
```

# Pandas Coding Interview Questions

## 29. How to set Index to a Pandas DataFrame?

- **Changing Index column:** In this example, the First Name column has been made the index column of DataFrame.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)

# set index using column
student_df = student_df.set_index('Name')
print(student_df)
```

**Output:**

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
         Age  Marks
Name
Kate      10     85
Harry     11     77
Sheila    12     91
```

- **Set Index using Multiple Column:** In this example, two columns will be made as an index column. The drop parameter is used to Drop the column and the append parameter is used to append passed columns to the already existing index column.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)

# set index using column
student_df = student_df.set_index(['Name', 'Marks'])
print(student_df)
```

**Output:**

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
              Age
Name   Marks
Kate   85      10
Harry  77      11
Sheila 91      12
```

- **Set index using a List:**

**Code Example :**

```
index = pd.Index(['x1', 'x2', 'x3'])
student_df = student_df.set_index(index)
print(student_df)
```

**Output:**

```
   Age  Marks
x1   10     85
x2   11     77
x3   12     91
```

- **Set multi-index using a list and column**

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
index = pd.Index(['x1', 'x2', 'x3'])
student_df = student_df.set_index([index, 'Name'])
print(student_df)
```

**Output:-**

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
           Age  Marks
   Name
x1 Kate     10     85
x2 Harry    11     77
x3 Sheila   12     91
```

## 30. How to add a row to a Pandas DataFrame?

**We can add a single row using DataFrame.loc:** We can add the row at the last in our dataframe. We can get the number of rows using len( `DataFrame.index` ) for determining the position at which we need to add the new row.

**Code Example :**

```
# Add row
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
student_df.loc[len(student_df.index)] = ['Alex', 19, 93]
print(student_df)
```

**Output:**

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
     Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
3    Alex   19     93
```

**We can also add a new row using the DataFrame.append() function:**

**Code Example :**

```
df2 = {'Name': 'Tom', 'Age': 18, 'Marks': 73}
student_df = student_df.append(df2, ignore_index = True)
print(student_df)
```

**Output:**

```
    Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
3    Alex   19     93
4     Tom   18     73
```

**We can also add multiple rows using the pandas.concat():** by creating a new dataframe of all the rows that we need to add and then appending this dataframe to the original dataframe.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12], 'Marks': [85,

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)

dict = {'Name':['Amy', 'Maddy'],
        'Age':[19, 12],
        'Marks':[93, 81]
       }
df2 = pd.DataFrame(dict)
print(df2)
df3 = pd.concat([df1, df2], ignore_index = True)

print(df3)
```

**Output:**

```
Name  Age  Marks
0    Kate   10     85
1   Harry   14     77
2  Sheila   12     91
    Name  Age  Marks
0    Amy   19     93
1  Maddy   12     81
     Name  Age  Marks
0   Harry   14     77
1  Sheila   12     91
2     Amy   19     93
3   Maddy   12     81
```

## 31. How to add a column to a Pandas DataFrame?

We first create the dataframe and then look into the various methods one by one.

**Code Example :**

```
# Add column
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
address = ['Chicago', 'London', 'Berlin']
student_df['Address'] = address

# Observe the result
print(student_df)
```

### Output:-

```
    Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
     Name  Age  Marks  Address
0    Kate   10     85  Chicago
1   Harry   11     77   London
2  Sheila   12     91   Berlin
```

- **By declaring a new list as a column.**

### Code Example :

```
# Add column
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
address = ['Chicago', 'London', 'Berlin']
student_df['Address'] = address

# Observe the result
print(student_df)
```

### Output:-

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
     Name  Age  Marks  Address
0    Kate   10     85  Chicago
1   Harry   11     77   London
2  Sheila   12     91   Berlin
```

- **By using DataFrame.insert():** It gives the freedom to add a column at any position we like and not just at the end. It also provides different options for inserting the column values.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
address = ['Chicago', 'London', 'Berlin']
student_df.insert(2, "Address", ['Chicago', 'London', 'Berlin'], True)
# Observe the result
print(student_df)
```

**Output:-**

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
     Name  Age  Address  Marks
0    Kate   10  Chicago     85
1   Harry   11   London     77
2  Sheila   12   Berlin     91
```

- **Using Dataframe.assign() method:** This method will create a new dataframe with a new column added to the old dataframe.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
df2 = student_df.assign(address=['Chicago', 'London', 'Berlin'])

# Observe the result
print(df2)
```

**Output:-**

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
     Name  Age  Marks  address
0    Kate   10     85  Chicago
1   Harry   11     77   London
2  Sheila   12     91   Berlin
```

- **By using a dictionary:** We can use a Python dictionary to add a new column in pandas DataFrame. Use an existing column as the key values and their respective values will be the values for a new column.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)
address = {'Chicago': 'Kate', 'London': 'Harry',
           'berlin': 'Sheila'}
student_df['Address'] = address

# Observe the output
print(student_df)
```

**Output:-**

```
    Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
     Name  Age  Marks  Address
0    Kate   10     85  Chicago
1   Harry   11     77   London
2  Sheila   12     91   berlin
```

## 32. How can we convert DataFrame into a NumPy array?

In order to convert a dataframe into a NumPy array we use `DataFrame.to_numpy()` method.

**Syntax:**

```
DataFrame.to_numpy(dtype=None, copy=False, na_value=_NoDefault.no_default)
```

**Parameters:**

- **dtype:** It accepts string or numpy.dtype value. It is an optional parameter.
- **copy:** It accepts a boolean value. The default value is set to False.
  It ensures that the returned value is not a view on another array. Setting the value of `copy=False` does not ensure that `to_numpy()` is no-copy. Whereas if `copy=True` it does ensure that a copy is made.
- **na_value:** It accepts the parameter of any datatype and it is an optional parameter. It specifies the value to be used for missing values. The default value is of the same data type as the object.

**Code Example :**

```
import pandas as pd

# initialize a dataframe
df = pd.DataFrame(
    [[10, 12, 33],
    [41, 53, 66],
    [17, 81, 19],
    [10, 11, 12]],
    columns=['X', 'Y', 'Z'])

# convert dataframe to numpy array
arr = df.to_numpy()

print('\nNumpy Array\n----------\n', arr)
print(type(arr))
```

**Output:**

```
Numpy Array
----------
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
<class 'numpy.ndarray'>
```

## 33. How will you compute the percentile of a numerical series in Pandas?

In order to compute percentile we use `numpy.percentile()` method.

**Syntax:**

```
numpy.percentile(a, q, axis=None, out=None, overwrite_input=False, method='linear', kee
```

It will calculate the q-th percentile of the given the data along the mentioned axis.

**Parameters:**

- **a:** It is an input array or object that can be converted to an array.
- **q:** It is the percentile or sequence of percentiles to be calculated. The value must be between `0` and `100` both inclusive.

### Code Example :

```
import pandas as pd
import random

A = [ random.randint(0,100) for i in range(10) ]
B = [ random.randint(0,100) for i in range(10) ]

df = pd.DataFrame({ 'field_A': A, 'field_B': B })
df


print(df.field_A.quantile(0.1)) # 10th percentile


print(df.field_A.quantile(0.5)) # same as median

print(df.field_A.quantile(0.9)) # 90th percentile
```

### Output:-

```
12.1
52.0
92.6
```

## 34. How to create Timedelta objects in Pandas?

**String**: In order to create a timedelta object using a string argument we pass a string literal.

### Code Example :

```
#importing necessary libraries
import pandas as pd
import numpy as np

#Conversion from string format to date format takes place using Timedelta method.
print (pd.Timedelta('20 days 12 hours 45 minutes 3 seconds'))
```

### Output:

```
20 days 12:45:03
```

**Integer*: What differs from string, in this case, is we just need to pass an integer value and the object will be created.

**Code Example :**

```
#importing necessary libraries
import pandas as pd
import numpy as np

print (pd.Timedelta(16,unit='h'))#h here is used for hours.
```

**Output:**

```
0 days 16:00:00
```

**Data Offsets**: In order to first learn how to create a timedelta object using data offset we first need to understand what data offset actually is. Data offsets are parameters like weeks, days, hours, minutes, seconds, milliseconds, microseconds, and nanoseconds. This when passed as an argument helps in the creation of the timedelta object.

**Code Example :**

```
#importing necessary libraries
import pandas as pd
import numpy as np

print (pd.Timedelta(days=2, hours = 16))
```

**Output:-**

```
2 days 16:00:00
```

**Code Example :**

```
#importing necessary libraries
import pandas as pd
import numpy as np

print (pd.Timedelta(days=2, hours = 6, minutes = 23))
```

**Output:-**

```
2 days 06:23:00
```

## 35. How do you split a DataFrame according to a boolean criterion?

We can create a mask to separate the dataframe and then use the inverse operator ( ~ ) to take the complement of the mask.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12], 'Marks': [85,

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)
df1 = df[df['Age'] > 10]

# printing df1
df1
```

**Output:-**

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   14     77
2  Sheila   12     91
Name    Age Marks
1   Harry   14  77
2   Sheila  12  91
```

## 36. How can we convert NumPy array into a DataFrame?

In order to convert a Numpy array into a DataFrame we first need to create a numpy array and then use the `pandas.DataFrame` the method along with specifying the/labels for rows and columns.

**Code Example :**

```
# Python program to Create a
# Pandas DataFrame from a Numpy
# array and specify the index
# column and column headers

# import required libraries
import numpy as np
import pandas as pd

# creating a numpy array
numpyArray = np.array([[115, 222, 343],
                       [323, 242, 356]])

# generating the Pandas dataframe
# from the Numpy array and specifying
# name of index and columns
dataframe = pd.DataFrame(data = numpyArray,
                         index = ["Row1", "Row2"],
                         columns = ["Column1",
                                    "Column2", "Column3"])

# printing the dataframe
print(dataframe)
```

**Output:-**

```
      Column1  Column2  Column3
Row1      115      222      343
Row2      323      242      356
```

## 37. How to delete a row in Pandas DataFrame?

The `drop()` method is used to delete a row in a DataFrame. If we set the value of the axis parameter as ‘ `0` ’ or do not mention it at all it will work for rows as the default value for the axis parameter is set to ‘ `0` ’, if we set the value to ‘ `1` ’ it will delete the column in the DataFrame.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)


# set index using column
student_df = student_df.set_index('Name')
print(student_df)

student_df.drop(["Harry"], inplace = True)
print(student_df)
```

**Output:-**

```
        Age  Marks
Name
Kate     10     85
Harry    11     77
Sheila   12     91
        Age  Marks
Name
Kate     10     85
Sheila   12     91
```

## 38. How to delete a column in Pandas DataFrame?

The `drop()` method is used to delete a column in a DataFrame. If we set the value of the axis parameter as ‘1’ it will work for a column if we set the value to ‘0’ it will delete the rows in the DataFrame.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
student_df.drop(["Age"], axis = 1, inplace = True)
print(student_df)
```

**Output:-**

```
     Name  Marks
0    Kate     85
1   Harry     77
2  Sheila     91
```

## 39. How to get frequency count of unique items in a Pandas DataFrame?

In order to get the frequency count of unique items in a Pandas DataFrame we can use the `Series.value_counts()` method.

**Code Example :**

```
# importing the module
import pandas as pd

# creating the series
s = pd.Series(data = [1,2,3,4,3,5,3,7,1])

# displaying the series
print(s)

# finding the unique count
print(s.value_counts())
```

**Output:-**

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

## 40. How to rename the index in a Pandas DataFrame?

In order to rename a DataFrame we use the `Dataframe.set_index()` method to give different values to the columns or the index values of DataFrame. Like in this example we will change the index label from 'Name' to 'FirstName'.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)

# set index using column
student_df = student_df.set_index('Name')
print(student_df)
student_df.index.names = ['FirstName']
print(student_df)
```

**Output:-**

```
       Age  Marks
Name
Kate     10     85
Harry    11     77
Sheila   12     91
           Age  Marks
FirstName
Kate        10     85
Harry       11     77
Sheila      12     91
```

## 41. How to reset the index in a Python Pandas DataFrame?

Inorder to reset the index of the DataFrame we use the `Dataframe.reset_index()` command. If the DataFrame has a MultiIndex, this method can also remove one or more levels.

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 11, 12], 'Marks': [85,

# create DataFrame from dict
student_df = pd.DataFrame(student_dict)
print(student_df)

# set index using column
student_df = student_df.set_index('Name')
print(student_df)

student_df.reset_index(drop=True, inplace=True)
print(student_df)
```

**Output:-**

```
    Name  Age  Marks
0    Kate   10     85
1   Harry   11     77
2  Sheila   12     91
        Age  Marks
Name
Kate     10     85
Harry    11     77
Sheila   12     91
   Age  Marks
0   10     85
1   11     77
2   12     91
```

## Conclusion

In this article, we have seen commonly asked pandas interview questions. These questions along with regular problem practice sessions will help you crack any pandas-based interview. We divided the article into four sections:

- The **BASIC** python pandas interview questions section contains questions based on theoretical concepts covering different segments like Data Structures in Pandas, Time Series, Statistical methods, etc.
- The **ADVANCED** python pandas interview questions section delves a little deeper into the conceptual section covering various methods like `join()`, `merge()`, `groupby()`, their functionalities, implementation, etc.
- The **DATA SCIENCE** python pandas interview questions section focuses on application-based questions the ones that a data scientist might face during his day-to-day work like formatting a dataframe, or working on data aggregation, etc.
- The **CODING** python pandas interview section focuses on questions that test the python coding skills along with the general concepts involved. You might be asked to write a code to calculate percentile or to convert a Numpy array to a DataFrame or vice-versa, etc.

Along with theoretical knowledge of pandas, there is an emphasis on the ability to write good-quality code as well. So keep learning and practising problems you'll no doubt succeed at any pandas interview.

# Pandas Interview Questions for Data Scientists

## 42. How can you find the row for which the value of a specific column is max or min?

We can find the row for which the value of a specific column is by using **idxmax** and **idxmin** functions.

**Code Example :**

```
import pandas as pd

data = {
  "sales": [23, 34, 56],
  "age": [50, 40, 30]
}

df = pd.DataFrame(data)

print(df.idxmax())
print(df.idxmin())
```

**Output:-**

```
sales    2
age      0
dtype: int64
sales    0
age      2
dtype: int64
```

## 43. Explain the GroupBy function in Pandas

Python pandas `Dataframe.groupby()` function is used for grouping the data according to the categories and applying a function to those categories. It helps in data aggregation in an efficient manner. It splits the data into groups based on some given criteria. The pandas objects can be split on any of their axes. In brief `groupby()` provides the mapping of labels to their respective group names.

**Syntax:**

```
DataFrame.groupby(by=None, axis=0, level=None, as_index=True, sort=True, group_keys=_Nc
```

**Code Example :**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12], 'Marks': [85,

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)

gk = df.groupby('Age')
gk.first()
```

**Output:-**

```
    Name  Age  Marks
0    Kate   10     85
1   Harry   14     77
2  Sheila   12     91
Name    Marks
Age
10  Kate    85
12  Sheila  91
14  Harry   77
```

## 44. How to format data in your Pandas DataFrame?

When we start working on a dataset or a set of data we need to perform some operations on the values in the DataFrame. At times these values might not be in the right format for you to work on it thus formatting of data is required. There are multiple ways in which we can format data in a Pandas DataFrame.

- One way is by **Replacing All Occurrences of a String in a DataFrame**. In order to replace Strings in our DataFrame, we can use `replace()` method i.e. all we need is to pass the values that we would like to change, followed by the values we want to replace them with.
- One other way is by **Removing Parts From Strings in the Cells of the DataFrame**. Removing unwanted parts of strings is cumbersome work. Luckily, there is a solution in place! We can do it easily by using `map()` function on the column result to apply the lambda function over each element or element-wise of the column.
- **Splitting Text in a Column into Multiple Rows in a DataFrame**. The process of splitting text into multiple rows is quite a complex task. We can do so by applying a function to the Pandas DataFrame's Columns or Rows.

## 45. What is the use of pandas.Dataframe.aggregate() function? Explain its syntax and parameters.

Data Aggregation is defined as the process of applying some aggregation function to one or more columns. It uses the following:

- **sum:** It is used to return the sum of the values for the requested axis.
- **min:** It is used to return a minimum of the values for the requested axis.
- **max:** It is used to return maximum values for the requested axis.

Its **Syntax** is:

```
DataFrame.aggregate(func=None, axis=0, *args, **kwargs)
```

Aggregate using one or more operations over the specified axis.

**Parameters:**

- **func:** It takes string, list, dictionary, or function values as input. It represents the function to use for data aggregation.
- **axis:** It takes in only two values '0' or '1'. 0 is for the index and 1 is for columns.
  **If 0 or 'index':** The function is applied to each column.
  **If 1 or 'columns':** The function is applied to each row.
  The default value is set to 0.

It returns the aggregated dataframe as the output.

**Code Example :**

```
import pandas as pd

data = {
  "x": [560, 240, 630],
  "y": [300, 1112, 452]
}

df = pd.DataFrame(data)

x = df.aggregate(["sum"])
y = df.aggregate(["min"])
z = df.aggregate(["max"])

print(x)
print(y)
print(z)
```

**Output:**

```
        x     y
sum  1430  1864
       x    y
min  240  300
       x     y
max  630  1112
```

## 46. How to get items of series A not present in series B?

In order to find items from series `A` that are not present in series B by using the `isin()` method combining it with the **Bitwise NOT** operator in pandas. We can understand this using a code example:

## Code Example :

```
# Importing pandas library
import pandas as pd

# Creating 2 pandas Series
series1 = pd.Series([12, 24, 38, 210, 110, 147, 929])
series2 = pd.Series([17, 83, 76, 54, 110, 929, 510])

print("Series1:")
print(series1)
print("\nSeries2:")
print(series2)

# Using Bitwise NOT operator along
# with pandas.isin()
print("\nItems of series1 not present in series2:")
res = series1[~series1.isin(series2)]
print(res)
```

## Output:-

```
Series1:
0     12
1     24
2     38
3    210
4    110
5    147
6    929
dtype: int64

Series2:
0     17
1     83
2     76
3     54
4    110
5    929
6    510
dtype: int64

Items of series1 not present in series2:
0     12
1     24
2     38
3    210
5    147
dtype: int64
```

## 47. Describe a few data operations in Pandas.

There are several useful data operations for DataFrame in Pandas, which are as follows:

1. **String Operation:** Pandas provide a set of string functions for working with string data. The following are the few operations on string data:
    - **lower():** Any strings in the index or series are converted to lowercase letters.
    - **upper():** Any strings in the index or series are converted to uppercase letters.
    - **strip():** This method eliminates spacing from every string in the Series/index, along with a new line.
    - **islower():** If all of the characters in the Series/Index string are lowercase, it returns True. Otherwise, False is returned.
    - **isupper():** If all of the characters in the Series/Index string are uppercase, it returns True. Otherwise, False is returned.
    - **split(' '):** It's a method that separates a string according to a pattern.
    - **cat(sep=' '):** With a defined separator, it concatenates series/index items.
    - **contains(pattern):** If a substring is available in the current element, it returns True; otherwise, it returns False.
    - **replace(a,b):** It substitutes the value b for the value a.
    - **startswith(pattern):** If all of the components in the series begin with a pattern, it returns True.
    - **endswith(pattern):** If all of the components in the series terminate in a pattern, it returns True.
    - **find(pattern):** It can be used to return the pattern's first occurrence.
    - **findall(pattern):** It gives you a list of all the times the pattern appears.
    - **swapcase:** It is used to switch the lower/upper case.
2. **Null values:** When no data is being sent to the items, a Null value/missing value can appear. There may be no values in the respective columns, which are commonly represented as NaN. Pandas provide several useful functions for identifying, deleting, and changing null values in Data Frames. The following are the functions.
    - **isnull():** isnull 's job is to return true if either of the rows has null values.
    - **notnull():** It is the inverse of the isnull() function, returning true values for non-null values.
    - **dropna():** This function evaluates and removes null values from rows and columns.
    - **fillna():** It enables users to substitute other values for the NaN values.
    - **replace():** It's a powerful function that can take the role of a regex, dictionary, string, series, and more.

## 48. Compare the Pandas methods: map(), applymap(), apply()

- The **map()** method is an elementwise method for only Pandas Series, it maps values of the Series according to input correspondence.

It accepts dicts, Series, or callable. Values that are not found in the dict are converted to NaN.

**Code Example:** We first create a dataframe and then apply the respective methods to it.

```
import pandas as pd

# Series generation
str_string = 'scalar'
str_series = pd.Series(list(str_string))
print("Original series\n" +
      str_series.to_string(index=False,
                           header=False), end='\n\n')

# Using apply method for converting characters
# present in the original series
new_str_series = str_series.map(str.upper)
print("Transformed series:\n" +
      new_str_series.to_string(index=False,
                               header=False), end='\n\n')
```

**Output:**

```
Original series
s
c
a
l
a
r

Transformed series:
S
C
A
L
A
R
```

- The **applymap()** method is an elementwise function for only DataFrames, it applies a function that accepts and returns a scalar to every element of a DataFrame.

It accepts callables only i.e. a Python function.

**Code Example :**

```
import pandas as pd

# initialize a dataframe
df = pd.DataFrame(
    [['a', 'b', 'c'],
    ['d','e','f'],
    ],
    columns=['X', 'Y', 'Z'])

print(df)

new_df = df.applymap(str.upper)
print("Transformed dataframe:\n" +
      new_df.to_string(index=False,
                       header=False), end='\n\n')
```

**Output:-**

```
   X  Y  Z
0  a  b  c
1  d  e  f

Transformed dataframe:
A B
D E F
```

- The **apply()** method also works elementwise, as it applies a function along the input axis of DataFrame. It is suited to more complex operations and aggregation.

It accepts the callables parameter as well.

**Code Example :**

```
import pandas as pd

# initialize a dataframe
df = pd.DataFrame(
    [[10, 12, 33],
    [41, 53, 66],
    [17, 81, 19],
    [10, 11, 12]],
    columns=['X', 'Y', 'Z'])

print(df)

new_df = df.apply(lambda x:x.sort_values(), axis = 1)

print("Transformed dataframe:\n" + \
    new_df.to_string(index = False,
            header = False), end = '\n\n')
```

**Output:-**

```
    X   Y   Z
0  10  12  33
1  41  53  66
2  17  81  19
3  10  11  12

Transformed dataframe:
10 12 33
41 53 66
17 81 19
10 11 12
```

## 49. What's the difference between pivot_table() and groupby()?

Both `pivot_table()` and `groupby()` are used to aggregate your dataframe. The major difference is in the shape of the result.

**Code Example:**

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12], 'Marks': [85,

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)
table = pd.pivot_table(df, index =['Name', 'Age'])
print(table)
```

## Output:-

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   14     77
2  Sheila   12     91
            Marks
        Name   Age
Harry   14      77
Kate    10      85
Sheila  12      91
```

## Code Example :

```
import pandas as pd

student_dict = {'Name': ['Kate', 'Harry', 'Sheila'], 'Age': [10, 14, 12], 'Marks': [85,

# create DataFrame from dict
df = pd.DataFrame(student_dict)
print(df)

gk = df.groupby('Age')
gk.first()
```

## Output:

```
     Name  Age  Marks
0    Kate   10     85
1   Harry   14     77
2  Sheila   12     91
Age  Name   Marks
10  Kate    85
12  Sheila  91
14  Harry   77
```

## 50. When to use merge() over concat() and vice-versa in Pandas?

The use of `concat()` function comes into play when combining **homogeneous** DataFrame, while the `merge()` function is considered first when combining complementary DataFrame.

If we need to merge vertically, we should always use `pandas.concat()` . whereas if need to merge horizontally via columns, we should go with `pandas.merge()` , which by default merges on the columns that are in common between the dataframes.

**Code Example :**

```
df1 = pd.DataFrame({'Key': ['b', 'b', 'a', 'c'], 'data1': range(4)})

df2 = pd.DataFrame({'Key': ['a', 'b', 'd'], 'data2': range(3)})

#Merge
# The 2 dataframes are merged on the basis of values in column "Key" as it is
# a common column in 2 dataframes

print(pd.merge(df1, df2))

#Concat
# df2 dataframe is appended at the bottom of df1

print(pd.concat([df1, df2]))
```

**Output:-**

```
  Key  data1  data2
0   b      0      1
1   b      1      1
2   a      2      0
  Key  data1  data2
0   b    0.0    NaN
1   b    1.0    NaN
2   a    2.0    NaN
3   c    3.0    NaN
0   a    NaN    0.0
1   b    NaN    1.0
2   d    NaN    2.0
```