syllabus
Getting Started with pandas: Introduction to pandas,
Library Architecture,
Features,
Applications,
Data Structures,
Series,
DataFrame,
Index Objects,
Essential Functionality
Reindexing, Dropping entries from an axis, Indexing, selection, and filtering),Sorting and ranking, Summarizing and Computing Descriptive Statistics, Unique Values, Value Counts, Handling Missing Data, filtering out missing data.

…………………………

Introduction to pandas,

Pandas is a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

Pandas is a powerful and open-source Python library. The Pandas library is used for data manipulation and analysis. Pandas consist of data structures and functions to perform efficient operations on data.

Pandas is well-suited for working with **tabular data**, such as **spreadsheets** or **SQL tables**.

Library Architecture

The architecture of the **Pandas** library is designed to offer efficient data manipulation and analysis capabilities, focusing on flexibility and performance. The architecture is built on **NumPy** for array processing, and **Cython** for speeding up computations. It also integrates with various I/O backends and other Python libraries. Let's dive deeper into the key components and their structure.

1. Core Data Structures

Pandas revolves around two primary data structures:

- **Series**: A one-dimensional labeled array, similar to a column in a database or spreadsheet. It can hold any data type (integers, floats, strings, etc.) and has an index associated with it.
- **DataFrame**: A two-dimensional, size-mutable, and potentially heterogeneous tabular data structure, similar to an Excel spreadsheet or SQL table. A DataFrame can hold data of different types (strings, integers, floats) across columns, with each column being a Pandas Series.

The **DataFrame** is essentially a collection of **Series** objects sharing a common index.

- ***Panel**: Panel is a somewhat deprecated data structure in pandas that facilitated 3D data representation (i.e., data that has dimensions in what might be considered rows, columns, and depth). As of pandas version 0.25, Panels are removed and no longer supported, and users are encouraged to use multi-index DataFrames or xarray for 3D data.*

Series	**DataFrame**
One- dimensional	Two- dimensional
Series elements must be homogenous.	Can be heterogeneous.
Immutable(size cannot be changed).	Mutable(size can be changeable).
Element wise computations.	Column wise computations.
Functionality is less.	Functionality is more.
Alignment not supported.	Alignment is supported.

2. Indexing and Alignment

Pandas has a powerful and flexible **indexing** system, which underpins much of its functionality:

- **Index**: Every Series and DataFrame in Pandas has an index (labels for rows and/or columns). The index can be of various types:
  - **RangeIndex** (default, a simple range of integers)
  - **DatetimeIndex** for time-based data
  - **MultiIndex** for hierarchical indexing (multiple levels of indexing)

This **Index** enables Pandas to efficiently handle and align data during operations, such as merging, joining, or aligning different data structures.

3. Pandas Core Architecture

The architecture of the Pandas library consists of several core modules and components:

a) The `pandas.core` module:

This is the heart of Pandas, containing the fundamental classes and functions that form the basis for Series, DataFrames, and related operations.

- **`pandas.core.series.Series`**: Implements the one-dimensional array with labeled axes.
- **`pandas.core.frame.DataFrame`**: Implements the two-dimensional tabular data structure.
- **`pandas.core.indexes`**: Implements various types of index objects (e.g., RangeIndex, DatetimeIndex, MultiIndex).
- **`pandas.core.reshape`**: Includes functions for reshaping data (pivoting, stacking, unstacking, etc.).
- **`pandas.core.groupby`**: Contains functionality for grouping and aggregating data.
- **`pandas.core.computation`**: Contains the evaluation and transformation features that allow you to apply custom functions across DataFrame columns.

b) The `pandas.util` module:

This module provides utility functions that help in internal operations such as checking for missing data, handling warnings, etc.

c) The `pandas.io` module:

This part of Pandas manages I/O operations, including reading and writing data from various formats (CSV, Excel, JSON, SQL, etc.). The **`pandas.io.parsers`** submodule, for example, handles reading data from CSV files.

d) The `pandas.tseries` module:

This module handles time series data. It provides functionality for working with date and time-related operations, like resampling, time-shifting, and rolling windows.

4. Underlying Technologies

a) NumPy:

Pandas relies heavily on **NumPy** for the underlying data structures, particularly the **ndarray** object. Both Series and DataFrames use **NumPy arrays** to store the actual data. This enables fast computation and efficient memory usage. Pandas uses NumPy's broadcasting and vectorization features to perform operations across data without explicitly writing loops.

b) Cython:

To improve performance, many of the core functions in Pandas are written using **Cython**, which is a superset of Python that allows for C-like performance while maintaining Python's syntax. This is crucial for operations like aggregations, mathematical computations, and data cleaning that need to be executed quickly.

c) Blaze (optional, for larger datasets):

Pandas can be extended with libraries like **Blaze** to handle even larger datasets that don't fit into memory. Blaze provides an interface to work with out-of-core datasets and distributed systems, such as **Dask** or **Spark**.

5. Optimization for Performance

Pandas incorporates several techniques to optimize for performance:

- **Vectorized Operations**: Pandas performs operations on entire columns or rows (Series) in a vectorized manner, meaning no explicit Python loops are necessary. This leverages the highly optimized **NumPy** array operations, which are performed in compiled C code, thus speeding up computation.
- **Memory Management**: Pandas is designed to handle large datasets efficiently. It uses **NumPy** arrays internally, which are memory-efficient and faster than Python lists. Operations on data structures are typically performed in-place, reducing memory usage.
- **Cython Optimizations**: Many of the time-consuming functions in Pandas (such as groupby, aggregation, etc.) are written in **Cython**, which provides a significant performance boost by compiling critical operations to C.
- **Efficient I/O**: Pandas can read and write data very efficiently from a variety of file formats. It uses external libraries (like **lxml** for XML parsing, **pyarrow** for Parquet, etc.) to accelerate these operations.

6. Functionality: Key Components

- **Grouping and Aggregation**:
  - Pandas provides the **groupby** functionality to split data into groups, apply transformations, and aggregate results.
  - The **pivot_table** method helps you summarize and aggregate data in a similar way to Excel's pivot tables.
- **Reshaping and Pivoting**:
  - Operations like **pivot**, **melt**, **stack**, and **unstack** allow for flexible reshaping of data, which is helpful for data analysis and reporting.
- **Time Series**:
  - Time series analysis is a key feature in Pandas. It allows resampling, shifting, and rolling window operations, which are crucial when working with time-based data.
- **Merging, Joining, and Concatenation**:

    - Pandas provides powerful methods for combining data from multiple sources through **merge()**, **concat()**, and **join()**, enabling SQL-like operations on DataFrames.

7. Extensibility and Compatibility

- **Third-Party Integrations**:
    - Pandas integrates seamlessly with libraries like **Matplotlib** (for plotting), **Seaborn** (for statistical graphics), **Scikit-learn** (for machine learning), and others.
- **Extensions**:
    - Pandas is extensible, allowing the addition of custom data types and functionalities. For instance, users can define their own custom data types and integrate them into Pandas workflows.

8. Threading and Parallelism (Optional)

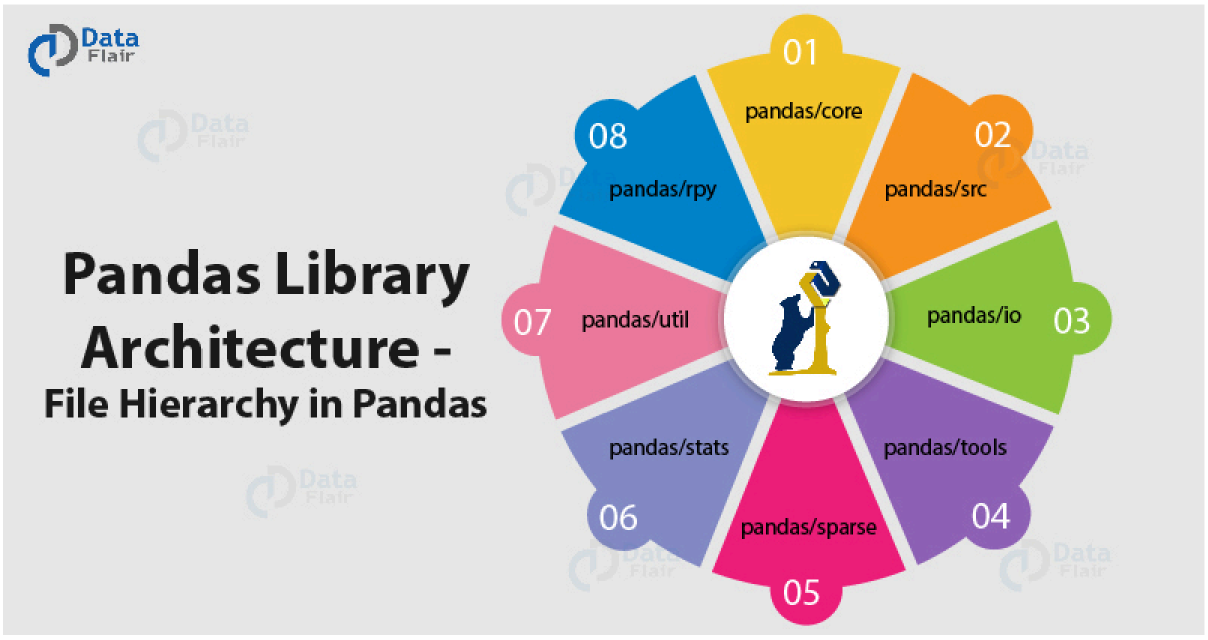
While Pandas itself is not inherently parallelized, you can extend its functionality to work with parallel computation using libraries such as **Dask** or **Modin**, which parallelize Pandas operations across multiple cores or nodes.

9. Future of Pandas Architecture

As the field of data science evolves, **Pandas** is also evolving to accommodate **big data** (through integration with distributed systems like **Dask** or **Spark**) and **machine learning workflows**. With the increasing size of datasets and complex operations, future versions of Pandas may introduce optimizations and better handling of out-of-core and distributed data.

Library Architecture

There are 8 types of files present in Pandas. The hierarchy of files is important to know the architecture of Pandas, so before starting with the architecture, let’s explore the hierarchy.



Pandas Library Architecture

The following list gives us an idea about the hierarchy of the files within Pandas Library Architecture:

1. pandas/core

In Pandas library architecture, this part consists of basic files about the data structures present within the library. For example, data structures – Series and DataFrames. There are various Python files within the core. The most important of them being:

- api.py: Important key modules which will be used later are imported using these files.
- base.py: This will provide the base for all the other classes present, like PandasObject and StringMIxin.
- common.py: It controls the common utility methods which help in handling various data structures.
- config.py: This helps to handle configurable objects found throughout the package.

These are the essential python classes which handle most of the working in the core of Pandas.

2. pandas/src

This contains algorithms which provide basic functionality to the library. The code here is usually written in C or Cython.

3. pandas/io

pandas/io, an essential part of the Pandas library architecture. This contains input and output tools which help Pandas handle files of various file formats. Essential modules found here are:

- api.py: This module handles various imports needed for input and output functions.
- auth.py: This module handles authentications and the methods dealing with it.
- common.py: Common functionality of input and output functions are taken care of by this module.
- data.py: This module helps to handle data with its input or output.

4. pandas/tools

The algorithms of pandas/tools are for auxiliary data. These help various functions like pivot, merge, join, concatenation, and other such functions for manipulating the data sets.

5. pandas/sparse

This part consists of sparse versions of various data structures like DataFrames and Series. A sparse version means that the data is mostly missing or unavailable.

6. pandas/stats

This part of the Pandas library architecture consists of a panel and linear regression and also contains moving window regression. Various statistics-related functions can be found in this portion.

7. pandas/util

Various utilities, testing tools, and development can be found here. In pandas/util, classes are used to test and debug any part of the library.

8. pandas/rpy

It consists of an interface to connect to R programming, called RPy2. Using Pandas with both R and Python can help you to have a much better grasp over data analysis.

The key features of Python Pandas

With Pandas module up and running, you can import your data into a DataFrame or Series and use Pandas' extensive functionality to manipulate, clean, and analyze that data. Key features and functions of Pandas include:

1. Data cleaning

Pandas offers various functions for cleaning and transforming your data, such as filling in missing values, dropping columns or rows, deleting NULL values and renaming columns.

2. Data filtering and selection

Pandas allow for a range of fine filtering and selection functions, based on highly granular conditions. So, no matter how complex the data is, you can extract the exact information you want.

3. Data aggregation

With Pandas, you can perform aggregation operations like groupby, pivot, and merge to summarize and restructure your data.

4. Data visualization

Pandas integrates with the popular data visualization library, Matplotlib, allowing you to create various types of plots and charts from your data.

The applications of Pandas in Python

What are the use cases for Pandas? Pandas is used across a range of data science and management fields, thanks to its army of applications:

1. Data cleaning and preprocessing

Pandas is an excellent tool for cleaning and preprocessing data. It offers various functions for handling missing values, transforming data, and reshaping data structures.

2. Data exploration

Pandas makes it easy to explore and understand your data. You can quickly calculate summary and basic statistics, filter multiple rows or tables, and visualize data using Pandas' integration with Matplotlib.

3. Feature engineering

Pandas provides robust functionality for creating new features from existing data, such as calculating aggregate statistics, creating dummy variables, and applying custom functions.

4. Time series analysis

Pandas has built-in support for handling time series data, streamlining work with time-stamped data, resampling operations, and rolling statistics calculations.

5. Data science

Pandas play a crucial role in preparing data for machine learning models. By cleaning, preprocessing, and transforming data with Pandas, you can create structured datasets that can be used with machine learning libraries like scikit-learn or TensorFlow.

Data Structures in Pandas

Intro to data structures

We’ll start with a quick, non-comprehensive overview of the fundamental data structures in pandas to get you started. The fundamental behavior about data types, indexing, axis labeling, and alignment apply across all of the objects. To get started, import NumPy and load pandas into your namespace:

DataFrame and Series are the two data structures that Pandas provides for processing data.

1) Series

A one-dimensional array capable of storing a variety of data types is how it is defined. The term "index" refers to the row labels of a series. We can without much of a stretch believe the rundown, tuple, and word reference into series utilizing "series' technique. Multiple columns cannot be included in a Series. Only one parameter exists:

Data: It can be any list, dictionary, or scalar value.

Creating Series from Array:

Before creating a Series, Firstly, we have to import the numpy module and then use array() function in the program.

```
import pandas as pd
import numpy as np
a = np.array(['R','a','g','h','u'])
b = pd.Series(a)
print("value in a -->",a)
print("type of a -->",type(a))
print("value in b -->",b)
print("type of b -->",type(b))
```

output

```
value in a --> ['R' 'a' 'g' 'h' 'u']

type of a --> <class 'numpy.ndarray'>

value in b --> 0    R

1    a

2    g

3    h

4    u

dtype: object

type of b --> <class 'pandas.core.series.Series'>
```

Example 2

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

# Creating the Series
sr = pd.Series(['New York', 'Chicago', 'Toronto', 'Lisbon'])

# Creating the row axis labels
sr.index = ['1 city', '2 city', '3 city', '4 city']

# Print the series
print(sr)
```

Output

```
C:\Users\REC\venky>python one.py

1 city    New York

2 city     Chicago

3 city     Toronto
```

```
4 city      Lisbon
dtype: object
```

Exampe3

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

# Creating the Series
sr = pd.Series([1000, 5000, 1500, 8222])

# Print the series
print(sr)

# return the data type
sr.dtype
```

Pandas **`Series.at`** attribute enables us to access a single value for a row/column label pair

```
sr.at[0]
```

Pandas series is a One-dimensional ndarray with axis labels. The labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index.

Pandas **`Series.iat`** attribute accesses a single value for a row/column pair by integer position.

```
sr.iat[0]
```

Pandas series is a One-dimensional ndarray with axis labels. The labels need not be unique but must be a hashable type. The object supports both integer- and label-based indexing and provides a host of methods for performing operations involving the index.

Pandas `Series.data` attribute returns the data pointer of the underlying data for the given Series object

```
# return the data pointer

sr.data
```

Use `Series.data` attribute to find the data pointer of the given Series object.

class pandas.Series(data=None, index=None,

dtype=None, name=None, copy=None, fastpath=<no_default>)[source]

pandas.Series.index

The base pandas index type.

The index (axis labels) of the Series.

Series.reindex

Conform Series to new index.

1) Example:

import pandas as pd

a=pd.Series([1,2,3,4,5])

print(a)

print('........')

print(a.index)

print('........')

print(a.reindex)

output:-

```
0    1

1    2

2    3

3    4

4    5

dtype: int64

........

RangeIndex(start=0, stop=5, step=1)

........

<bound method Series.reindex of 0    1

1    2

2    3

3    4

4    5

dtype: int64>
```

The ExtensionArray of the data backing this Series or Index.

example

```
import pandas as pd

a=pd.Series([1, 2, 3]).array

print(a)
```

Value

pandas.Series.values

property Series.values[source]

Return Series as ndarray of ndarray-like depending on the dtype.

example:

```
import pandas as pd

a=pd.Series([1, 2, 3]).values

print(a)
```

output

```
[1 2 3]
```

example:

```
import pandas as pd

a=pd.Series([1, 2, 3,4,5,6])

print(a.values)
```

Output:

```
[1 2 3 4 5 6]
```

example:

```
import pandas as pd

a=pd.Series(list('aabc')).astype('category').values

print(a)
```

----

pandas.Series.dtype

property Series.dtype[source]

Return the dtype object of the underlying data.

example:

```
import pandas as pd

s = pd.Series([1, 2, 3])

print(s.dtype)
```

Output:

```
int64
```

----

pandas.Series.shape

Return a tuple of the shape of the underlying data.

Example

```
import pandas as pd
s = pd.Series([1, 2, 3])
print(s.shape)
```

Output:

```
(3,)
```

----

pandas.Series.nbytes

Return the number of bytes in the underlying data.

Example:

```
import pandas as pd
s = pd.Series([1, 2, 3])
print(s.nbytes)
```

Output:

```
24
```

----

pandas.Series.size

Return the number of elements in the underlying data.

Example

```
import pandas as pd

s = pd.Series(['a', 'aa', 12.32,3])

print(s.size)
```

----

pandas.Series.memory_usage

Return the memory usage of the Series.

The memory usage can optionally include the contribution

of the index and of elements of object dtype.

Example

```
import pandas as pd

s = pd.Series(['a', 'aa', 12.32,123])

print(s.memory_usage())

print(s.nbytes)
```

Output:

```
164

32
```

Index Objects

In pandas, `Index` objects are used to label the rows or columns of `Series` and `DataFrame` objects. When working with a `Series`, the `Index` object holds the labels or identifiers of the data points, which allows for more intuitive data manipulation and retrieval.

Key Characteristics of `Index` Objects in a `Series`:

1. Immutable: You cannot change the values of an index once it's created.
2. Supports Labels: They can contain labels (which are typically strings or integers) that are used to reference the data.
3. Alignment: Pandas automatically aligns data in a `Series` based on the index labels.

Example of a pandas `Index` in a Series

```
import pandas as pd
# Create a simple Series
data = [10, 20, 30]
index = ['a', 'b', 'c']
series = pd.Series(data, index=index)
# Show the Series
print(series)
# Access the Index of the Series
print("Index of the Series:")
print(series.index)
```

output

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

Index of the Series:
Index(['a', 'b', 'c'], dtype='object')
```

Methods to Work with Index:

- Accessing Index: `series.index` returns the `Index` object of the Series.
- Index Operations: You can perform operations like slicing or searching within an `Index` object.

Example of Index Slicing and Searching:

```
# Slicing the index
print(series.index[:2])  # Output: Index(['a', 'b'], dtype='object')

# Checking if a label exists in the index
print('b' in series.index)  # Output: True
```

Common Methods of Index Objects:

- `.is_unique`: Check if the index has unique values.
- `.duplicated()`: Check which elements of the index are duplicated.
- `.get_loc()`: Return the integer location of a label.

```
print(series.index.is_unique)  # Output: True
print(series.index.get_loc('b'))  # Output: 1
```

Essential Functionality Reindexing

DataFrame

What is a DataFrame?

A DataFrame is a two-dimensional, labeled data structure with columns of potentially different types (similar to a table in a database or a spreadsheet).

How do you create a DataFrame from a dictionary?

- **You can create a DataFrame using `pd.DataFrame(data_dict)`, where `data_dict` is a dictionary of lists or arrays.**

How can you check the shape of a DataFrame?

- **Use the `.shape` attribute, e.g., `df.shape`, which returns a tuple `(rows, columns)`.**

What method is used to view the first few rows of a DataFrame?

- **The `head()` method, e.g., `df.head()`.**

How do you get the column names of a DataFrame?

- **Use the `.columns` attribute, e.g., `df.columns`.**

How can you select a single column from a DataFrame?

- **You can select a column using `df['column_name']` or `df.column_name`.**

How do you filter rows based on a condition?

- **Use boolean indexing, e.g., `df[df['column_name'] > 10]`.**

How can you add a new column to a DataFrame?

- **You can add a new column like this: `df['new_column'] = values`.**

What is the difference between `apply()` and `map()`?

- **`apply()` is used for applying a function along a DataFrame axis (rows/columns), while `map()` is typically used for element-wise transformations on a Series.**

How do you drop a column in a DataFrame?

- **Use the `drop()` method with `axis=1`, e.g., `df.drop('column_name', axis=1)`.**

How do you group a DataFrame by a column?

- **Use the `groupby()` method, e.g., `df.groupby('column_name')`.**

How can you sort a DataFrame by one or more columns?

- **Use the `sort_values()` method, e.g., `df.sort_values('column_name')`.**

What is the purpose of the `pivot()` function in Pandas?

- **The `pivot()` function reshapes data by converting unique values from one column into multiple columns.**

How do you rename columns in a DataFrame?

- **Use the `rename()` method, e.g., `df.rename(columns={'old_name': 'new_name'})`.**

How can you check for missing values in a DataFrame?

- **Use the `isnull()` method, e.g., `df.isnull().sum()`.**

Index Objects,

1 What is the purpose of indexing in Pandas?

- Indexing is used to label and select rows and columns in a DataFrame or Series, providing easy access to data for analysis and manipulation.

2 How can you set a custom index for a Pandas DataFrame?

- You can use the `set_index()` method, e.g., `df.set_index('column_name')`.

3 What is the difference between `loc` and `iloc` in Pandas?

- `loc` is label-based indexing, used for selecting rows/columns by labels. `iloc` is integer-location based indexing, used for selecting by index position.

4 How can you reset the index of a DataFrame?

- Use the `reset_index()` method, e.g., `df.reset_index()`.

5 What does the `[]` operator do in Pandas?

- It is used for selecting columns in a DataFrame or rows in a Series. You can also perform conditional filtering.

6 What is the default index in a Pandas DataFrame?

- The default index is a RangeIndex, which is a sequence of integers starting from 0.

7 How can you select rows where a column value meets a condition?

- You can use boolean indexing, e.g., `df[df['column_name'] > 10]`.

8 Can you perform slicing with Pandas DataFrames?

- Yes, you can slice rows using `loc` or `iloc`, e.g., `df.loc[2:5]` or `df.iloc[2:5]`.

10 How do you select a specific cell using `loc`?

- Use `df.loc[row_label, column_label]`.

11 What is a MultiIndex in Pandas?

- **A MultiIndex allows you to have multiple levels of indexing on rows or columns, enabling hierarchical data representation.**

Essential Functionality

Reindexing, Dropping entries from an axis, Indexing, selection, and filtering),Sorting and

ranking, Summarizing and Computing Descriptive Statistics, Unique Values, Value Counts,

Handling Missing Data, filtering out missing data.

DataFrame

A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

Example

[Get your own Python Server]

Create a simple Pandas DataFrame:

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

#load data into a DataFrame object:
df = pd.DataFrame(data)
print(df)
```

Locate Row

As you can see from the result above, the DataFrame is like a table with rows and columns.

Pandas use the `loc` attribute to return one or more specified row(s)

Example

```
print(df.loc[0])
```

Example

Return row 0 and 1:

```
#use a list of indexes:

print(df.loc[[0, 1]])
```

Note: When using `[]`, the result is a Pandas DataFrame.

Named Indexes

With the `index` argument, you can name your own indexes.

Example

Add a list of names to give each row a name:

```
import pandas as pd

data = {
  "calories": [420, 380, 390],
  "duration": [50, 40, 45]
}

df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
print(df)
```

Locate Named Indexes

Use the named index in the `loc` attribute to return the specified row(s).

Example

Return "day2":

```
#refer to the named index:

print(df.loc["day2"])
```

Index Objects

An Index is used to label the rows of a DataFrame or elements in a Series. These labels can be numbers, strings, or dates, and they help you to identify the data. One key thing to remember about Pandas indexes is that they are immutable, meaning you cannot change their size once created.

The Index Class

The Index class is a basic object for storing all index types in Pandas objects. It provides the basic functionality for accessing and manipulating data.

Key Features of Index Object

Immutable: Index object is a immutable sequence, which cannot modify once it is created.
Alignment: Index ensures that data from different DataFrames or Series can be combined correctly, based on the index values.
Slicing: Index allows fast slicing and retrieval of data based on labels.

Syntax

Following is the syntax of the Index class −

```
class pandas.Index(data=None, dtype=None, copy=False, name=None,
tupleize_cols=True)
```

data: The data for the index, which can be an array-like structure (like a list or numpy array) or another index object.

dtype: It specifies the data type for the index values, If not provided, Pandas will decide the data type based on the index values.

copy: It is a boolean parameter (True or False), which, specifies to create a copy of the input data.

name: This parameter gives a label to the index.

data: It is also a boolean parameter (True or False), When True, it tries to create MultiIndex if possible.

NumericIndex

A NumericIndex is the basic index type in Pandas, it contains numerical values. NumericIndex is a default index and Pandas automatically assigns this if you did not provided any index.

```
import pandas as pd

# Generate some data for DataFrame
data = {
   'Name': ['Steve', 'Lia', 'Vin', 'Katie'],
   'Age': [32, 28, 45, 38],
   'Gender': ['Male', 'Female', 'Male', 'Female'],
   'Rating': [3.45, 4.6, 3.9, 2.78]
}
# Creating the DataFrame
df = pd.DataFrame(data)

# Display the DataFrame
print(df)

print("\nDataFrame Index Object Type:",df.index.dtype)
```

Categorical Index

The CategoricalIndex is used to deal the duplicate labels. This index is efficient in terms of memory usage and handling the large number of duplicate elements.

```
import pandas as pd



# Creating a CategoricalIndex

categories = pd.CategoricalIndex(['a','b', 'a', 'c'])

df = pd.DataFrame({'Col1': [50, 70, 90, 60], 'Col2':[1, 3, 5, 8]},
index=categories)

print("Input DataFrame:\n",df)

print("\nDataFrame Index Object Type:",df.index.dtype)
```

IntervalIndex

An IntervalIndex is used to represent intervals (ranges) in your data. This type of index will be created using the interval_range() method.

Example

Following example creates a DataFrame with IntervalIndex using the interval_range() method.

```
import pandas as pd

# Creating a IntervalIndex
interval_idx = pd.interval_range(start=0, end=4)

# Creating a DataFrame with IntervalIndex
df = pd.DataFrame({'Col1': [1, 2, 3, 4], 'Col2':[1, 3, 5, 8]},
index=interval_idx)

print("Input DataFrame:\n",df)

print("\nDataFrame Index Object Type:",df.index.dtype)
```

MultiIndex

Pandas **MultiIndex** is used to represent multiple levels or layers in index of Pandas data structures, which is also called as hierarchical.

```
import pandas as pd


# Create MultiIndex
arrays = [[1, 1, 2, 2], ['red', 'blue', 'red', 'blue']]
multi_idx = pd.MultiIndex.from_arrays(arrays,
names=('number', 'color'))

# Create a DataFrame with MultiIndex
df = pd.DataFrame({'Col1': [1, 2, 3, 4], 'Col2':[1, 3, 5,
8]}, index=multi_idx)

print("MultiIndexed DataFrame:\n",df)
```

Basic DataFrame Operations:

- Filtering:

```
import pandas as pd

# From a dictionary

data = {

    'Name': ['Adam','Alice', 'Bob', 'Charlie'],

    'Age': [54,25, 30, 35],

    'City': ['Chicago

    ','New York', 'Los Angeles', 'Chicago']

}

df = pd.DataFrame(data)

print(df)

print('-----------age>25')

# Filter rows where Age is greater than 25

print(df[df['Age'] > 25])
```

In the above code it display age >25

Grouping:

```
print('-----------groupby city')

# Group by 'City' and calculate the average Age

print(df.groupby('City')['Age'].mean())
```

Working with Missing Data in Pandas

In Pandas, missing values are represented by None or NaN, which can occur due to uncollected data or incomplete entries. Let’s explore how to detect, handle, and fill in missing values in a DataFrame to ensure accurate analysis.

Checking for Missing Values in Pandas DataFrame

To identify and handle the missing values, Pandas provides two useful functions: isnull() and notnull(). These functions help detect whether a value is NaN or not, making it easier to clean and preprocess data in a DataFrame or Series.

1. Checking for Missing Values Using isnull()

isnull() returns a DataFrame of Boolean values, where True represents missing data (NaN). This is useful when you want to locate and address missing data within a dataset.

Example 1: Detecting Missing Values in a DataFrame

```
# Importing pandas and numpy

import pandas as pd

import numpy as np


# Sample DataFrame with missing values

data = {'First Score': [100, 90, np.nan, 95],

        'Second Score': [30, 45, 56, np.nan],

        'Third Score': [np.nan, 40, 80, 98]}


df = pd.DataFrame(data)
```

```
# Checking for missing values using isnull()

missing_values = df.isnull()


print(missing_values)
```

Example 2: Filtering Data Based on Missing Values

In this case, the isnull() function is applied to the “Gender” column to filter and display rows with missing gender information.

```
import pandas as pd


data = pd.read_csv("employees.csv")

bool_series = pd.isnull(data["Gender"])

missing_gender_data = data[bool_series]

print(missing_gender_data)
```

Checking for Missing Values Using notnull()

notnull() returns a DataFrame of Boolean values, where True indicates non-missing data. This function can be useful when you want to focus on the rows that contain valid, non-missing data.

Example 3: Detecting Non-Missing Values in a DataFrame

```
# Importing pandas and numpy

import pandas as pd

import numpy as np


# Sample DataFrame with missing values
```

```
data = {'First Score': [100, 90, np.nan, 95],

        'Second Score': [30, 45, 56, np.nan],

        'Third Score': [np.nan, 40, 80, 98]}


df = pd.DataFrame(data)


# Checking for non-missing values using notnull()

non_missing_values = df.notnull()


print(non_missing_values)
```

Example 4: Filtering Data with Non-Missing Values

This code snippet uses the notnull() function to filter out rows where the "Gender" column does not have missing values.

```python
# Importing pandas

import pandas as pd


# Reading data from a CSV file

data = pd.read_csv("employees.csv")


# Identifying non-missing values in the 'Gender' column

non_missing_gender = pd.notnull(data["Gender"])


# Filtering rows where 'Gender' is not missing

non_missing_gender_data = data[non_missing_gender]
```

```
display(non_missing_gender_data)
```

Filling Missing Values in Pandas Using fillna(), replace(), and interpolate()

When working with missing data in Pandas, the fillna(), replace(), and interpolate() functions are commonly used to fill NaN values. These functions allow you to replace missing values with a specific value or use interpolation techniques.

1. Filling Missing Values with a Specific Value Using fillna()

The fillna() function is used to replace missing values (NaN) with a specified value. For example, you can fill missing values with 0.

Example: Fill Missing Values with Zero

```
import pandas as pd

import numpy as np


dict = {'First Score': [100, 90, np.nan, 95],

        'Second Score': [30, 45, 56, np.nan],

        'Third Score': [np.nan, 40, 80, 98]}


df = pd.DataFrame(dict)


# Filling missing values with 0

df.fillna(0)
```

2. Filling Missing Values with the Prev/Next Value Using fillna

You can use the pad method to fill missing values with the previous value, or bfill to fill with the next value. We will be using the above dataset for the demonstration.

Example: Fill with Previous Value (Forward Fill)

df.fillna(method='pad') # Forward fill

Example: Fill with Next Value (Backward Fill)

```
df.fillna(method='bfill')   # Backward fill
```

Example: Fill NaN Values with ‘No Gender’ using fillna()

```
import pandas as pd

import numpy as np


data = pd.read_csv("employees.csv")

# Print records from 10th row to 24th row

data[10:25]
```

Now we are going to fill all the null values in Gender column with “No Gender”

```
# filling a null values using fillna()

data["Gender"].fillna('No Gender', inplace = True)

data[10:25]
```

3. Replacing Missing Values Using replace()

Use replace() to replace NaN values with a specific value like -99.

Example: Replace NaN with -99

```
import pandas as pd

import numpy as np


data = pd.read_csv("employees.csv")
```

```
data[10:25]
```

Now, we are going to replace the all Nan value in the data frame with -99 value.

```
data.replace(to_replace=np.nan, value=-99)
```

4. Filling Missing Values Using interpolate()

The interpolate() function fills missing values using interpolation techniques, such as the linear method.

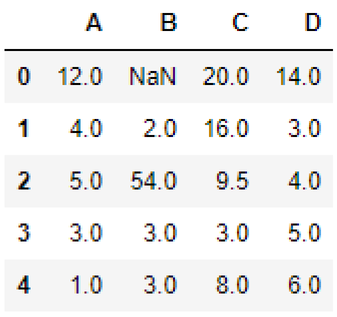
Example: Linear Interpolation

```
# importing pandas as pd

import pandas as pd


# Creating the dataframe

df = pd.DataFrame({"A": [12, 4, 5, None, 1],

                   "B": [None, 2, 54, 3, None],

                   "C": [20, 16, None, 3, 8],

                   "D": [14, 3, None, None, 6]})


# Print the dataframe

print(df)
```

Let’s interpolate the missing values using Linear method. Note that Linear method ignore the index and treat the values as equally spaced.

```
# to interpolate the missing values

df.interpolate(method ='linear', limit_direction ='forward')
```

Output:



	A	B	C	D
0	12.0	NaN	20.0	14.0
1	4.0	2.0	16.0	3.0
2	5.0	54.0	9.5	4.0
3	3.0	3.0	3.0	5.0
4	1.0	3.0	8.0	6.0

This method fills missing values by treating the data as equally spaced.

Dropping Missing Values in Pandas Using dropna()

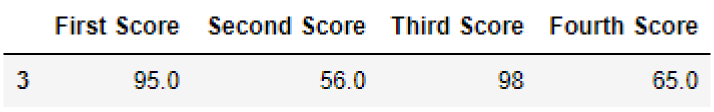
The dropna()function in Pandas removes rows or columns with NaN values. It can be used to drop data based on different conditions.

1. Dropping Rows with At Least One Null Value

Use dropna() to remove rows that contain at least one missing value.

Example: Drop Rows with At Least One NaN

```
import pandas as pd

import numpy as np


dict = {'First Score': [100, 90, np.nan, 95],

         'Second Score': [30, np.nan, 45, 56],

         'Third Score': [52, 40, 80, 98],

         'Fourth Score': [np.nan, np.nan, np.nan, 65]}


df = pd.DataFrame(dict)


# Drop rows with at least one missing value

df.dropna()
```

Output: 

	First Score	Second Score	Third Score	Fourth Score
3	95.0	56.0	98	65.0

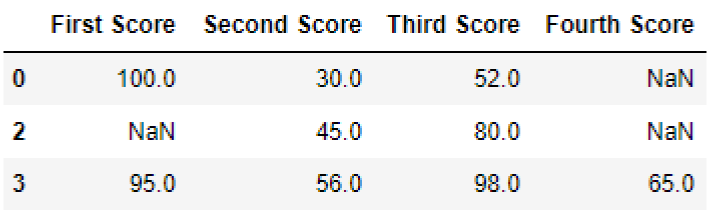
2. Dropping Rows with All Null Values

You can drop rows where all values are missing using dropna(how=’all’).

Example: Drop Rows with All NaN Values

```python
dict = {'First Score': [100, np.nan, np.nan, 95],

        'Second Score': [30, np.nan, 45, 56],

        'Third Score': [52, np.nan, 80, 98],

        'Fourth Score': [np.nan, np.nan, np.nan, 65]}


df = pd.DataFrame(dict)


# Drop rows where all values are missing

df.dropna(how='all')
```

Output:



	First Score	Second Score	Third Score	Fourth Score
0	100.0	30.0	52.0	NaN
2	NaN	45.0	80.0	NaN
3	95.0	56.0	98.0	65.0

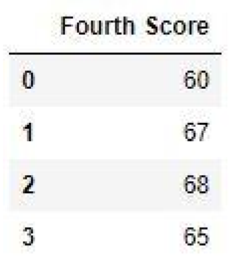
3. Dropping Columns with At Least One Null Value

To remove columns that contain at least one missing value, use dropna(axis=1).

Example: Drop Columns with At Least One NaN

```python
dict = {'First Score': [100, np.nan, np.nan, 95],
         'Second Score': [30, np.nan, 45, 56],

         'Third Score': [52, np.nan, 80, 98],

         'Fourth Score': [60, 67, 68, 65]}


df = pd.DataFrame(dict)


# Drop columns with at least one missing value

df.dropna(axis=1)
```

Output :



	Fourth Score
0	60
1	67
2	68
3	65

4. Dropping Rows with Missing Values in CSV Files

When working with data from CSV files, you can drop rows with missing values using dropna().

Example: Drop Rows with NaN in a CSV File

```python
import pandas as pd


data = pd.read_csv("employees.csv")


# Drop rows with any missing value

new_data = data.dropna(axis=0, how='any')


# Compare lengths of original and new dataframes
```

```
print("Old data frame length:", len(data))

print("New data frame length:", len(new_data))

print("Rows with at least one missing value:", (len(data) -
len(new_data)))
```

Output :

```
Old data frame length: 1000

New data frame length: 764

Rows with at least one missing value: 236
```

Since the difference is 236, there were 236 rows which had at least 1 Null value in any column.