**Pandas Tutorials**

Pandas is an open-source software library designed for **data manipulation** and **analysis**. It provides data structures like series and DataFrames to easily clean, transform and analyze large datasets and integrates with other Python libraries, such as NumPy and Matplotlib.

It offers functions for data transformation, aggregation and visualization, which are important for analysis. Created by Wes McKinney in 2008, Pandas widely used by data scientists, analysts and researchers worldwide. Pandas revolves around two primary Data structures: Series (1D) for single columns and DataFrame (2D) for tabular data enabling efficient data manipulation.

***Important Facts to Know :***

* ***DataFrames:*** *It is a two-dimensional data structure constructed with rows and columns, which is more similar to Excel spreadsheet.*
* ***pandas:*** *This name is derived for the term “panel data” which is* [*econometrics*](https://www.geeksforgeeks.org/econometrics-meaning-examples-theory-and-methods/) *terms of data sets.*

**What is Pandas Used for?**

With pandas, you can perform a wide range of data operations, including

* Reading and writing data from various file formats like CSV, Excel and SQL databases.
* Cleaning and preparing data by handling missing values and filtering entries.
* Merging and joining multiple datasets seamlessly.
* Reshaping data through pivoting and stacking operations.
* Conducting statistical analysis and generating descriptive statistics.
* Visualizing data with integrated plotting capabilities.

**Why Learn Pandas**

Here’s why it’s worth learning:

* It offers a simple and intuitive way to work with structured data, especially using DataFrames.
* Makes data exploration easy, so you can quickly understand patterns or spot issues.
* Saves time by reducing the need for complex code.
* It’s widely used in industries like finance, healthcare, marketing and research.
* A must-have skill for data science, analytics and machine learning roles.

**Pandas DataFrame**

A [DataFrame](https://www.geeksforgeeks.org/python-pandas-dataframe/" \t "_blank) is a two-dimensional, size-mutable and potentially heterogeneous tabular data structure with labeled axes (rows and columns).

* [Creating a DataFrame](https://www.geeksforgeeks.org/creating-a-pandas-dataframe/)

**Creating an Empty DataFrame**

An empty DataFrame in pandas is a table with no data but can have defined column names and indexes. It is useful for setting up a structure before adding data dynamically. An empty DataFrame can be created just by calling a dataframe constructor.

import pandas as pd

df = pd.DataFrame()

print(df)

**Creating a DataFrame from a List**

A simple way to create a [DataFrame is by using a single list.](https://www.geeksforgeeks.org/create-a-pandas-dataframe-from-lists/" \t "_blank) Pandas automatically assigns index values to the rows when you pass a list.

* Each item in the list becomes a row.
* The DataFrame consists of a single unnamed column.

import pandas as pd

lst = ['Geeks', 'For', 'Geeks', 'is',

'portal', 'for', 'Geeks']

df = pd.DataFrame(lst)

print(df)

**Creating DataFrame from dict of Numpy Array**

We can create a Pandas DataFrame using a dictionary of [NumPy arrays](https://www.geeksforgeeks.org/basics-of-numpy-arrays/). Each key in the dictionary represents a column name and the corresponding NumPy array provides the values for that column.

import numpy as np

import pandas as pd

data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

df = pd.DataFrame(data, columns=['A', 'B', 'C'])

print(df)

**Creating a DataFrame from a List of Dictionaries**

We can also [create dataframe using List of Dictionaries.](https://www.geeksforgeeks.org/create-pandas-dataframe-from-lists-using-dictionary/) It represents data where each dictionary corresponds to a row. This method is useful for handling structured data from APIs or JSON files. It is commonly used in web scraping and API data processing since JSON responses often contain lists of dictionaries.

import pandas as pd

dict = {'name':["aparna", "pankaj", "sudhir", "Geeku"],

'degree': ["MBA", "BCA", "M.Tech", "MBA"],

'score':[90, 40, 80, 98]}

df = pd.DataFrame(dict)

print(df)

* [Pandas Dataframe Index](https://www.geeksforgeeks.org/pandas-dataframe-index/?ref=ml_lbp)

**Pandas Dataframe Index**

Index in pandas dataframe act as reference for each row in dataset. It can be numeric or based on specific column values. The default index is usually a **RangeIndex** starting from 0, but you can customize it for better data understanding. You can easily access the current index of a dataframe using the index attribute. **Let's us understand with the help of an example:**

import pandas as pd

data = {'Name': ['John', 'Alice', 'Bob', 'Eve', 'Charlie'],

'Age': [25, 30, 22, 35, 28],

'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],

'Salary': [50000, 55000, 40000, 70000, 48000]}

df = pd.DataFrame(data)

print(df.index) # Accessing the index

**2. Setting a Custom Index**

**To set a custom index**, you can use the [set\_index()](https://www.geeksforgeeks.org/python-pandas-dataframe-set_index/" \t "_blank)method, allowing you to set a custom index based on a column, such as Name or Age.

*# Set 'Name' column as the index*

df\_with\_index = df.set\_index('Name')

print(df\_with\_index)

There are various operations you can perform with the DataFrame index, such as resetting it, changing it, or indexing with loc[]. Let's understand these as well:

**3. Resetting the Index**

If you need to reset the index back to **default integer index**, use[reset\_index()](https://www.geeksforgeeks.org/python-pandas-dataframe-reset_index/) method. This will convert the **current index into a regular column and create a new default index.**

*# Reset the index back to the default integer index*

df\_reset = df.reset\_index()

print(df\_reset)

**4. Indexing with loc**

The [loc[]](https://www.geeksforgeeks.org/python-pandas-dataframe-loc/)method in pandas allows to access rows and columns of a dataFrame using their labels, making it easy to retrieve specific data points.

row = df.loc['Alice']

print(row)

The **df.loc[]** function in Pandas is a very powerful method used for **label-based indexing** in DataFrames. It allows you to access a group of rows and columns by labels or a boolean array. The index or columns must be labels, not integer positions. It's one of the most commonly used functions for data selection and manipulation in Pandas.

**Syntax:**

python

CopyEdit

df.loc[row\_label, column\_label]

* **row\_label**: This can be:
  + A single label (row index).
  + A list of labels (multiple row indices).
  + A slice of labels.
  + A boolean array (same length as the number of rows).
* **column\_label**: This can be:
  + A single column label.
  + A list of column labels.
  + A slice of column labels.
  + A boolean array (same length as the number of columns).

**Key Features:**

1. **Label-based selection**: Works with row and column **labels** (i.e., index names or column names).
2. **Includes the endpoint in slices**: When using slices, df.loc[] includes the endpoint (contrary to df.iloc[]).
3. **Supports Boolean indexing**: You can use conditions to filter rows or columns based on boolean values.
4. **Can modify the DataFrame**: You can use df.loc[] to modify the data in a specific row/column.

**Examples of df.loc[]:**

**1. Select a Single Row by Label:**

python

CopyEdit

import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6]}

df = pd.DataFrame(data, index=['a', 'b', 'c'])

# Select row with label 'a'

print(df.loc['a'])

**Output:**

less

CopyEdit

A 1

B 4

Name: a, dtype: int64

**2. Select a Single Element by Row and Column Label:**

python

CopyEdit

# Select the element at row 'a' and column 'B'

print(df.loc['a', 'B'])

**Output:**

CopyEdit

4

**3. Select Multiple Rows and Columns by Labels:**

python

CopyEdit

# Select rows 'a' and 'c', and columns 'A' and 'B'

print(df.loc[['a', 'c'], ['A', 'B']])

**Output:**

css

CopyEdit

A B

a 1 4

c 3 6

**4. Select a Range of Rows Using Label Slicing:**

# Select rows from 'a' to 'b' (inclusive of 'b')

print(df.loc['a':'b'])

**Output:**

A B

a 1 4

b 2 5

**5. Select All Rows for a Specific Column:**

# Select all rows of column 'B'

print(df.loc[:, 'B'])

**Output:**

a 4

b 5

c 6

Name: B, dtype: int64

**6. Select Rows Based on Conditions (Boolean Indexing):**

# Select rows where column 'A' is greater than 1

print(df.loc[df['A'] > 1])

**Output:**

A B

b 2 5

c 3 6

**7. Modify Values Using df.loc[]:**

python

CopyEdit

# Change value of 'B' in row 'a' to 10

df.loc['a', 'B'] = 10

print(df)

**Output:**

A B

a 1 10

b 2 5

c 3 6

**8. Select Multiple Rows and Columns Using Boolean Arrays:**

# Select rows where column 'A' is greater than 1 and only 'A' column

print(df.loc[df['A'] > 1, 'A'])

**Output:**

b 2

c 3

Name: A, dtype: int64

**Detailed Breakdown of Parameters:**

1. **row\_label**:
   * Can be a single label or a list of labels (i.e., index values).
   * You can use a slice of labels, which will include the endpoint.
   * Boolean indexing is supported (you can filter rows based on a condition).
2. **column\_label**:
   * Can be a single column label or a list of column labels.
   * You can slice columns as well.
   * Boolean indexing is also possible for selecting specific columns based on conditions.

**Key Points to Remember:**

* **Includes the endpoint** when using slicing, unlike df.iloc[] which excludes the endpoint.
* **Supports both row and column indexing** simultaneously.
* Allows **modification** of DataFrame elements in place.
* Supports **Boolean indexing** for filtering data based on conditions.

**df.iloc in Pandas**

df.iloc[] is an indexer in Pandas used for **integer-location based indexing**. It allows you to select data from a DataFrame using row and column positions (integer indices), rather than the labels (which is done by df.loc[]). This makes df.iloc[] very useful when you want to work with the position of rows and columns directly.

**🔹 Basic Syntax:**

df.iloc[row\_indexer, column\_indexer]

* **row\_indexer**: The index or range of indices to select the rows.
* **column\_indexer**: The index or range of indices to select the columns.

Both row and column indexers are **0-based** (i.e., the first row/column has index 0).

**🔸 Parameters for df.iloc[]:**

| **Parameter** | **Description** |
| --- | --- |
| **row\_indexer** | The row(s) to select. It can be a single integer, a list of integers, a slice, or a boolean array. |
| **column\_indexer** | The column(s) to select. Similar to row\_indexer, it can be a single integer, a list of integers, a slice, or a boolean array. |

**🔹 Examples:**

**1. Selecting a single row:**

To select a specific row by its integer index:

import pandas as pd

data = {'A': [1, 2, 3], 'B': [4, 5, 6], 'C': [7, 8, 9]}

df = pd.DataFrame(data)

# Select the first row (index 0)

print(df.iloc[0])

**Output:**

A 1

B 4

C 7

Name: 0, dtype: int64

**2. Selecting a specific element (row, column):**

To select an element from a specific row and column using integer indices:

# Select the element at row 1 and column 2 (indexing starts from 0)

print(df.iloc[1, 2])

**Output:**

8

**3. Selecting multiple rows:**

You can select multiple rows by passing a list of indices:

# Select rows 0 and 2

print(df.iloc[[0, 2]])

**Output:**

A B C

0 1 4 7

2 3 6 9

**4. Selecting rows with a range (slice):**

You can use slices to select a range of rows:

# Select rows 1 to 2 (exclusive of index 3)

print(df.iloc[1:3])

**Output:**

css

CopyEdit

A B C

1 2 5 8

2 3 6 9

**5. Selecting all rows but specific columns:**

To select all rows for specific columns (e.g., the first two columns):

# Select all rows for the first two columns

print(df.iloc[:, :2])

**Output:**

css

CopyEdit

A B

0 1 4

1 2 5

2 3 6

**6. Selecting specific rows and specific columns:**

You can select a specific subset of rows and columns by combining row and column indexers:

# Select rows 0 and 2, and columns 0 and 1

print(df.iloc[[0, 2], [0, 1]])

**Output:**

A B

0 1 4

2 3 6

**7. Using boolean arrays for selection:**

You can use a boolean array for row selection. This is less common but can be useful in certain scenarios:

# Boolean array for selecting rows

print(df.iloc[[True, False, True], :])

**Output:**

A B C

0 1 4 7

2 3 6 9

**🔹 Common Methods with df.iloc[]:**

Although df.iloc[] is mainly for selecting rows and columns based on their integer position, you can combine it with other methods for filtering, manipulating, or analyzing data:

**1. .values: Convert DataFrame to a NumPy array:**

# Convert the values of the first two rows to a NumPy array

arr = df.iloc[:2].values

print(arr)

**Output:**

[[1 4 7]

[2 5 8]]

**2. .shape: Get the shape of the selected rows and columns:**

# Get the shape of the selection

print(df.iloc[:2, :2].shape)

**Output:**

(2, 2)

**3. .head() and .tail(): Preview first or last few rows:**

# Select the first 3 rows

print(df.iloc[:3].head())

**4. .iloc for setting values:**

You can also use iloc to modify values in your DataFrame:

# Change the value at row 0, column 1

df.iloc[0, 1] = 100

print(df)

**Output:**

A B C

0 1 100 7

1 2 5 8

2 3 6 9

**🔹 Key Differences: df.loc[] vs df.iloc[]**

* **df.loc[]**: Label-based indexing. It uses row and column labels (names).
* **df.iloc[]**: Integer-location based indexing. It uses the integer positions of rows and columns.

Example:

* df.loc[0, 'A']: Selects the value in row labeled 0 and column labeled 'A'.
* df.iloc[0, 0]: Selects the value in the first row and first column (index 0).

**🧠 Summary:**

* **df.iloc[]** is used for **integer-based** indexing, meaning you select rows and columns based on their integer positions (starting from 0).
* It supports various indexers, including integers, lists of integers, slices, and boolean arrays.
* It's especially useful for programmatically selecting or modifying parts of a DataFrame based on their location in the dataset.

**5. Changing the Index**

Change the index of dataFrame, with help of [set\_index()](https://www.geeksforgeeks.org/python-pandas-dataframe-set_index/" \t "_blank) method; allows to set one or more columns as the new index.

*# Set 'Age' as the new index*

df\_with\_new\_index = df.set\_index('Age')

print(df\_with\_new\_index)

Here are some Key Takeaways:

* Use .loc[] for label-based row selection and set\_index() to set custom indices.
* Access the index with .index and reset\_index() restores the default index, with an option to drop the old index.
* [Pandas Access DataFrame](https://www.geeksforgeeks.org/pandas-access-dataframe/)

Accessing a dataframe in pandas involves retrieving, exploring, and manipulating**data stored within this structure.**The most basic form of accessing a DataFrame is simply referring to it by its variable name. This will display the entire DataFrame, which includes all rows and columns.

import pandas as pd

data = {'Name': ['John', 'Alice', 'Bob', 'Eve', 'Charlie'],

'Age': [25, 30, 22, 35, 28],

'Gender': ['Male', 'Female', 'Male', 'Female', 'Male'],

'Salary': [50000, 55000, 40000, 70000, 48000]}

df = pd.DataFrame(data)

# Display the entire DataFrame

print(df)

**Output**

Name Age Gender Salary

0 John 25 Male 50000

1 Alice 30 Female 55000

2 Bob 22 Male 40000

3 Eve 35 Female 70000

4 Charlie 28 Male 48000

In addition to accessing the entire DataFrame there are several other methods to effectively retrieve and manipulate data within a Pandas DataFrame. Let's have a look on that:

**1. Accessing Columns From DataFrame**

Columns in a DataFrame can be accessed individually using bracket notation Accessing a column retrieves that column as a Series, which can then be further manipulated.

*# Access the 'Age' column*

age\_column = df['Age']

print(age\_column)

**Output**

0 25

1 30

2 22

3 35

4 28

Name: Age, dtype: int64

**2. Accessing Rows by Index**

To access specific rows in a DataFrame, you can use[iloc](https://www.geeksforgeeks.org/python-pandas-series-iloc/) (for positional indexing) or [loc](https://www.geeksforgeeks.org/python-pandas-dataframe-loc/) (for label-based indexing). These methods allow you to retrieve rows based on their index positions or labels.

*# Access the row at index 1 (second row)*

second\_row = df.iloc[1]

print(second\_row)

**Output**

Name Alice

Age 30

Gender Female

Salary 55000

Name: 1, dtype: object

**3. Accessing Multiple Rows or Columns**

You can access multiple rows or columns at once by passing a list of column names or index positions. This is useful when you need to select several columns or rows for further analysis.

*# Access the first three rows and the 'Name' and 'Age' columns*

subset = df.loc[0:2, ['Name', 'Age']]

print(subset)

**Output**

Name Age

0 John 25

1 Alice 30

2 Bob 22

**4. Accessing Rows Based on Conditions**

Pandas allows you to **filter rows** based on conditions, which can be very powerful for exploring subsets of data that meet specific criteria.

*# Access rows where 'Age' is greater than 25*

filtered\_data = df[df['Age'] > 25]

print(filtered\_data)

**Output**

Name Age Gender Salary

1 Alice 30 Female 55000

3 Eve 35 Female 70000

4 Charlie 28 Male 48000

**5. Accessing Specific Cells with at and iat**

If you need to access a specific cell, you can use the[.at[]](https://www.geeksforgeeks.org/python-pandas-dataframe-at/) method for label-based indexing and the[.iat[]](https://www.geeksforgeeks.org/python-pandas-dataframe-iat/) method for integer position-based indexing. These are optimized for fast access to single values.

*# Access the 'Salary' of the row with label 2*

salary\_at\_index\_2 = df.at[2, 'Salary']

print(salary\_at\_index\_2)

**Output**

40000

**Here are some Key Takeaways:**

1. Access a DataFrame by its variable name to view all data, and use bracket notation for columns and**loc/iloc**for rows.
2. Retrieve multiple rows or columns simultaneously by passing lists of names or indices.
3. Filter rows based on conditions to explore specific subsets of data effectively.

* [Indexing and Selecting Data with Pandas](https://www.geeksforgeeks.org/indexing-and-selecting-data-with-pandas/)

**Indexing in Pandas**refers to selecting specific rows and columns from a DataFrame. It allows you to subset data in various ways, such as selecting all rows with specific columns, some rows with all columns, or a subset of both rows and columns. This technique is also known as *Subset Selection*.

Let’s learn how to use different techniques for indexing and selecting data with Pandas.

**Indexing Data using the [] Operator**

The most straightforward way to index data in Pandas is by using the [**[] operator**](https://www.geeksforgeeks.org/python-pandas-dataframe/). This method can be used to select individual columns or multiple columns.

**Selecting a Single Column**

To select a single column, you simply reference the column name inside square brackets:

import pandas as pd

# Load the data

data = pd.read\_csv("nba.csv", index\_col="Name")

print("Dataset")

display(data.head(5))

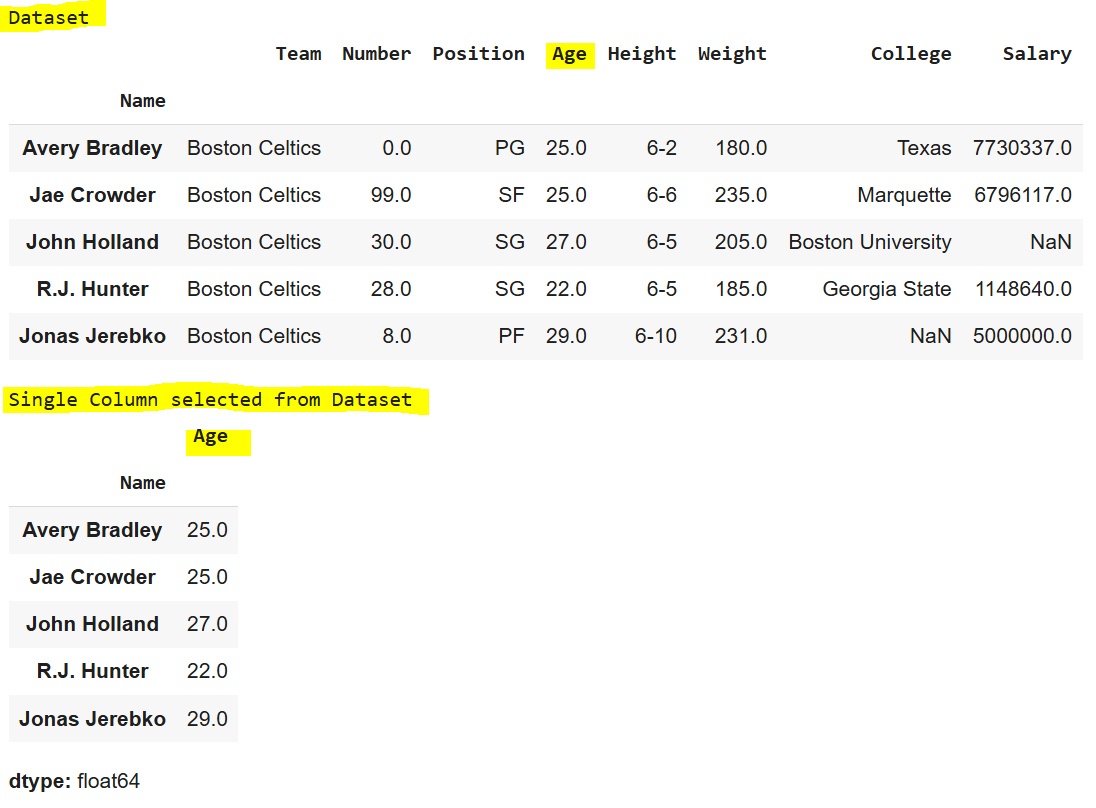
# Select a single column

first = data["Age"]

print("\nSingle Column selected from Dataset")

display(first.head(5))

**Output:**



**Selecting Multiple Columns**

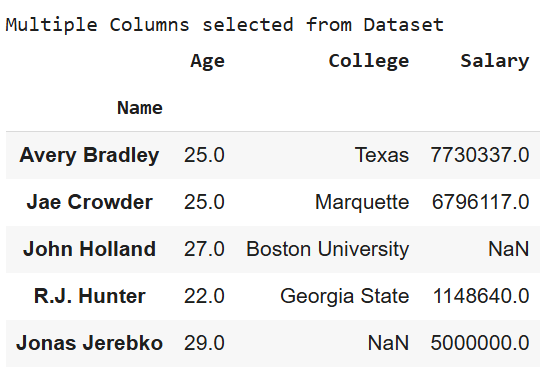
To select multiple columns, pass a list of column names:

first = data[["Age", "College", "Salary"]]

print("\nMultiple Columns selected from Dataset")

display(first.head(5))

**Output:**



Pandas offers several indexing methods to efficiently extract elements, rows, and columns from a DataFrame. These methods, while similar, have distinct behaviors. The four main types of indexing in Pandas are:

1. **DataFrame[]:** Known as the indexing operator, used for basic selection.
2. **DataFrame.loc[]:** Label-based indexing for selecting data by row/column labels.
3. **DataFrame.iloc[]:**Position-based indexing for selecting data by row/column integer positions.

Together, these indexing methods, also called*“indexers,”* are the most common ways to access data in a Pandas DataFrame.

**Indexing a DataFrame using .loc[ ]**

The[**.loc[]**](https://www.geeksforgeeks.org/python-pandas-dataframe-loc/) function in Pandas is used for selecting data by row and column labels. Unlike the indexing operator, **.loc[]** can select subsets of rows and columns simultaneously, offering flexibility in data retrieval.

**Selecting a single row**

To select a single row, provide the row label inside the **.loc[]** function:

# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("nba.csv", index\_col ="Name")

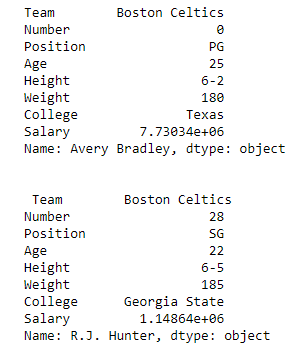
# retrieving row by loc method

first = data.loc["Avery Bradley"]

second = data.loc["R.J. Hunter"]

print(first, "\n\n\n", second)

**Output:**



As shown in the output image, two series were returned since there was only one parameter both of the times.

**Selecting multiple rows**

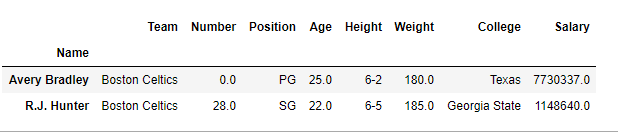
For multiple rows, pass a list of row labels to .loc[]:

# Select multiple rows

first = data.loc[["Avery Bradley", "R.J. Hunter"]]

display(first)

**Output:**



**Selecting Specific Rows and Columns**

To select specific rows and columns, provide both row labels and column names as lists:

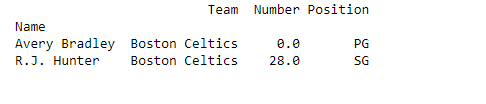
Dataframe.loc[["row1", "row2"], ["column1", "column2", "column3"]]

# Select two rows and three columns

first = data.loc[["Avery Bradley", "R.J. Hunter"], ["Team", "Number", "Position"]]

print(first)

**Output:**



**Selecting all of the rows and some columns**

To select all rows and specific columns, use a colon [:] to indicate all rows, followed by the list of column names:

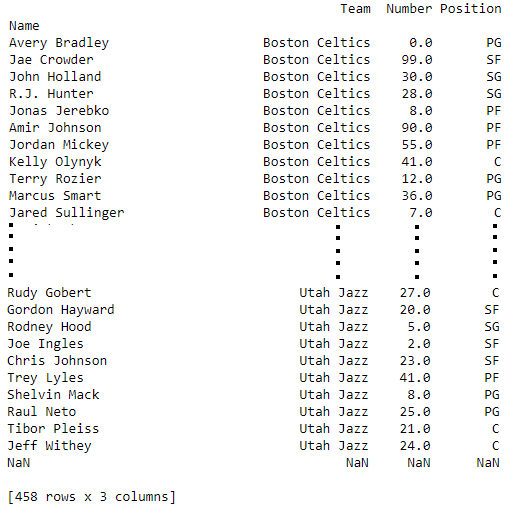
Dataframe.loc[:, ["column1", "column2", "column3"]]

# Select all rows and specific columns

first = data.loc[:, ["Team", "Number", "Position"]]

print(first)

**Output:**



The **.loc[]**method in Pandas offers a powerful way to index and filter data using labels, making it a core tool for data selection.

**Indexing a DataFrame using .iloc[ ]**

The [**.iloc[]**](https://www.geeksforgeeks.org/python-extracting-rows-using-pandas-iloc/)function in Pandas allows data selection based on integer positions (rather than labels). It is similar to **.loc[]**, but only accepts integer-based indices to specify rows and columns.

**Selecting a Single Row**

To select a single row using **.iloc[]**, provide the integer position of the row:

import pandas as pd

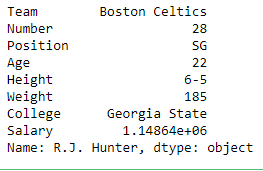
data = pd.read\_csv("nba.csv", index\_col="Name")

# Select a single row by position

row2 = data.iloc[3]

print(row2)

**Output:**



**Selecting Multiple Rows**

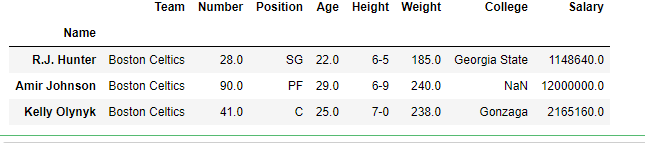
To select multiple rows, pass a list of integer positions:

# Select multiple rows by position

row2 = data.iloc[[3, 5, 7]]

display(row2)

**Output:**



**Selecting Specific Rows and Columns**

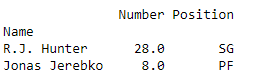
To select specific rows and columns, provide integer positions for both rows and columns:

# Select two rows and two columns by position

row2 = data.iloc[[3, 4], [1, 2]]

print(row2)

**Output:**



**Selecting All Rows and Some Columns**

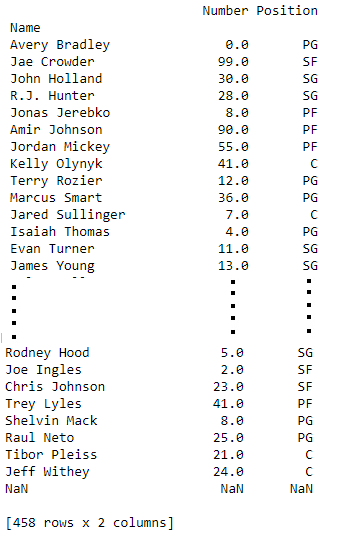
To select all rows and specific columns, use a colon **[:]** for all rows and a list of column positions:

# Select all rows and specific columns

row2 = data.iloc[:, [1, 2]]

print(row2)

**Output:**



The**.iloc[]** method is a powerful way to select data by position, offering flexibility in accessing subsets of a DataFrame based on integer indexing.

Other methods for indexing in a Pandas DataFrame include:

deprecated ones:

| **Function** | **Description** |
| --- | --- |
| [DataFrame.head()](https://www.geeksforgeeks.org/python-pandas-dataframe-series-head-method/) | Return top n rows of a DataFrame. |
| [DataFrame.tail()](https://www.geeksforgeeks.org/python-pandas-dataframe-series-tail-method/) | Return bottom n rows of a DataFrame. |
| [DataFrame.at[]](https://www.geeksforgeeks.org/python-pandas-dataframe-at/) | Access a single value for a row/column label pair. |
| [DataFrame.iat[]](https://www.geeksforgeeks.org/python-pandas-dataframe-iat/) | Access a single value for a row/column pair by integer position. |
| [DataFrame.lookup()](https://www.geeksforgeeks.org/label-based-indexing-to-the-pandas-dataframe/) | Label-based “fancy indexing” function for DataFrame. |
| [DataFrame.pop()](https://www.geeksforgeeks.org/python-pandas-dataframe-pop/) | Return item and drop from DataFrame. |
| [DataFrame.xs()](https://www.geeksforgeeks.org/python-pandas-series-xs/) | Return a cross-section (row(s) or column(s)) from the DataFrame. |
| [DataFrame.get()](https://www.geeksforgeeks.org/python-pandas-dataframe-get/) | Get item from object for given key (e.g., DataFrame column). |
| [DataFrame.isin()](https://www.geeksforgeeks.org/python-pandas-dataframe-isin/) | Return a boolean DataFrame showing whether each element is contained in values. |
| [DataFrame.where()](https://www.geeksforgeeks.org/python-pandas-dataframe-where/) | Return an object of the same shape with entries from self where cond is True, otherwise from other. |
| [DataFrame.mask()](https://www.geeksforgeeks.org/python-pandas-dataframe-mask/) | Return an object of the same shape with entries from self where cond is False, otherwise from other. |
| [DataFrame.query()](https://www.geeksforgeeks.org/pandas-query-method/) | Query the columns of a DataFrame with a boolean expression. |
| [DataFrame.insert()](https://www.geeksforgeeks.org/python-pandas-dataframe-insert/) | Insert a column into DataFrame at a specified location. |

This table contains only active, non-deprecated methods for DataFrame indexing in Pandas. Let me know if you need more details!

* [Slicing Pandas Dataframe](https://www.geeksforgeeks.org/slicing-indexing-manipulating-and-cleaning-pandas-dataframe/)

**Slicing Pandas Dataframe**

Slicing a [Pandas DataFrame](https://www.geeksforgeeks.org/python-pandas-dataframe/) is a important skill for extracting specific data subsets. Whether you want to select rows, columns or individual cells, Pandas provides efficient methods like[iloc[] and loc[]](https://www.geeksforgeeks.org/difference-between-loc-and-iloc-in-pandas-dataframe/). In this guide we’ll explore how to use integer-based and label-based indexing to slice DataFrames effectively.

**Create a Custom Dataframe**

Let’s import pandas library and create pandas dataframe from custom nested list.

**import** **pandas** **as** **pd**

player\_list = [['M.S.Dhoni', 36, 75, 5428000],

['A.B.D Villers', 38, 74, 3428000],

['V.Kohli', 31, 70, 8428000],

['S.Smith', 34, 80, 4428000],

['C.Gayle', 40, 100, 4528000],

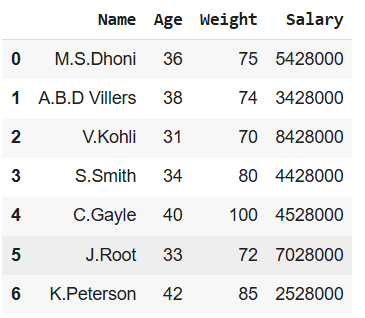
['J.Root', 33, 72, 7028000],

['K.Peterson', 42, 85, 2528000]]

df = pd.DataFrame(player\_list, columns=['Name', 'Age', 'Weight', 'Salary'])

print(df)

Output:



**Slicing Using iloc[] (Integer-Based Indexing)**

The[iloc[] method](https://www.geeksforgeeks.org/python-extracting-rows-using-pandas-iloc/)in Pandas allows us to extract specific rows and columns based on their integer positions starting from 0. Each number represents a position in the DataFrame not the actual label of the row or column.

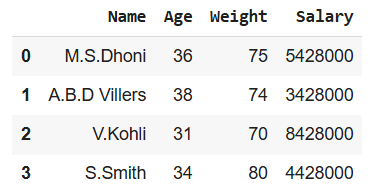
**Slicing Rows in dataframe**

Row slicing means selecting a specific set of rows from the DataFrame while keeping all columns.

df1 = df.iloc[0:4]

print(df1)

**Output**:



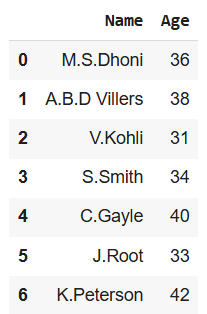
**Slicing Columns in dataframe**

Column slicing means selecting a specific set of columns from the DataFrame while keeping all rows.

df1 = df.iloc[:, 0:2]

print(df1)

**Output**:



**Selecting a Specific Cell in a Pandas DataFrame**

If you need a single value from a DataFrame you can specify the exact row and column position

value = df.iloc[2, 3]

print("Specific Cell Value:", value)

**Output**:

*Specific Cell Value: 8428000*

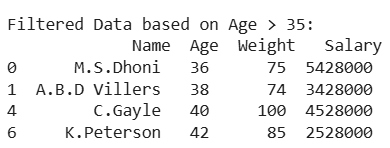
**Using Boolean Conditions in a Pandas DataFrame**

Instead of selecting rows by index we can use Boolean conditions (e.g., Age > 35) to filter rows dynamically.

data = df[df['Age'] > 35].iloc[:, :]

print("\nFiltered Data based on Age > 35:\n", data)

**Output**:



**Slicing Using loc[]**

We can also implement slicing using the loc function in Pandas but there are some limitations to be aware of. The loc function relies on labels meaning that if your DataFrame has custom labels instead of default integer indices you need to be careful with how you specify them.

**Slicing Rows in Dataframe**

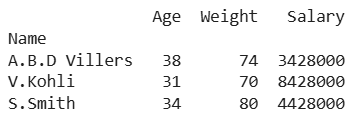
With loc[] we can extract a range of rows by their labels instead of integer positions.

df.set\_index('Name', inplace=True)

custom = df.loc['A.B.D Villers':'S.Smith']

print(custom)

**Output**:



**Selecting Specified cell in Dataframe**

[loc[]](https://www.geeksforgeeks.org/python-pandas-dataframe-loc/) allows us to fetch a specific value based on row and column labels

value = df.loc['V.Kohli', 'Salary']

print("\nValue of the Specific Cell (V.Kohli, Salary):", value)

**Output**:

*Value of the Specific Cell (V.Kohli, Salary): 8428000*

* [Filter Pandas Dataframe with multiple conditions](https://www.geeksforgeeks.org/filter-pandas-dataframe-with-multiple-conditions/)

**Filter Pandas Dataframe with multiple conditions**

The reason is dataframe may be having multiple columns and multiple rows. Selective display of columns with limited rows is always the expected view of users. To fulfill the user's expectations and also help in machine deep learning scenarios, filtering of Pandas dataframe with multiple conditions is much necessary.

Let us see the different ways to do the same.

**Creating a sample dataframe to proceed further**

*# import module*

**import** **pandas** **as** **pd**

*# assign data*

dataFrame = pd.DataFrame({'Name': [' RACHEL ', ' MONICA ', ' PHOEBE ',

' ROSS ', 'CHANDLER', ' JOEY '],

'Age': [30, 35, 37, 33, 34, 30],

'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

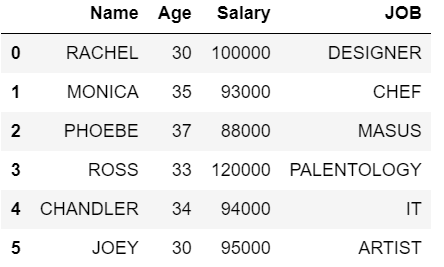
'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',

'IT', 'ARTIST']})

*# display dataframe*

display(dataFrame)

**Output:**



**Filter Pandas Dataframe with multiple conditions Using loc**

Here we will get all rows having Salary greater or equal to 100000 and Age < 40 and their JOB starts with ‘D’ from the dataframe. Print the details with Name and their JOB. For the above requirement, we can achieve this by using [**loc**](https://www.geeksforgeeks.org/python-pandas-extracting-rows-using-loc/). It is used to access single or more rows and columns by label(s) or by a boolean array. loc works with column labels and indexes.

*# import module*

**import** **pandas** **as** **pd**

*# assign data*

dataFrame = pd.DataFrame({'Name': [' RACHEL ', ' MONICA ', ' PHOEBE ',

' ROSS ', 'CHANDLER', ' JOEY '],

'Age': [30, 35, 37, 33, 34, 30],

'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',

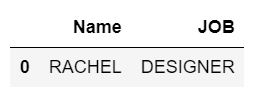
'IT', 'ARTIST']})

*# filter dataframe*

display(dataFrame.loc[(dataFrame['Salary']>=100000) & (dataFrame['Age']< 40) & (dataFrame['JOB'].str.startswith('D')),

['Name','JOB']])

**Output:**



Output resolves for the given conditions and finally, we are going to show only 2 columns namely Name and JOB.

**Filter Pandas Dataframe Using NumPy**

Here will get all rows having Salary greater or equal to 100000 and Age < 40 and their JOB starts with ‘D’ from the data frame. We need to use [NumPy](https://www.geeksforgeeks.org/python-numpy/).

*# import module*

**import** **pandas** **as** **pd**

**import** **numpy** **as** **np**

*# assign data*

dataFrame = pd.DataFrame({'Name': [' RACHEL', ' MONICA', ' PHOEBE', 'ROSS', 'CHANDLER', ' JOEY'],

'Age': [30, 35, 37, 33, 34, 30],

'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY', 'IT', 'ARTIST']})

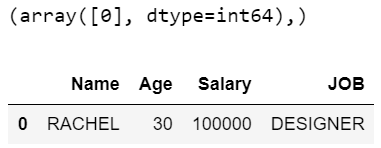
*# filter dataframe*

filtered\_values = np.where((dataFrame['Salary']>=100000) & (dataFrame['Age']< 40) & (dataFrame['JOB'].str.startswith('D')))

print(filtered\_values)

display(dataFrame.loc[filtered\_values])

**Output:**



In the above example, *print(filtered\_values)* will give the output as *(array([0], dtype=int64),)*  which indicates the first row with index value 0 will be the output. After that output will have 1 row with all the columns and it is retrieved as per the given conditions.

**Filter Pandas Dataframe Using Query (eval and query works only with columns)**

In this approach, we get all rows having Salary lesser or equal to 100000 and Age < 40, and their JOB starts with ‘C’ from the dataframe. Its just query the columns of a DataFrame with a single or more Boolean expressions and if multiple, it is having & condition in the middle.

*# import module*

**import** **pandas** **as** **pd**

*# assign data*

dataFrame = pd.DataFrame({'Name': [' RACHEL ', ' MONICA ', ' PHOEBE ',

' ROSS ', 'CHANDLER', ' JOEY '],

'Age': [30, 35, 37, 33, 34, 30],

'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

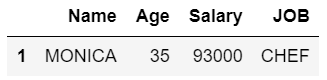
'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',

'IT', 'ARTIST']})

*# filter dataframe*

display(dataFrame.query('Salary <= 100000 & Age < 40 & JOB.str.startswith("C").values'))

**Output:**



**Pandas Boolean indexing multiple conditions standard way ("Boolean indexing" works with values in a column only)**

In this approach, we get all rows having Salary lesser or equal to 100000 and Age < 40 and their JOB starts with ‘P’ from the dataframe. In order to select the subset of data using the values in the dataframe and applying Boolean conditions, we need to follow these ways

*# import module*

**import** **pandas** **as** **pd**

*# assign data*

dataFrame = pd.DataFrame({'Name': [' RACHEL ', ' MONICA ', ' PHOEBE ',

' ROSS ', 'CHANDLER', ' JOEY '],

'Age': [30, 35, 37, 33, 34, 30],

'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

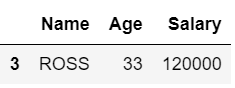
'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',

'IT', 'ARTIST']})

*# filter dataframe*

display(dataFrame[(dataFrame['Salary']>=100000) & (dataFrame['Age']<40) & dataFrame['JOB'].str.startswith('P')][['Name','Age','Salary']])

**Output:**



We are mentioning a list of columns that need to be retrieved along with the Boolean conditions and since many conditions, it is having '&'.

**Eval multiple conditions  ("eval" and "query" works only with columns )**

Here, we get all rows having Salary lesser or equal to 100000 and Age < 40 and their JOB starts with ‘A’ from the dataframe.

*# import module*

**import** **pandas** **as** **pd**

*# assign data*

dataFrame = pd.DataFrame({'Name': [' RACHEL ', ' MONICA ', ' PHOEBE ',

' ROSS ', 'CHANDLER', ' JOEY '],

'Age': [30, 35, 37, 33, 34, 30],

'Salary': [100000, 93000, 88000, 120000, 94000, 95000],

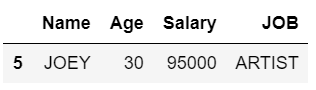
'JOB': ['DESIGNER', 'CHEF', 'MASUS', 'PALENTOLOGY',

'IT', 'ARTIST']})

*# filter dataframe*

display(dataFrame[dataFrame.eval("Salary <=100000 & (Age <40) & JOB.str.startswith('A').values")])

**Output:**



Dataframes are a very essential concept in Python and filtration of data is required can be performed based on various conditions. They can be achieved in any one of the above ways. Points to be noted:

* *loc* works with column labels and indexes.
* *eval* and *query* works only with columns.
* *Boolean indexing* works with values in a column only
* [Merging, Joining and Concatenating Dataframes](https://www.geeksforgeeks.org/python-pandas-merging-joining-and-concatenating/)

[Pandas DataFrame](https://www.geeksforgeeks.org/python-pandas-dataframe/) is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labelled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. We can join, merge, and concat dataframe using different methods.

In Dataframe df.merge(),df.join(), and df.concat() methods help in joining, merging and concating different dataframe.

**Concatenating DataFrame**

In order to concat dataframe, we use concat() function which helps in concatenating a dataframe. We can concat a dataframe in many different ways, they are:

* Concatenating DataFrame using .concat()
* Concatenating DataFrame by setting logic on axes
* Concatenating DataFrame using .append()
* Concatenating DataFrame by ignoring indexes
* Concatenating DataFrame with group keys
* Concatenating with mixed ndims

**Concatenating DataFrame using .concat() :**

In order to concat a dataframe, we use .concat() function this function concat a dataframe and returns a new dataframe.

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

data1 = {'Name': ['Jai', 'Princi', 'Gaurav', 'Anuj'],

'Age': [27, 24, 22, 32],

'Address': ['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],

'Qualification': ['Msc', 'MA', 'MCA', 'Phd']}

# Define a dictionary containing employee data

data2 = {'Name': ['Abhi', 'Ayushi', 'Dhiraj', 'Hitesh'],

'Age': [17, 14, 12, 52],

'Address': ['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],

'Qualification': ['Btech', 'B.A', 'Bcom', 'B.hons']}

​

# Convert the dictionary into DataFrame

df = pd.DataFrame(data1, index=[0, 1, 2, 3])

# Convert the dictionary into DataFrame

df1 = pd.DataFrame(data2, index=[4, 5, 6, 7])

print(df, "\n\n", df1)

Now we apply .concat function in order to concat two dataframe.

# using a .concat() method

frames = [df, df1]

res1 = pd.concat(frames)

res1

**Output :**  
As shown in the output image, we have created two dataframe after concatenating we get one dataframe.

**Concatenating DataFrame by setting logic on axes :**  
In order to concat dataframe, we have to set different logic on axes. We can set axes in the following three ways:

* Taking the union of them all, join='outer'. This is the default option as it results in zero information loss.
* Taking the intersection, join='inner'.
* Use a specific index, as passed to the join\_axes argument.

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

data1 = {'Name': ['Jai', 'Princi', 'Gaurav', 'Anuj'],

'Age': [27, 24, 22, 32],

'Address': ['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],

'Qualification': ['Msc', 'MA', 'MCA', 'Phd'],

'Mobile No': [97, 91, 58, 76]}

# Define a dictionary containing employee data

data2 = {'Name': ['Gaurav', 'Anuj', 'Dhiraj', 'Hitesh'],

'Age': [22, 32, 12, 52],

'Address': ['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],

'Qualification': ['MCA', 'Phd', 'Bcom', 'B.hons'],

'Salary': [1000, 2000, 3000, 4000]}

# Convert the dictionary into DataFrame

df = pd.DataFrame(data1, index=[0, 1, 2, 3])

# Convert the dictionary into DataFrame

df1 = pd.DataFrame(data2, index=[2, 3, 6, 7])

print(df, "\n\n", df1)

Now we set axes join = inner for intersection of dataframe.

# applying concat with axes

# join = 'inner'

res2 = pd.concat([df, df1], axis=1, join='inner')

res2

**Output :**  
As shown in the output image, we get the intersection of dataframe.

Now we set axes join = outer for union of dataframe.

# using a .concat for

# union of dataframe

res2 = pd.concat([df, df1], axis=1, sort=False)

res2

**Output :**  
As shown in the output image, we get the union of dataframe

Now we used a specific index, as passed to the join\_axes argument

# using join\_axes

res3 = pd.concat([df, df1], axis=1, join\_axes=[df.index])

res3

**Output :**

**Concatenating DataFrame with group keys :**

In order to concat dataframe with group keys, we override the column names with the use of the keys argument. Keys argument is to override the column names when creating a new DataFrame based on existing Series.

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],

'Age':[27, 24, 22, 32],

'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],

'Qualification':['Msc', 'MA', 'MCA', 'Phd']}

# Define a dictionary containing employee data

data2 = {'Name':['Abhi', 'Ayushi', 'Dhiraj', 'Hitesh'],

'Age':[17, 14, 12, 52],

'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],

'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame

df = pd.DataFrame(data1,index=[0, 1, 2, 3])

# Convert the dictionary into DataFrame

df1 = pd.DataFrame(data2, index=[4, 5, 6, 7])

print(df, "\n\n", df1)

Now we use keys as an argument.

# using keys

frames = [df, df1 ]

res = pd.concat(frames, keys=['x', 'y'])

res

**Output :**

**Concatenating with mixed ndims :**

User can concatenate a mix of Series and DataFrame. The Series will be transformed to DataFrame with the column name as the name of the Series.

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

5

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],

6

'Age':[27, 24, 22, 32],

7

'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],

8

'Qualification':['Msc', 'MA', 'MCA', 'Phd']}

9

10

# Convert the dictionary into DataFrame

11

df = pd.DataFrame(data1,index=[0, 1, 2, 3])

12

13

# creating a series

14

s1 = pd.Series([1000, 2000, 3000, 4000], name='Salary')

15

16

print(df, "\n\n", s1)

Now we are going to mix Series and dataframe together.

1

# combining series and dataframe

2

res = pd.concat([df, s1], axis=1)

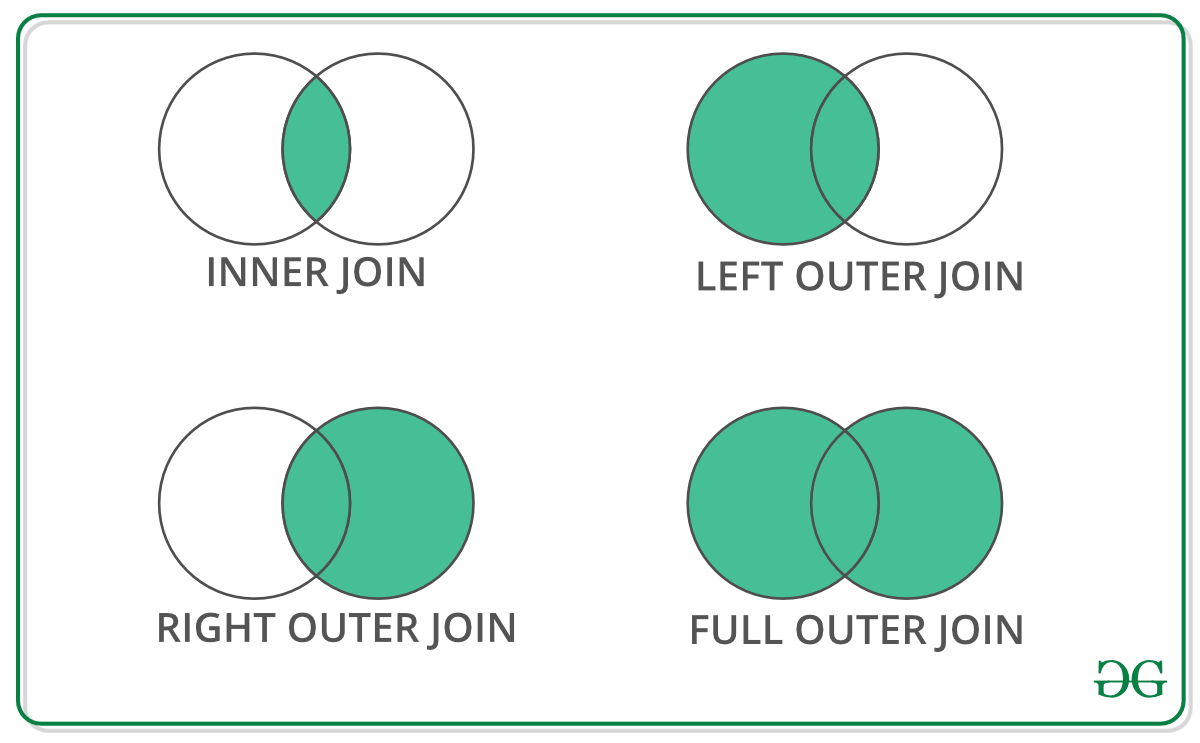
3

4

res

**Output :  
Merging DataFrame**

Pandas have options for high-performance in-memory merging and joining. When we need to combine very large DataFrames, joins serve as a powerful way to perform these operations swiftly. Joins can only be done on two DataFrames at a time, denoted as left and right tables. The key is the common column that the two DataFrames will be joined on. It’s a good practice to use keys which have unique values throughout the column to avoid unintended duplication of row values. Pandas provide a single function, merge(), as the entry point for all standard database join operations between DataFrame objects.

There are four basic ways to handle the join (inner, left, right, and outer), depending on which rows must retain their data.  
**Code #1 :** Merging a dataframe with one unique key combination

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

data1 = {'key': ['K0', 'K1', 'K2', 'K3'],

'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],

'Age':[27, 24, 22, 32],}

# Define a dictionary containing employee data

data2 = {'key': ['K0', 'K1', 'K2', 'K3'],

'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],

'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame

df = pd.DataFrame(data1)

# Convert the dictionary into DataFrame

df1 = pd.DataFrame(data2)

print(df, "\n\n", df1)

Now we are using .merge() with one unique key combination.

# using .merge() function

res = pd.merge(df, df1, on='key')

res

**Output :**  
  
**Code #2:** Merging dataframe using multiple join keys.

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

data1 = {'key': ['K0', 'K1', 'K2', 'K3'],

'key1': ['K0', 'K1', 'K0', 'K1'],

'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],

'Age':[27, 24, 22, 32],}

# Define a dictionary containing employee data

data2 = {'key': ['K0', 'K1', 'K2', 'K3'],

'key1': ['K0', 'K0', 'K0', 'K0'],

'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],

'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame

df = pd.DataFrame(data1)

# Convert the dictionary into DataFrame

df1 = pd.DataFrame(data2)

print(df, "\n\n", df1)

Now we merge dataframe using multiple keys.

# merging dataframe using multiple keys

res1 = pd.merge(df, df1, on=['key', 'key1'])

res1

**Output :**  
**Merging dataframe using how in an argument:**

We use how argument to merge specifies how to determine which keys are to be included in the resulting table. If a key combination does not appear in either the left or right tables, the values in the joined table will be NA. Here is a summary of the how options and their SQL equivalent names:

| **MERGE METHOD** | **JOIN NAME** | **DESCRIPTION** |
| --- | --- | --- |
| **left** | LEFT OUTER JOIN | Use keys from left frame only |
| **right** | RIGHT OUTER JOIN | Use keys from right frame only |
| **outer** | FULL OUTER JOIN | Use union of keys from both frames |
| **inner** | INNER JOIN | Use intersection of keys from both frames |

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

data1 = {'key': ['K0', 'K1', 'K2', 'K3'],

'key1': ['K0', 'K1', 'K0', 'K1'],

'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],

'Age':[27, 24, 22, 32],}

# Define a dictionary containing employee data

data2 = {'key': ['K0', 'K1', 'K2', 'K3'],

'key1': ['K0', 'K0', 'K0', 'K0'],

'Address':['Nagpur', 'Kanpur', 'Allahabad', 'Kannuaj'],

'Qualification':['Btech', 'B.A', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame

df = pd.DataFrame(data1)

# Convert the dictionary into DataFrame

df1 = pd.DataFrame(data2)

print(df, "\n\n", df1)

Now we set how = 'left' in order to use keys from left frame only.

# using keys from left frame

res = pd.merge(df, df1, how='left', on=['key', 'key1'])

res

**Output :**

 Now we set how = 'right' in order to use keys from right frame only.

# using keys from right frame

res1 = pd.merge(df, df1, how='right', on=['key', 'key1'])

res1

**Output :**

 Now we set how = 'outer' in order to get union of keys from dataframes.

# getting union of keys

res2 = pd.merge(df, df1, how='outer', on=['key', 'key1'])

res2

**Output :**

 Now we set how = 'inner' in order to get intersection of keys from dataframes.

# getting intersection of keys

res3 = pd.merge(df, df1, how='inner', on=['key', 'key1'])

res3

**Output :**

**Joining DataFrame**

In order to join dataframe, we use .join() function this function is used for combining the columns of two potentially differently-indexed DataFrames into a single result DataFrame.

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],

'Age':[27, 24, 22, 32]}

# Define a dictionary containing employee data

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],

'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame

df = pd.DataFrame(data1,index=['K0', 'K1', 'K2', 'K3'])

# Convert the dictionary into DataFrame

df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])

print(df, "\n\n", df1)

Now we are use .join() method in order to join dataframes

# joining dataframe

res = df.join(df1)

res

**Output :**

 Now we use how = 'outer' in order to get union

# getting union

res1 = df.join(df1, how='outer')

res1

**Output :**

**Joining dataframe using on in an argument :**

In order to join dataframes we use on in an argument. join() takes an optional on argument which may be a column or multiple column names, which specifies that the passed DataFrame is to be aligned on that column in the DataFrame.

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

data1 = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],

'Age':[27, 24, 22, 32],

'Key':['K0', 'K1', 'K2', 'K3']}

# Define a dictionary containing employee data

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kannuaj'],

'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame

df = pd.DataFrame(data1)

# Convert the dictionary into DataFrame

df1 = pd.DataFrame(data2, index=['K0', 'K2', 'K3', 'K4'])

print(df, "\n\n", df1)

Now we are using .join with “on” argument.

# using on argument in join

res2 = df.join(df1, on='Key')

res2

**Output :**

**Joining singly-indexed DataFrame with multi-indexed DataFrame :**

In order to join singly indexed dataframe with multi-indexed dataframe, the level will match on the name of the index of the singly-indexed frame against a level name of the multi-indexed frame.

# importing pandas module

import pandas as pd

# Define a dictionary containing employee data

data1 = {'Name':['Jai', 'Princi', 'Gaurav'],

'Age':[27, 24, 22]}

# Define a dictionary containing employee data

data2 = {'Address':['Allahabad', 'Kannuaj', 'Allahabad', 'Kanpur'],

'Qualification':['MCA', 'Phd', 'Bcom', 'B.hons']}

# Convert the dictionary into DataFrame

df = pd.DataFrame(data1, index=pd.Index(['K0', 'K1', 'K2'], name='key'))

index = pd.MultiIndex.from\_tuples([('K0', 'Y0'), ('K1', 'Y1'),

('K2', 'Y2'), ('K2', 'Y3')],

names=['key', 'Y'])

# Convert the dictionary into DataFrame

df1 = pd.DataFrame(data2, index= index)

print(df, "\n\n", df1)

Now we join singly indexed dataframe with multi-indexed dataframe.

# joining singly indexed with

# multi indexed

result = df.join(df1, how='inner')

* [Sorting Pandas DataFrame](https://www.geeksforgeeks.org/how-to-sort-pandas-dataframe/)

Pandas provides a powerful method called**sort\_values() that allows to sort the DataFrame based on one or more columns**. The method can sort in both ascending and descending order, handle missing values, and even apply custom sorting logic. To immediately understand how sorting works, let’s look at a simple example:

**1. Sort DataFrame by One Column Value**

To sort a DataFrame by a single column, you use the sort\_values() method and specify the column name using the by parameter.

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, 30, 35, 40],

'Score': [85, 90, 95, 80]}

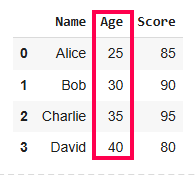
df = pd.DataFrame(data)

# Sorting by 'Age' in ascending order

sorted\_df = df.sort\_values(by='Age')

print(sorted\_df)

**Output:**



*Sort Pandas DataFrame*

In this example, the **DataFrame is sorted by the Age column in ascending order**. Now let’s dive deeper into how this works.

Sorting is essential when dealing with large datasets as it helps organize and interpret data more efficiently. **In Pandas, the sort\_values() method allows you to sort a DataFrame by one or more columns. By default, it sorts in ascending order but can be customized with various parameters.**

***Key Parameters of sort\_values():***

* ***by:*** *Specifies the column(s) to sort by.*
* ***ascending:*** *Boolean (default True). If False, sorts in descending order.*
* ***inplace:*** *If True, modifies the original DataFrame; otherwise returns a new sorted DataFrame.*
* ***na\_position:*** *Specifies whether to place NaN values at the beginning (‘first’) or end (‘last’).*
* ***ignore\_index:*** *If True, resets the index after sorting.*

By default, the sorting is done in ascending order. If you want to sort in descending order, you can set the ascending parameter to False.

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],'Age': [25, 30, 35, 40],'Score': [85, 90, 95, 80]}

df = pd.DataFrame(data)

# Sorting by 'Age' in descending order

sorted\_df = df.sort\_values(by='Age',ascending=False)

print(sorted\_df)

**Output**

Name Age Score

3 David 40 80

2 Charlie 35 95

1 Bob 30 90

0 Alice 25 85

**2. Sort DataFrame by Multiple Columns**

Sometimes, you need to sort your data based on multiple criteria. **For example, you might want to sort by age and then by name**. You can achieve this by passing a list of column names to the by parameter.

import pandas as pd

data = {'Name': ['Alice', 'Bob', 'Charlie', 'David'],

'Age': [25, 30, 35, 40],

'Score': [85, 90, 95, 80]}

df = pd.DataFrame(data)

# Sorting by 'Score' in ascending order

sorted\_df = df.sort\_values(by=['Age', 'Score'])

print(sorted\_df)

**Output**

Name Age Score

0 Alice 25 85

1 Bob 30 90

2 Charlie 35 95

3 David 40 80

This will first sort by Age, and if there are ties (same age), it will then sort by Score. You can also specify different sort orders for each column by using the ascending parameter with a list of boolean values.

**3. Sort DataFrame with Missing Values**

When datasets contain missing values, sorting behavior can be controlled using na\_position parameter in sort\_values(). By default, missing values are placed last, but you can place them first if needed.

import pandas as pd

data\_with\_nan = {"Name": ["Alice", "Bob", "Charlie", "David"],"Age": [28, 22, None, 22]}

df\_nan = pd.DataFrame(data\_with\_nan)

# Sort by 'Age', placing missing values first

sorted\_df = df\_nan.sort\_values(by="Age", na\_position="first")

print(sorted\_df)

**Output**

Name Age

2 Charlie NaN

1 Bob 22.0

3 David 22.0

0 Alice 28.0

The na\_position='first' option moves rows with NaN values to the top during sorting.

**Choosing the Sorting Algorithm**

Pandas allows you to specify the **sorting algorithm using the kind parameter**. The available options are:

* **'quicksort'**: Quicksort is a highly efficient, divide-and-conquer sorting algorithm. It selects a “pivot” element and partitions the dataset into two halves: one with elements smaller than the pivot and the other with elements greater than the pivot.
* **'mergesort'**: Divides the dataset into smaller subarrays, sorts them, and then merges them back together in sorted order.
* **'heapsort'**: Heapsort is another comparison-based sorting algorithm that builds a heap data structure to systematically extract the largest or smallest element and reorder the dataset.

To better demonstrate the behavior and benefits of using the 'mergesort' algorithm, particularly its stability, let’s modify the example to include duplicate values in the column being sorted.

import pandas as pd

# Create a DataFrame with duplicate 'Age' values

data = {

"Name": ["Alice", "Bob", "Charlie", "David", "Eve"],

"Age": [28, 22, 25, 22, 28],

"Score": [85, 90, 95, 80, 88]

}

9

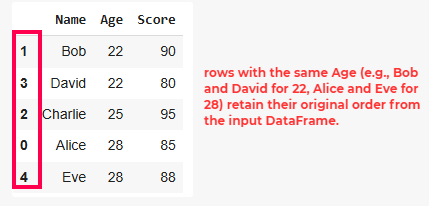
df = pd.DataFrame(data)

# Sort the DataFrame by 'Age' using the 'mergesort' algorithm

sorted\_df = df.sort\_values(by='Age', kind='mergesort')

print(sorted\_df)

**Output:**



*Sort Pandas DataFrame*

Stability ensures that the **relative order of rows with equal values in the sorting column is preserved.**

**Custom Sorting with Key Functions**

You can also apply custom sorting logic using the key parameter. For example, let’s say you want to sort strings ignoring case sensitivity:

import pandas as pd

data = {

"Name": ["Alice", "Bob", "Charlie", "David", "Eve"],

"Age": [28, 22, 25, 22, 28],

"Score": [85, 90, 95, 80, 88]

}

df = pd.DataFrame(data)

sorted\_df = df.sort\_values(by='Name', key=lambda col: col.str.lower())

print(sorted\_df)

**Output**

Name Age Score

0 Alice 28 85

1 Bob 22 90

2 Charlie 25 95

3 David 22 80

4 Eve 28 88

This ensures that names are sorted alphabetically without considering case differences.

**Key Takeaways:**

* **sort\_values()** is versatile and allows sorting by one or multiple columns.
* You can control whether sorting is ascending or descending using the ascending parameter.
* Missing values (NaN) can be placed at either the beginning or end using the na\_position parameter.
* Custom sorting logic can be applied using the **key** parameter.
* [Pivot Table in Pandas](https://www.geeksforgeeks.org/how-to-create-a-pivot-table-in-python-using-pandas/)

A pivot table is a statistical table that summarizes a substantial table like a big dataset. It is part of data processing. This summary in pivot tables may include mean, median, sum, or other statistical terms. Pivot tables are originally associated with MS Excel but we can create a pivot table in [Pandas](https://www.geeksforgeeks.org/pandas-tutorial/)using [Python](https://www.geeksforgeeks.org/python-programming-language/)using the [Pandas Dataframe](https://www.geeksforgeeks.org/python-pandas-dataframe/) pivot\_table() method.

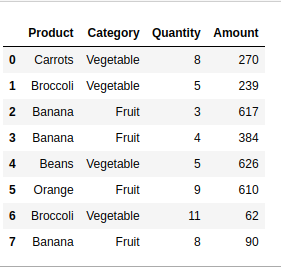
**Creating a Sample DataFrame**

Let’s first create a dataframe that includes Sales of Fruits.

* Python3

|  |
| --- |
| # importing pandas  **import** pandas as pd    # creating dataframe  df **=** pd.DataFrame({'Product': ['Carrots', 'Broccoli', 'Banana', 'Banana',                                 'Beans', 'Orange', 'Broccoli', 'Banana'],                     'Category': ['Vegetable', 'Vegetable', 'Fruit', 'Fruit',                                  'Vegetable', 'Fruit', 'Vegetable', 'Fruit'],                     'Quantity': [8, 5, 3, 4, 5, 9, 11, 8],                     'Amount': [270, 239, 617, 384, 626, 610, 62, 90]})  df |

**Output**



**Create a Pivot Table in Pandas**

Below are some examples to understand how we can create a pivot table in Pandas in Python:

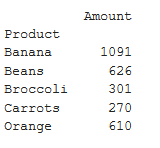
**Example 1: Get the Total Sales of Each Product**

In this example, the DataFrame ‘df’ is transformed using a pivot table, aggregating the total ‘Amount’ for each unique ‘Product’ and displaying the result with the sum of amounts for each product.

* Python3

|  |
| --- |
| pivot **=** df.pivot\_table(index**=**['Product'],                         values**=**['Amount'],                         aggfunc**=**'sum')  print(pivot) |

**Output**



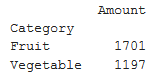
**Example 2: Get the Total Sales of Each Category**

In this example, a pivot table is created from the DataFrame ‘df’ to summarize the total ‘Amount’ sales for each unique ‘Category,’ employing the ‘sum’ aggregation function, and the result is printed.

* Python3

|  |
| --- |
| # creating pivot table of total  # sales category-wise aggfunc = 'sum'  pivot **=** df.pivot\_table(index**=**['Category'],                         values**=**['Amount'],                         aggfunc**=**'sum')  print(pivot) |

**Output**



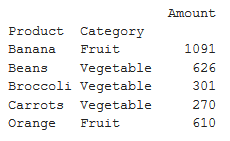
**Example 3: Get Total Sales by Category and Product Both**

In this example, a pivot table is generated from the DataFrame ‘df’ to showcase the total ‘Amount’ sales for unique combinations of ‘Product’ and ‘Category,’ utilizing the ‘sum’ [aggregation function](https://www.geeksforgeeks.org/aggregate-functions-in-sql/). The resulting pivot table is then printed.

* Python3

|  |
| --- |
| pivot **=** df.pivot\_table(index**=**['Product', 'Category'],                         values**=**['Amount'], aggfunc**=**'sum')  print(pivot) |

**Output**



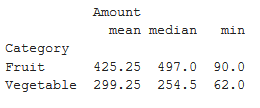
**Example 4: Get the Mean, Median, Minimum Sale by Category**

In this example, a pivot table is created from the DataFrame ‘df’ to display the median, mean, and minimum ‘Amount’ values categorized by ‘Category.’ The aggregation functions ‘median,’ ‘mean,’ and ‘min’ are applied, and the resulting pivot table is printed.

* Python3

|  |
| --- |
| # 'mean', 'min'} will get median, mean and  # minimum of sales respectively  pivot **=** df.pivot\_table(index**=**['Category'], values**=**['Amount'],                         aggfunc**=**{'median', 'mean', 'min'})  print(pivot) |

**Output**



**Example 5: Get the Mean, Median, Minimum Sale by Product**

In this example, a pivot table is generated from the DataFrame ‘df’ to showcase the median, mean, and minimum ‘Amount’ values for each unique ‘Product.’ The aggregation functions ‘median,’ ‘mean,’ and ‘min’ are applied, resulting in the pivot table, which is then printed.

* Python3

|  |
| --- |
| pivot **=** df.pivot\_table(index**=**['Product'], values**=**['Amount'],                         aggfunc**=**{'median', 'mean', 'min'})  print(pivot) |

**Output**

