Pandas Tutorial

**Pandas Series**

A[Series](https://www.geeksforgeeks.org/python-pandas-series/) is a one-dimensional labeled array capable of holding any data type (integers, strings, floating-point numbers, Python objects, etc.). It’s similar to a column in a spreadsheet or a database table.

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

**Creating a Pandas Series**

A [**Pandas Series**](https://www.geeksforgeeks.org/python-pandas-series/) is like a single column of data in a spreadsheet.***It is a one-dimensional array that can hold many types of data such as numbers, words or even other Python objects.*** Each value in a Series is associated with an **index**, which makes data retrieval and manipulation easy. This article explores multiple ways to create a Pandas Series with step-by-step explanations and examples.

**Creating an Empty Pandas Series**

An empty Series contains no data and can be useful when we plan to add values later. we can create an empty Series using the [pd.Series()](https://www.geeksforgeeks.org/python-pandas-series/" \t "_blank) function. By default an empty Series has a float64 data type. If we need a different data type specify it using the dtype parameter

1

import pandas as pd

ser = pd.Series()

print(ser)

**Output:**

*Series([], dtype: float64)*

**Creating a Series from a NumPy Array**

If we already have data stored in a [**NumPy array**](https://www.geeksforgeeks.org/basics-of-numpy-arrays/)we can easily convert it into a Pandas Series. This is helpful when working with numerical data.

import pandas as pd

import numpy as np

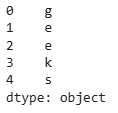
​

data = np.array(['g', 'e', 'e', 'k', 's'])

ser = pd.Series(data)

print(ser)

**Output:**



*Series Using Numpy Arrays*

**Creating a Series from a List**

we can [create a Series by passing a Python **list**](https://www.geeksforgeeks.org/creating-a-pandas-series-from-lists/) to the pd.Series() function. Pandas automatically assigns an index to each element starting from 0. This is a simple way to store and manipulate data.

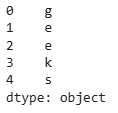
import pandas as pd

data\_list = ['g', 'e', 'e', 'k', 's']

ser = pd.Series(data\_list)

print(ser)

**Output:**



**Creating a Series from a Dictionary**

A dictionary in Python stores data as key-value pairs. When we [convert Dictionary into a Pandas Series](https://www.geeksforgeeks.org/creating-a-pandas-series-from-dictionary/) the keys become index labels and the values become the data. This method is useful for labeled data preserving structure and enabling quick access. Below is an example.

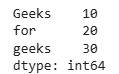
import pandas as pd

data\_dict = {'Geeks': 10, 'for': 20, 'geeks': 30}

ser = pd.Series(data\_dict)

print(ser)

**Output:**



*Series using Dictionary*

**Creating a Series Using NumPy Functions**

In order to [create a series using numpy function.](https://www.geeksforgeeks.org/create-pandas-series-using-numpy-functions/) Some commonly used NumPy functions for generating sequences include [numpy.linspace()](https://www.geeksforgeeks.org/numpy-linspace-python/" \t "_blank) for creating evenly spaced numbers over a specified range and [numpy.random.randn()](https://www.geeksforgeeks.org/numpy-random-randn-python/" \t "_blank) for generating random numbers from a normal distribution. This is particularly useful when working with scientific computations, statistical modeling or large datasets

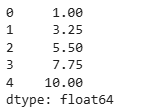
import numpy as np

import pandas as pd

ser = pd.Series(np.linspace(1, 10, 5))

print(ser)

**Output:**



*Series using Numpy Functions*

**Creating a Series Using range()**

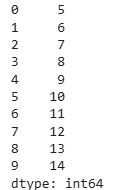
The [range()](https://www.geeksforgeeks.org/python-range-function/) function in Python is commonly used to generate sequences of numbers and it can be easily converted into a Pandas Series. This is particularly useful for creating a sequence of values in a structured format without need of manually specify each element. Below is an how range() can be used to create a Series.

import pandas as pd

ser = pd.Series(range(5, 15))

print(ser)

**Output:**



*Series using range()*

**Creating a Series Using List Comprehension**

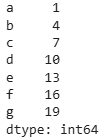
[List comprehension](https://www.geeksforgeeks.org/python-list-comprehension/)is a concise way to generate sequences and apply transformations in a single line of code. This method is useful when we need to create structured sequences dynamically. Below is an example demonstrating how list comprehension is used to create a Series with a custom index.

import pandas as pd

ser=pd.Series(range(1,20,3), index=[x for x in 'abcdefg'])

print(ser)

**Output:**



*Using List Comprehension*

Pandas provides multiple ways to create a Series. Understanding these methods will help us work efficiently with data in Pandas making it easier to analyze and process real-world datasets. With these examples we are now able to create and work with Pandas Series in different ways. Happy coding!

* [Accessing elements of a Pandas Series](https://www.geeksforgeeks.org/accessing-elements-of-a-pandas-series/)

Pandas Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). Labels need not be unique but must be a hashable type. An element in the series can be accessed similarly to that in an **ndarray**. Elements of a series can be accessed in two ways:

* **Accessing Element from Series with Position**
* **Accessing Element Using Label (index)**

In this article, we are using “nba.csv” file, to download the CSV, click [here](https://media.geeksforgeeks.org/wp-content/uploads/nba.csv).

**Accessing Element from Series with Position**

In order to access the series element refers to the index number. Use the [index](https://www.geeksforgeeks.org/python-list-index/)operator **[ ]** to access an element in a series. The index must be an integer.

In order to access multiple elements from a series, we use [Slice operation](https://www.geeksforgeeks.org/python-list-slicing/). Slice operation is performed on Series with the use of the colon(**:**). To print elements from beginning to a range use **[:Index]**, to print elements from end-use **[:-Index]**, to print elements from specific Index till the end use **[Index:]**, to print elements within a range, use [Start Index:End Index] and to print whole Series with the use of slicing operation, use **[:]**. Further, to print the whole Series in reverse order, use**[::-**1].

**Accessing the First Element of Series**

In this example, a [Pandas](https://www.geeksforgeeks.org/pandas-tutorial/)Series named ‘ser’ is created from a NumPy array ‘data’ containing the elements ‘g’, ‘e’, ‘e’, ‘k’, ‘s’, ‘f’, ‘o’, ‘r’, ‘g’, ‘e’, ‘e’, ‘k’, ‘s’. The first element of the series is accessed and printed using `print(ser[0])`

*# import pandas and numpy*

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

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

*# creating simple array*

data = np.array(['g', 'e', 'e', 'k', 's', 'f',

'o', 'r', 'g', 'e', 'e', 'k', 's'])

ser = pd.Series(data)

*# retrieve the first element*

print(ser[0])

**Output:**

**Accessing First 5 Elements of Series**

In this example, a Pandas Series named ‘ser’ is created from a [NumPy](https://www.geeksforgeeks.org/numpy-tutorial/)array ‘data’ containing the elements ‘g’, ‘e’, ‘e’, ‘k’, ‘s’, ‘f’, ‘o’, ‘r’, ‘g’, ‘e’, ‘e’, ‘k’, ‘s’. The first five elements of the series are accessed and printed using print(ser[:5]).

*# import pandas and numpy*

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

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

*# creating simple array*

data = np.array(['g', 'e', 'e', 'k', 's', 'f',

'o', 'r', 'g', 'e', 'e', 'k', 's'])

ser = pd.Series(data)

*# retrieve the first element*

print(ser[:5])

**Output:**

0 g  
1 e  
2 e  
3 k  
4 s  
dtype: object

**Accessing Last 10 Elements of Series**

In this example, a [Pandas Series](https://www.geeksforgeeks.org/python-pandas-series/) named ‘ser’ is created from a NumPy array ‘data’ containing the elements ‘g’, ‘e’, ‘e’, ‘k’, ‘s’, ‘f’, ‘o’, ‘r’, ‘g’, ‘e’, ‘e’, ‘k’, ‘s’. The last 10 elements of the series are accessed and printed using `print(ser[-10:])`.

*# import pandas and numpy*

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

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

*# creating simple array*

data = np.array(['g', 'e', 'e', 'k', 's', 'f',

'o', 'r', 'g', 'e', 'e', 'k', 's'])

ser = pd.Series(data)

*# retrieve the first element*

print(ser[-10:])

**Output:**

3 k  
4 s  
5 f  
6 o  
7 r  
8 g  
9 e  
10 e  
11 k  
12 s  
dtype: object

**Accessing First 5 Elements of Series in nba.csv File**

In this example, the Pandas module is imported, and a DataFrame ‘df’ is created by reading data from a [CSV file](https://www.geeksforgeeks.org/working-csv-files-python/) named “nba.csv” using `pd.read\_csv`. A Pandas Series ‘ser’ is then created by selecting the ‘Name’ column from the DataFrame. Finally, the first 10 elements of the series are accessed and displayed using ser.head(10).

*# importing pandas module*

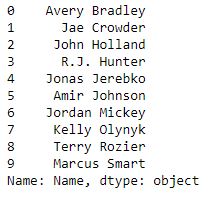
**import** **pandas** **as** **pd**

*# making data frame*

df = pd.read\_csv("nba.csv")

ser = pd.Series(df['Name'])

ser.head(10)

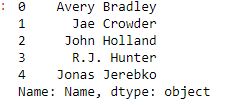


Now we access first 5 elements of series.

*# get first five names*

ser[:5]

**Output:**



**Access an Element in Pandas Using Label**

In order to access an element from series, we have to set values by index label. A Series is like a fixed-size dictionary in that you can get and set values by index label. Here, we will access an element in Pandas using label.

**Accessing a Single Element Using index Label**

In this example, a Pandas Series ‘ser’ is created from a [NumPy array](https://www.geeksforgeeks.org/basics-of-numpy-arrays/) ‘data’ with custom indices provided. The element at index 16 is accessed and printed using `print(ser[16])`.

*# import pandas and numpy*

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

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

*# creating simple array*

data = np.array(['g', 'e', 'e', 'k', 's', 'f',

'o', 'r', 'g', 'e', 'e', 'k', 's'])

ser = pd.Series(data, index=[10, 11, 12, 13, 14,

15, 16, 17, 18, 19, 20, 21, 22])

*# accessing a element using index element*

print(ser[16])

**Output:**

o

**Accessing a Multiple Element Using index Label**

In this example, a Pandas Series ‘ser’ is created from a NumPy array ‘data’ with custom indices provided. Multiple elements at indices 10, 11, 12, 13, and 14 are accessed and printed using `print(ser[[10, 11, 12, 13, 14]])`.

*# import pandas and numpy*

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

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

*# creating simple array*

data = np.array(['g', 'e', 'e', 'k', 's', 'f',

'o', 'r', 'g', 'e', 'e', 'k', 's'])

ser = pd.Series(data, index=[10, 11, 12, 13, 14,

15, 16, 17, 18, 19, 20, 21, 22])

*# accessing a multiple element using*

*# index element*

print(ser[[10, 11, 12, 13, 14]])

**Output:**

10 g  
11 e  
12 e  
13 k  
14 s  
dtype: object

**Access Multiple Elements by Providing Label of Index**

In this example, a Pandas Series ‘ser’ is created using NumPy’s [arange()](https://www.geeksforgeeks.org/numpy-arrange-in-python/) function with values from 3 to 8 and custom indices. Elements at indices ‘a’, ‘d’ are accessed and printed using `print(ser[[‘a’, ‘d’]])`.

*# importing pandas and numpy*

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

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

ser = pd.Series(np.arange(3, 9), index=['a', 'b', 'c', 'd', 'e', 'f'])

print(ser[['a', 'd']])

**Output:**

a 3.0  
d 6.0  
dtype: float64

**Accessing a Multiple Element Using Index Label in nba.csv File**

In this example, the Pandas module is imported, and a [DataFrame](https://www.geeksforgeeks.org/python-pandas-dataframe/)‘df’ is created by reading data from a CSV file named “nba.csv” using `pd.read\_csv`. A Pandas Series ‘ser’ is then created by selecting the ‘Name’ column from the DataFrame. Finally, the first 10 elements of the series are accessed and displayed using `ser.head(10)`.

*# importing pandas module*

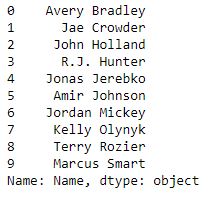
**import** **pandas** **as** **pd**

*# making data frame*

df = pd.read\_csv("nba.csv")

ser = pd.Series(df['Name'])

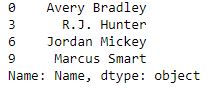
ser.head(10)



Now we access an multiple element using index label.

ser[[0, 3, 6, 9]]

**Output:**



* [Binary Operations on Series](https://www.geeksforgeeks.org/binary-operations-on-pandas-dataframe-and-series/)

Binary operations involve applying mathematical or logical operations on two objects, typically DataFrames or Series, to produce a new result. Let's learn how binary operations work in Pandas, focusing on their usage with DataFrames and Series.

The most common binary operations include:

* **Arithmetic operations**: Addition, subtraction, multiplication, division, etc.
* **Comparison operations**: Equal to, not equal to, greater than, less than, etc.
* **Logical operations**: And, or, etc.

Pandas makes it easy to perform these operations element-wise (i.e., on a per-row or per-column basis), which is particularly useful when working with large datasets.

**Binary Operations on Pandas Series**

**1. Arithmetic Operations on Series**

Arithmetic operations between two Series is applied element-wise. The index labels must align for the operation to work. If the indexes don’t match, Pandas will fill in missing values with NaN.

**Example: Adding Two Series**

import pandas as pd

s1 = pd.Series([10, 20, 30], index=['a', 'b', 'c'])

s2 = pd.Series([1, 2, 3], index=['a', 'b', 'c'])

# Adding the two Series

result = s1 + s2

print(result)

**Output**

a 11

b 22

c 33

dtype: int64

**2. Comparison Operations on Series**

Comparison operations return a Series of boolean values, indicating whether the comparison is True or False for each corresponding element.

**Example: Checking Equality**

import pandas as pd

s1 = pd.Series([10, 20, 30])

s2 = pd.Series([10, 25, 30])

# Comparing the two Series

result = s1 == s2

print(result)

**Output**

0 True

1 False

2 True

dtype: bool

**Binary Operations on Pandas DataFrame**

**1. Arithmetic Operations on DataFrames**

Similar to Series, DataFrame arithmetic operations apply element-wise between two DataFrames.

**Note:**The DataFrames must have the same shape or matching indexes and columns.

**Example: Subtracting DataFrames**

import pandas as pd

df1 = pd.DataFrame({'A': [10, 20, 30], 'B': [40, 50, 60]})

df2 = pd.DataFrame({'A': [1, 2, 3], 'B': [4, 5, 6]})

# Subtracting the DataFrames

result = df1 - df2

print(result)

**Output**

A B

0 9 36

1 18 45

2 27 54

**2. Comparison Operations on DataFrames**

Like Series, comparison operations on DataFrames return a DataFrame of boolean values. These boolean values indicate whether the corresponding elements are equal or satisfy other comparison conditions.

**Example: Checking Greater Than**

import pandas as pd

df1 = pd.DataFrame({'A': [10, 20, 30], 'B': [40, 50, 60]})

df2 = pd.DataFrame({'A': [5, 15, 35], 'B': [30, 60, 55]})

# Checking if elements of df1 are greater than df2

result = df1 > df2

print(result)

**Output**

A B

0 True True

1 True False

2 False True

**2. Comparison Operations on DataFrames**

Like Series, comparison operations on DataFrames return a DataFrame of boolean values. These boolean values indicate whether the corresponding elements are equal or satisfy other comparison conditions.

**Example: Checking Greater Than**

import pandas as pd

df1 = pd.DataFrame({'A': [10, 20, 30], 'B': [40, 50, 60]})

df2 = pd.DataFrame({'A': [5, 15, 35], 'B': [30, 60, 55]})

# Checking if elements of df1 are greater than df2

result = df1 > df2

print(result)

**Output**

A B

0 True True

1 True False

2 False True

**Logical Operations on DataFrame and Series**

Pandas also supports logical operations (AND, OR, etc.) on DataFrames and Series. These are commonly used for filtering and applying conditions.

**Example: Logical AND on Series**

import pandas as pd

s1 = pd.Series([True, False, True])

s2 = pd.Series([False, False, True])

# Applying logical AND

result = s1 & s2

print(result)

**Output**

0 False

1 False

2 True

dtype: bool

**Handling Missing Data in Binary Operations**

When performing binary operations on DataFrames or Series, missing data (NaN) can affect the results. Pandas handles missing data based on the operation:

* Arithmetic operations involving NaN will generally return NaN (e.g., NaN + 1 = NaN).
* Logical operations involving NaN might return False or True, depending on the operation.

**Example: Arithmetic with NaN**

import pandas as pd

df1 = pd.DataFrame({'A': [1, 2, None], 'B': [4, None, 6]})

df2 = pd.DataFrame({'A': [1, None, 3], 'B': [None, 5, 6]})

# Adding the DataFrames

result = df1 + df2

print(result)

**Output**

A B

0 2.0 NaN

1 NaN NaN

2 NaN 12.0

As seen above, where there is missing data (None or NaN), the result becomes NaN.

By leveraging these operations, you can perform complex calculations, comparisons, and transformations on your data, making Pandas a powerful tool for data analysis

* [Pandas Series Index() Methods](https://www.geeksforgeeks.org/python-pandas-series-index/)

[Pandas](https://www.geeksforgeeks.org/introduction-to-pandas-in-python/) Series is a **one-dimensional labeled array** capable of holding any data type (integers, strings, floats, etc.), with each element having an associated label known as its **index**. The**Series.index attribute** in Pandas allows users to get or set the index labels of a Series object, enhancing data accessibility and retrieval efficiency.**Example:**

import pandas as pd

data = pd.Series([10, 20, 30, 40], index=['a', 'b', 'c', 'd'])

# Accessing the index

print("Original Index:", data.index)

# Modifying the index

data.index = ['w', 'x', 'y', 'z']

print("Modified Series:\n", data)

**Output**

Original Index: Index(['a', 'b', 'c', 'd'], dtype='object')

Modified Series:

w 10

x 20

y 30

z 40

dtype: int64

**Explanation:** This code creates a Pandas Series with custom index labels (‘a’, ‘b’, ‘c’, ‘d’) and retrieves the index using**data.index.** It then updates the index to (‘w’, ‘x’, ‘y’, ‘z’).

**Syntax**

*Series.index # Access index labels*

*Series.index = new\_index # Modify index labels*

**Parameter:** This method does not take any parameter.

**Returns:** Index labels of the Series.

**Functionality:**

* Retrieves the current index labels of the Series.
* Can be used to set new index labels.
* Supports both unique and duplicate index labels.
* Useful for locating elements efficiently within a Series.

**Examples of Pandas Series Index() Attribute**

**Example 1.**Assigning Duplicate Index Labels

**Pandas**allows assigning duplicate index labels, which can be useful in cases where multiple elements share the same category.

import pandas as pd

series = pd.Series(['New York', 'Chicago', 'Toronto', 'Lisbon'])

# Creating the row axis labels

series.index = ['City 1', 'City 1', 'City 3', 'City 3']

print(series)

**Output**

City 1 New York

City 1 Chicago

City 3 Toronto

City 3 Lisbon

dtype: object

**Explanation:**Even with duplicate labels (**‘City 1’ and ‘City 3’** appearing twice), Pandas maintains the Series structure and ensures data integrity.

**Example 2.**Retrieving Index Labels

The **Series.index attribute** can also be used to retrieve the current index labels of a Series.

import pandas as pd

Date = ['1/1/2018', '2/1/2018', '3/1/2018', '4/1/2018']

idx\_name = ['Day 1', 'Day 2', 'Day 3', 'Day 4']

sr = pd.Series(data = Date,index = idx\_name)

print(sr.index)

**Output**

Index(['Day 1', 'Day 2', 'Day 3', 'Day 4'], dtype='object')

**Explanation:**The index labels (‘Day 1’ to ‘Day 4’) are assigned to a Series and retrieved using series.index.

**Example 3.** Resetting Index to Default

If needed, we can reset the index to default integer values.

import pandas as pd

Date = ['1/1/2018', '2/1/2018', '3/1/2018', '4/1/2018']

idx\_name = ['Day 1', 'Day 2', 'Day 3', 'Day 4']

sr = pd.Series(data = Date, # Series Data

index = idx\_name # Index

)

​

# Resetting index to default

sr.reset\_index(drop=True, inplace=True)

print(sr)

**Output**

0 1/1/2018

1 2/1/2018

2 3/1/2018

3 4/1/2018

dtype: object

**Explanation: reset\_index(drop=True, inplace=True)**removes the custom index and replaces it with the default integer index while modifying the Series in place.

* [Create a Pandas Series from array](https://www.geeksforgeeks.org/create-a-pandas-series-from-array/)

A [**Pandas Series**](https://www.geeksforgeeks.org/python-pandas-series/) is a one-dimensional labeled array that stores various data types, including numbers (integers or floats), strings, and Python objects. It is a fundamental data structure in the Pandas library used for efficient data manipulation and analysis. In this guide we will explore two simple methods to create a Pandas **Series** from a NumPy array.

**Creating a Pandas Series Without an Index**

By default when you create a Series from a NumPy array Pandas automatically assigns a numeric index starting from **0**. Here pd.Series(data) converts the array into a **Pandas Series** automatically assigning an index.

import pandas as pd

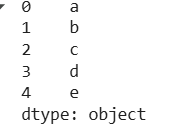
import numpy as np

data = np.array(['a', 'b', 'c', 'd', 'e'])

s = pd.Series(data)

print(s)

**Output:**



**Explanation:**

* The default index starts from **0** and increments by **1**.
* The data type (dtype: object) means it stores text values

**Creating a Pandas Series With a Custom Index**

In this method we specify custom indexes instead of using Pandas’ default numerical indexing. This is useful when working with structured data, such as **employee IDs, timestamps, or product codes** where meaningful indexes enhance data retrieval and analysis.

import pandas as pd

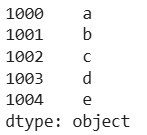
import numpy as np

data = np.array(['a', 'b', 'c', 'd', 'e'])

s = pd.Series(data, index=[1000, 1001, 1002, 1003, 1004])

print(s)

**Output:**



**Explanation:**

* Custom indexes (**1000, 1001, 1002…**) replace the default ones and allow meaningful data representation.
* Custom indexing enhances **data retrieval**, and make easier to access specific values directly using meaningful labels (e.g., s[1002] instead of s[2]).

Creating a Pandas Series from a NumPy array is simple and efficient. You can use the default index for quick access or assign a custom index for better data organization.