Common Python Libraries

Unit-7

Content

- Numpy: Introduction; Array Creating; Dimensions; Data Types, Array Attributes, Indexing and Slicing; Array Copy and View; Creating Array from Numerical Range; Array Broadcasting; Iterating Over Array; Sorting and Searching; Statistical Functions
- Pandas: Series and DataFrames; Creating DataFrames; The head and tail Functions; Attributes; Working with Missing Data; Indexing, Slicing, and Subsetting; Merging and Joining DataFrames; Working with CSV Files
- Matplotlib: Introduction; Marker; Line; Color; Label; Grid Lines; Subplot; Scatter Plot; Bar Graph; Histogram, pie chart and Box plot.

Numpy

Introduction

NumPy is a Python package. It stands for 'Numerical Python'.

NumPy is a powerful library in Python used for numerical computations. It provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. Here’s a brief introduction to its key features and functionality:

Numeric, the ancestor of NumPy, was developed by Jim Hugunin. Another package Numarray was also developed, having some additional functionalities. In 2005, Travis Oliphant created NumPy package by incorporating the features of Numarray into Numeric package. There are many contributors to this open source project.

Operations using NumPy

- Using NumPy, a developer can perform the following operations:
- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

Why Use NumPy?

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.

NUMPY − ENVIRONMENT

Standard Python distribution doesn't come bundled with NumPy module. A lightweight alternative is to install NumPy using popular Python package installer, pip.

pip3 install numpy

or

python3 setup.py install

To test whether NumPy module is properly installed, try to import it from Python prompt.

import numpy

If it is not installed, the following error message will be displayed.

Traceback (most recent call last):

File "<pyshell#0>", line 1, in <module>

import numpy

ImportError: No module named 'numpy'

Alternatively, NumPy package is imported using the following syntax:

```
import numpy as np
```

```
import numpy as np

arr = np.array([1,2,3,4,5])

print(arr)
```

Array in Numpy

In numpy, an array is the fundamental data structure. It is the powerful way to store and manipulate large collections of elements of same data type, offering significant advantages over standard Python list for numerical computing.

In Numpy, an array class is known as ndarray.

The primary array type in NumPy is ndarray (n-dimensional array), which is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers.

When you access an element in a NumPy array, you use a tuple of indices to specify its position. For example, in a 1-dimensional array, you would use a single integer index. In a 2-dimensional array, you would use a tuple (row_index, column_index).

Creating an array

Creating arrays in NumPy can be done in several ways

From Python Lists or Tuples:

You can create a NumPy array from a Python list or tuple using the np.array() function:

```
import numpy as np

# Create a 1-dimensional array from a list
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1)

# Create a 2-dimensional array (matrix) from a nested list
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
```

Using Built-in Functions

NumPy provides several built-in functions to create arrays with specific initial values:

Zeros and Ones: Create arrays of zeros or ones with a specified shape.

```
zeros_arr = np.zeros((2, 3))  # 2x3 array of zeros

ones_arr = np.ones((3, 2))    # 3x2 array of ones
```

Empty: Create an uninitialized array of a specified shape. The values in the array will be whatever happens to already exist at that memory location.

```
empty_arr = np.empty((2, 2))  # 2x2 uninitialized array
```

Arange: Create an array with values that are evenly spaced within a given interval.

```
range_arr = np.arange(0, 10, 2)  # array([0, 2, 4, 6, 8])
```

Linspace: Create an array with values that are evenly spaced over a specified interval.

```
linspace_arr = np.linspace(0, 1, 5)  # array([0.  , 0.25, 0.5 , 0.75, 1.  ]
```

Random: Create an array with random values from a uniform distribution.

```
random_arr = np.random.rand(2, 2)  # 2x2 array with random values [0, 1)
print(random_arr)
```

Creating a array with specific values

np.full(shape,value): create a array filled with specific values

Import numpy as np

array_with_seven = np.full((3,4),7)

creating array with ranged values

To create a NumPy array with ranged values, you have a few options depending on whether you want integers or floating-point numbers and whether you want to specify the step size or let NumPy decide it automatically.

np.arange() is similar to Python's built-in range() but returns an array instead of a list.

np.arange(start, stop, step)

```
import numpy as np

# Example 1: arange with start, stop, step
arr1 = np.arange(0, 10, 2)  # array([0, 2, 4, 6, 8])
print(arr1)

# Example 2: arange with only stop (start defaults to 0, step defaults to 1)
arr2 = np.arange(5)  # array([0, 1, 2, 3, 4])
print(arr2)
```

Using np.linspace()

np.linspace() generates evenly spaced numbers over a specified interval.

Syntax:

np.linspace(start, stop, num=50, endpoint=True)

start: Starting value of the range.

stop: Ending value of the range.

num (optional): Number of samples to generate (defaults to 50).

endpoint (optional): Whether to include stop in the range (default is True).

```
import numpy as np

# Example 1: linspace with start, stop, num
arr1 = np.linspace(0, 1, num=5)  # array([0.  , 0.25, 0.5 , 0.75, 1.  ])
print(arr1)

# Example 2: linspace with endpoint=False
arr2 = np.linspace(0, 10, num=5, endpoint=False)  # array([0., 2., 4., 6., 8.])
print(arr2)
```

creating identity matrix

To create an identity matrix in NumPy, you can use the **np.eye(n)** function. An identity matrix is a square matrix (same number of rows and columns) where all diagonal elements are 1 and all other elements are 0.

It is convenient way to generate identity matrix

The size of identity matrix is determined by the value passed to **n**

Using np.eye()

```
import numpy as np
# Creating a 3x3 identity matrix
identity_3 = np.eye(3)
print("3x3 Identity Matrix:")
print(identity_3)

# Creating a 4x4 identity matrix
identity_4 = np.eye(4)
print("\n4x4 Identity Matrix:")
print(identity_4)
```

creating array from string

To create a NumPy array from a string in Python, you typically need to parse the string into individual numeric values and then convert those values into an array.

np.array(string, dtype)

Steps to Create an Array from a String:

Parse the String: Extract numeric values from the string.

Convert to Numeric Format: Convert these values into a format that NumPy can use, typically floating-point numbers.

Create the NumPy Array: Use NumPy's array creation functions to construct the array.

```
import numpy as np
str_values = "1.2 3.4 5.6 7.8"

# Step 1: Parse the string into a list of numbers
numbers = [float(num_str) for num_str in str_values.split()]

# Step 2: Convert the list of numbers into a NumPy array
arr = np.array(numbers)

# Print the resulting array
print("Array from string:", arr)
```

Dimensions Array

Zero-dimensional Array (Scalar)

A zero-dimensional array in NumPy is essentially a scalar value. You can create a zero-dimensional array using np.array() with a single value:

```
import numpy as np

# Creating a zero-dimensional array (scalar)

arr_scalar = np.array(42)

print("Zero-dimensional Array (Scalar):")

print(arr_scalar)
```

Output:

```
Zero-dimensional Array (Scalar):

42
```

One-dimensional Array

A one-dimensional array in NumPy is similar to a Python list:

```
# Creating a 1D array
arr_1d = np.array([1, 2, 3, 4, 5])
print("\n1D Array:")
print(arr_1d)
```

Output:

```
1D Array:
[1 2 3 4 5]
```

Two-dimensional Array (Matrix)

A two-dimensional array in NumPy is like a table with rows and columns:

```
# Creating a 2D array (3x3 matrix)
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
print("\n2D Array (Matrix):")
print(arr_2d)
```

Output

2D Array (Matrix):

[[1 2 3]

 [4 5 6]

 [7 8 9]]

Three-dimensional Array

A three-dimensional array extends the concept to a cube or block of data:

```
# Creating a 3D array (2x3x4 array)
arr_3d = np.array([
                    [[1, 2, 3, 4],[5, 6, 7, 8], [9, 10, 11, 12]],
                    [[13, 14, 15, 16], [17, 18, 19, 20], [21, 22, 23, 24]]
                    ])
print("\n3D Array:")
print(arr_3d)
```

Creating 3D array using reshape()

The reshape() function in NumPy allows you to change the shape (dimensions) of an array without changing its data.

Reshaping is a fundamental operation when working with arrays of various dimensions.

```
import numpy as np
# Create a 1D array with 12 elements
arr = np.arange(12)
print("Original 1D array:")
print(arr)

# Reshape the 1D array into a 3x4 2D array
arr_2d = arr.reshape(3, 4)
print("\nReshaped 2D array (3x4):")
print(arr_2d)
```

Output:

Original 1D array:

[0 1 2 3 4 5 6 7 8 9 10 11]

Reshaped 2D array (3x4):

[[0 1 2 3]

 [4 5 6 7]

 [8 9 10 11]]

Data Types

In NumPy, data types (dtypes) specify the kind of elements that are contained within an array.

Choosing the appropriate data type is essential for memory efficiency, better performance and accurate calculation.

Each dtype defines the type and size of data elements, ensuring efficient memory usage and performance.

Common NumPy Data Types

- Basic numeric data types
- Integer (int): Represents whole numbers(positive, negative, or zero), with various sizes like
  - int8: Stores 8-bit signed integers, ranging from -128 to 127.
  - int16: Stores 16-bit signed integers, ranging from -32,768 to 32,767.
  - int32: Stores 32-bit signed integers, ranging from -2,147,483,648 to 2,147,483,647.
  - int64: Stores 64-bit signed integers, ranging from roughly -9.2 quintillion to 9.2 quintillion
- Unsigned Integer (uint): Similar to integers but can only store non-negative values, offering the same size variations
  - uint8: Stores unsigned 8-bit integers, ranging from 0 to 255. Useful for representing percentages, pixel values (grayscale images), or other non-negative whole numbers within this range.
  - uint16: Stores unsigned 16-bit integers, ranging from 0 to 65,535.
  - uint32: Stores unsigned 32-bit integers, ranging from 0 to 4,294,967,295.
  - uint64: Stores unsigned 64-bit integers, with a massive range of positive values

- Floating-point (float): Stores numbers with decimal points, with
  - float16: Half- precision floating-point format(2 bytes)
  - float32: Single- precision floating-point format(typically 6-7 decimal digit of precision)
  - float64: Double- precision floating-point format(typically 15-16 decimal digit of precision)

```
import numpy as np
pi = 3.1415926535897793
pi_float16 = np.float16(pi_exact)
pi_float32 = np.float32(pi_exact)
pi_float64 = np.float64(pi_exact)

print("Original pi:", pi_exact)
print("pi (float16):", pi_float16)  # Least precise, might lose some digits
print("pi (float32):", pi_float32)  # More precise than float16
print("pi (float64):", pi_float64)  # Most precise
```

Complex (complex): Represents complex numbers with real and imaginary parts, available in complex64 and complex128 for 64-bit and 128-bit precision.

Boolean (bool): Represents logical values, True or False.

Data type object

In the context of NumPy, a data type object (often referred to as a dtype object) isn't a specific data type itself, but rather an object that describes the properties of how data elements in a NumPy array are stored in memory. It essentially defines the blueprint for the array's elements.

Purpose:

Specifies the data type (e.g., integer, floating-point, string) of elements in a NumPy array.

Determines the size (in bits) allocated for each element.

Defines any additional information specific to the data type, like byte order or field formats for structured data types.

Example:

```
import numpy as np
# Create an array of integers
int_array = np.array([1, 2, 3])

# Get the data type object of the array
data_type = int_array.dtype

# Print the data type object
print(data_type)

#This code will typically output something like int32, indicating that the array int_array stores its elements as 32-bit signed integers.
```

Using Data Type Objects:

Specifying Data Type during Array Creation: You can explicitly define the data type using the dtype parameter in the np.array() function.

```
float_array = np.array([1.2, 3.5], dtype=np.float64)  # Specifying double precision
```

1. Using dtype parameter in np.array():

```
import numpy as np
# Create an array of integers with explicit data type (int32)
int_array = np.array([1, 2, 3], dtype=np.int32)
print(int_array.dtype)  # Output: int32

# Create an array of floats with single precision (float32)
float_array = np.array([1.2, 3.5], dtype=np.float32)
print(float_array.dtype)  # Output: float32

# Create an array of booleans
bool_array = np.array([True, False, True], dtype=np.bool_)
print(bool_array.dtype)  # Output: bool
```

2. Using NumPy functions with built-in data types:

np.zeros (creates array filled with zeros):

```
# Create array of 5 zeros with double precision
zeros_array = np.zeros(5, dtype=np.float64)
print(zeros_array.dtype)  # Output: float64
```

np.ones (creates array filled with ones):

```
# Create array of 10 ones with unsigned 8-bit integers
ones_array = np.ones(10, dtype=np.uint8)
print(ones_array.dtype)  # Output: uint8
```

Array Attributes

NumPy arrays come with several important attributes that provide valuable information about the array's structure and properties. These attributes include the shape, size, number of dimensions, data type, and more.

Key Attributes of NumPy Arrays

- ndarray.ndim: Number of dimensions (axes) of the array.
- ndarray.shape: Tuple of integers representing the size of the array in each dimension.
- ndarray.size: Total number of elements in the array.
- ndarray.dtype: Data type of the elements in the array.
- ndarray.itemsize: Size (in bytes) of each element in the array.
- ndarray.nbytes: Total number of bytes consumed by the elements of the array.
- ndarray.T: Transpose of the array (for 2D arrays and higher).
- ndarray.real: The real part of the array.
- ndarray.imag: The imaginary part of the array
- ndarray.flag: Information about the memory layout of the array

Examples of Using NumPy Array Attributes

1. Number of Dimensions: ndim

```
import numpy as np


# Create a 3x4 array of integers
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8],
                [9, 10, 11, 12]])


print("Array:")
print(arr)
print("\nNumber of dimensions (ndim):", arr.ndim)
```

Output

Number of dimensions (ndim): 2

Shape of the Array: shape

The shape attribute returns a tuple representing the dimensions of the array.

```
# Creating a 2D array
arr = np.array([[1, 2, 3], [4, 5, 6]])

print("Array:")
print(arr)
print("Shape of the array (shape):", arr.shape)
```

Output

Array:

[[1 2 3]

 [4 5 6]]

Shape of the array (shape): (2, 3)

Total Number of Elements: size

The size attribute returns the total number of elements in the array.

```
# Creating a 2D array
arr = np.array([[1, 2, 3], [4, 5, 6]])

print("Array:")
print(arr)
print("Total number of elements (size):", arr.size)
```

Output

Array:

[[1 2 3]

 [4 5 6]]

Total number of elements (size): 6

Data Type of Elements: dtype

The dtype attribute returns the data type of the elements in the array.

```
# Creating a 2D array

arr = np.array([[1, 2, 3], [4, 5, 6]])


print("Array:")

print(arr)

print("Data type of elements (dtype):", arr.dtype)
```

Output

Array:

[[1 2 3]

 [4 5 6]]

Data type of elements (dtype): int64

Size of Each Element (in Bytes): itemsize

The itemsize attribute returns the size (in bytes) of each element in the array.

```
# Creating a 2D array
arr = np.array([[1, 2, 3], [4, 5, 6]])

print("Array:")
print(arr)
print("Size of each element (itemsize):", arr.itemsize, "bytes")
```

Output

Array:

[[1 2 3]

 [4 5 6]]

Size of each element (itemsize): 8 bytes

Total Number of Bytes: nbytes

The nbytes attribute returns the total number of bytes consumed by the elements of the array.

```
# Creating a 2D array
arr = np.array([[1, 2, 3], [4, 5, 6]])


print("Array:")
print(arr)
print("Total number of bytes (nbytes):", arr.nbytes, "bytes")
```

Array:

[[1 2 3]

 [4 5 6]]

Total number of bytes (nbytes): 48 bytes

Transpose of the Array: T

The T attribute returns the transpose of the array, which swaps the axes of the array. For 2D arrays, this means converting rows to columns and vice versa.

```
# Creating a 2D array

arr = np.array([[1, 2, 3], [4, 5, 6]])


print("Original Array:")

print(arr)

print("\nTransposed Array (T):")

print(arr.T)
```

Original Array:

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

Transposed Array (T):

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

Real Part of the Array: ndarray.real

The real attribute returns the real part of the elements in the array. This is particularly useful when dealing with complex numbers.

```
import numpy as np

# Creating a complex array
complex_arr = np.array([1+2j, 3+4j, 5+6j])

print("Original Array:")
print(complex_arr)

print("\nReal Part of the Array (real):")
print(complex_arr.real)
```

Original Array:

[1.+2.j 3.+4.j 5.+6.j]

Real Part of the Array (real):

[1. 3. 5.]

Imaginary Part of the Array: ndarray.imag

The imag attribute returns the imaginary part of the elements in the array.

```
# Creating a complex array
complex_arr = np.array([1+2j, 3+4j, 5+6j])


print("Original Array:")
print(complex_arr)


print("\nImaginary Part of the Array (imag):")
print(complex_arr.imag)
```

Original Array:

[1.+2.j 3.+4.j 5.+6.j]

Imaginary Part of the Array (imag):

[2. 4. 6.]

Information About the Memory Layout of the Array: ndarray.flags

The flags attribute returns information about the memory layout of the array. This includes information such as whether the array is contiguous in memory (C_CONTIGUOUS), whether it is writable, etc.

- C_CONTIGUOUS: Indicates whether the array is stored in a single, contiguous block of memory (row-major order, C-style).
- F_CONTIGUOUS: Indicates whether the array is stored in column-major order (Fortran-style).
- OWNDATA: Indicates whether the array owns the memory it uses or if it is a view of another array.
- WRITEABLE: Indicates whether the array is writable.
- ALIGNED: Indicates whether the data is aligned in memory for efficient access.
- WRITEBACKIFCOPY: Used in advanced scenarios involving temporary copies of arrays.
- UPDATEIFCOPY: Similar to WRITEBACKIFCOPY, used for temporary copies.

```
# Creating a simple array
arr = np.array([[1, 2, 3], [4, 5, 6]])

print("Array:")
print(arr)

print("\nMemory Layout Information (flags):")
print(arr.flags)
```

Array:
[[1 2 3]
 [4 5 6]]

Memory Layout Information (flags):
  C_CONTIGUOUS : True
  F_CONTIGUOUS : False
  OWNDATA : True
  WRITEABLE : True
  ALIGNED : True
  WRITEBACKIFCOPY : False
  UPDATEIFCOPY : False

Indexing and Slicing

Indexing

- Accessing the elements of a numPy array at certain index is known as indexing in numPy array.
- Refers to accessing a specific element within a sequence using its position.
- Sequences have an inherent order, and each element has a unique index, starting from 0.
- You use square brackets [] to access elements by their index.

Slicing

- A technique to extract a portion of a sequence based on a range of indices.
- Uses square brackets [] with a colon : separating the start and end indices.
- Similar to indexing, slicing starts from 0 and goes up to, but doesn't include, the end index.

Indexing in NumPy Arrays

Basic Indexing: You can access elements of a NumPy array using square brackets and indices. NumPy arrays are zero-indexed, meaning the first element has index 0.

Example

```
import numpy as np
# Create a sequence of integers from
a = np.arange(10, 1, -2)
print("\n A sequential array with a negative step: \n",a)

# Indexes are specified inside the np.array method.
narr = a[np.array([3, 1, 2 ])]
print("\n Elements at these indices are:\n",narr)
```

A sequential array with a negative step:

[10 8 6 4 2]

Elements at these indices are:

[4 8 6]

```
import numpy as np

# NumPy array with elements from 1 to 9
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

# Index values can be negative.
arr = x[np.array([1, 3, -3])]
print("\n Elements are : \n",arr)
```

Elements are:

[2 4 7]

Get fourth and fifth element and print them

```
import numpy as np

# NumPy array with elements from 1 to 9
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])

print(arr[4]+ arr[5])
```

11

```
import numpy as np
# Creating a 2D array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print("Original Array:")
print(arr)

# Accessing specific elements
print("\nElement at index (0, 0):", arr[0, 0])
print("Element at index (1, 2):", arr[1, 2])
```

Output

Original Array:

[[1 2 3]

 [4 5 6]

 [7 8 9]]

Element at index (0, 0): 1

Element at index (1, 2): 6

numPy array Slicing

Array Slicing is the process of extracting a portion of an array.

With slicing, we can easily access elements in the array. It can be done on one or more dimensions of a NumPy array.

Syntax of NumPy Array Slicing

array[start:stop:step]

Here,

start - index of the first element to be included in the slice

stop - index of the last element (exclusive)

step - step size between each element in the slice

Note: When we slice arrays, the start index is inclusive but the stop index is exclusive.

If we omit start, slicing starts from the first element

If we omit stop, slicing continues up to the last element

If we omit step, default step size is 1

1D NumPy Array Slicing

```
import numpy as np

# create a 1D array
array1 = np.array([1, 3, 5, 7, 8, 9, 2, 4, 6])

# slice array1 from index 2 to index 6 (exclusive)
print(array1[2:6])  # [5 7 8 9]

# slice array1 from index 0 to index 8 (exclusive) with a step size of 2
print(array1[0:8:2])  # [1 5 8 2]

# slice array1 from index 3 up to the last element
print(array1[3:])  # [7 8 9 2 4 6]

# items from start to end
print(array1[:])   # [1 3 5 7 8 9 2 4 6]
```

Modify Array Elements Using Slicing

1. Using start Parameter

```
import numpy as np


# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])


# modify elements from index 3 onwards
numbers[3:] = 20
print(numbers)


# Output: [ 2  4  6 20 20 20]
```

2. Using stop Parameter

```
import numpy as np

# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])

# modify the first 3 elements
numbers[:3] = 40
print(numbers)

# Output: [40 40 40  8 10 12]
```

3. Using start and stop parameter

```
import numpy as np

# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])

# modify elements from indices 2 to 5
numbers[2:5] = 22
print(numbers)

# Output: [2 4 22 22 22 12]
```

Using start, stop, and step parameter

```
import numpy as np

# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])

# modify every second element from indices 1 to 5
numbers[1:5:2] = 16
print(numbers)

# Output: [ 2 16  6 16 10 12]
```

NumPy Array Negative Slicing

```
import numpy as np

# create a numpy array
numbers = np.array([2, 4, 6, 8, 10, 12])

# slice the last 3 elements of the array
# using the start parameter
print(numbers[-3:])    # [8 10 12]

# slice elements from 2nd-to-last to 4th-to-last element
# using the start and stop parameters
print(numbers[-5:-2])    # [4 6 8]

# slice every other element of the array from the end
# using the start, stop, and step parameters
print(numbers[-1::-2])   # [12 8 4]
```

Array Copy and View

In NumPy, you can create copies or views of arrays. Understanding the difference between a copy and a view is crucial for efficient memory usage and data manipulation.

Copy of Array

A copy of an array creates a new array object with its own data. Modifying the copy will not affect the original array, and vice versa.

```
import numpy as np

# Creating an original array
original_array = np.array([1, 2, 3, 4, 5])

# Creating a copy of the array
copied_array = original_array.copy()

print("Original Array:")
print(original_array)

print("\nCopied Array:")
print(copied_array)

# Modifying the copied array
copied_array[0] = 99

print("\nModified Copied Array:")
print(copied_array)

print("\nOriginal Array after modifying the copy:")
print(original_array)
```

Original Array:

[1 2 3 4 5]

Copied Array:

[1 2 3 4 5]

Modified Copied Array:

[99 2 3 4 5]

Original Array after modifying the copy:

[1 2 3 4 5]

View of Array

A view of an array is a new array object that looks at the same data as the original array. Modifying the view will affect the original array, and vice versa.

```
# Creating an original array
original_array = np.array([1, 2, 3, 4, 5])

# Creating a view of the array
view_array = original_array.view()

print("Original Array:")
print(original_array)

print("\nView Array:")
print(view_array)

# Modifying the view array
view_array[0] = 99

print("\nModified View Array:")
print(view_array)

print("\nOriginal Array after modifying the view:")
print(original_array)
```

Original Array:

[1 2 3 4 5]

View Array:

[1 2 3 4 5]

Modified View Array:

[99 2 3 4 5]

Original Array after modifying the view:

[99 2 3 4 5]

The Difference Between Copy and View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

Creating array from numerical range

Creating arrays from numerical ranges allows for efficient generation of arrays with specific numerical sequences which are essential for various data manipulation task.

There are several functions designed to create arrays with specific numerical ranges:

1. numpy.arange()

The numpy.arange() function returns evenly spaced values within a given interval.

Syntax:

numpy.arange([start,]stop, [step,]dtype=None)

start: The starting value of the sequence (default is 0).

stop: The end value of the sequence (not included in the array).

step: The difference between each pair of consecutive values (default is 1).

dtype: The data type of the output array.

```
import numpy as np

# Creating an array from 0 to 9
arr1 = np.arange(10)
print("Array from 0 to 9:", arr1)

# Creating an array from 5 to 20 with a step of 3
arr2 = np.arange(5, 20, 3)
print("Array from 5 to 20 with step 3:", arr2)

# Creating an array from 10 to 1 with a step of -1
arr3 = np.arange(10, 0, -1)
print("Array from 10 to 1 with step -1:", arr3)
```

2. numpy.linspace()

The numpy.linspace() function returns evenly spaced numbers over a specified interval.

Syntax:

numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)

- start: The starting value of the sequence.
- stop: The end value of the sequence.
- num: Number of samples to generate (default is 50).
- endpoint: If True, stop is the last sample; otherwise, it is not included (default is True).
- retstep: If True, return the step size (default is False).
- dtype: The data type of the output array.

```
# Creating an array with 5 evenly spaced values from 0 to 1
arr4 = np.linspace(0, 1, 5)
print("Array with 5 evenly spaced values from 0 to 1:", arr4)

# Creating an array with 10 evenly spaced values from 1 to 10
arr5 = np.linspace(1, 10, 10)
print("Array with 10 evenly spaced values from 1 to 10:", arr5)

# Creating an array from 1 to 2 with 5 values, returning the step size
arr6, step = np.linspace(1, 2, 5, retstep=True)
print("Array from 1 to 2 with 5 values:", arr6)
print("Step size:", step)
```

3. numpy.logspace()

The numpy.logspace() function returns numbers spaced evenly on a log scale.

Syntax:
numpy.logspace(start, stop, num=50, endpoint=True, base=10.0, dtype=None)

start: The starting value of the sequence (base**start).
stop: The end value of the sequence (base**stop).
num: Number of samples to generate (default is 50).
endpoint: If True, stop is the last sample; otherwise, it is not included (default is True).
base: The base of the logarithm (default is 10.0).
dtype: The data type of the output array.

Parameters

- start: The starting value of the sequence, expressed as the base-10 logarithm. If base is not specified, base defaults to 10.0 (i.e., base**start).
- stop: The end value of the sequence, expressed as the base-10 logarithm. If base is not specified, base defaults to 10.0 (i.e., base**stop).
- num: The number of samples to generate. This determines the length of the output array. Default is 50.
- endpoint: If True, stop is the last value in the generated sequence. If False, stop is not included. Default is True.
- base: The base of the logarithm. Default is 10.0.
- dtype: The data type of the output array. If not specified, the data type is inferred from the inputs.

```
import numpy as np

# Generating an array with 5 values spaced evenly on a log scale from 1 to 100
arr = np.logspace(start=0, stop=2, num=5)
print("Array with values spaced evenly on a log scale from 1 to 100:")
print(arr)
```

Array with values spaced evenly on a log scale from 1 to 100:

[1.   3.16227766 10.   31.6227766 100.   ]

Array Broadcasting

The term broadcasting refers to the ability of NumPy to treat arrays with different dimensions during arithmetic operations.

This process involves certain rules that allow the smaller array to be ‘broadcast’ across the larger one, ensuring that they have compatible shapes for these operations.

Broadcasting is not limited to two arrays; it can be applied over multiple arrays as well.

Compatibility Rules for Broadcasting

Broadcasting only works with compatible arrays. NumPy compares a set of array dimensions from right to left.

Every set of dimensions must be compatible with the arrays to be broadcastable. A set of dimension lengths is compatible when

- All the input array have same shape
- The two arrays are compatible in a dimension if they have the same size in the dimension or if one of the arrays has size 1 in that dimension.
- The arrays can be broadcast together if they are compatible with all dimensions.
- In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension.

Array Broadcasting Examples

Array Broadcasting of Single-Dimensional Array

```
import numpy as np

a = np.array([17, 11, 19])

print(a)

b = 3

print(b)

c = a + b

print(c)
```

Array Broadcasting of Two-Dimensional Array

```
import numpy as np

A = np.array([[11, 22, 33], [10, 20, 30]])

print(A)

b = 4

print(b)

C = A + b

print(C)
```

Iterating over array

In python For and While loop is used to iterate through arrays.

NumPy package contains an iterator object numpy.nditer.

It is an efficient multidimensional iterator object using which it is possible to iterate over an array.

Each element of an array is visited using Python’s standard Iterator interface.

Using a for loop:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

for element in arr:
    print(element)
```

Using nditer:

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for element in np.nditer(arr):
    print(element)
```

Sorting and Searching

Sorting

Sorting refers to arranging data in a particular format.

Sorting algorithm specifies the way to arrange data in a particular order.

Most common orders are in numerical or lexicographical order.

In Numpy, we can perform various sorting operations using the various functions that are provided in the library like sort, lexsort, argsort etc.

np.sort

Syntax:

np.sort(a, axis=-1, kind=None, order=None)

a: Array to be sorted.

axis: Axis along which to sort. Default is -1 (last axis).

kind: Sorting algorithm. Options are 'quicksort', 'mergesort', 'heapsort', and 'stable'.

order: When a is an array with fields defined, this argument specifies which fields to compare first.

```
import numpy as np

arr = np.array([3, 1, 2, 5, 4])

sorted_arr = np.sort(arr)

print("Sorted array:", sorted_arr)

# Two-dimensional array

arr_2d = np.array([[3, 2, 1], [6, 5, 4]])

sorted_arr_2d_axis0 = np.sort(arr_2d, axis=0)  # Sort along rows

sorted_arr_2d_axis1 = np.sort(arr_2d, axis=1)  # Sort along columns

print("Sorted 2D array along axis 0:\n", sorted_arr_2d_axis0)

print("Sorted 2D array along axis 1:\n", sorted_arr_2d_axis1)
```

np.argsort

The numpy.argsort() function returns the indices that would sort an array.

This means that instead of returning the sorted array itself, it returns an array of indices that indicates the order in which elements should be arranged to achieve a sorted array.

Syntax:

np.argsort(a, axis=-1, kind=None, order=None)

```
import numpy as np


arr = np.array([3, 1, 2, 5, 4])
sorted_indices = np.argsort(arr)
print("Indices that would sort the array:", sorted_indices)
sorted_arr = arr[sorted_indices]
print("Array sorted using indices:", sorted_arr)
```

Indices that would sort the array: [1 2 0 4 3]

Array sorted using indices: [1 2 3 4 5]

np.lexsort

The numpy.lexsort() function performs an indirect stable sort using a sequence of keys. This means it returns an array of indices that sorts the data according to the order specified by the keys. The sort is stable, meaning that the order of elements that compare equal is preserved.

Syntax:

np.lexsort(keys, axis=-1)

keys: Sequence of arrays to sort by. The last key is primary.

axis: Axis to sort along.

```
import numpy as np

names = np.array(['John', 'Anne', 'Zoe', 'Ben'])
scores = np.array([85, 90, 85, 95])
sorted_indices = np.lexsort((scores, names))
sorted_names = names[sorted_indices]
sorted_scores = scores[sorted_indices]
print("Sorted names:", sorted_names)
print("Sorted scores:", sorted_scores)
```

Output

Original names: ['John' 'Anne' 'Zoe' 'Ben']

Original scores: [85 90 85 95]

Sorted indices: [1 3 0 2]

Sorted names: ['Anne' 'Ben' 'John' 'Zoe']

Sorted scores: [90 95 85 85]

Explanation

Original Arrays:

- Names: ′John′,′Anne′,′Zoe′,′Ben″John', 'Anne', 'Zoe', 'Ben″John′,′Anne′,′Zoe′,′Ben′
- Scores: 85,90,85,9585, 90, 85, 9585,90,85,95

Keys for Sorting:

- The primary key for sorting is the last key in the tuple: names.
- The secondary key is the first key in the tuple: scores.

Sorting Process:

- np.lexsort((scores, names)) returns indices 1,3,0,21, 3, 0, 21,3,0,2 that would sort the names array primarily and scores secondarily.

Searching

Searching involves finding specific elements or indices in an array. NumPy provides several functions for searching.

One fundamental approach involves element wise comparison, where boolean mask are generated based on condition or criteria applied to the elements of an array.

np.where

The np.where function returns the indices of elements in an input array where a specified condition is true. It can also return elements chosen from x or y based on the condition.

element wise comparison in numpy

Element-wise comparison in NumPy allows you to compare each element of an array with the corresponding element of another array or with a scalar value. NumPy provides several functions and operators to perform these comparisons.
Comparison Operators

== : Element-wise equality comparison

!= : Element-wise inequality comparison

> : Element-wise greater-than comparison

< : Element-wise less-than comparison

>= : Element-wise greater-than-or-equal-to comparison

<= : Element-wise less-than-or-equal-to comparison

Example

```
import numpy as np
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([4, 3, 2, 1])
# Element-wise equality
print("arr1 == arr2:", arr1 == arr2)
# Element-wise inequality
print("arr1 != arr2:", arr1 != arr2)
# Element-wise greater-than
print("arr1 > arr2:", arr1 > arr2)
# Element-wise less-than
print("arr1 < arr2:", arr1 < arr2)
# Element-wise greater-than-or-equal-to
print("arr1 >= arr2:", arr1 >= arr2)
# Element-wise less-than-or-equal-to
print("arr1 <= arr2:", arr1 <= arr2)
```

Using numpy.where()

The numpy.where() function is a versatile tool that allows you to perform element-wise conditional operations on arrays.

You can use it to get the indices of elements that satisfy a certain condition or to select elements from one of two arrays based on a condition.

Syntax

numpy.where(condition, [x, y])

condition: An array-like object that represents the condition to be checked.

x: (Optional) An array-like object. Elements from x are selected where condition is True.

y: (Optional) An array-like object. Elements from y are selected where condition is False.

If only the condition is provided, np.where() returns the indices of the elements that are True.

Examples

Finding Indices Where Condition is True

You can use np.where() to find the indices of elements in an array that satisfy a certain condition.

```
import numpy as np


arr = np.array([3, 1, 2, 5, 4])

indices = np.where(arr > 2)

print("Indices of elements greater than 2:", indices)

print("Elements greater than 2:", arr[indices])
```

Output

Indices of elements greater than 2: (array([0, 3, 4]),)

Elements greater than 2: [3 5 4]

Conditional Selection from Two Arrays

You can also use np.where() to select elements from two arrays based on a condition.

```
import numpy as np


arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([1, 2, 3, 4, 5])
condition = arr1 > 25
result = np.where(condition, arr1, arr2)
print("Condition:", condition)
print("Resulting array:", result)
```

Output

Condition: [False False True True True]

Resulting array: [1 2 30 40 50]

Statistical Functions

NumPy provides a wide range of statistical functions to perform various statistical operations on arrays.

These functions are essential for data analysis and include measures of central tendency, dispersion, and other statistical properties.

Here’s an overview of some commonly used statistical functions in NumPy:

Descriptive statistics

Descriptive statistics include measures such as mean, median, variance, and standard deviation that summarize the main features of a dataset.

- Mean (np.mean): Computes the arithmetic mean of array elements.
- Median (np.median): Computes the median of array elements.
- Variance (np.var): Computes the variance of array elements.
- Standard Deviation (np.std): Computes the standard deviation of array elements.

Percentiles

Percentiles indicate the relative standing of a value within a dataset.

- Percentile (np.percentile): Computes the nth percentile of the data along the specified axis.

Min and Max

- Minimum (np.min): Computes the smallest value in an array.
- Maximum (np.max): Computes the largest value in an array.

Histograms

Histograms are used to estimate the probability distribution of a continuous variable.

- Histogram (np.histogram): Computes the histogram of a dataset, which includes bin counts and bin edges.

Correlation and Covariance

- Correlation (np.corrcoef): Computes the Pearson correlation coefficient matrix of the input arrays.
- Covariance (np.cov): Computes the covariance matrix of the input arrays.

Random Sampling

NumPy provides functions to generate random samples and perform operations related to probability distributions.

- np.random.rand(): Generates random numbers from a uniform distribution.
- np.random.randn(): Generates random numbers from a standard normal distribution.
- np.random.choice(): Generates random samples from a given array.

Statistical Test

- Numpy provide functions for conduction statistical tests, such as numpy.ttest_ind() for performing a t-test between two samples

Most common numpy statistical tool

Numpy.amin() and numpy.amax() function

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

min_value = np.amin(arr) # Minimum value in the entire array

min_axis_0 = np.amin(arr, axis=0) # Minimum value along axis 0 (column-wise minimum)

min_axis_1 = np.amin(arr, axis=1) # Minimum value along axis 1 (row-wise minimum)

print("Minimum value in the entire array:", min_value)

print("Minimum value along axis 0:", min_axis_0)

print("Minimum value along axis 1:", min_axis_1)
```

Find Median Using NumPy

```
import numpy as np


# create a 1D array with 5 elements
array1 = np.array([1, 2, 3, 4, 5])
# calculate the median
median = np.median(array1)
print(median)
# Output: 3.0
```

Compute Mean Using NumPy

We use the np.mean() function to calculate the mean value. For example,

```
import numpy as np

# create a numpy array

marks = np.array([76, 78, 81, 66, 85])

# compute the mean of marks

mean_marks = np.mean(marks)

print(mean_marks)

# Output: 77.2
```

Standard Deviation of NumPy Array

```
import numpy as np


# create a numpy array
marks = np.array([76, 78, 81, 66, 85])
# compute the standard deviation of marks
std_marks = np.std(marks)
print(std_marks)
# Output: 6.803568381206575
```

Computing the variance of array elements

```
import numpy as np


arr = np.array([1, 2, 3, 4, 5])

variance = np.var(arr)


print("Variance:", variance)

#Output Variance: 2.0
```

Compute Percentile of NumPy Array

In NumPy, we use the percentile() function to compute the nth percentile of a given array.

```
import numpy as np
array1 = np.array([1, 3, 5, 7, 9, 11, 13, 15, 17, 19])


# compute the 25th percentile of the array
result1 = np.percentile(array1, 25)
print("25th percentile:",result1)


# compute the 75th percentile of the array
result2 = np.percentile(array1, 75)
print("75th percentile:",result2)
```

Pandas

Introduction

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

Pandas is a powerful and versatile library that simplifies the tasks of data manipulation in Python.

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

The Pandas library is an essential tool for data analysts, scientists, and engineers working with structured data in Python.

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.

What is Python Pandas used for?

The Pandas library is generally used for data science, but have you wondered why? This is because the Pandas library is used in conjunction with other libraries that are used for data science.

It is built on top of the NumPy library which means that a lot of the structures of NumPy are used or replicated in Pandas.

The data produced by Pandas is often used as input for plotting functions in Matplotlib, statistical analysis in SciPy, and machine learning algorithms in Scikit-learn.

Python's Pandas library is the best tool to analyze, clean, and manipulate data.

Here is a list of things that we can do using Pandas.

- Data set cleaning, merging, and joining.
- Easy handling of missing data (represented as NaN) in floating point as well as non-floating point data.
- Columns can be inserted and deleted from DataFrame and higher-dimensional objects.
- Powerful group by functionality for performing split-apply-combine operations on data sets.
- Data Visualization.

Getting Started with Pandas

Installing Pandas

- pip install pandas

Importing Pandas

After the Pandas have been installed in the system, you need to import the library. This module is generally imported as follows:

- import pandas as pd

Pandas Series

A Pandas Series is like a column in a table.

A pandas Series is a one-dimensional array-like object that can hold any data type.

It is similar to a column in a spreadsheet or a database table.

Creating a Series

You can create a Series from a list, NumPy array, or dictionary.

From a list:

```
import pandas as pd
data = [1, 2, 3, 4, 5]
series = pd.Series(data)
print(series)
```

Output:

0 1

1 2

2 3

3 4

4 5

dtype: int64

From a NumPy array:

```
import numpy as np

array = np.array([10, 20, 30, 40, 50])

series = pd.Series(array)

print(series)
```

Output:

0 10

1 20

2 30

3 40

4 50

dtype: int64

From a dictionary:

```
data = {'a': 100, 'b': 200, 'c': 300}

series = pd.Series(data)

print(series)
```

Output:

```
a    100
b    200
c    300
dtype: int64
```

Basic Operations

You can perform arithmetic operations on Series.

```
series1 = pd.Series([1, 2, 3, 4, 5])
series2 = pd.Series([10, 20, 30, 40, 50])
result = series1 + series2
print(result)
```

Output:

```
0    11
1    22
2    33
3    44
4    55
dtype: int64
```

Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

Example

print(myvar[0])

Create Labels

With the index argument, you can name your own labels.

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a, index = ["x", "y", "z"])

print(myvar)
```

When you have created labels, you can access an item by referring to the label.

Example

```
print(myvar["y"]) # Return the value of "y"
```

DataFrames

Data sets in Pandas are usually multi-dimensional tables, called DataFrames.

Series is like a column, a DataFrame is the whole table.

A pandas DataFrame is a two-dimensional, size-mutable, and heterogeneous tabular data structure with labeled axes (rows and columns). It is similar to a spreadsheet or SQL table and is one of the most important data structures in pandas.

Creating a DataFrame

Creating a DataFrame using List

Creating a Pandas DataFrame from lists involves organizing data into a two-dimensional structure where each list represents a row of data. You can specify column names explicitly or allow Pandas to assign default column names. Finally, by passing these lists to the pd.DataFrame constructor, you effortlessly instantiate a DataFrame ready for data analysis and manipulation.

Example

Example:

```
import pandas as pd
data_set = [
        ['Alankrit', 45],
        ['Anamol', 40],
        ['haravi', 49]
]
df = pd.DataFrame (data_set, columns=[ 'Name', 'Marks'])
print (df)
```

Output

	Name	Marks
0	Alankrit	45
1	Anamol	40
2	Aaravi	49

From a List of Dictionaries

You can create a DataFrame from a list of dictionaries, where each dictionary represents a row of data.

```
data = [
    {'Name': 'Alice', 'Age': 25, 'City': 'New York'},
    {'Name': 'Bob', 'Age': 30, 'City': 'Los Angeles'},
    {'Name': 'Charlie', 'Age': 35, 'City': 'Chicago'}
]
df = pd.DataFrame(data)
print(df)
```

Output

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

From a NumPy Array

You can create a DataFrame from a NumPy array, specifying column names separately.

```
import numpy as np

array = np.array([
    ['Alice', 25, 'New York'],
    ['Bob', 30, 'Los Angeles'],
    ['Charlie', 35, 'Chicago']
])
df = pd.DataFrame(array, columns=['Name', 'Age', 'City'])
print(df)
```

Output

```
      Name Age          City
0    Alice  25      New York
1      Bob  30  Los Angeles
2  Charlie  35       Chicago
```

Accessing Data in a DataFrame

Accessing Columns

You can access a single column or multiple columns.

```
print(df['Name'])  # Single column

print(df[['Name', 'Age']])  # Multiple columns
```

Output for single column:

```
0      Alice
1        Bob
2    Charlie
Name: Name, dtype: object
```

Output for multiple columns:

```
      Name  Age
0    Alice   25
1      Bob   30
2  Charlie   35
```

Accessing Rows

You can use .loc and .iloc for accessing rows by label or integer location respectively.

```
print(df.loc[0])  # Accessing the first row by label

print(df.iloc[0]) # Accessing the first row by integer location
```

Output for the first row:

```
Name      Alice
Age        25
City    New York
Name: 0, dtype: object
```

Creating a DataFrame using Excel File

You can use the pd.read_excel() function to read an Excel file and create a DataFrame.

Example Excel File (example.xlsx)

Assume you have an Excel file named example.xlsx with the following data in a sheet named "Sheet1":

Name	Age	City
Alice	25	New York
Bob	30	Los Angeles
Charlie	35	Chicago

Reading the Excel File

```
import pandas as pd

# Read the entire sheet
df = pd.read_excel('example.xlsx', sheet_name='Sheet1')
print(df)
```

Output

```
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

Reading Specific Columns

If you only want to read specific columns, you can use the usecols parameter:

```
df = pd.read_excel('example.xlsx', sheet_name='Sheet1', usecols=['Name', 'City'])
print(df)
```

Creating a DataFrame using CSV File

Reading a CSV File

You can use the pd.read_csv() function to read a CSV file and create a DataFrame.

Example CSV File (example.csv)

Assume you have a CSV file named example.csv with the following data:

Name,Age,City

Alice,25,New York

Bob,30,Los Angeles

Charlie,35,Chicago

Reading the CSV File

```
import pandas as pd

# Read the entire CSV file
df = pd.read_csv('example.csv')
print(df)
```

Output

```
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago
```

Reading Specific Columns

If you only want to read specific columns, you can use the usecols parameter:

```
df = pd.read_csv('example.csv', usecols=['Name', 'City'])
print(df)
```

Output

	Name	City
0	Alice	New York
1	Bob	Los Angeles
2	Charlie	Chicago

Creating a DataFrame using SQL Database

Creating a DataFrame using data from an SQL database in pandas is also a common and straightforward process.

Prerequisites

Ensure you have the necessary packages installed:

- pandas
- SQLAlchemy for creating the connection
- The appropriate database driver for your SQL database (e.g., psycopg2 for PostgreSQL, pymysql for MySQL, etc.)

You can install these packages using pip:

pip install pandas sqlalchemy psycopg2 pymysql

```
import pandas as pd
import mysql.connector
#Connect to MySQL database
conn =mysql.connector.connect(
host= "your _host",
user= "your _username", password="your _password", database="your _database"
#SQLqueryot fetchdata
query ="SELECT *FROM your _table"
# Read data from MySQL into DataFrame
df = pd.read_sal(query, conn) #Close the connection
conn. close() print(df)
```

The head and tail Functions

The head and tail functions are essential data analysis and programming tools, notably in the context of Python’s popular pandas package.

In the world of data analysis using Python's pandas library, the head and tail functions are indispensable tools for getting a quick grasp of your dataset.

What is head() Function?

The head() function is primarily used to view the first few rows of a dataset. It helps users quickly get an overview of the data and its structure. Analysts can check column names, data types, and the data itself by displaying the initial records. The head() function is available in many programming languages, including Python and R.

Syntax

The head function is defined as follows:

dataframe.head(**N**)

N refers to the number of rows. If no parameter is passed, the first five rows are returned.

Note: **The head function also supports negative values of N. In that case, all rows except the last N rows are returned.**

Example

```
import pandas as pd

# Creating a dataframe
df = pd.DataFrame({'Sports': ['Football', 'Cricket', 'Baseball', 'Basketball',
                    'Tennis', 'Table-tennis', 'Archery', 'Swimming', 'Boxing']})
print(df.head()) # By default it prints first 5 rows
print('\n')
print(df.head(3)) # Printing first 3 rows
print('\n')
print(df.head(-2)) # Printing all except the last 2 rows
```

	Sports
0	Football
1	Cricket
2	Baseball
3	Basketball
4	Tennis

	Sports
0	Football
1	Cricket
2	Baseball

	Sports
0	Football
1	Cricket
2	Baseball
3	Basketball
4	Tennis
5	Table-tennis
6	Archery

```
import pandas as pd

# Sample DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David', 'Eva', 'Frank'],
    'Age': [25, 30, 35, 40, 45, 50],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix', 'Philadelphia']
}

df = pd.DataFrame(data)
print(df.head()) # Display the first 5 rows (default)
print(df.head(3)) # Display the first 3 rows
```

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago
3	David	40	Houston
4	Eva	45	Phoenix

	Name	Age	City
0	Alice	25	New York
1	Bob	30	Los Angeles
2	Charlie	35	Chicago

What is tail() Function?

The tail() function offers a rapid view of the final few rows of a dataset, just like the head() does. It is especially helpful when working with huge datasets because it enables users to check that the data is full and spot any trends or outliers at the dataset’s end.

Syntax

The tail function is defined as follows:

dataframe.tail(**N**)

N refers to the number of rows. If no parameter is passed, the first last rows are returned.

Note : **The tail function also supports negative values of N. In that case, all rows except the first N rows are returned.**

Example

```
import pandas as pd
# Creating a dataframe
df = pd.DataFrame({'Sports': ['Football', 'Cricket', 'Baseball', 'Basketball',
                    'Tennis', 'Table-tennis', 'Archery', 'Swimming', 'Boxing']})
print(df.tail()) # By default
print('\n')
print(df.tail(3)) # Printing last 3 rows
print('\n')
print(df.tail(-2)) # Printing all except the first 2 rows
```

	Sports
4	Tennis
5	Table-tennis
6	Archery
7	Swimming
8	Boxing

	Sports
6	Archery
7	Swimming
8	Boxing

	Sports
2	Baseball
3	Basketball
4	Tennis
5	Table-tennis
6	Archery
7	Swimming
8	Boxing

Attributes in DataFrames

Pandas attributes offer crucial insights into the properties and organization of DataFrame and Series objects without modifying their contents. These characteristics provide useful metadata that helps users comprehend the types, sizes, shapes, and labels of the data.

Let us see such attributes and methods in Python Pandas for DataFrame:

- dtypes: Return the dtypes in the DataFrame
- ndim: Return the number of dimensions of the DataFrame
- columns: Return the column labels of the DataFrame
- values: Returns the data stored in the dataframe as a NumPy array
- size: Return the number of elements in the DataFrame.
- shape: Return the dimensionality of the DataFrame in the form of a tuple.
- index: Return the index of the DataFrame
- T: Transpose the rows and columns

shape

The pandas.DataFrame.shape is used to return the dimensionality of the DataFrame in the form of a tuple.

```
import pandas as pd

# Dataset
data = {
    'Student': ["Amit", "John", "Jacob", "David", "Steve"],
    'Rank': [1, 4, 3, 5, 2],
    'Marks': [95, 70, 80, 60, 90]
}

# Create a DataFrame using the DataFrame() method with index
res = pd.DataFrame(data, index=['RowA', 'RowB', 'RowC', 'RowD', 'RowE'], )
print("Student Records\n\n", res)

# Return the dimensionality of the DataFrame, Result in a Tuple form
print("\nDimensionality:\n", res.shape)
```

Output

Student Records

	Student	Rank	Marks
RowA	Amit	1	95
RowB	John	4	70
RowC	Jacob	3	80
RowD	David	5	60
RowE	Steve	2	90

Dimensionality:

(5, 3)

index

The pandas.DataFrame.index is used to return the index of the DataFrame.

```
import pandas as pd
 # Dataset
data = {
    'Student': ["Amit", "John", "Jacob", "David", "Steve"],
    'Rank': [1, 4, 3, 5, 2],
    'Marks': [95, 70, 80, 60, 90]
}

# Create a DataFrame using the DataFrame() method with index
res = pd.DataFrame(data, index=['RowA', 'RowB', 'RowC', 'RowD', 'RowE'], )

# Display the Records
print("Student Records\n\n", res)

# Return the index of the DataFrame
print("\nDataFrame Index:\n", res.index)
```

Output

Student Records

	Student	Rank	Marks
RowA	Amit	1	95
RowB	John	4	70
RowC	Jacob	3	80
RowD	David	5	60
RowE	Steve	2	90

DataFrame Index:

Index(['RowA', 'RowB', 'RowC', 'RowD', 'RowE'], dtype='object')

column attributes

```
import pandas as pd
# Create a sample DataFrame
data = {
    'Name': ['Apple', 'Banana', 'Cherry'],
    'Color': ['Red', 'Yellow', 'Red'],
    'Qty': [10, 20, 15]
}
df = pd.DataFrame(data)

print("Original column names:", df.columns) # Get the original column names

df.columns = ['Fruit Name', 'Fruit Color', 'Quantity'] # Set new column names
print("New column names:", df.columns)

fruit_names = df['Fruit Name'] # Accessing a single column
print("Fruit Names:")
print(fruit_names)

fruit_info = df[['Fruit Name', 'Quantity']] # Accessing multiple columns
print("\nFruit Names and Quantities:")
print(fruit_info)
```

Output

```
Original column names: Index(['Name', 'Color', 'Qty'], dtype='object')

New column names: Index(['Fruit Name', 'Fruit Color', 'Quantity'], dtype='object')

Fruit Names:

0     Apple

1    Banana

2    Cherry

Name: Fruit Name, dtype: object

Fruit Names and Quantities:

  Fruit Name  Quantity

0      Apple        10

1     Banana        20

2     Cherry        15
```

dtypes

The pandas.DataFrame.dtypes is used to return the dtypes in the DataFrame.

```
import pandas as pd

# Dataset
data = {
    'Student': ["Amit", "John", "Jacob", "David", "Steve"],
    'Rank': [1, 4, 3, 5, 2],
    'Marks': [95, 70, 80, 60, 90]
}

# Create a DataFrame using the DataFrame() method with index
res = pd.DataFrame(data, index=['RowA', 'RowB', 'RowC', 'RowD', 'RowE'], )

# Display the Records
print("Student Records\n\n", res)

# Datatypes in the DataFrame
print("\nDatatypes:\n", res.dtypes)
```

Output

```
import pandas as pd

# Dataset
data = {
    'Student': ["Amit", "John", "Jacob", "David", "Steve"],
    'Rank': [1, 4, 3, 5, 2],
    'Marks': [95, 70, 80, 60, 90]
}

# Create a DataFrame using the DataFrame() method with index
res = pd.DataFrame(data, index=['RowA', 'RowB', 'RowC', 'RowD', 'RowE'], )

# Display the Records
print("Student Records\n\n", res)

# Datatypes in the DataFrame
print("\nDatatypes:\n", res.dtypes)
```

Values

```
import pandas as pd
#Create a sample DataFrame
data = {'Name': ['Aaravi', 'Eleeja', 'Alankrit', 'Anamol'],
        'Age': (21, 24, 23, 31]}
df = pd. DataFrame(data)
# Accessing data as a NumPy array
print("DataFrame as a Numpy array:")
print(df.values)
# Accessing a specific element using indexing
print("\n Accessing element at row 1, column 0:")
name = df. values(1, 0] # Got namo at row 1 (index 1), column 0
print (name)
```

DataFrame as a NumPy array:

[['Aaravi' 21)

('Elooja' 21]

('Alankrit' 23]

('Anamol" 31))

Accessing element at row 1, column 0:

Eleeja

ndim

The pandas.DataFrame.ndim is used to return the number of dimensions of the DataFrame.

```
import pandas as pd
# Dataset
data = {
    'Student': ["Amit", "John", "Jacob", "David", "Steve"],
    'Rank': [1, 4, 3, 5, 2],
    'Marks': [95, 70, 80, 60, 90]
}

# Create a DataFrame using the DataFrame() method with index
res = pd.DataFrame(data, index=['RowA', 'RowB', 'RowC', 'RowD', 'RowE'], )

# Display the Records
print("Student Records\n\n", res)

# Number of Dimensions in the DataFrame
print("\nNumber of Dimensions:\n", res.ndim)
```

Output

Student Records

	Student	Rank	Marks
RowA	Amit	1	95
RowB	John	4	70
RowC	Jacob	3	80
RowD	David	5	60
RowE	Steve	2	90

Number of Dimensions:
2

size

The pandas.DataFrame.size is used to return the number of elements in the DataFrame.

```
import pandas as pd

# Dataset
data = {
    'Student': ["Amit", "John", "Jacob", "David", "Steve"],
    'Rank': [1, 4, 3, 5, 2],
    'Marks': [95, 70, 80, 60, 90]
}

# Create a DataFrame using the DataFrame() method with index
res = pd.DataFrame(data, index=['RowA', 'RowB', 'RowC', 'RowD', 'RowE'], )

# Display the Records
print("Student Records\n\n", res)

# Number of elements in the DataFrame
print("\nNumber of Elements:\n", res.size)
```

Output

Student Records

	Student	Rank	Marks
RowA	Amit	1	95
RowB	John	4	70
RowC	Jacob	3	80
RowD	David	5	60
RowE	Steve	2	90

Number of Elements:
15

T

The pandas.DataFrame.T is used to Transpose the rows and columns.

```
import pandas as pd

# Dataset
data = {
    'Student': ["Amit", "John", "Jacob", "David", "Steve"],
    'Rank': [1, 4, 3, 5, 2],
    'Marks': [95, 70, 80, 60, 90]
}

# Create a DataFrame using the DataFrame() method with index
res = pd.DataFrame(data, index=['RowA', 'RowB', 'RowC', 'RowD', 'RowE'], )

# Display the Records
print("Student Records\n\n", res)

# Return the Transpose
print("\nTranspose:\n", res.T)
```

Student Records

	Student	Rank	Marks
RowA	Amit	1	95
RowB	John	4	70
RowC	Jacob	3	80
RowD	David	5	60
RowE	Steve	2	90

Transpose:

	RowA	RowB	RowC	RowD	RowE
Student	Amit	John	Jacob	David	Steve
Rank	1	4	3	5	2
Marks	95	70	80	60	90

Working with Missing Data

Data can have null or missing values and is frequently gathered from a variety of sources. Any dataset may contain missing values, which can be brought on by a number of factors including careless data entry, missing information, or deliberate omissions. Since they might result in inaccurate analysis or outcomes, they are a common concern in data analysis. As a result, handling these missing variables is crucial before moving on to more analysis.

Pandas offers a number of techniques to handle these missing data in an efficient manner. Using functions like **isnull()** or **notnull()**, which enable Boolean masking to find null values within a DataFrame or Series, is one popular method for identifying missing data. Once missing numbers have been located, they can be handled with **fillna()** or another technique to replace them with a desired value or approach, like mean, median, or mode. Alternatively, **dropna()** can be used to eliminate missing data completely, either row-wise or column-wise, depending on what the demands of the application are.

Example identifying missing values

```
import pandas as pd
import numpy as np
# Create a sample DataFrame with missing values
data = {
    'Fruit Name': ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry'],
    'Fruit Color': ['Red', np.nan, 'Red', 'Brown', np.nan],
    'Quantity': [10, 20, np.nan, 15, 10],
    'Price per Unit': [0.5, 0.2, 0.8, np.nan, 1.0]
}

df = pd.DataFrame(data)

# Display the DataFrame with missing values
print("Original DataFrame with Missing Values:")
print(df)

# Identifying missing data
print("\nIdentifying Missing Data:")
print(df.isnull())
```

Output

Original DataFrame with Missing Values:

	Fruit Name	Fruit Color	Quantity	Price per Unit
0	Apple	Red	10.0	0.5
1	Banana	NaN	20.0	0.2
2	Cherry	Red	NaN	0.8
3	Date	Brown	15.0	NaN
4	Elderberry	NaN	10.0	1.0

Identifying Missing Data:

	Fruit Name	Fruit Color	Quantity	Price per Unit
0	False	False	False	False
1	False	True	False	False
2	False	False	True	False
3	False	False	False	True
4	False	True	False	False

```
# Removing missing data
df_dropped = df.dropna()
print("\nDataFrame after Dropping Rows with Missing Values:")
print(df_dropped)

# Filling missing data with a specific value
df_filled = df.fillna(0)
print("\nDataFrame after Filling Missing Values with 0:")
print(df_filled)
```

DataFrame after Dropping Rows with Missing Values:

	Fruit Name	Fruit Color	Quantity	Price per Unit
0	Apple	Red	10.0	0.5

DataFrame after Filling Missing Values with 0:

	Fruit Name	Fruit Color	Quantity	Price per Unit
0	Apple	Red	10.0	0.5
1	Banana	0	20.0	0.2
2	Cherry	Red	0.0	0.8
3	Date	Brown	15.0	0.0
4	Elderberry	0	10.0	1.0

Indexing in DataFrame

In Pandas, indexing is the act of choosing and gaining access to particular data inside a DataFrame or Series. Pandas indexes provide for efficient data retrieval and manipulation by acting as a unique identifier for every row or element in the data structure. Pandas has a number of indexing strategy types that each provide unique functionality to meet distinct needs for data processing.

Pandas provides several ways to index and select data,

- including label-based indexing with .loc,
- position-based indexing with .iloc, and
- boolean indexing.

```
import pandas as pd
# Create a sample DataFrame
data = {
    'Fruit Name': ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry'],
    'Fruit Color': ['Red', 'Yellow', 'Red', 'Brown', 'Purple'],
    'Quantity': [10, 20, 15, 5, 7],
    'Price per Unit': [0.5, 0.2, 0.8, 1.0, 1.5]
}

df = pd.DataFrame(data)

# Set the index to 'Fruit Name'
df.set_index('Fruit Name', inplace=True)

# Display the DataFrame
print("DataFrame with 'Fruit Name' as Index:")
print(df)

# Label-based indexing using .loc
print("\nLabel-based indexing with .loc:")
print(df.loc['Apple'])  # Access a single row
print(df.loc[['Apple', 'Cherry']])  # Access multiple rows
print(df.loc['Apple', 'Price per Unit'])  # Access a single value
print(df.loc['Apple':'Cherry', ['Quantity', 'Price per Unit']])  # Slicing rows and columns
```

DataFrame with 'Fruit Name' as Index:

Fruit Name	Fruit Color	Quantity	Price per Unit
Apple	Red	10	0.5
Banana	Yellow	20	0.2
Cherry	Red	15	0.8
Date	Brown	5	1.0
Elderberry	Purple	7	1.5

Label-based indexing with .loc:

Fruit Color	Red
Quantity	10
Price per Unit	0.5

Name: Apple, dtype: object

Fruit Name	Fruit Color	Quantity	Price per Unit
Apple	Red	10	0.5
Cherry	Red	15	0.8

Price per Unit 0.5

Fruit Name	Quantity	Price per Unit
Apple	10	0.5
Banana	20	0.2
Cherry	15	0.8

```
# Position-based indexing using .iloc
print("\nPosition-based indexing with .iloc:")
print(df.iloc[0])  # Access a single row
print(df.iloc[[0, 2]])  # Access multiple rows
print(df.iloc[0, 2])  # Access a single value
print(df.iloc[0:3, [2, 3]])  # Slicing rows and columns

# Boolean indexing
print("\nBoolean indexing:")
print(df[df['Quantity'] > 10])  # Select rows based on a condition
print(df[df['Fruit Color'] == 'Red'])  # Select rows based on another condition
```

Position-based indexing with .iloc:
Fruit Color Red
Quantity 10
Price per Unit 0.5
Name: Apple, dtype: object
Fruit Color Quantity Price per Unit
Fruit Name
Apple Red 10 0.5
Cherry Red 15 0.8
15
Quantity Price per Unit
Fruit Name
Apple 10 0.5
Banana 20 0.2
Cherry 15 0.8

Boolean indexing:
Fruit Color Quantity Price per Unit
Fruit Name
Banana Yellow 20 0.2
Cherry Red 15 0.8
Fruit Color Quantity Price per Unit
Fruit Name
Apple Red 10 0.5
Cherry Red 15 0.8

Slicing in DataFrame

In Pandas, slicing is the process of choosing a portion of data from a DataFrame or Series according to predetermined standards or ranges. Users can use it to extract certain segments of the data that fit certain criteria or span particular ranges of rows and columns. Pandas allows for slicing along the rows and columns of the DataFrame, giving users more flexibility when manipulating and analyzing data.

There are several ways to slice a DataFrame, including using the .loc and .iloc indexers for label-based and position-based slicing, respectively.

- **Slicing with .loc (Label-based):**
  - The .loc indexer is used for slicing based on the labels of rows and columns.
- **Slicing with .iloc (Position-based):**
  - The .iloc indexer is used for slicing based on the integer positions of rows and columns.
- **Boolean slicing**
  - allows you to select rows or columns based on conditions.
- **Hierarchical indexing, or multi-level indexing**
  - allows you to have multiple index levels (or tiers) on a DataFrame

Try multi Indexing yourself

Example

```
import pandas as pd
# Create a sample DataFrame
data = {
    'Fruit Name': ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry'],
    'Fruit Color': ['Red', 'Yellow', 'Red', 'Brown', 'Purple'],
    'Quantity': [10, 20, 15, 5, 7],
    'Price per Unit': [0.5, 0.2, 0.8, 1.0, 1.5]
}
df = pd.DataFrame(data)
# Set the 'Fruit Name' column as the index
df.set_index('Fruit Name', inplace=True)
# Display the DataFrame
print("DataFrame with 'Fruit Name' as Index:")
print(df)

# Label-based slicing with .loc
print("\nLabel-based slicing with .loc:")
# Slicing rows
print(df.loc['Apple':'Cherry'])  # Select rows from 'Apple' to 'Cherry'
# Slicing columns
print(df.loc[:, 'Quantity':'Price per Unit'])  # Select columns from 'Quantity' to 'Price per Unit'
# Slicing both rows and columns
print(df.loc['Apple':'Cherry', 'Quantity':'Price per Unit'])  # Select rows and columns
```

Output

DataFrame with 'Fruit Name' as Index:

Fruit Name	Fruit Color	Quantity	Price per Unit
Apple	Red	10	0.5
Banana	Yellow	20	0.2
Cherry	Red	15	0.8
Date	Brown	5	1.0
Elderberry	Purple	7	1.5

Label-based slicing with .loc:

Fruit Name	Fruit Color	Quantity	Price per Unit
Apple	Red	10	0.5
Banana	Yellow	20	0.2
Cherry	Red	15	0.8

Fruit Name	Quantity	Price per Unit
Apple	10	0.5
Banana	20	0.2
Cherry	15	0.8

Fruit Name	Quantity	Price per Unit
Apple	10	0.5
Banana	20	0.2
Cherry	15	0.8

```
# Position-based slicing with .iloc
print("\nPosition-based slicing with .iloc:")
# Slicing rows
print(df.iloc[0:3])  # Select the first three rows
# Slicing columns
print(df.iloc[:, 1:3])  # Select the second and third columns
# Slicing both rows and columns
print(df.iloc[0:3, 1:3])  # Select the first three rows and the second and third columns
```

Position-based slicing with .iloc:

Fruit Name	Color	Quantity	Price per Unit
Apple	Red	10	0.5
Banana	Yellow	20	0.2
Cherry	Red	15	0.8

Fruit Name	Quantity	Price per Unit
Apple	10	0.5
Banana	20	0.2
Cherry	15	0.8

Fruit Name	Quantity	Price per Unit
Apple	10	0.5
Banana	20	0.2
Cherry	15	0.8

```
# Boolean slicing to select rows where 'Quantity' > 10
print("\nRows where 'Quantity' > 10:")
print(df[df['Quantity'] > 10])

# Boolean slicing to select rows where 'Fruit Color' is 'Red'
print("\nRows where 'Fruit Color' is 'Red':")
print(df[df['Fruit Color'] == 'Red'])
```

Rows where 'Quantity' > 10:

	Fruit Name	Fruit Color	Quantity	Price per Unit
1	Banana	Yellow	20	0.2
2	Cherry	Red	15	0.8

Rows where 'Fruit Color' is 'Red':

	Fruit Name	Fruit Color	Quantity	Price per Unit
0	Apple	Red	10	0.5
2	Cherry	Red	15	0.8

Subsetting a DataFrame

The process of choosing a desired set of rows and columns from a data frame is known as subsetting.

Using a variety of criteria, subsetting in Pandas entails choosing particular rows and columns from a DataFrame or Series to create a more condensed and targeted subset. When only a piece of the dataset is needed for additional analysis or visualization, this technique is essential for data analysis activities.

Pandas has a number of subsetting techniques that make it easy for users to extract the required subset.

- Boolean indexing, in which data is chosen according to Boolean conditions, is a popular method for subsetting data in Pandas.
- Label-based subsetting, which selects data based on the labels of rows and columns, is another popular subsetting method in Pandas.
- Positional subsetting, in which data is chosen according to integer places along the rows and columns of the DataFrame, is another feature supported by Pandas.

```
import pandas as pd
# Create a sample DataFrame
data = { 'A': [11, 12, 13, 14, 15],
         'B': [16, 17, 18, 19, 20],
         ‘C’: [21, 22, 23, 24, 25]}
index = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
df = pd. DataFrame(data, index=index)

# Display the original DataFrame
print("Original DataFrame:")
print (df)

# Boolean indexing
print("\n Boolean indexing:")
print(df[df['A'] › 13]) # Select rows where values in column 'A' are greater than 13

# Label-based subsetting using loc[]
print("\n Label-based subsetting using loc(]:")
print(df.loc[['b', 'd'], ['B', 'C']]) # Select specific rows and columns by labels

# Positional subsetting using iloc[]
print("\n Positional subsetting using iloc[]:")
print(df.iloc[1:4, 1:3]) # Select rows 2 to 4 and columns 2 to 3
```

Merging and Joining dataframe

Merging and joining DataFrames in pandas is essential for combining data from multiple sources.

This can be done using various methods, including merge(), join(), and concat().

merge()

The merge() function is similar to SQL joins and allows you to combine DataFrames based on common columns or indices. You can perform inner, outer, left, and right joins.

Syntax:

pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False)

Output

```
import pandas as pd

# Create sample DataFrames
df1 = pd.DataFrame({
    'EmployeeID': [1, 2, 3, 4],
    'Name': ['Alice', 'Bob', 'Charlie', 'David']
})

df2 = pd.DataFrame({
    'EmployeeID': [3, 4, 5, 6],
    'Department': ['HR', 'IT', 'Finance', 'Marketing']
})

# Merge DataFrames on 'EmployeeID'
merged_df = pd.merge(df1, df2, on='EmployeeID', how='inner')

print("Merged DataFrame (Inner Join):")
print(merged_df)
```

Merged DataFrame (Inner Join):

	EmployeeID	Name	Department
0	3	Charlie	HR
1	4	David	IT

```
# Other types of joins
left_join = pd.merge(df1, df2, on='EmployeeID', how='left')
right_join = pd.merge(df1, df2, on='EmployeeID', how='right')
outer_join = pd.merge(df1, df2, on='EmployeeID', how='outer')

print("\nLeft Join:")
print(left_join)
print("\nRight Join:")
print(right_join)
print("\nOuter Join:")
print(outer_join)
```

Left Join:

	EmployeeID	Name	Department
0	1	Alice	NaN
1	2	Bob	NaN
2	3	Charlie	HR
3	4	David	IT

Right Join:

	EmployeeID	Name	Department
0	3	Charlie	HR
1	4	David	IT
2	5	NaN	Finance
3	6	NaN	Marketing

Outer Join:

	EmployeeID	Name	Department
0	1	Alice	NaN
1	2	Bob	NaN
2	3	Charlie	HR
3	4	David	IT
4	5	NaN	Finance
5	6	NaN	Marketing

join()

The join() method is used to join DataFrames based on the index or a key column. It’s typically used when you want to join on the index rather than columns.

Basic Syntax:

df1.join(df2, how='left', on=None, lsuffix='', rsuffix='')

```
# Create sample DataFrames with index
df3 = pd.DataFrame({
    'EmployeeID': [1, 2, 3, 4],
    'Name': ['Alice', 'Bob', 'Charlie', 'David']
}).set_index('EmployeeID')

df4 = pd.DataFrame({
    'Department': ['HR', 'IT', 'Finance', 'Marketing']
}, index=[3, 4, 5, 6])

# Join DataFrames on index
joined_df = df3.join(df4, how='inner')

print("Joined DataFrame:")
print(joined_df)
```

Working with CSV

- Comma Separated Values files, or CSV files for short, are just plain text documents that use a certain structure to arrange tabular data.
- It is a simple text file, hence the only data it can include are legible ASCII or Unicode letters.
- CSV files typically employ a comma to separate every data value.
- DataFrames can be easily filled with data for analysis or exported back to CSV format with Pandas' handy utilities for reading and writing CSV files. Here's a step-by-step tutorial with code to use pandas to import, read, and manipulate CSV files:
    - **Import the pandas library**
    - Read the CSV file into a pandas DataFrame
    - Explore the DataFrame
    - Perform operations on the DataFrame
    - Export the DataFrame to a CSV file (optional)

Reading a CSV File

Example:

Assume you have a CSV file named data.csv with the following content:

Name,Age,City

Alice,30,New York

Bob,25,Los Angeles

Charlie,35,Chicago

```
import pandas as pd

# Read CSV file into DataFrame
df = pd.read_csv('data.csv')

# Display the DataFrame
print("DataFrame read from CSV:")
print(df)
```

Output

DataFrame read from CSV:

	Name	Age	City
0	Alice	30	New York
1	Bob	25	Los Angeles
2	Charlie	35	Chicago

Writing a DataFrame to a CSV File

```
# Create a DataFrame
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [30, 25, 35],
    'City': ['New York', 'Los Angeles', 'Chicago']
}
df = pd.DataFrame(data)


# Write DataFrame to a CSV file
df.to_csv('output.csv', index=False)
print("DataFrame written to 'output.csv'.")
```

Matlpotlib

- Introduction
- Marker
- Line
- Color
- Label
- Grid Line
- Subplot
- Scatter Plot
- Bar Graph
- Histogram
- pie chart and Box plot

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

It is widely used for generating plots, histograms, power spectra, bar charts, error charts, scatterplots, etc.

Matplotlib is designed to work seamlessly with NumPy arrays and is a fundamental tool for data visualization in Python.

To install Matplotlib we simply use the following command.

- pip install matplotlib

Another option is to utilize a Python installation like as Anaconda, Spyder, or others that come with Matplotlib already installed. After the successful installation we need to import the Matplotlib to perform various functionalities. Importing Matplotlib follows simple code as shown below.

import matplotlib

After importing the module, let's check the version of Matpotlib with the following code. Version is stored under _version_ attribute.

import matplotlib print (matplotlib._ version)

#Output 2.0.0

The 'pyplot' submodule contains the majority of Matplotlib's utilities, which are typically imported via the plt alias as shown below.

import matplotlib pyplot as plt

After importing, pltcan be used to refer pyplot.

Plotting

Multiple Points

Default X Points

Marker

MarkerSize

Marker Color

Line Style

Label

Grid Lines

Sub Plot

Scatter Plot

Color Map

Bar Graph

Histogram

Pie Chart

Box Plot