Module - 1

1. **With python code explain the following basic array manipulations on 1dimensional and 2dimensional data**
   (i)Indexing of arrays,
   (ii)Slicing of arrays,
   (iii) Joining and splitting of arrays

Numpy Arrays

Numpy (Numerical Python) is a fundamental library for numerical computing in Python, providing powerful N-dimensional array objects and a collection of routines for processing these arrays. It is extensively used in data science for efficient array operations.

(i) Indexing of Arrays

- **Definition:** Array indexing is the method of accessing individual elements or specific parts of a NumPy array using square brackets [] and integer indices.
- Similar to Python list indexing, you can access elements in a 1D array by their position, starting from 0.
- Negative indices allow accessing elements from the end of the array, with -1 representing the last element.
- For multidimensional arrays, you access items using a comma-separated tuple of indices to specify the row and column (and further dimensions).
- Array elements can be modified by assigning new values using indexing, but be aware of silent truncation if the new value's type doesn't match the array's fixed data type.

Example Code

```
In [1]: import numpy as np # Import the NumPy library

In [2]: x1 = np.array([5, 0, 3, 3, 7, 9]) # Create a 1D NumPy array
           x1
Out[2]: array([5, 0, 3, 3, 7, 9])

In [3]: x1[0] # Access the first element (index 0)
Out[3]: 5

In [4]: x1[-1] # Access the last element using negative indexing
Out[4]: 9

In [5]: x2 = np.array([[3, 5, 2, 4], # Create a 2D NumPy array
                       [7, 6, 8, 8],
                       [1, 6, 7, 7]])
         x2
Out[5]: array([[3, 5, 2, 4],
               [7, 6, 8, 8],
               [1, 6, 7, 7]])

In [6]: x2[0, 0] # Access the element at row 0, column 0
Out[6]: 3

In [7]: x2[2, -1] # Access the element at row 2, last column
Out[7]: 7

In [8]: x2[0, 0] = 12 # Modify the element at row 0, column 0
```

```
           x2
Out[8]: array([[12, 5, 2, 4],
               [ 7, 6, 8, 8],
               [ 1, 6, 7, 7]])


In [9]: x1[0] = 3.14159 # Assign a float to an integer array (truncation occurs)
        x1
Out[9]: array([3, 0, 3, 3, 7, 9])
```

Explanation of Example Code

- **`x1[0]`:** Demonstrates accessing the first element (at index 0) of a 1D array.
- **`x1[-1]`:** Shows how to access the last element of a 1D array using negative indexing.
- **`x2[0, 0]`:** Illustrates accessing an individual element in a 2D array by specifying both row and column indices (first row, first column).
- **`x2[2, -1]`:** Shows accessing an element in a 2D array using a combination of positive row index and negative column index (third row, last column).
- **`x2[0, 0] = 12`:** Demonstrates how to modify the value of an element at a specific index in a 2D array.
- **`x1[0] = 3.14159`:** Highlights that NumPy arrays have a fixed data type, leading to silent truncation when a float is assigned to an integer array.

(ii) Slicing of Arrays

- **Definition:** Array slicing is a method to extract subarrays (subsets of elements) from a NumPy array using the slice notation, marked by the colon (:) character. The basic syntax is `x[start:stop:step]`.
- `start` is the inclusive beginning index, `stop` is the exclusive ending index, and `step` is the increment between elements.
- Unspecified `start`, `stop`, or `step` default to the dimension's beginning (0), end (size), and 1 respectively for positive steps.
- A negative `step` value reverses the order of elements; in this case, the `start` and `stop` defaults are automatically swapped.
- For multidimensional arrays, slices are applied independently to each dimension, separated by commas.

Example Code

```
In [1]: import numpy as np # Import the NumPy library

In [2]: x = np.arange(10) # Create a 1D array with values from 0 to 9
           x
Out[2]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [3]: x[:5] # Access the first five elements
Out[3]: array([0, 1, 2, 3, 4])

In [4]: x[5:] # Access elements from index 5 to the end
Out[4]: array([5, 6, 7, 8, 9])

In [5]: x[4:7] # Access a middle subarray from index 4 up to (but not including) 7
Out[5]: array([4, 5, 6])

In [6]: x[::2] # Access every other element
Out[6]: array([0, 2, 4, 6, 8])
```

```
In [7]: x[::-1] # Reverse the entire array
Out[7]: array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])

In [8]: x2 = np.array([[12, 5, 2, 4], # Create a 2D NumPy array
                       [ 7, 6, 8, 8],
                       [ 1, 6, 7, 7]])
        x2
Out[8]: array([[12, 5, 2, 4],
               [ 7, 6, 8, 8],
               [ 1, 6, 7, 7]])

In [9]: x2[:2, :3] # Access the first two rows and first three columns
Out[9]: array([[12, 5, 2],
               [ 7, 6, 8]])

In [10]: x2[::-1, ::-1] # Reverse both rows and columns
Out[10]: array([[ 7,  7,  6,  1],
                [ 8,  8,  6,  7],
                [ 4,  2,  5, 12]])
```

Explanation of Example Code

- **`x[:5]`:** Shows how to extract elements from the beginning up to a specific index (exclusive) in a 1D array.
- **`x[5:]`:** Demonstrates extracting elements from a specific index (inclusive) to the end of a 1D array.
- **`x[4:7]`:** Illustrates slicing a specific middle portion of a 1D array using `start:stop`.
- **`x[::2]`:** Shows how to use the `step` parameter to select every Nth element (here, every other element).
- **`x[::-1]`:** Explains how a negative `step` reverses the entire 1D array.
- **`x2[:2, :3]`:** Shows how to slice both rows and columns in a 2D array simultaneously to obtain a specific subarray (first two rows, first three columns).
- **`x2[::-1, ::-1]`:** Highlights reversing both dimensions of a 2D array by using negative steps for both row and column slices.

(iii) Joining and Splitting of Arrays

- **Definition:** Array joining (concatenation) combines two or more arrays into a single, larger array. Conversely, array splitting divides a single array into multiple smaller arrays.
- **`np.concatenate()`:** This versatile function combines arrays along a specified axis; by default, it concatenates along the first axis (rows for 2D arrays).
- **`np.vstack()` and `np.hstack()`:** These are convenience functions for vertical (row-wise) and horizontal (column-wise) stacking, particularly useful for combining arrays of compatible but potentially mixed dimensions.
- **`np.split()`:** This function divides an array into multiple subarrays based on a list of indices where the splits should occur, resulting in N+1 subarrays for N split points.
- **`np.vsplit()` and `np.hsplit()`:** These are specialized functions for splitting arrays vertically (by row) and horizontally (by column), providing more direct ways to divide 2D arrays.

Example Code

```
In [1]: import numpy as np # Import the NumPy library

In [2]: x = np.array([1, 2, 3]) # Create the first 1D array
        y = np.array([3, 2, 1]) # Create the second 1D array
        np.concatenate([x, y]) # Join two 1D arrays
Out[2]: array([1, 2, 3, 3, 2, 1])
```

```
In [3]: z = [99, 99, 99] # Create a Python list
        print(np.concatenate([x, y, z])) # Join multiple arrays/lists
Out[3]: [ 1  2  3  3  2  1 99 99 99]

In [4]: grid = np.array([[1, 2, 3], # Create a 2D array
                         [4, 5, 6]])
        print(f"Original 2D grid:\n{grid}")
Original 2D grid:
[[1 2 3]
 [4 5 6]]

In [5]: # Concatenate along the first axis (rows by default)
           np.concatenate([grid, grid])
Out[5]: array([[1, 2, 3],
               [4, 5, 6],
               [1, 2, 3],
               [4, 5, 6]])

In [6]: # Concatenate along the second axis (columns)
        np.concatenate([grid, grid], axis=1)
Out[6]: array([[1, 2, 3, 1, 2, 3],
               [4, 5, 6, 4, 5, 6]])

In [7]: x_1d = np.array([1, 2, 3]) # Create a 1D array for vertical stacking
        grid_2d = np.array([[9, 8, 7], # Create a 2D array for stacking
                            [6, 5, 4]])
        # Vertically stack a 1D array onto a 2D array
        np.vstack([x_1d, grid_2d])
Out[7]: array([[1, 2, 3],
               [9, 8, 7],
               [6, 5, 4]])

In [8]: y_col = np.array([[99], # Create a 2D column array
                          [99]])
        # Horizontally stack a 2D array with a column array
        np.hstack([grid_2d, y_col])
Out[8]: array([[ 9,  8,  7, 99],
               [ 6,  5,  4, 99]])
```

```
#Split()

In [9]: x_split = np.array([1, 2, 3, 99, 99, 3, 2, 1]) # Create a 1D array for splitting
        x1, x2, x3 = np.split(x_split, [3, 5]) # Split at indices 3 and 5
        print(f"Original 1D array x_split: {x_split}")
        print(f"x1: {x1}, x2: {x2}, x3: {x3}")
Original 1D array x_split: [ 1  2  3 99 99  3  2  1]
x1: [1 2 3], x2: [99 99], x3: [3 2 1]

In [10]: grid_split = np.arange(16).reshape((4, 4)) # Create a 2D array for splitting
         print(f"Original 2D grid_split:\n{grid_split}")
Original 2D grid_split:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
```

```
In [11]: upper, lower = np.vsplit(grid_split, [2]) # Split vertically after the 2nd row
         print(f"Upper part:\n{upper}")
         print(f"Lower part:\n{lower}")
Upper part:
[[0 1 2 3]
 [4 5 6 7]]
Lower part:
[[ 8  9 10 11]
 [12 13 14 15]]

In [12]: left, right = np.hsplit(grid_split, [2]) # Split horizontally after the 2nd
column
         print(f"Left part:\n{left}")
         print(f"Right part:\n{right}")
Left part:
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
Right part:
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

---

2. Explain the role of IPython and Jupyter in data analysis.

IPython and Jupyter play a pivotal role in modern data analysis workflows by providing interactive, reproducible, and collaborative environments. They bridge the gap between code execution, narrative documentation, and visualization, making data exploration, modeling, and communication significantly more efficient.

The Role of IPython in Data Analysis

- **Enhanced Interactive Shell:** IPython significantly improves the standard Python interpreter, offering features like tab completion and object introspection for faster data exploration.
- **Magic Commands (Line):** These start with `%` and operate on a single line, used for tasks like timing code (`%timeit`) or running external scripts (`%run`).
- **Magic Commands (Cell):** These start with `%%` and apply to the entire cell, useful for executing shell commands (`%%bash`) or writing to files (`%%writefile`).
- **Seamless Library Integration:** IPython is designed to work smoothly with scientific libraries, enabling rich display of DataFrames and inline rendering of plots.
- **Persistent Session History:** It maintains a detailed history of commands and outputs across sessions, allowing users to recall and reuse previous work effortlessly.
- **In-depth Object Introspection:** Users can easily inspect objects by typing `?` or `??` after a variable or function, revealing docstrings and source code.
- **Integrated Debugging:** IPython provides tools for interactive debugging, allowing users to step through code and inspect variables when errors occur.
- **System Shell Access:** It allows direct execution of system shell commands (e.g., `!ls`, `!pwd`) for managing files and interacting with the OS.

The Role of Jupyter in Data Analysis

- **Interactive Computational Environment:** Jupyter Notebooks are web-based documents combining live code, rich text, and visualizations, ideal for interactive data exploration and analysis.
- **Reproducibility:** Notebooks capture the entire analysis process, from data loading to results, making it easy to reproduce steps and findings.
- **Communication & Collaboration:** Highly shareable notebooks facilitate effective communication of insights, allowing code, results, and explanations to coexist for seamless collaboration.
- **Rich Output Support:** Jupyter supports diverse output types, including formatted HTML tables, interactive plots, images, and mathematical equations for enhanced presentation.
- **Extensible Platform:** Its architecture is extensible, with JupyterLab offering an advanced interface, widgets enabling interactive controls, and multi-kernel support for over 40 languages.
- **Literate Programming:** Jupyter promotes "literate programming" by allowing users to interleave executable code with narrative text, fostering clearer understanding and documentation.
- **Version Control Integration:** Notebooks can be easily tracked and version-controlled using tools like Git, though managing diffs for `.ipynb` files requires special tools.
- **Dashboard Creation:** Tools like Voilà can transform notebooks into interactive web applications or dashboards, allowing non-technical users to interact with the analysis.
- **Remote Server Capabilities:** Notebooks can be run on remote servers, enabling access to powerful computing resources and collaborative environments.
- **Data Visualization Powerhouse:** Its integration with libraries like Matplotlib, Seaborn, and Plotly makes it a central hub for creating compelling data visualizations

---

3. Explain how NumPy handles structured data with examples..

NumPy arrays are typically homogeneous, meaning all elements are of the same data type. However, NumPy also provides **structured arrays**, which allow for the efficient storage and manipulation of heterogeneous data, similar to records or rows in a database table. Each element in a structured array is a "record" composed of named fields, each with its own data type.

How NumPy Handles Structured Data

- **Definition:** Structured arrays are NumPy arrays with compound data types, allowing each element to hold different types of data under named fields.
- **Creating Structured Arrays:** You define a compound `dtype` using a dictionary with 'names' (field names) and 'formats' (data types for each field), then pass this to array creation functions.
- **Accessing Fields:** Individual fields within the structured array can be accessed by their name using dictionary-like indexing (e.g., `data['name']`).
- **Record Access:** Entire records (rows) can be accessed using standard integer indexing (e.g., `data[0]`), returning a single structured array element.
- **Heterogeneous Data Storage:** They provide an efficient way to store related but differently typed data together, like a table with columns for name (string), age (integer), and weight (float).
- **Fixed Type Behavior:** Like regular NumPy arrays, structured arrays have fixed types; attempting to insert a value of an incompatible type will result in truncation or type casting.

Example Code

```
In [1]: import numpy as np # Import the NumPy library

In [2]: name = ['Alice', 'Bob', 'Cathy', 'Doug'] # Sample list of names
        age = [25, 45, 37, 19] # Sample list of ages
        weight = [55.0, 85.5, 68.0, 61.5] # Sample list of weights
```

```
In [3]: # Create a structured array using a compound data type specification
        # 'U10' for Unicode string of max 10 chars, 'i4' for 4-byte integer, 'f8' for 8-byte
float
        data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                                  'formats':('U10', 'i4', 'f8')})
        print(data.dtype) # Print the compound data type
Out[3]: [('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]

In [4]: # Populate the structured array with the sample data
        data['name'] = name
        data['age'] = age
        data['weight'] = weight
        print(data) # Print the populated structured array
Out[4]: [('Alice', 25, 55. ) ('Bob', 45, 85.5) ('Cathy', 37, 68. )
        ('Doug', 19, 61.5)]

In [5]: print(data['name']) # Access the 'name' field (returns a view of the names)
Out[5]: ['Alice' 'Bob' 'Cathy' 'Doug']

In [6]: print(data[0]) # Access the first record (row)
Out[6]: ('Alice', 25, 55.)

In [7]: print(data['age'][2]) # Access the age of the third person (index 2)
Out[7]: 37
```

Explanation of Example Code

- **`data = np.zeros(4, dtype=...)`:** Demonstrates creating an empty structured array for 4 records by defining a `dtype` with field `names` and their respective `formats`.
- **`print(data.dtype)`:** Shows the resulting compound data type, including the endianness symbol (e.g., < for little-endian).
- **`data['name'] = name`:** Illustrates populating entire fields (columns) of the structured array from lists.
- **`print(data)`:** Displays the complete structured array, showing each record with its heterogeneous data.
- **`print(data['name'])`:** Shows how to access data from a specific field across all records, behaving like column selection.
- **`print(data[0])`:** Demonstrates accessing an entire record (row) using standard integer indexing.
- **`print(data['age'][2])`:** Shows combined indexing to access a specific value from a specific field of a specific record.

---

4. With Python program explain the various numpy functions used for sorting of arrays

Sorting arrays is a fundamental operation in data analysis, arranging elements in a specified order (ascending or descending). NumPy provides highly optimized functions for sorting, crucial for preparing data for analysis, search operations, or presenting results.

(i) Fast Sorting in NumPy: `np.sort` and `np.argsort`

- **Definition:** NumPy offers `np.sort()` for returning a sorted copy of an array and the `.sort()` method for performing an in-place sort.
- **`np.sort()` Function:** This function returns a new, sorted array without modifying the original input array.
- **`.sort()` Method:** This method sorts the array *in-place*, meaning it directly modifies the original array and returns `None`.

- **`np.argsort()` Function:** Instead of returning the sorted values, `np.argsort()` returns the *indices* that would sort the array.
- **Multidimensional Sorting:** Both `np.sort()` and `np.argsort()` support an `axis` argument to sort along specific rows (`axis=1`) or columns (`axis=0`) of multidimensional arrays.
- **Default Algorithm:** By default, `np.sort` uses a quicksort algorithm, offering efficient performance for most applications.

Example Code

```
In [1]: import numpy as np
In [2]: x = np.array([2, 1, 4, 3, 5]) # Create a 1D NumPy array
        np.sort(x) # Return a sorted copy without modifying x
Out[2]: array([1, 2, 3, 4, 5])

In [3]: x.sort() # Sort the array in-place
        print(x) # Print the modified array
Out[3]: [1 2 3 4 5]

In [4]: x = np.array([2, 1, 4, 3, 5]) # Recreate array for argsort example
        i = np.argsort(x) # Get the indices that would sort x
        print(i) # Print the sorted indices
Out[4]: [1 0 3 2 4]

In [5]: print(x[i]) # Use fancy indexing with argsort result to construct the sorted
array
Out[5]: [1 2 3 4 5]

In [6]: rand = np.random.RandomState(42) # Set random state for reproducibility
        X = rand.randint(0, 10, (4, 6)) # Create a 2D array of random integers
        print(f"Original 2D array:\n{X}") # Print the original 2D array
Original 2D array:
[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]

In [7]: # Sort each column of X independently (axis=0)
        np.sort(X, axis=0)
Out[7]: array([[2, 1, 4, 0, 1, 5],
               [5, 2, 5, 4, 3, 7],
               [6, 3, 7, 4, 6, 7],
               [7, 6, 7, 4, 9, 9]])

In [8]: # Sort each row of X independently (axis=1)
        np.sort(X, axis=1)
Out[8]: array([[3, 4, 6, 6, 7, 9],
               [2, 3, 4, 6, 7, 7],
               [1, 2, 4, 5, 7, 7],
               [0, 1, 4, 5, 5, 9]])
```

Explanation of Example Code

- **`np.sort(x)`:** Shows how `np.sort` returns a new sorted array, leaving the original `x` unchanged.
- **`x.sort()`:** Demonstrates the in-place sorting method, where the `x` array itself is modified.

- **`np.argsort(x):`** Illustrates how `argsort` returns an array of indices that, if used to index `x`, would yield a sorted array.
- **`x[i]:`** Confirms that using the `argsort` result with fancy indexing effectively sorts the original array.
- **`np.sort(X, axis=0):`** Shows how to sort elements along each column (vertically) in a 2D array.
- **`np.sort(X, axis=1):`** Demonstrates sorting elements along each row (horizontally) in a 2D array.

(ii) Partial Sorts: Partitioning with `np.partition`

- **Definition:** `np.partition()` is a NumPy function that reorders an array such that the element at a specified Kth position is in its sorted place.
- **Left Partition:** All elements smaller than the Kth element are moved to its left, in arbitrary order.
- **Right Partition:** All elements larger than or equal to the Kth element are moved to its right, also in arbitrary order.
- **Efficiency:** It's more efficient than a full sort when only the K smallest (or largest) elements are needed, without regard for the order of other elements.
- **Multidimensional Partitioning:** Similar to `np.sort`, `np.partition` also supports the `axis` argument for partitioning along rows or columns in multidimensional arrays.
- **`np.argpartition():`** A related function that returns the indices that would partition the array according to the specified K.

Example Code

```
In [1]: import numpy as np
In [2]: x = np.array([7, 2, 3, 1, 6, 5, 4]) # Create a 1D array for partitioning
        np.partition(x, 3) # Partition the array such that the 3rd smallest element is in place
Out[2]: array([2, 1, 3, 4, 6, 5, 7])

In [3]: rand = np.random.RandomState(42) # Set random state for reproducibility
        X = rand.randint(0, 10, (4, 6)) # Recreate the 2D array
        print(f"Original 2D array:\n{X}") # Print the original 2D array
Original 2D array:
[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]

In [4]: # Partition each row of X, ensuring the 2nd smallest element is in its sorted position
within each row
        np.partition(X, 2, axis=1)
Out[4]: array([[3, 4, 6, 7, 6, 9],
               [2, 3, 4, 7, 6, 7],
               [1, 2, 4, 5, 7, 7],
               [0, 1, 4, 5, 9, 5]])
```

Explanation of Example Code

- **`np.partition(x, 3):`** Shows how partitioning works on a 1D array, placing the 3rd smallest element (index 3) in its correct sorted position, with smaller elements to its left.
- **`np.partition(X, 2, axis=1):`** Demonstrates partitioning each row of a 2D array independently, ensuring that the element that would be at index 2 if sorted (the 3rd smallest) is in that position within each row.

Module 2

---

1. **Explain the concept of Pandas objects and their types**

Pandas objects are enhanced versions of NumPy arrays, specifically designed to handle the complexities of real-world, labeled, and heterogeneous data. They provide robust structures for data manipulation, analysis, and cleaning, forming the indispensable bedrock of modern data science workflows in Python. The three fundamental Pandas data structures are `Series`, `DataFrame`, and `Index`.

(i) Pandas Series Object

- **Definition:** A Pandas `Series` is a one-dimensional labeled array capable of holding any data type.
- **Structure:** It combines a sequence of `values` (a NumPy array) and an `index` for explicit data identification.
- **Access:** Data can be accessed using both explicit index labels and implicit integer positions.
- **Flexible Index:** The index can consist of various data types, including strings, integers, or dates, beyond simple numerical order.
- **Data Alignment:** Operations between Series automatically align data based on their labels, handling missing entries gracefully.
- **Broad Creation:** Series can be created from lists, NumPy arrays, Python dictionaries, or scalar values.

Example Code

```
In [1]: import pandas as pd # Standard import for Pandas

In [2]: s_list = pd.Series([0.25, 0.5, 0.75, 1.0]) # Create from a list
           print(f"Series from list:\n{s_list}")
Out[2]: Series from list:
        0    0.25
        1    0.50
        2    0.75
        3    1.00
        dtype: float64

In [3]: s_dict = pd.Series({'London': 8.9, 'Paris': 2.1}) # Create from a dictionary
           print(f"Series from dictionary:\n{s_dict}")
Out[3]: Series from dictionary:
        London    8.9
        Paris     2.1
        dtype: float64

In [4]: print(f"Value at index 1: {s_list[1]}") # Access by implicit integer index
           print(f"Value for 'London': {s_dict['London']}") # Access by explicit label
Out[4]: Value at index 1: 0.5
        Value for 'London': 8.9
```

Explanation of Example Code

- **`pd.Series([0.25, ...])`:** Shows basic Series creation from a list, demonstrating the default integer index.
- **`pd.Series({'London': ..., 'Paris': ...})`:** Illustrates creating a Series from a dictionary, where keys become explicit labels.
- **`s_list[1]` and `s_dict['London']`:** Highlight accessing Series elements using both implicit integer positions and explicit labels.

(ii) Pandas DataFrame Object

- **Definition:** A Pandas `DataFrame` is a two-dimensional labeled data structure with columns of potentially different types, analogous to a spreadsheet.
- **Structure:** It features both a row `index` and distinct column `columns` (which is also an `Index` object), enabling dual-axis labeling.
- **Access:** Data can be accessed by column name (dictionary-style), or by row/column labels (`.loc`) and positions (`.iloc`).
- **Collection of Series:** Internally, a DataFrame functions as a dictionary of aligned `Series` objects, where each Series represents a column.
- **Robust Operations:** DataFrames offer powerful methods for selection, filtering, aggregation, and reshaping, crucial for complex data analysis.
- **Versatile Creation:** DataFrames can be built from dictionaries of Series/arrays, lists of dictionaries, NumPy arrays, or external files.

Example Code

```
In [1]: import pandas as pd # Standard import for Pandas
           import numpy as np # Standard import for NumPy

In [2]: data = {'city': ['London', 'Paris'], # Data for DataFrame
                'population_millions': [8.9, 2.1]}
           df = pd.DataFrame(data) # Create DataFrame from dictionary of lists
           print(f"DataFrame created:\n{df}")
Out[2]: DataFrame created:
             city  population_millions
        0  London                  8.9
        1   Paris                  2.1

In [3]: print(f"DataFrame columns: {df.columns.tolist()}") # Access column labels
           print(f"DataFrame index: {df.index.tolist()}") # Access row labels
Out[3]: DataFrame columns: ['city', 'population_millions']
        DataFrame index: [0, 1]

In [4]: print(f"Access 'city' column:\n{df['city']}") # Access a column by name
Out[4]: Access 'city' column:
        0    London
        1     Paris
        Name: city, dtype: object

In [5]: print(f"Access row by label (using .loc): {df.loc[0].tolist()}") # Access a row
by label (example using default int label)
           print(f"Access row by position (using .iloc): {df.iloc[1].tolist()}") #
Access a row by integer position
Out[5]: Access row by label (using .loc): ['London', 8.9]
        Access row by position (using .iloc): ['Paris', 2.1]
```

Explanation of Example Code

- **`pd.DataFrame(data)`:** Shows creating a DataFrame from a dictionary of lists, which is a common and intuitive method.
- **`df.columns` and `df.index`:** Demonstrate accessing the explicit column and row `Index` objects.
- **`df['city']`:** Illustrates accessing an entire column by its label, which returns a Series.
- **`df.loc[0]` and `df.iloc[1]`:** Highlight the primary methods for explicit (label-based) and implicit (position-based) row access.

(iii) Pandas Index Object

- **Definition:** A Pandas `Index` is an immutable array-like object that holds axis labels (for `Series` or `DataFrame` rows/columns).
- **Purpose:** It provides labels for efficient data access, enables automatic data alignment, and supports high-performance lookups.
- **Immutability:** `Index` objects are immutable, meaning their values cannot be changed in-place after creation, ensuring data integrity.
- **Set-like Operations:** `Index` objects efficiently support standard set operations like union, intersection, and difference, crucial for data joins.
- **Specialized Types:** Pandas includes various `Index` subclasses (e.g., `DatetimeIndex`, `MultiIndex`) for specialized data types and structures.

Example Code

```
In [1]: import pandas as pd # Standard import for Pandas

In [2]: ind = pd.Index([2, 3, 5]) # Create a simple Index object
           print(f"Index object: {ind}")
Out[2]: Index object: Index([2, 3, 5], dtype='int64')

In [3]: print(f"Element at position 1: {ind[1]}") # Access element by integer position
Out[3]: Element at position 1: 3

In [4]: indA = pd.Index([1, 3, 5]) # Create two Index objects for set operations
        indB = pd.Index([2, 3, 5])
        print(f"Intersection: {indA & indB}") # Perform intersection
        print(f"Union: {indA | indB}") # Perform union
Out[4]: Intersection: Int64Index([3, 5], dtype='int64')
        Union: Int64Index([1, 2, 3, 5], dtype='int64')
```

Explanation of Example Code

- **`pd.Index([2, 3, 5])`:** Shows basic creation of an `Index` object from a list.
- **`ind[1]`:** Illustrates accessing an element within the `Index` object by its integer position.
- **`indA & indB` and `indA | indB`:** Demonstrate the set-like operations (intersection and union) that `Index` objects support, crucial for data alignment.

---

2. **Write a Python program to demonstrate hierarchical indexing in Pandas.**

Hierarchical indexing, also known as **MultiIndex**, is a powerful feature in Pandas that allows you to incorporate multiple levels of index labels within a single index. This enables the compact representation of higher-dimensional data within the familiar one-dimensional Series and two-dimensional DataFrame objects, offering a structured way to organize complex data.

(i) A Multiply Indexed Series

- **Definition:** Hierarchical indexing in a Series means each data point is indexed by more than one key, typically structured as tuples.
- **The Bad Way (Tuple Indexing):** Initially, one might use Python tuples as keys in a Series, allowing basic indexing based on multiple values.

- **The Better Way (Pandas MultiIndex):** Pandas provides a `MultiIndex` type that offers superior operations and efficiency compared to rudimentary tuple-based indexing.
- **MultiIndex Creation:** A `MultiIndex` can be explicitly created from a list of tuples using `pd.MultiIndex.from_tuples()`.
- **Hierarchical Representation:** Re-indexing a Series with a `MultiIndex` displays data in a clear hierarchical format, where blank entries indicate identical higher-level index values.
- **Efficient Slicing:** Multi-indexing enables concise and efficient slicing and selection of data across specific levels using Pandas' optimized syntax.

Example Code

```
In [1]: import pandas as pd # Import the Pandas library
           import numpy as np # Import NumPy (standard practice)

In [2]: # Data for states' populations across two years
           index_tuples = [('California', 2000), ('California', 2010),
                           ('New York', 2000), ('New York', 2010),
                           ('Texas', 2000), ('Texas', 2010)]
           populations = [33871648, 37253956,
                          18976457, 19378102,
                          20851820, 25145561]

In [3]: # Create a Series using Python tuples as keys (demonstrating the initial approach)
           pop_tuple_index = pd.Series(populations, index=index_tuples)
           print(f"Series with tuple index:\n{pop_tuple_index}")
Out[3]: Series with tuple index:
        (California, 2000) 33871648
        (California, 2010) 37253956
        (New York, 2000) 18976457
        (New York, 2010) 19378102
        (Texas, 2000) 20851820
        (Texas, 2010) 25145561
        dtype: int64

In [4]: # Slice the Series based on the tuple index
           print(f"\nSliced Series (tuple index):\n{pop_tuple_index[('California',
2010):('Texas', 2000)]}")
Out[4]: Sliced Series (tuple index):
        (California, 2010) 37253956
        (New York, 2000) 18976457
        (New York, 2010) 19378102
        (Texas, 2000) 20851820
        dtype: int64

In [5]: # Select all values from 2010 (demonstrating the inefficiency of tuple indexing)
           print(f"\nValues from 2010 (inefficient tuple method):\n{pop_tuple_index[[i for i in
pop_tuple_index.index if i[1] == 2010]]}")
Out[5]: Values from 2010 (inefficient tuple method):
        (California, 2010) 37253956
        (New York, 2010) 19378102
        (Texas, 2010) 25145561
        dtype: int64

In [6]: # Convert the tuple index to a proper Pandas MultiIndex
           multi_index = pd.MultiIndex.from_tuples(index_tuples)
           print(f"\nGenerated MultiIndex:\n{multi_index}")
Out[6]: Generated MultiIndex:
        MultiIndex(levels=[['California', 'New York', 'Texas'], [2000, 2010]],
                   labels=[[0, 0, 1, 1, 2, 2], [0, 1, 0, 1, 0, 1]])
```

```
In [7]: # Reindex the Series with the MultiIndex for hierarchical representation
           pop = pop_tuple_index.reindex(multi_index)
           print(f"\nMulti-indexed Series (better way):\n{pop}")
Out[7]: Multi-indexed Series (better way):
        California  2000    33871648
                    2010    37253956
        New York    2000    18976457
                    2010    19378102
        Texas       2000    20851820
                    2010    25145561
        dtype: int64

In [8]: # Select all data for which the second index is 2010 using MultiIndex slicing
           print(f"\nPopulations for 2010 (efficient MultiIndex method):\n{pop[:, 2010]}")
Out[8]: Populations for 2010 (efficient MultiIndex method):
        California    37253956
        New York      19378102
        Texas         25145561
        dtype: int64
```

Explanation of Example Code

- **Initial `pop_tuple_index` (In [3, 4, 5]):** Demonstrates the "bad way" of using simple Python tuples as index keys, highlighting its basic functionality and limitations for complex selections.
- **`multi_index = pd.MultiIndex.from_tuples(index_tuples)` (In [6]):** Shows the explicit creation of a Pandas `MultiIndex` object from the original list of tuples.
- **`pop = pop_tuple_index.reindex(multi_index)` (In [7]):** Illustrates how applying the `MultiIndex` transforms the Series into a clear hierarchical representation.
- **`pop[:, 2010]` (In [8]):** Demonstrates the efficient and clean slicing syntax provided by `MultiIndex` for selecting data across index levels.

(ii) MultiIndex as Extra Dimension and in DataFrames

- **Definition:** A `MultiIndex` effectively represents extra dimensions of data, allowing conversion between Series and DataFrames.
- **`unstack()` Method:** This method pivots the innermost index level of a Multi-indexed Series to become new columns in a DataFrame.
- **DataFrame with MultiIndex:** A `DataFrame` can be created or extended with a `MultiIndex` for its rows or columns, enabling higher-dimensional data representation.
- **MultiIndex for Columns:** Hierarchical indexing can be applied to `DataFrame` columns, creating structured headers useful for complex, multi-dimensional datasets.

Example Code

```
In [1]: import pandas as pd # Import Pandas
           import numpy as np # Import NumPy

In [2]: # Recreate the Multi-indexed Series 'pop' (setup for demonstration)
           index_tuples = [('California', 2000), ('California', 2010)] # Reduced tuples
           populations = [33871648, 37253956]
           multi_index = pd.MultiIndex.from_tuples(index_tuples, names=['state', 'year'])
           pop = pd.Series(populations, index=multi_index)

In [3]: # Convert the Multi-indexed Series to a conventionally indexed DataFrame using unstack()
           pop_df_unstacked = pop.unstack()
           print(f"Series unstacked to DataFrame:\n{pop_df_unstacked}")
```

```
Out[3]: Series unstacked to DataFrame:
        year          2000      2010
        state
        California  33871648  37253956

In [4]: # Add another column of demographic data to the DataFrame with MultiIndex
           under18_populations = [9267089, 9284094] # Corresponding to 'pop'
           pop_df_extended = pd.DataFrame({'total': pop, 'under18': under18_populations})
           print(f"\nDataFrame with MultiIndex and multiple columns:\n{pop_df_extended}")
Out[4]: DataFrame with MultiIndex and multiple columns:
                        total  under18
        state      year
        California 2000   33871648  9267089
                   2010   37253956  9284094

In [5]: # Demonstrate MultiIndex for columns (simplified medical data example)
           # Create minimal MultiIndex for rows and columns
           rows_idx = pd.MultiIndex.from_product([[2013], [1, 2]], names=['year', 'visit'])
           cols_idx = pd.MultiIndex.from_product([['Bob'], ['HR', 'Temp']], names=['subject',
'type'])
           # Create simple data
           simple_data = np.array([[31.0, 38.7], [44.0, 37.7]])

           health_data = pd.DataFrame(simple_data, index=rows_idx, columns=cols_idx)
           print(f"\nDataFrame with MultiIndex on rows and columns:\n{health_data}")
Out[5]: DataFrame with MultiIndex on rows and columns:
        subject   Bob
        type       HR Temp
        year visit
        2013 1     31.0 38.7
             2     44.0 37.7
```

Explanation of Example Code

- **`pop.unstack()` (In [3]):** Demonstrates how a Multi-indexed Series can be converted into a DataFrame, with one index level becoming columns, effectively viewing the MultiIndex as an extra dimension.
- **`pd.DataFrame({'total': pop, 'under18': ...})` (In [4]):** Shows how a DataFrame can be constructed or extended with additional columns while maintaining its MultiIndex for rows.
- **`health_data = pd.DataFrame(simple_data, index=rows_idx, columns=cols_idx)` (In [5]):** Illustrates creating a DataFrame with MultiIndex on *both* rows and columns using simplified data and index generation, representing higher-dimensional data.

---

3. Discuss the significance of DataFrame and Series in Pandas.

The significance of `DataFrame` and `Series` in Pandas lies in their powerful capabilities for structuring, manipulating, and analyzing real-world, often messy, datasets. They bridge the gap between raw data and actionable insights, forming the bedrock of most data science workflows in Python.

Significance of Pandas Series

- **Labeled One-Dimensional Data:** A `Series` is a 1D labeled array, making data self-describing and easily comprehensible. Its explicit `Index` allows intuitive access beyond numerical positions.

- **Automatic Data Alignment:** Operations between Series automatically align data based on labels, preventing common errors. This ensures calculations are performed on corresponding data points.
- **Foundation for Tabular Data:** A `Series` is a fundamental building block for `DataFrame` columns, supporting diverse data types. It is also crucial for handling time-series data efficiently.
- **Vectorized Computations:** Leverages NumPy for vectorized operations, ensuring high performance for numerical computations. This allows for fast processing without explicit Python loops.
- **Flexible Indexing:** Supports various index types (integers, strings, dates), providing powerful and flexible ways to organize and retrieve data. This enhances data organization beyond simple arrays.
- **Integrated Mathematical Operations:** Supports a wide range of mathematical operations (e.g., aggregation, element-wise) directly on the entire Series. This simplifies numerical computations across the dataset.
- **Named for Clarity:** Each Series can be assigned a `name` attribute, which improves data clarity when used as a column in a DataFrame. This aids in better data identification and communication.

Significance of Pandas DataFrame

- **Tabular Data Representation:** A `DataFrame` represents 2D tabular data, like a spreadsheet or SQL table, with labeled rows and columns. This structure naturally maps to most real-world datasets.
- **Dual-Axis Labeling & Heterogeneous Columns:** It has explicit labels for rows and columns, offering rich context for data points. Columns can hold different data types, accommodating mixed datasets seamlessly.
- **Comprehensive Data Manipulation:** DataFrames offer powerful methods for filtering, selecting, grouping, joining, and reshaping data. These highly optimized operations simplify complex data tasks.
- **Robust Missing Data Handling:** Built-in capabilities robustly handle missing data (`NaN`), a pervasive issue in real-world datasets. This simplifies data cleaning and ensures reliable analysis outcomes.
- **Hierarchical Indexing Support:** Fully integrates hierarchical (multi-level) indexing for both rows and columns, enabling compact representation of high-dimensional data. This extends its capacity beyond simple 2D tables.
- **Extensive Input/Output (I/O) Capabilities:** DataFrames facilitate easy reading and writing data from/to various file formats (CSV, Excel, SQL). This makes them central to data ingestion and output pipelines.
- **Column-Oriented Operations:** Provides intuitive syntax for working with entire columns, allowing for easy selection, addition, or modification of data. This simplifies common data transformation workflows.
- **Integrated Statistical Analysis:** Offers direct methods for common statistical operations (e.g., mean, median, standard deviation) across rows or columns. This provides quick insights into data distributions.

Combined Significance

- **Intuitive & Powerful Interface:** Together, `Series` and `DataFrame` provide an incredibly intuitive and expressive interface for complex data operations. They abstract away low-level array details.
- **Foundation for Data Analysis:** They are the primary data structures underpinning virtually all advanced data analysis and machine learning tasks performed with Pandas. Their design directly addresses real-world data challenges.

---

4. List the methods used to operate on Null values and Also demonstrate the use of these methods to handle the null values present in the data with examples.

Pandas treats `None` and `NaN` (Not a Number) interchangeably to indicate missing or null values. To effectively manage these, Pandas provides several useful methods for detecting, removing, and replacing null values in its data structures.

The key methods are:

- **`isnull()`**: Generates a Boolean mask indicating where values are missing.
- **`notnull()`**: The opposite of `isnull()`, indicating where values are not missing.
- **`dropna()`**: Returns a filtered version of the data, removing null values.
- **`fillna()`**: Returns a copy of the data with missing values filled or imputed.

Below is a demonstration of these methods using examples directly from the provided text.

Detecting Null Values

Pandas data structures include `isnull()` and `notnull()` to detect null data, both returning a Boolean mask.

```
In [1]: import pandas as pd
           import numpy as np # Used for np.nan

In [2]: data = pd.Series([1, np.nan, 'hello', None])
           print(f"Original Series:\n{data}")
Out[2]: Original Series:
        0        1
        1      NaN
        2    hello
        3     None
        dtype: object

In [3]: print(f"isnull() output:\n{data.isnull()}")
Out[3]: isnull() output:
        0    False
        1     True
        2    False
        3     True
        dtype: bool

In [4]: # Using notnull() as a Boolean mask to select non-null data
           print(f"Data after filtering with notnull():\n{data[data.notnull()]}")
Out[4]: Data after filtering with notnull():
        0        1
        2    hello
        dtype: object
```

Dropping Null Values

The `dropna()` method removes null (NA) values. Its behavior depends on whether it's applied to a Series or DataFrame, and the specified parameters.

For a Series:

```
In [5]: data = pd.Series([1, np.nan, 'hello', None]) # Re-using Series from above
           print(f"Original Series:\n{data}")
Out[5]: Original Series:
        0        1
```

```
        1      NaN
        2    hello
        3     None
        dtype: object

In [6]: print(f"Series after dropna():\n{data.dropna()}")
Out[6]: Series after dropna():
        0        1
        2    hello
        dtype: object
```

For a DataFrame:

For DataFrames, `dropna()` can remove full rows or columns containing nulls, with various options for control.

```
In [7]: df = pd.DataFrame([[1, np.nan, 2],
                           [2, 3, 5],
                           [np.nan, 4, 6]])
           print(f"Original DataFrame:\n{df}")
Out[7]: Original DataFrame:
             0    1  2
        0  1.0  NaN  2
        1  2.0  3.0  5
        2  NaN  4.0  6

In [8]: # Default dropna(): Drops any row containing a null value (axis='rows',
how='any')
           print(f"DataFrame after default dropna():\n{df.dropna()}")
Out[8]: DataFrame after default dropna():
             0    1  2
        1  2.0  3.0  5

In [9]: # Dropna() along columns: Drops any column containing a null value
(axis='columns')
           print(f"DataFrame after
dropna(axis='columns'):\n{df.dropna(axis='columns')}")
Out[9]: DataFrame after dropna(axis='columns'):
           2
        0  2
        1  5
        2  6

In [10]: # Adding a new column entirely of NaN for 'how' demonstration
            df[3] = np.nan
            print(f"DataFrame with new NaN column:\n{df}")
Out[10]: DataFrame with new NaN column:
             0    1  2   3
        0  1.0  NaN  2 NaN
        1  2.0  3.0  5 NaN
        2  NaN  4.0  6 NaN

In [11]: # Drop columns only if ALL values are null (how='all')
            print(f"DataFrame after dropna(axis='columns',
how='all'):\n{df.dropna(axis='columns', how='all')}")
Out[11]: DataFrame after dropna(axis='columns', how='all'):
```

```
     0    1  2
0  1.0  NaN  2
1  2.0  3.0  5
2  NaN  4.0  6

In [12]: # Drop rows based on a minimum number of non-null values (thresh=3)
         print(f"DataFrame after dropna(axis='rows',
thresh=3):\n{df.dropna(axis='rows', thresh=3)}")
Out[12]: DataFrame after dropna(axis='rows', thresh=3):
         0    1  2   3
1  2.0  3.0  5 NaN
```

Filling Null Values

The `fillna()` method replaces null values with a specified valid value.

For a Series:

Python

```
In [13]: data = pd.Series([1, np.nan, 2, None, 3], index=list('abcde'))
         print(f"Original Series for fillna():\n{data}")
Out[13]: Original Series for fillna():
         a    1.0
         b    NaN
         c    2.0
         d    NaN
         e    3.0
         dtype: float64

In [14]: # Fill NA entries with a single value (e.g., 0)
         print(f"Series after fillna(0):\n{data.fillna(0)}")
Out[14]: Series after fillna(0):
         a    1.0
         b    0.0
         c    2.0
         d    0.0
         e    3.0
         dtype: float64

In [15]: # Forward-fill: Propagate the previous valid value forward
         print(f"Series after
fillna(method='ffill'):\n{data.fillna(method='ffill')}")
Out[15]: Series after fillna(method='ffill'):
         a    1.0
         b    1.0
         c    2.0
         d    2.0
         e    3.0
         dtype: float64

In [16]: # Back-fill: Propagate the next valid value backward
         print(f"Series after
fillna(method='bfill'):\n{data.fillna(method='bfill')}")
Out[16]: Series after fillna(method='bfill'):
         a    1.0
```

```
b    2.0
c    2.0
d    3.0
e    3.0
dtype: float64
```

For a DataFrame:

`fillna()` for DataFrames offers similar options, plus the ability to specify an axis for the fill operation.

```
In [17]: df_for_fill = pd.DataFrame([[1, np.nan, 2, np.nan], # Re-using df state before
extra column addition
                                     [2, 3, 5, np.nan],
                                     [np.nan, 4, 6, np.nan]])
            print(f"Original DataFrame for fillna():\n{df_for_fill}")
Out[17]: Original DataFrame for fillna():
             0    1  2   3
        0  1.0  NaN  2 NaN
        1  2.0  3.0  5 NaN
        2  NaN  4.0  6 NaN

In [18]: # Forward-fill across columns (axis=1)
             print(f"DataFrame after fillna(method='ffill',
axis=1):\n{df_for_fill.fillna(method='ffill', axis=1)}")
Out[18]: DataFrame after fillna(method='ffill', axis=1):
             0    1    2    3
        0  1.0  1.0  2.0  2.0
        1  2.0  3.0  5.0  5.0
        2  NaN  4.0  6.0  6.0
```

---

5. **Develop Python code for constructing the Pandas DataFrame objects from the following**
   (i) Single Series object,
   (ii) List of dicts,
   (iii) Dictionary of Series objects,
   (iv) Two-dimensional NumPy array

The Pandas `DataFrame` is a fundamental data structure, often described as a generalization of a NumPy array or a specialization of a Python dictionary. It is designed to handle two-dimensional, tabular data with both flexible row indices and flexible column names.

A `DataFrame` can be thought of as a sequence of aligned `Series` objects, where "aligned" means they share the same index. This allows for rich, labeled data handling.

Here's how to construct `DataFrame` objects from various common data sources, as per the provided text and examples:

```
In [1]: import pandas as pd
           import numpy as np
```

```
           # population Series (implicitly created from previous section's context)
           population_dict = {'California': 38332521, 'Texas': 26448193, 'New York':
19651127,
                              'Florida': 19552860, 'Illinois': 12882135}
           population = pd.Series(population_dict)

           # area Series
           area_dict = {'California': 423967, 'Texas': 695662, 'New York': 141297,
                        'Florida': 170312, 'Illinois': 149995}
           area = pd.Series(area_dict)

           print("--- Constructing DataFrame Objects ---")
Out[1]: --- Constructing DataFrame Objects ---
```

(i) From a Single Series object

- A single-column DataFrame can be constructed directly from a Series. The Series's index becomes the DataFrame's row index.

```
In [2]: print("\n(i) Constructing from a Single Series object:")
           df_from_series = pd.DataFrame(population, columns=['population'])
           print(df_from_series)
Out[2]:
        (i) Constructing from a Single Series object:
                    population
        California    38332521
        Texas         26448193
        New York      19651127
        Florida       19552860
        Illinois      12882135
```

Explanation of Example Code

- `pd.DataFrame(population, columns=['population'])`: This directly creates a DataFrame with `population` Series data. It assigns the Series's index as the DataFrame's row index and sets the column name to 'population'.

(ii) From a List of dicts

- Each dictionary in the list becomes a row in the DataFrame.
- Keys of the dictionaries become column names; missing keys are filled with `NaN`.

```
In [3]: print("\n(ii) Constructing from a List of dicts:")
           list_of_dicts = [{'a': i, 'b': 2 * i} for i in range(3)]
           df_from_list_dicts = pd.DataFrame(list_of_dicts)
           print(df_from_list_dicts)
Out[3]:
        (ii) Constructing from a List of dicts:
           a  b
        0  0  0
        1  1  2
        2  2  4

In [4]: print("\n(ii) From a List of dicts (with missing keys):")
```

```
           list_of_dicts_missing = [{'a': 1, 'b': 2}, {'b': 3, 'c': 4}]
           df_from_list_dicts_missing = pd.DataFrame(list_of_dicts_missing)
           print(df_from_list_dicts_missing)
Out[4]:
        (ii) From a List of dicts (with missing keys):
             a    b    c
        0  1.0  2.0  NaN
        1  NaN  3.0  4.0
```

Explanation of Example Code

- `pd.DataFrame(list_of_dicts)`: Creates a DataFrame where each dictionary in the list corresponds to a row. The keys (`'a'`, `'b'`) become the column labels.
- `pd.DataFrame(list_of_dicts_missing)`: Demonstrates that if keys are missing in some dictionaries, Pandas automatically fills those corresponding cells with `NaN` (Not a Number).

(iii) From a Dictionary of Series objects

- The keys of the dictionary become column names.
- The values are `Series` objects, which are aligned by their index to form DataFrame columns.

```
In [5]: print("\n(iii) Constructing from a Dictionary of Series objects:")
           df_from_dict_series = pd.DataFrame({'population': population, 'area': area})
           print(df_from_dict_series)
Out[5]:
        (iii) Constructing from a Dictionary of Series objects:
                      area  population
        California   423967    38332521
        Texas        695662    26448193
        New York     141297    19651127
        Florida      170312    19552860
        Illinois     149995    12882135
```

Explanation of Example Code

- `pd.DataFrame({'population': population, 'area': area})`: Constructs a DataFrame from a dictionary. `'population'` and `'area'` become column names, and their respective `Series` objects populate the columns.
- Pandas automatically aligns the data based on the shared index of the `population` and `area` Series.

(iv) From a Two-dimensional NumPy array

- Given a 2D NumPy array, you can create a DataFrame.
- Column and index names can be specified; otherwise, integer indices are used by default.

```
In [6]: print("\n(iv) Constructing from a Two-dimensional NumPy array:")
           df_from_numpy = pd.DataFrame(np.random.rand(3, 2),
                                        columns=['foo', 'bar'],
                                        index=['a', 'b', 'c'])
           print(df_from_numpy)
Out[6]:
        (iv) Constructing from a Two-dimensional NumPy array:
              foo       bar
        a  0.123456  0.789012 # (Values will vary due to np.random.rand)
```

```
b  0.345678  0.567890
c  0.901234  0.234567
```

Explanation of Example Code

- `pd.DataFrame(np.random.rand(3, 2), ...)`: The `np.random.rand(3, 2)` creates a 3x2 NumPy array, which forms the core data.
- `columns=['foo', 'bar']`: Assigns specific names to the DataFrame's columns.
- `index=['a', 'b', 'c']`: Assigns specific labels to the DataFrame's rows (index).

Module 3

---

1. Explain vectorized string operations in Pandas with examples

Pandas provides a powerful set of **vectorized string operations**, which are crucial for cleaning and manipulating string data, especially in real-world datasets that often contain inconsistencies or missing values. Unlike standard Python string methods that would fail on missing data or require verbose loops for arrays, Pandas' vectorized operations handle entire Series or Index objects efficiently and gracefully.

This capability is exposed through the `.str` accessor available on Pandas Series and Index objects containing string data.

```
In [1]: import numpy as np
           import pandas as pd

           # Demonstrating Python's limitations for vectorized string operations
           x = np.array([2, 3, 5, 7, 11, 13])
           print(f"NumPy vectorized arithmetic:\n{x * 2}")
Out[1]: NumPy vectorized arithmetic:
        [ 4  6 10 14 22 26]

In [2]: data_py = ['peter', 'Paul', 'MARY', 'gUIDO']
           print(f"Python list comprehension (no None):\n{[s.capitalize() for s in
data_py]}")
Out[2]: Python list comprehension (no None):
        ['Peter', 'Paul', 'Mary', 'Guido']

In [3]: data_with_none = ['peter', 'Paul', None, 'MARY', 'gUIDO']
           # This will raise an AttributeError if uncommented
           # [s.capitalize() for s in data_with_none]
           print("Python list comprehension with None would raise
AttributeError:\n'NoneType' object has no attribute 'capitalize'")
Out[3]: Python list comprehension with None would raise AttributeError:
        'NoneType' object has no attribute 'capitalize'
```

Introducing Pandas String Operations

- Pandas' `.str` attribute addresses the need for vectorized string operations and correctly handles missing data.
- It allows applying string methods to every element of a Series while skipping `None` or `NaN` values without error.

```
In [4]: names = pd.Series(data_with_none)
           print(f"Pandas Series with None:\n{names}")
Out[4]: Pandas Series with None:
        0    peter
        1     Paul
        2     None
        3     MARY
        4    gUIDO
        dtype: object

In [5]: print(f"Pandas .str.capitalize() output:\n{names.str.capitalize()}")
Out[5]: Pandas .str.capitalize() output:
```

```
0    Peter
1     Paul
2     None
3     Mary
4    Guido
dtype: object
```

Explanation of Example Code

- `names = pd.Series(data_with_none)`: Creates a Pandas Series that includes `None` (missing) values.
- `names.str.capitalize()`: Applies the `capitalize()` string method to each non-null element in the Series. It correctly handles `None` values by propagating them without raising an error.

Methods Similar to Python String Methods

- Nearly all of Python's built-in string methods (e.g., `lower()`, `len()`, `startswith()`, `split()`) are mirrored by Pandas' vectorized `.str` methods.
- These methods return Series of various types, such as strings, numbers (for length), Booleans, or lists, depending on the operation.

```
In [6]: monte = pd.Series(['Graham Chapman', 'John Cleese', 'Terry Gilliam',
                           'Eric Idle', 'Terry Jones', 'Michael Palin'])
           print(f"Original Series:\n{monte}")
Out[6]: Original Series:
        0    Graham Chapman
        1       John Cleese
        2     Terry Gilliam
        3         Eric Idle
        4       Terry Jones
        5     Michael Palin
        dtype: object

In [7]: print(f"monte.str.lower():\n{monte.str.lower()}")
Out[7]: monte.str.lower():
        0    graham chapman
        1       john cleese
        2     terry gilliam
        3         eric idle
        4       terry jones
        5     michael palin
        dtype: object

In [8]: print(f"monte.str.len():\n{monte.str.len()}")
Out[8]: monte.str.len():
        0    14
        1    11
        2    13
        3     9
        4    11
        5    13
        dtype: int64

In [9]: print(f"monte.str.startswith('T'):\n{monte.str.startswith('T')}")
Out[9]: monte.str.startswith('T'):
        0    False
        1    False
        2     True
        3    False
```

```
        4     True
        5    False
        dtype: bool

In [10]: print(f"monte.str.split():\n{monte.str.split()}")
Out[10]: monte.str.split():
        0    [Graham, Chapman]
        1       [John, Cleese]
        2     [Terry, Gilliam]
        3         [Eric, Idle]
        4       [Terry, Jones]
        5     [Michael, Palin]
        dtype: object
```

Explanation of Example Code

- `monte.str.lower()`: Converts all characters in each string to lowercase, returning a Series of strings.
- `monte.str.len()`: Calculates the length of each string, returning a Series of integers.
- `monte.str.startswith('T')`: Checks if each string begins with 'T', returning a Series of Boolean values.
- `monte.str.split()`: Splits each string by whitespace into a list of substrings, returning a Series of lists.

Methods Using Regular Expressions

- Pandas includes methods that accept regular expressions, mirroring Python's `re` module conventions.
- These allow powerful pattern-based string examination, extraction, and replacement.

```
In [11]: print(f"monte.str.extract('([A-Za-z]+)'):\n{monte.str.extract('([A-Za-z]+)')}")
Out[11]: monte.str.extract('([A-Za-z]+)'):
             0
        0  Graham
        1    John
        2   Terry
        3    Eric
        4   Terry
        5 Michael

In [12]:
print(f"monte.str.findall(r'^[^AEIOU].*[^aeiou]$'):\n{monte.str.findall(r'^[^AEIOU].*[^a
eiou]$')}")
Out[12]: monte.str.findall(r'^[^AEIOU].*[^aeiou]$'):
        0    [Graham Chapman]
        1                  []
        2     [Terry Gilliam]
        3                  []
        4       [Terry Jones]
        5     [Michael Palin]
        dtype: object
```

Explanation of Example Code

- `monte.str.extract('([A-Za-z]+)')`: Uses a regex to extract the first contiguous group of letters from each string. It returns a DataFrame where matched groups are columns.
- `monte.str.findall(r'^[^AEIOU].*[^aeiou]$')`: Finds all occurrences (in this case, the whole string if it matches) that start and end with a consonant using regex. It returns a Series of lists.

Miscellaneous Methods

- Pandas provides other convenient methods for operations like vectorized item access, slicing, and creating dummy variables.
- These can be chained together for complex transformations.

```
In [13]: print(f"monte.str[0:3]:\n{monte.str[0:3]}")
Out[13]: monte.str[0:3]:
        0    Gra
        1    Joh
        2    Ter
        3    Eri
        4    Ter
        5    Mic
        dtype: object

In [14]: print(f"monte.str.split().str.get(-1):\n{monte.str.split().str.get(-1)}")
Out[14]: monte.str.split().str.get(-1):
        0    Chapman
        1     Cleese
        2    Gilliam
        3       Idle
        4      Jones
        5      Palin
        dtype: object

In [15]: full_monte = pd.DataFrame({'name': monte,
                                    'info': ['B|C|D', 'B|D', 'A|C', 'B|D', 'B|C',
                                             'B|C|D']})
            print(f"Original DataFrame 'full_monte':\n{full_monte}")
Out[15]: Original DataFrame 'full_monte':
                 name   info
        0    Graham Chapman  B|C|D
        1       John Cleese    B|D
        2     Terry Gilliam    A|C
        3         Eric Idle    B|D
        4       Terry Jones    B|C
        5     Michael Palin  B|C|D

In [16]:
print(f"full_monte['info'].str.get_dummies('|'):\n{full_monte['info'].str.get_dummies('|
')}")
Out[16]: full_monte['info'].str.get_dummies('|'):
           A  B  C  D
        0  0  1  1  1
        1  0  1  0  1
        2  1  0  1  0
        3  0  1  0  1
        4  0  1  1  0
        5  0  1  1  1
```

Explanation of Example Code

- `monte.str[0:3]`: Performs vectorized slicing, extracting the first three characters from each string. This is equivalent to `monte.str.slice(0, 3)`.
- `monte.str.split().str.get(-1)`: Chaines two string methods; first `split()` creates lists of words, then `get(-1)` extracts the last element (last name) from each list.
- `full_monte['info'].str.get_dummies('|')`: Converts a string column containing delimited categories into a DataFrame of binary indicator variables. It creates a new column for each unique category.

---

2. Develop a python program to demonstrate the arithmetic, comparison and bitwise operator using pandas.eval() function

The `pandas.eval()` function allows for efficient computation of operations on DataFrames and Series using string expressions. This can lead to significant performance improvements, especially for large datasets, by leveraging the underlying Numexpr library to perform operations more efficiently than standard Python.

`pandas.eval()` supports a wide range of operations, including arithmetic, comparison, and bitwise operators.

```
In [1]: import pandas as pd
           import numpy as np

           # Set up DataFrames for demonstration
           nrows, ncols = 10000, 10 # Reduced size for quicker demonstration
           rng = np.random.RandomState(42)
           df1, df2, df3, df4, df5 = (pd.DataFrame(rng.rand(nrows, ncols)) for i in range(5))

           print("--- Demonstrating pandas.eval() ---")
Out[1]: --- Demonstrating pandas.eval() ---
```

(i) Arithmetic Operators

- `pd.eval()` supports all standard arithmetic operators (+, -, *, /, **).
- Expressions are provided as strings, which Pandas parses and computes efficiently.

```
In [2]: # Compute a complex arithmetic expression using standard Pandas
           result1_arith = -df1 * df2 / (df3 + df4) - df5

In [3]: # Compute the same expression using pd.eval()
           result2_arith = pd.eval('-df1 * df2 / (df3 + df4) - df5')

In [4]: # Verify that both methods produce the same result
           print(f"Arithmetic results are close: {np.allclose(result1_arith, result2_arith)}")
Out[4]: Arithmetic results are close: True
```

Explanation of Example Code

- `result1_arith = -df1 * df2 / (df3 + df4) - df5`: This line performs a series of arithmetic operations using standard Pandas DataFrame syntax.
- `result2_arith = pd.eval('-df1 * df2 / (df3 + df4) - df5')`: This line performs the *exact same* arithmetic operations, but by providing the expression as a string to `pd.eval()`.
- `np.allclose(result1_arith, result2_arith)`: Verifies that the results from both computation methods are numerically equivalent, confirming `pd.eval()`'s accuracy.

(ii) Comparison Operators

- `pd.eval()` supports all comparison operators (<, >, <=, >=, ==, !=).
- Chained comparison expressions are also supported for concise Boolean logic.

```
In [5]: # Compute a chained comparison expression using standard Pandas
           result1_comp = (df1 < df2) & (df2 <= df3) & (df3 != df4)

In [6]: # Compute the same expression using pd.eval()
           result2_comp = pd.eval('df1 < df2 <= df3 != df4')

In [7]: # Verify that both methods produce the same result
           print(f"Comparison results are close: {np.allclose(result1_comp, result2_comp)}")
Out[7]: Comparison results are close: True
```

Explanation of Example Code

- `result1_comp = (df1 < df2) & (df2 <= df3) & (df3 != df4)`: This performs multiple comparison operations and combines them using bitwise `&` (AND) in standard Pandas.
- `result2_comp = pd.eval('df1 < df2 <= df3 != df4')`: This uses `pd.eval()` to compute the same chained comparison expression, which is more concise.
- `np.allclose(result1_comp, result2_comp)`: Confirms the equivalence of the Boolean results from both computation methods.

(iii) Bitwise Operators

- `pd.eval()` supports the bitwise `&` (AND) and `|` (OR) operators.
- It also supports the use of the literal `and` and `or` keywords in Boolean expressions, which can be more readable.

```
In [8]: # Compute a bitwise expression using standard Pandas
           result1_bitwise = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)

In [9]: # Compute the same expression using pd.eval() with bitwise operators
           result2_bitwise = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')

In [10]: # Verify that both methods produce the same result
            print(f"Bitwise results are close (using & |): {np.allclose(result1_bitwise,
result2_bitwise)}")
Out[10]: Bitwise results are close (using & |): True

In [11]: # Compute the same expression using pd.eval() with literal 'and'/'or'
            result3_bitwise = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')

In [12]: # Verify equivalence between bitwise and literal Boolean operators in pd.eval()
            print(f"Bitwise vs. literal Boolean results are close: {np.allclose(result1_bitwise,
result3_bitwise)}")
Out[12]: Bitwise vs. literal Boolean results are close: True
```

Explanation of Example Code

- `result1_bitwise = (df1 < 0.5) & (df2 < 0.5) | (df3 < df4)`: This performs Boolean comparisons and combines them using standard Pandas bitwise operators (`&`, `|`).
- `result2_bitwise = pd.eval('(df1 < 0.5) & (df2 < 0.5) | (df3 < df4)')`: Shows the same expression computed with `pd.eval()` using bitwise operators as a string.
- `result3_bitwise = pd.eval('(df1 < 0.5) and (df2 < 0.5) or (df3 < df4)')`: Demonstrates using the more Pythonic literal `and` and `or` keywords within the `pd.eval()` string for Boolean operations.

- `np.allclose(...)`: All `np.allclose` calls confirm that `pd.eval()` produces identical results to standard Pandas operations, highlighting its accuracy and efficiency.

---

3. **Develop a python program to create a Vectorized string and for Applying the pandas string methods along with regular expression on the created string**

Pandas provides **vectorized string operations** that allow you to efficiently apply string methods and regular expressions across entire Series or Index objects containing string data. This capability is exposed through the `.str` accessor.

```
In [1]: import pandas as pd
           import numpy as np # Used for np.nan, if needed

           print("--- Python Program: Vectorized String Operations ---")
Out[1]: --- Python Program: Vectorized String Operations ---
```

Creating a Vectorized String (Pandas Series)

- In Pandas, a "vectorized string" is typically a `pd.Series` object whose elements are strings.
- The `.str` accessor allows applying string methods to all elements simultaneously.

```
In [2]: # Create a Pandas Series (our 'vectorized string') with sample names, including
a None value
           names_data = ['peter', 'Paul', None, 'MARY', 'gUIDO', 'john doe']
           names_series = pd.Series(names_data)
           print(f"Created Vectorized String (Pandas Series):\n{names_series}")
Out[2]: Created Vectorized String (Pandas Series):
        0       peter
        1        Paul
        2        None
        3        MARY
        4       gUIDO
        5    john doe
        dtype: object
```

Explanation of Example Code

- `names_series = pd.Series(names_data)`: A Pandas Series is created from a list of strings, including a `None` value to show `NaN` handling. This Series acts as our "vectorized string."

Applying Pandas String Methods

- Pandas' `.str` accessor provides methods mirroring Python's built-in string functions and advanced regular expression capabilities.
- These methods operate element-wise on the Series, handling `None`/`NaN` values gracefully.

(i) Basic Pandas String Methods

- These methods apply standard string operations (e.g., case conversion, length, splitting) to each element.
- They return a Series of strings, numbers, Booleans, or lists, depending on the operation.

```
In [3]: print(f"Applying .str.capitalize():\n{names_series.str.capitalize()}")
```

```
Out[3]: Applying .str.capitalize():
        0       Peter
        1        Paul
        2        None
        3        Mary
        4       Guido
        5    John doe
        dtype: object

In [4]: print(f"\nApplying .str.len():\n{names_series.str.len()}")
Out[4]:
        Applying .str.len():
        0    5.0
        1    4.0
        2    NaN
        3    4.0
        4    5.0
        5    8.0
        dtype: float64

In [5]: print(f"\nApplying .str.split():\n{names_series.str.split()}")
Out[5]:
        Applying .str.split():
        0        [peter]
        1         [Paul]
        2           None
        3         [MARY]
        4        [gUIDO]
        5    [john, doe]
        dtype: object
```

Explanation of Example Code

- `names_series.str.capitalize()`: Capitalizes the first letter of each string and converts the rest to lowercase, correctly handling the `None` value.
- `names_series.str.len()`: Calculates the character length of each string, resulting in `NaN` for the `None` entry.
- `names_series.str.split()`: Splits each string by whitespace into a list of words, returning a Series of lists.

(ii) Regular Expression Methods

- The `.str` accessor includes methods that accept regular expressions for powerful pattern matching, extraction, and replacement.
- These methods are highly effective for complex text analysis and data cleaning.

```
In [6]: print(f"Applying .str.extract('([A-Za-z]+)' to get first
word):\n{names_series.str.extract('([A-Za-z]+)')}")
Out[6]:
        Applying .str.extract('([A-Za-z]+)' to get first word):
             0
        0  peter
        1   Paul
        2   None
        3   MARY
        4  gUIDO
```

```
        5   john

In [7]: print(f"\nApplying .str.contains('a', case=False) to check for 'a' (case-
insensitive):\n{names_series.str.contains('a', case=False)}")
Out[7]:
        Applying .str.contains('a', case=False) to check for 'a' (case-insensitive):
        0     True
        1     True
        2    False
        3     True
        4    False
        5     True
        dtype: bool
```

Explanation of Example Code

- `names_series.str.extract('([A-Za-z]+)')`: Uses a regular expression to extract the first sequence of alphabetic characters from each string.
- `names_series.str.contains('a', case=False)`: Checks if each string contains the letter 'a' (ignoring case) using a regular expression, returning a Boolean Series.

---

4. **Write a Python script to demonstrate handling and analyzing time-series data**

Time-series data involves data indexed in time order, used in stock prices, weather records, etc.
Python's `pandas`, `numpy`, and `datetime` simplify **creation, parsing, indexing, slicing, and analysis** of time-series data.

Key Concepts:

- **Timestamp:** A specific point in time (e.g., '2025-07-02 10:00').
- **DatetimeIndex:** An index of timestamps used in Series/DataFrames.
- **Timedelta:** Duration representing differences in dates/times.
- **Rolling Analysis:** Calculates moving statistics for analysis.

```
In [1]: import pandas as pd
           import numpy as np
           from datetime import datetime

           print("--- Python Script: Pandas Time-Series Data Handling ---")
Out[1]: --- Python Script: Pandas Time-Series Data Handling ---
```

Pandas Time Series: Core Objects and Creation

- Pandas' primary time series objects are `Timestamp` (for specific points in time), `Timedelta` (for durations), and `Period` (for fixed-frequency intervals).
- Their corresponding index structures are `DatetimeIndex`, `TimedeltaIndex`, and `PeriodIndex`.
- `pd.to_datetime()` is a versatile function to parse various date formats into `Timestamp` or `DatetimeIndex`.

```
In [2]: # Create a Timestamp object from a string
           pd_timestamp = pd.to_datetime("July 4th, 2015")
           print(f"1. Pandas Timestamp object: {pd_timestamp}")
Out[2]: 1. Pandas Timestamp object: 2015-07-04 00:00:00
```

```
In [3]: # Create a DatetimeIndex from a list of mixed date formats
           datetime_index = pd.to_datetime([datetime(2024, 1, 1), '2024-01-02', 'Jan 3, 2024'])
           print(f"\n2. DatetimeIndex from mixed formats:\n{datetime_index}")
Out[3]:
        2. DatetimeIndex from mixed formats:
        DatetimeIndex(['2024-01-01', '2024-01-02', '2024-01-03'], dtype='datetime64[ns]',
freq=None)

In [4]: # Create a Timedelta object (duration)
           pd_timedelta = pd.Timedelta('1 hour 30 minutes')
           print(f"\n3. Pandas Timedelta object: {pd_timedelta}")
Out[4]: 3. Pandas Timedelta object: 0 days 01:30:00

In [5]: # Create a Period object (fixed-frequency interval)
           pd_period = pd.Period('2024-07', freq='M')
           print(f"\n4. Pandas Period object: {pd_period}")
Out[5]: 4. Pandas Period object: 2024-07
```

Explanation of Example Code

- `pd.to_datetime(...)`: Demonstrates creating `Timestamp` and `DatetimeIndex` from various string and `datetime` inputs.
- `pd.Timedelta(...)`: Shows how to define a specific duration using a string.
- `pd.Period(...)`: Illustrates creating a fixed-frequency time interval, here representing July 2024.

Time-Indexed Series and Data Selection

- Pandas' strength lies in using `DatetimeIndex` to index Series or DataFrames.
- This allows for powerful and intuitive time-based data selection, including partial string indexing.

```
In [6]: # Create a Series with a DatetimeIndex
           data_values = [100, 150, 120, 180, 200, 160]
           time_index = pd.date_range('2024-07-01', periods=6, freq='D')
           ts_series = pd.Series(data_values, index=time_index)
           print(f"1. Time-indexed Series:\n{ts_series}")
Out[6]: 1. Time-indexed Series:
        2024-07-01    100
        2024-07-02    150
        2024-07-03    120
        2024-07-04    180
        2024-07-05    200
        2024-07-06    160
        Freq: D, dtype: int64

In [7]: # Select data for a specific date
           data_on_date = ts_series['2024-07-03']
           print(f"\n2. Data on '2024-07-03': {data_on_date}")
Out[7]: 2. Data on '2024-07-03': 120

In [8]: # Slice data using a date range
           data_slice = ts_series['2024-07-02':'2024-07-04']
           print(f"\n3. Data sliced from '2024-07-02' to '2024-07-04':\n{data_slice}")
Out[8]:
        3. Data sliced from '2024-07-02' to '2024-07-04':
        2024-07-02    150
        2024-07-03    120
```

```
        2024-07-04    180
        Freq: D, dtype: int64

In [9]: # Select data for a specific month (using partial string indexing)
           data_for_month = ts_series['2024-07']
           print(f"\n4. Data for July 2024:\n{data_for_month}")
Out[9]:
        4. Data for July 2024:
        2024-07-01    100
        2024-07-02    150
        2024-07-03    120
        2024-07-04    180
        2024-07-05    200
        2024-07-06    160
        Freq: D, dtype: int64
```

Explanation of Example Code

- `pd.date_range(...)`: Generates a sequence of daily `Timestamp` objects used as the index.
- `ts_series = pd.Series(...)`: Creates a Series indexed by these `Timestamp` objects.
- `ts_series['2024-07-03']`: Shows precise selection of a data point by its date.
- `ts_series['2024-07-02':'2024-07-04']`: Demonstrates slicing data over a continuous date range.
- `ts_series['2024-07']`: Illustrates selecting all data points within a specific month by providing only the year and month.

Generating Regular Time Sequences

- Pandas provides dedicated functions to create sequences of dates, periods, or timedeltas with specified frequencies.

```
In [10]: # Generate a daily date range
            daily_dates = pd.date_range(start='2024-01-01', periods=5, freq='D')
            print(f"1. Daily DatetimeIndex (5 days):\n{daily_dates}")
Out[10]: 1. Daily DatetimeIndex (5 days):
        DatetimeIndex(['2024-01-01', '2024-01-02', '2024-01-03', '2024-01-04',
                       '2024-01-05'],
                      dtype='datetime64[ns]', freq='D')

In [11]: # Generate an hourly date range
            hourly_dates = pd.date_range(start='2024-07-01 10:00', periods=3, freq='H')
            print(f"\n2. Hourly DatetimeIndex (3 hours):\n{hourly_dates}")
Out[11]:
        2. Hourly DatetimeIndex (3 hours):
        DatetimeIndex(['2024-07-01 10:00:00', '2024-07-01 11:00:00',
                       '2024-07-01 12:00:00'],
                      dtype='datetime64[ns]', freq='H')

In [12]: # Generate a monthly period range
            monthly_periods = pd.period_range(start='2024-01', periods=4, freq='M')
            print(f"\n3. Monthly PeriodIndex (4 months):\n{monthly_periods}")
Out[12]:
        3. Monthly PeriodIndex (4 months):
        PeriodIndex(['2024-01', '2024-02', '2024-03', '2024-04'], dtype='period[M]')

In [13]: # Generate a time delta range
```

```
            delta_range = pd.timedelta_range(start='0 days', periods=3, freq='10min')
            print(f"\n4. TimedeltaIndex (3 periods, 10 min each):\n{delta_range}")
Out[13]:
        4. TimedeltaIndex (3 periods, 10 min each):
        TimedeltaIndex(['0 days 00:00:00', '0 days 00:10:00', '0 days 00:20:00'],
dtype='timedelta64[ns]', freq='10min')
```

Explanation of Example Code

- `pd.date_range(...)`: Creates sequences of `DatetimeIndex` objects with specified `start` dates, `periods`, and `freq` (e.g., 'D' for daily, 'H' for hourly).
- `pd.period_range(...)`: Generates a `PeriodIndex` for fixed-frequency intervals, such as 'M' for monthly.
- `pd.timedelta_range(...)`: Creates a `TimedeltaIndex` for sequences of durations, useful for representing time intervals.

Module 4

---

1. Explain the general tips for creating visualizations with Matplotlib

General Matplotlib Tips

Before creating plots with Matplotlib, it helps to know a few useful basics about how to use the library.

Importing Matplotlib

Like we use np for NumPy and pd for Pandas, we use these standard imports for Matplotlib:

```
import matplotlib as mpl
import matplotlib.pyplot as plt
```

We'll mostly use plt throughout, as it provides an easy interface for plotting.

Setting Styles

Matplotlib supports styles to change the appearance of your plots easily.

```
plt.style.use('classic')
```

This sets the classic Matplotlib style.

show() or No show()? – How to Display Plots

How you view plots depends on **where you're using Matplotlib**: in a script, an IPython terminal, or an IPython notebook.

Plotting from a Script

If you're writing a Python script, use plt.show() to display your plots:

```
# ------- file: myplot.py ------
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(0, 10, 100)
plt.plot(x, np.sin(x))
plt.plot(x, np.cos(x))
plt.show()
```

Run this script from a terminal:

$ python myplot.py

- plt.show() opens a window with the plot.
- It interacts with your system's graphical backend.
- **Use it only once per session**, usually at the end of the script.

Using it multiple times may cause unexpected behavior.

Plotting from an IPython Shell

If you're using Matplotlib in an IPython terminal, enable Matplotlib mode using:

```
%matplotlib
```

Then:

```
import matplotlib.pyplot as plt
```

- Plot commands will now automatically open a window.
- To update an existing plot, use `plt.draw()`.
- You don't need `plt.show()` in this mode.

Plotting from an IPython Notebook

In a Jupyter (IPython) notebook, use %matplotlib to plot interactively.

Two options for embedding plots:

- `%matplotlib` notebook → Interactive plots
- `%matplotlib inline` → Static image plots (used in this book)

Example:

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 10, 100)
fig = plt.figure()
plt.plot(x, np.sin(x), '-')
plt.plot(x, np.cos(x), '--')
```

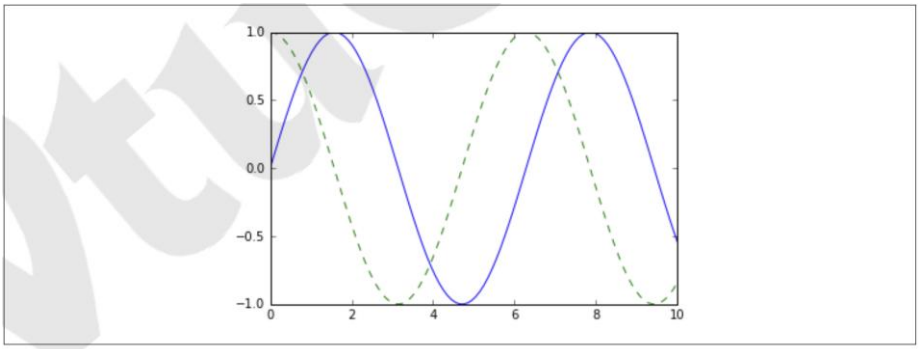

Figure 4-1. Basic plotting example

Saving Figures to File

You can save plots using savefig():

```
fig.savefig('my_figure.png')
```

Now a file named my_figure.png will be saved in your current directory.

You can verify it using:

```
!ls -lh my_figure.png
```

To view it inside the notebook:

from IPython.display import Image

```
Image('my_figure.png')
```

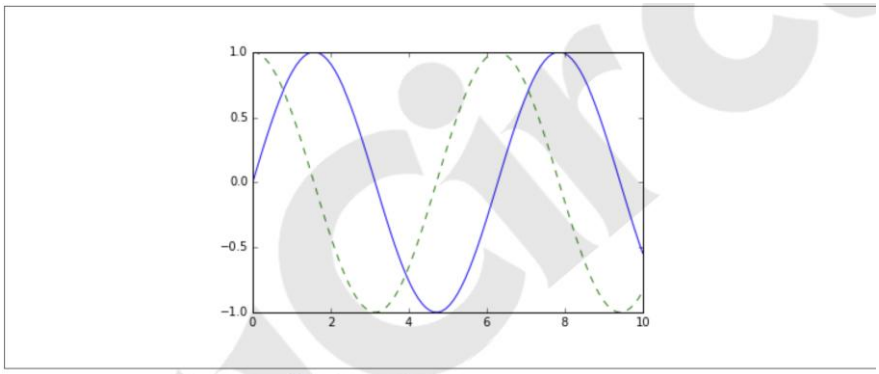

Figure 4-2. PNG rendering of the basic plot

File Format Support

Matplotlib supports many file formats (depending on your system). You can check supported formats:

```
In[8]: fig.canvas.get_supported_filetypes()
Out[8]:
{
    'eps':  'Encapsulated Postscript',
    'jpeg': 'Joint Photographic Experts Group',
    'jpg':  'Joint Photographic Experts Group',
    'pdf':  'Portable Document Format',
    'pgf':  'PGF code for LaTeX',
    'png':  'Portable Network Graphics',
    'ps':   'Postscript',
    'raw':  'Raw RGBA bitmap',
    'rgba': 'Raw RGBA bitmap',
    'svg':  'Scalable Vector Graphics',
    'svgz': 'Scalable Vector Graphics',
    'tif':  'Tagged Image File Format',
    'tiff': 'Tagged Image File Format'
}
```

When saving a figure, you **don't need to use** plt.show().

2. Discuss the role of Seaborn in enhancing data visualizations.

Matplotlib is a powerful and widely-used tool for creating plots. However, even experienced users often find some limitations:

Common Issues with Matplotlib:

- **Old default styles** (especially before version 2.0), based on MATLAB from 1999.
- **Low-level API** — creating advanced plots takes a lot of code.
- **Poor integration with Pandas DataFrames** — you often have to extract and reshape data manually.

Seaborn:

Seaborn is a library built on top of Matplotlib that fixes many of these issues:

- Comes with **better default styles and colors**.
- Provides **simple high-level functions** for common statistical plots.
- **Works smoothly with Pandas DataFrames**, using column labels intelligently.

Seaborn vs. Matplotlib: A Comparison

Let's look at the same plot made using both Matplotlib and Seaborn.

Using Matplotlib

In[1]:

```
import matplotlib.pyplot as plt
plt.style.use('classic')
%matplotlib inline
import numpy as np
import pandas as pd
```

In[2]:

```
# Create some data
rng = np.random.RandomState(0)
x = np.linspace(0, 10, 500)
y = np.cumsum(rng.randn(500, 6), 0)
```

In[3]:

```
# Plot the data with Matplotlib defaults
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left');
```

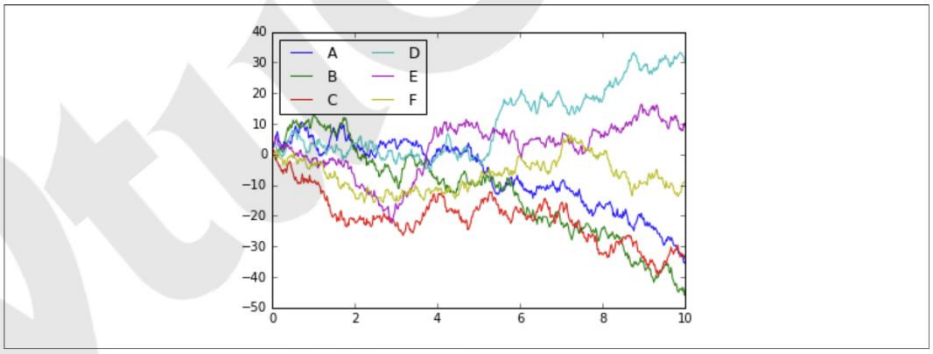

Figure 4-111. Data in Matplotlib's default style

The plot contains all the necessary data, but looks a bit outdated and not very visually appealing.

Using Seaborn

Seaborn can **override Matplotlib's settings** to give a better look, even with the same code.

In[4]:

```
import seaborn as sns
sns.set()
```

In[5]:

```
# same plotting code as above!
plt.plot(x, y)
plt.legend('ABCDEF', ncol=2, loc='upper left');
```

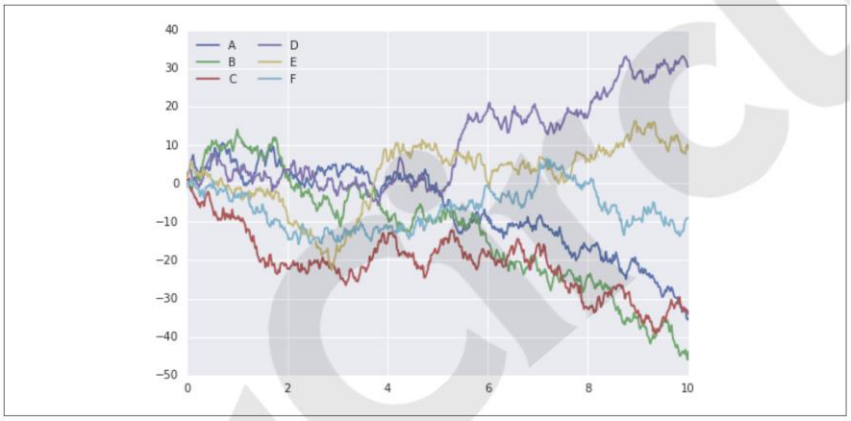

Figure 4-112. Data in Seaborn's default style

The same plot now looks more modern and polished — just by calling sns.set()!

Feature	Matplotlib	Seaborn
Style Defaults	Basic (old-looking pre-v2.0)	Clean and modern
High-Level Plotting Functions	Limited	Many built-in statistical plots
Pandas DataFrame Integration	Manual	Automatic and smart
Ease of Use	Requires more code	Simple and intuitive

Seaborn doesn't replace Matplotlib, but **makes it much easier and more elegant** to create attractive and informative plots — especially for statistical and DataFrame-based visualizations.

---

3. Develop Python code for simple line plot with specific line color, line styles, Axes Limits and Labeling

Simple Line Plot with Enhancements

In[1]: `%matplotlib inline`

```
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

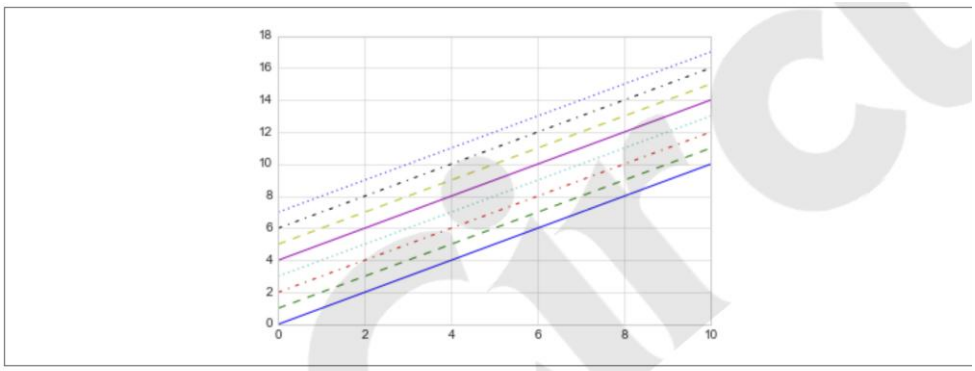

Figure 4-10. Example of various line styles

Plotting the Data with Line Styles and Colors

In[2]:

```
x = np.linspace(0, 10, 1000)
# Plot multiple lines with specific styles and colors
plt.plot(x, np.sin(x), '-g', label='sin(x)')   # solid green line
plt.plot(x, np.cos(x), ':b', label='cos(x)')   # dotted blue line
```

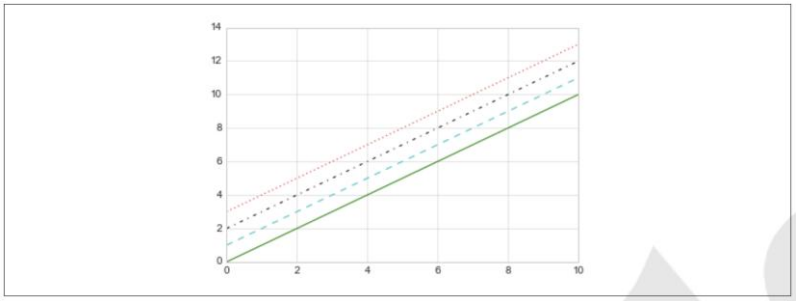

Figure 4-11. Controlling colors and styles with the shorthand syntax

Setting Axes Limits

In[3]:

```
plt.axis([0, 10, -1.5, 1.5])  # [xmin, xmax, ymin, ymax]
```

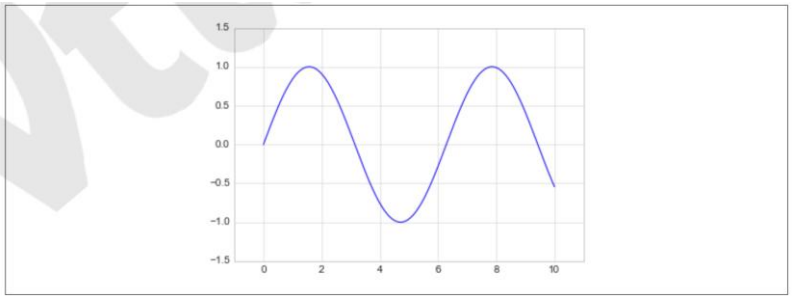

Figure 4-12. Example of setting axis limits

Adding Title and Axis Labels

In[4]:

```
plt.title("Trigonometric Functions")

plt.xlabel("x")

plt.ylabel("Function value")
```

Adding a Legend

In[5]:

```
plt.legend()
```

---

4. With python program explain simple scatter plot using plot() and scatter() function of matplotlib.

Scatter plots are similar to line plots, but instead of connecting points with lines, they show individual data points using symbols like dots or circles.

In[1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
plt.style.use('seaborn-whitegrid')
import numpy as np
```

Scatter Plots with plt.plot

You can use plt.plot() to create scatter plots using marker symbols.

In[2]:

```
x = np.linspace(0, 10, 30)
y = np.sin(x)
plt.plot(x, y, 'o', color='black')
```

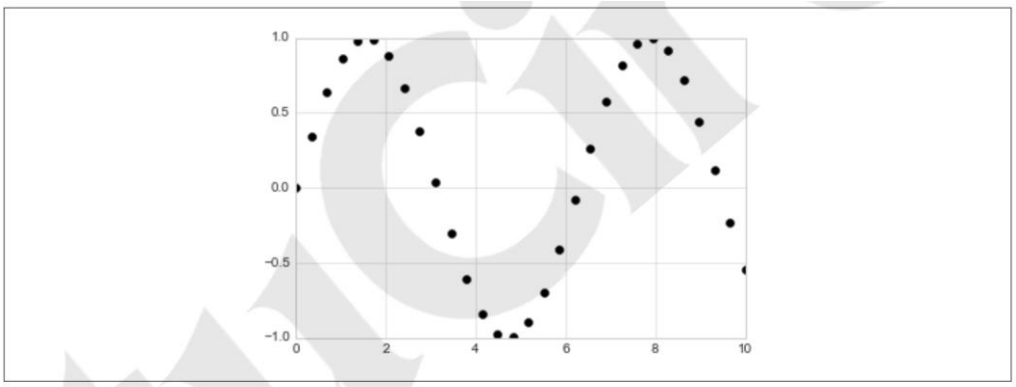

Figure 4-20. Scatter plot example

The third argument ('o') specifies the **marker type**. You can use different marker codes like 'x', 'v', 's', etc.

Marker Styles Demonstration

In[3]:

```
rng = np.random.RandomState(0)
for marker in ['o', '.', ',', 'x', '+', 'v', '^', '<', '>', 's', 'd']:
    plt.plot(rng.rand(5), rng.rand(5), marker, label="marker='{0}'".format(marker))

plt.legend(numpoints=1)
plt.xlim(0, 1.8)
```

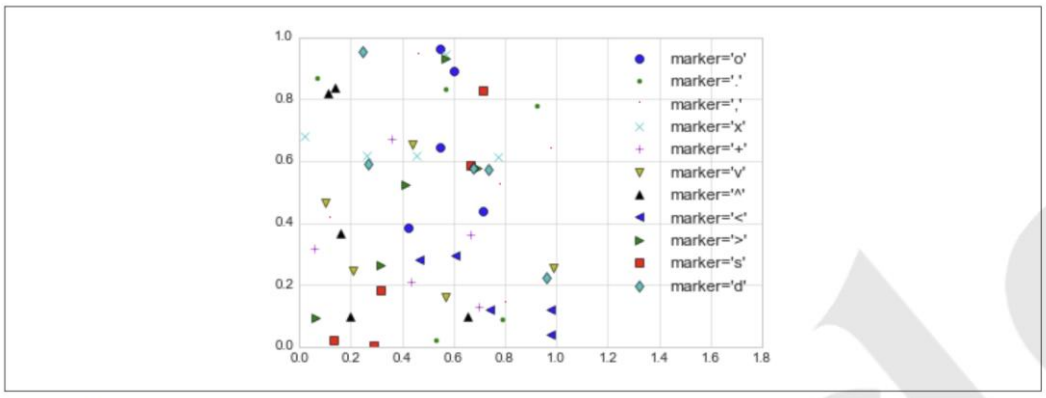

Figure 4-21. Demonstration of point numbers

Combining Markers with Lines

You can mix markers, lines, and color codes.

In[4]:

```
plt.plot(x, y, '-ok')  # line (-), circle marker (o), black (k)
```

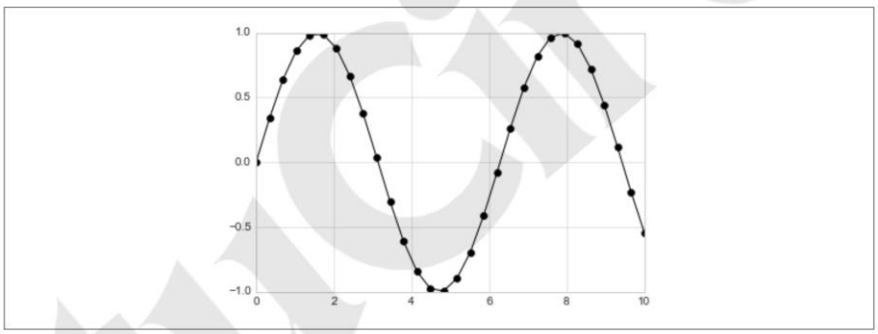

Figure 4-22. Combining line and point markers

Customizing Marker and Line Properties

In[5]:

```
plt.plot(x, y, '-p', color='gray',
         markersize=15, linewidth=4,
         markerfacecolor='white',
         markeredgecolor='gray',
         markeredgewidth=2)
```

```
plt.ylim(-1.2, 1.2)
```

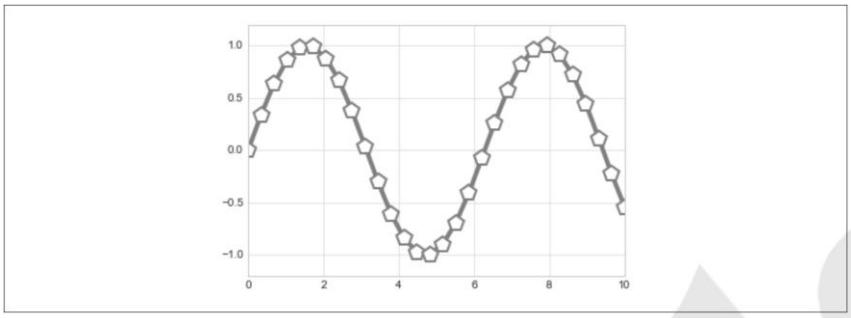

Figure 4-23. Customizing line and point numbers

Scatter Plots with plt.scatter

`plt.scatter()` offers more control over individual point appearance (color, size, etc.).

In[6]:

```
plt.scatter(x, y, marker='o')
```

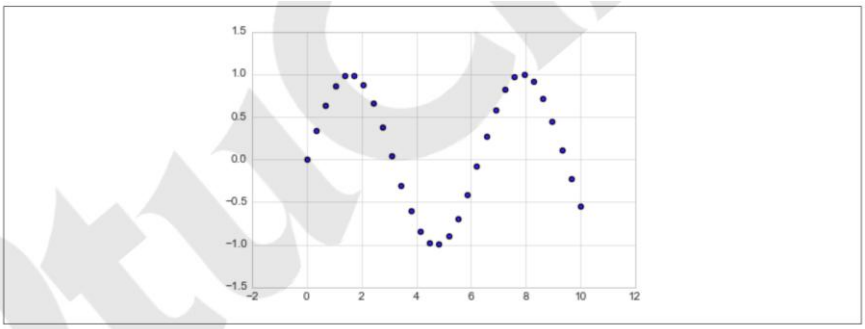

Figure 4-24. A simple scatter plot

Color, Size, and Transparency in Scatter Plots

In[7]:

```
rng = np.random.RandomState(0)
x = rng.randn(100)
y = rng.randn(100)
colors = rng.rand(100)
sizes = 1000 * rng.rand(100)

plt.scatter(x, y, c=colors, s=sizes, alpha=0.3, cmap='viridis')
plt.colorbar()  # shows color scale
```

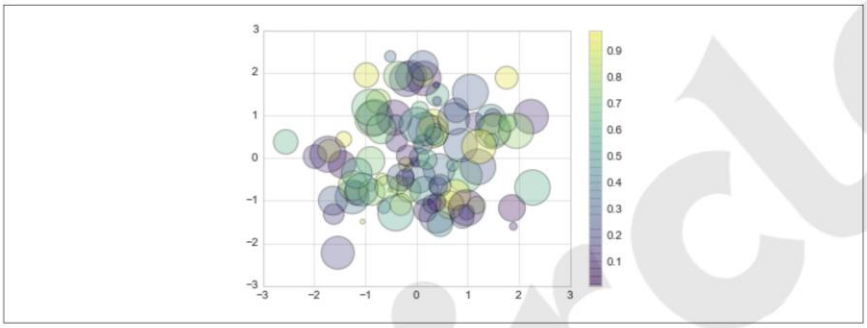

Figure 4-25. Changing size, color, and transparency in scatter points

- c=colors maps values to colors
- s=sizes defines size of each point
- alpha controls transparency

Multifeature Scatter Plot (Iris Dataset)

In[8]:

```
from sklearn.datasets import load_iris
iris = load_iris()
features = iris.data.T  # transpose to get columns

plt.scatter(features[0], features[1], alpha=0.2,
            s=100 * features[3], c=iris.target, cmap='viridis')
plt.xlabel(iris.feature_names[0])
plt.ylabel(iris.feature_names[1])
```

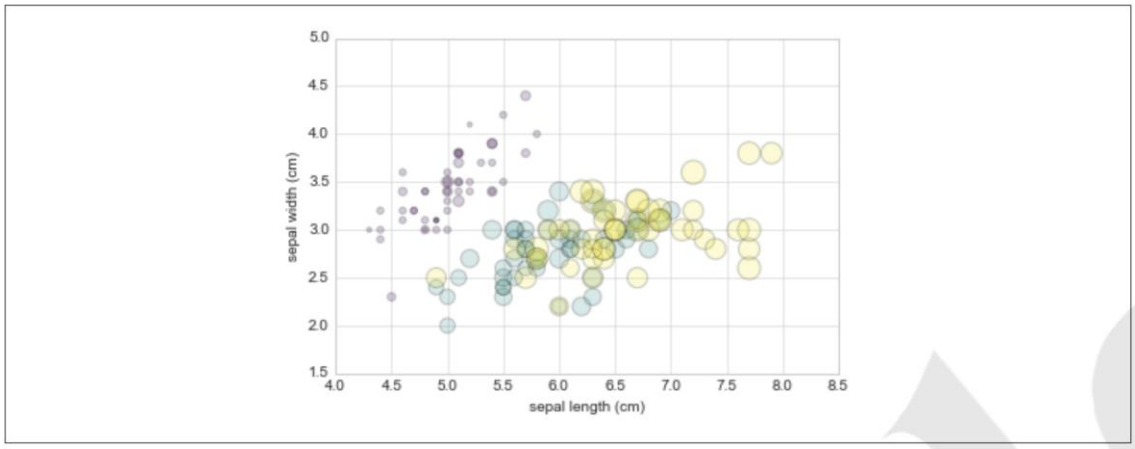

Figure 4-26. Using point properties to encode features of the Iris data

This scatter plot shows:

- x = sepal length
- y = sepal width
- size = petal width
- color = species

We are visualizing **4 dimensions** of the data at once.

plot vs scatter: A Note on Efficiency

- For **large datasets**, prefer plt.plot() — it is **faster**, since all points share the same appearance.
- plt.scatter() is **more flexible**, but **slower**, since each point can have different styles (color, size, etc.).

---

5. With example illustrate the creation of histogram, KDE and pairplot using seaborn

In[1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
import pandas as pd
```

Histogram using plt.hist

You can use Matplotlib's plt.hist to draw histograms for each column.

In[2]:

```
data = np.random.multivariate_normal([0, 0], [[5, 2], [2, 2]], size=2000)
data = pd.DataFrame(data, columns=['x', 'y'])

for col in 'xy':
    plt.hist(data[col], density=True, alpha=0.5)
```

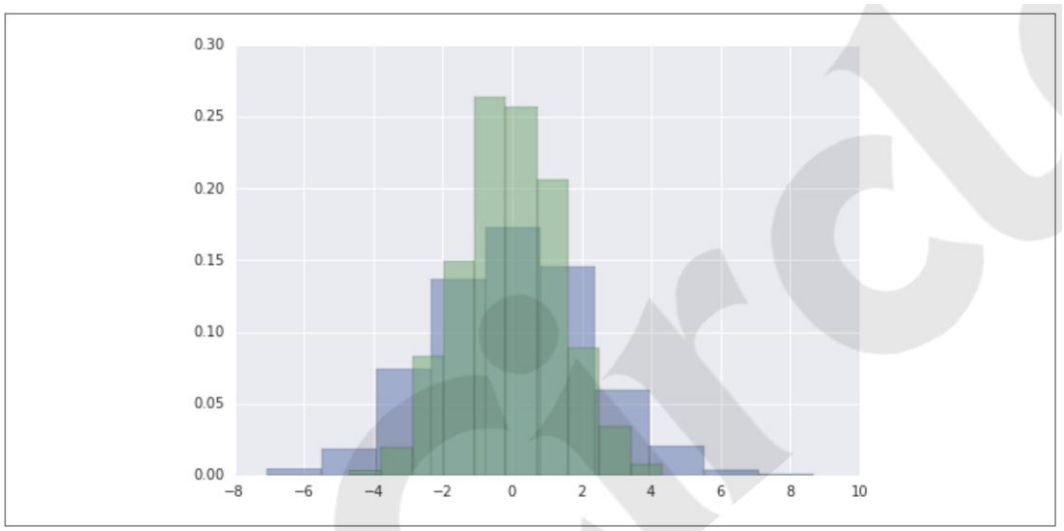

Figure 4-113. Histograms for visualizing distributions

normed=True is replaced with density=True in newer versions of Matplotlib.

KDE using sns.kdeplot

Seaborn makes it easy to visualize smoothed distributions using **kernel density estimation**.

In[3]:

```
for col in 'xy':
    sns.kdeplot(data[col], shade=True)
```

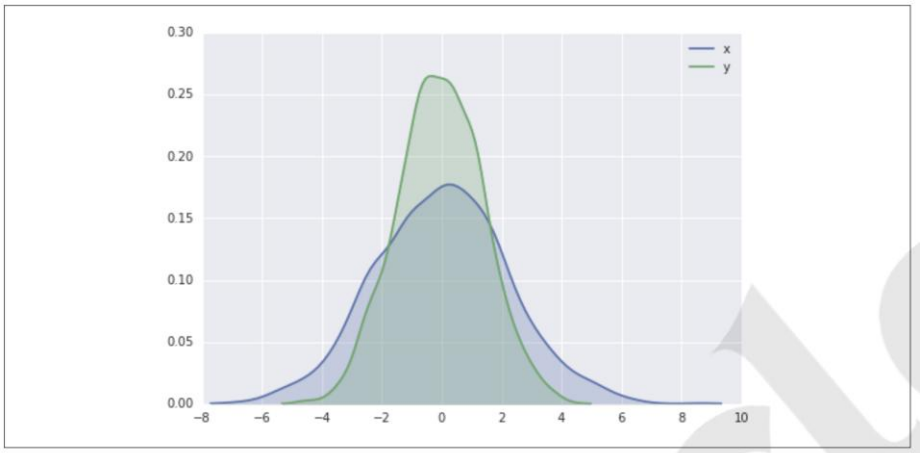

Figure 4-114. Kernel density estimates for visualizing distributions

Combining Histogram and KDE using sns.distplot

distplot allows combining both histogram and KDE.
(Note: distplot is deprecated in newer Seaborn versions; use displot instead.)

In[4]:

```
sns.distplot(data['x'])
sns.distplot(data['y'])
```

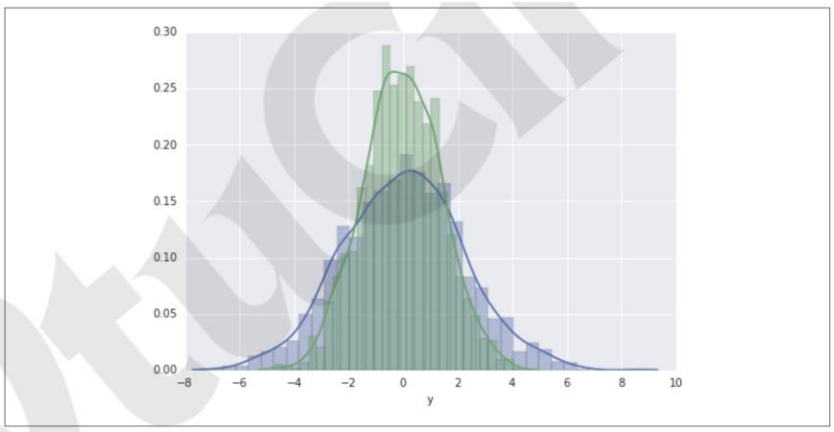

Figure 4-115. Kernel density and histograms plotted together

2D KDE Plot using sns.kdeplot

You can pass the entire DataFrame to get a 2D KDE plot.

In[5]:

```
sns.kdeplot(data)
```

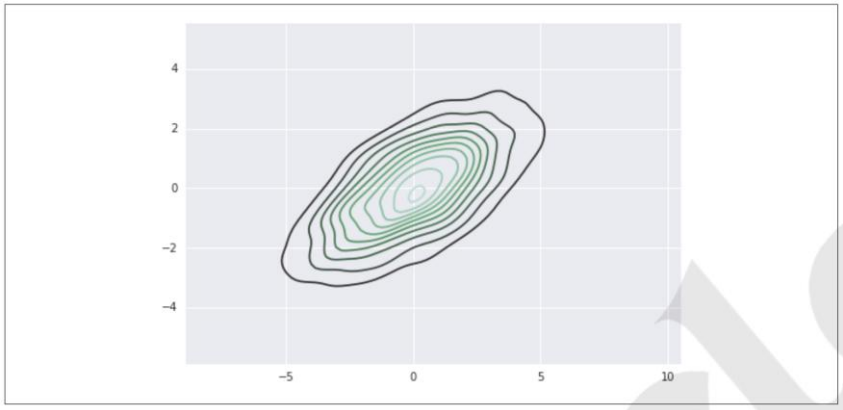

Figure 4-116. A two-dimensional kernel density plot

🔗 Joint Distribution using sns.jointplot (KDE)

Joint plots show both joint and marginal distributions.

In[6]:

```
with sns.axes_style('white'):
    sns.jointplot("x", "y", data, kind='kde')
```

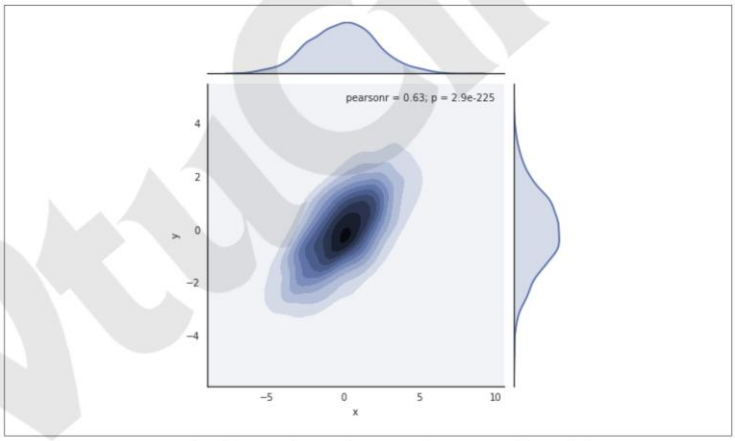

Figure 4-117. A joint distribution plot with a two-dimensional kernel density estimate

Joint Plot with Hex Bins

You can also show joint distributions using **hexagonal bins**.

In[7]:

```
with sns.axes_style('white'):
    sns.jointplot("x", "y", data, kind='hex')
```

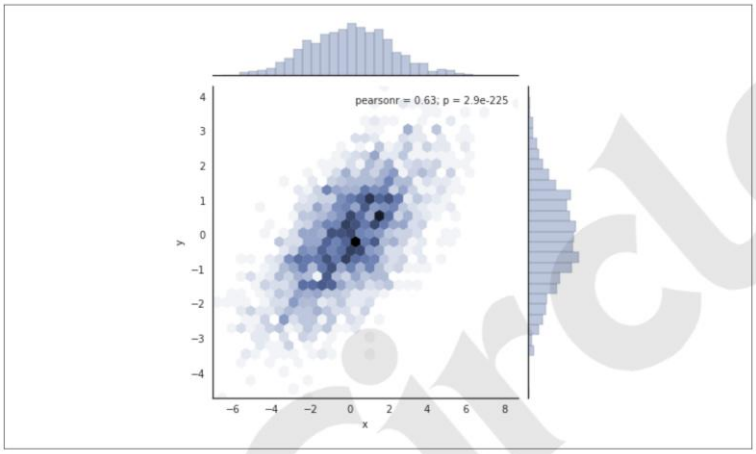

Figure 4-118. A joint distribution plot with a hexagonal bin representation

Pair Plot using sns.pairplot

Pair plots help visualize **multivariate relationships** in high-dimensional datasets.
Let's use the built-in **Iris dataset**.

In[8]:

```
iris = sns.load_dataset("iris")
sns.pairplot(iris, hue='species', height=2.5)
```

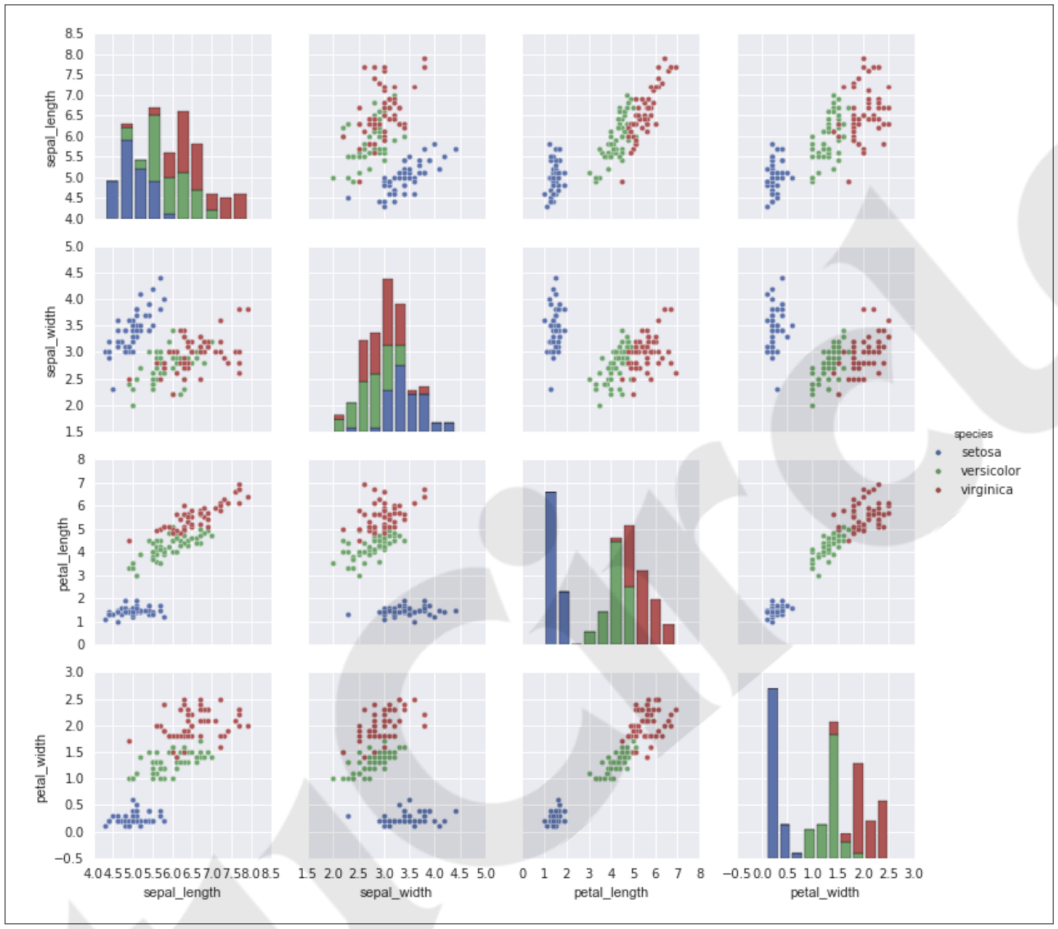

Figure 4-119. A pair plot showing the relationships between four variables

Summary

Plot Type	Function
Histogram	plt.hist
KDE (1D)	sns.kdeplot
KDE + Histogram	sns.distplot
KDE (2D)	sns.kdeplot(data)
Joint KDE	sns.jointplot
Joint Hex	sns.jointplot(..., kind='hex')
Pair Plot	sns.pairplot

Module 5

---

1. **Compare supervised and unsupervised learning with examples.**

Supervised Learning

Supervised learning algorithms are trained on a labeled dataset, meaning each input data point is paired with a corresponding correct output or "label." The algorithm learns to map inputs to outputs by identifying patterns and relationships within this labeled data. Once trained, the model can then predict outputs for new, unseen input data.

Key Characteristics:

- **Labeled Data:** Requires training data where the desired output for each input is known. This often means human effort is involved in labeling the data.
- **Goal-Oriented:** The primary goal is to predict a specific outcome or classify data into predefined categories.
- **Direct Feedback:** During training, the algorithm receives direct feedback on its predictions (the "correct answer" is provided), allowing it to adjust its internal parameters to minimize errors.
- **Clear Objectives:** The problems solved by supervised learning typically have well-defined objectives, such as predicting a value or classifying an item.

Types of Supervised Learning Problems:

- **Classification:** The output variable is a category or a class (e.g., "spam" or "not spam," "dog" or "cat," "malignant" or "benign").
- **Regression:** The output variable is a continuous numerical value (e.g., house price, temperature, stock price).

Examples:

- **Email Spam Detection:**
  - **Data:** Emails labeled as "spam" or "not spam."
  - **Process:** The model learns features (e.g., specific words, sender addresses, links) that differentiate spam from legitimate emails.
  - **Output:** Predicts whether a new incoming email is spam or not.
- **Predicting House Prices:**
  - **Data:** Historical data of houses including features like square footage, number of bedrooms, location, and their corresponding sale prices.
  - **Process:** The model learns the relationship between house features and their prices.
  - **Output:** Predicts the likely sale price of a new house based on its features.
- **Medical Diagnosis:**
  - **Data:** Patient symptoms, test results, and confirmed diagnoses (e.g., "has disease A" or "does not have disease A").
  - **Process:** The model learns to associate specific symptoms and test results with particular diseases.
  - **Output:** Suggests a diagnosis for a new patient based on their symptoms and test results.

Unsupervised Learning

Unsupervised learning algorithms work with unlabeled data. There are no predefined output labels or "correct answers" provided during training. Instead, the algorithm's goal is to discover hidden patterns, structures, relationships, or groupings within the data on its own.

Key Characteristics:

- **Unlabeled Data:** Uses data without any predefined target variables or labels.
- **Exploratory:** The primary goal is to explore the intrinsic structure of the data, discover hidden insights, and simplify complex datasets.
- **No Direct Feedback:** The algorithm works independently, without external guidance, to find inherent patterns.
- **Discovering Structure:** The problems solved often involve finding natural groupings, anomalies, or simplifying high-dimensional data.

Types of Unsupervised Learning Problems:

- **Clustering:** Grouping similar data points together based on their inherent characteristics.
- **Association Rule Mining:** Discovering relationships or dependencies between variables in large datasets (e.g., "customers who bought X also bought Y").
- **Dimensionality Reduction:** Reducing the number of features or variables in a dataset while retaining as much important information as possible.

Examples:

- **Customer Segmentation:**
  - **Data:** Unlabeled customer purchase history, Browse behavior, demographics.
  - **Process:** The algorithm identifies natural groups of customers who exhibit similar behaviors or characteristics.
  - **Output:** Segments customers into distinct groups (e.g., "high-value shoppers," "occasional buyers," "bargain hunters") for targeted marketing, without prior knowledge of these segments.
- **Anomaly Detection (Fraud Detection):**
  - **Data:** Unlabeled transaction data (normal transactions).
  - **Process:** The algorithm learns the "normal" patterns of transactions.
  - **Output:** Flags transactions that deviate significantly from the learned normal patterns as potential fraud, even if it has never seen a "fraudulent" label.
- **News Article Categorization:**
  - **Data:** A large collection of unlabeled news articles.
  - **Process:** The algorithm analyzes the text content to identify common themes and keywords.
  - **Output:** Groups articles into clusters that represent different topics (e.g., "Sports," "Politics," "Technology") without being explicitly told what those topics are.

Comparison Summary:

Feature	Supervised Learning	Unsupervised Learning
Data Type	Labeled data (input-output pairs)	Unlabeled data
Goal	Predict outcomes, classify data	Discover hidden patterns, structures, groupings, anomalies
Feedback	Direct feedback (correct answers provided)	No direct feedback; finds inherent structure

Problem Type	Regression, Classification	Clustering, Association, Dimensionality Reduction
Complexity	Generally less complex to evaluate (known output)	Can be more complex to interpret and validate results
Human Effort	High for data labeling	Lower for data labeling, higher for result interpretation
Applications	Spam detection, image recognition, price prediction, medical diagnosis	Customer segmentation, anomaly detection, recommendation systems, data exploration

---

2. Write a Python script to demonstrate classification using decision trees in Scikit-Learn

- Decision trees are intuitive non-parametric algorithms used for classification. They operate by asking a series of questions about the data's features, designed to progressively narrow down and classify objects.
- In machine learning, these questions typically translate into "axis-aligned splits" within the data, where each step divides the dataset into two groups based on a cutoff value for one of the features.
- This binary splitting approach is highly efficient, as each well-chosen question can significantly reduce the number of options, quickly leading to a classification even among many classes.

Example Code

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import seaborn as sns; sns.set()
        from sklearn.datasets import make_blobs
        from sklearn.tree import DecisionTreeClassifier
```

Explanation of Example Code

- **import numpy as np, matplotlib.pyplot as plt, seaborn as sns; sns.set():**
  - These lines **import** core Python libraries for scientific computing (NumPy), plotting (Matplotlib), and enhancing plot aesthetics (Seaborn). They are standard for data science tasks in Python.
- **from sklearn.datasets import make_blobs:**
  - This line **imports** `make_blobs`, a utility function from Scikit-Learn. It is used to **generate** synthetic datasets with distinct clusters, which are ideal for demonstrating classification algorithms.
- **from sklearn.tree import DecisionTreeClassifier:**
  - This line **imports** the `DecisionTreeClassifier` class, which is Scikit-Learn's implementation of the decision tree algorithm specifically for classification tasks.

Next, we prepare a two-dimensional dataset with multiple class labels to demonstrate the decision tree's classification capabilities.

Example Code

```
In [2]: X, y = make_blobs(n_samples=300, centers=4,
                          random_state=0, cluster_std=1.0)
          plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow');
```

Figure 5-68. Data for the decision tree classifier

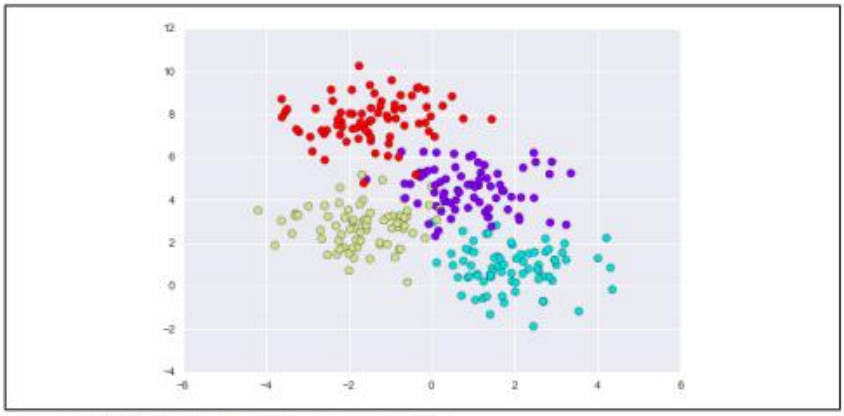

Figure 5-68. Data for the decision tree classifier

Explanation of Example Code

- **X, y = make_blobs(n_samples=300, centers=4, random_state=0, cluster_std=1.0):**
  - This line **generates** a synthetic dataset with 300 data points.
  - `X` **represents** the feature matrix (2D coordinates of each point).
  - `y` **represents** the corresponding target labels (class assignments for each point), with `centers=4` indicating four distinct classes.
  - `random_state=0` **ensures** the reproducibility of the generated data, meaning the same dataset will be created every time this code is run.
- **plt.scatter(X[:, 0], X[:, 1], c=y, s=50, cmap='rainbow'):**
  - This line **creates** a scatter plot to **visualize** the generated data.
  - `X[:, 0]` and `X[:, 1]` **are used** as the x and y coordinates respectively.
  - `c=y` **colors** each point according to its class label, allowing for clear visual distinction between the four generated clusters.

We proceed to train a Decision Tree Classifier on this prepared data.

Example Code

```
In [3]: tree = DecisionTreeClassifier().fit(X, y)
```

Explanation of Example Code

- **tree = DecisionTreeClassifier().fit(X, y):**
  - This line **instantiates** a `DecisionTreeClassifier` object.
  - The `.fit(X, y)` method **trains** the decision tree model. During this process, the algorithm **learns** a series of optimal axis-aligned splits and corresponding cutoff values based on the input features `X` and their true class labels `y`, aiming to correctly classify the training data. The resulting `tree` object holds the learned decision rules.

To effectively visualize the classification boundaries determined by our trained Decision Tree, we define a helper function.

Example Code

```
In [4]: def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):
            ax = ax or plt.gca()
            # Plot the training points
            ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,
                       clim=(y.min(), y.max()), zorder=3)
            ax.axis('tight')
            ax.axis('off')
            xlim = ax.get_xlim()
            ylim = ax.get_ylim()
            # fit the estimator
            model.fit(X, y)
            xx, yy = np.meshgrid(np.linspace(*xlim, num=200),
                                 np.linspace(*ylim, num=200))
            Z = model.predict(np.c_[xx.ravel(), yy.ravel()]).reshape(xx.shape)
            # Create a color plot with the results
            n_classes = len(np.unique(y))
            contours = ax.contourf(xx, yy, Z, alpha=0.3,
                                   levels=np.arange(n_classes + 1) - 0.5,
                                   cmap=cmap, clim=(y.min(), y.max()),
                                   zorder=1)
            ax.set(xlim=xlim, ylim=ylim)
```

Explanation of Example Code

- **def visualize_classifier(model, X, y, ax=None, cmap='rainbow'):**
  - This line **defines** a reusable utility function designed to **plot** the decision boundaries of any given classifier model.
  - It **takes** the trained `model`, the feature data `X`, and the target labels `y` as primary inputs.
- **ax.scatter(X[:, 0], X[:, 1], c=y, ...):**
  - Within the function, this line **plots** the original `X` data points on the scatter plot.
  - The points **are colored** according to their true labels (`y`), serving as a visual reference of the input data distribution.
- **model.fit(X, y); xx, yy = np.meshgrid(...); Z = model.predict(...).reshape(xx.shape):**
  - The `model` **is fitted** to the data again (if not already fitted) for consistency within the visualization.
  - A dense grid of points `(xx, yy)` covering the plot area **is created**.
  - The `model.predict()` method **is then used** to obtain a predicted class label `Z` for every point on this grid. This `Z` array effectively maps each region of the plot to the class that the model would assign to it.
- **contours = ax.contourf(xx, yy, Z, ...):**
  - This line **generates** a filled contour plot.
  - It **uses** the `Z` array (predicted classes on the grid) to **draw** distinct, color-filled regions. These regions **represent** the learned decision boundaries of the classifier, illustrating how the model partitions the feature space for classification.

Finally, we call our visualization function to display the results of the Decision Tree classification.

Example Code

Python

```
In [5]: visualize_classifier(DecisionTreeClassifier(), X, y)
```

Figure 5-70. Visualization of a decision tree classification

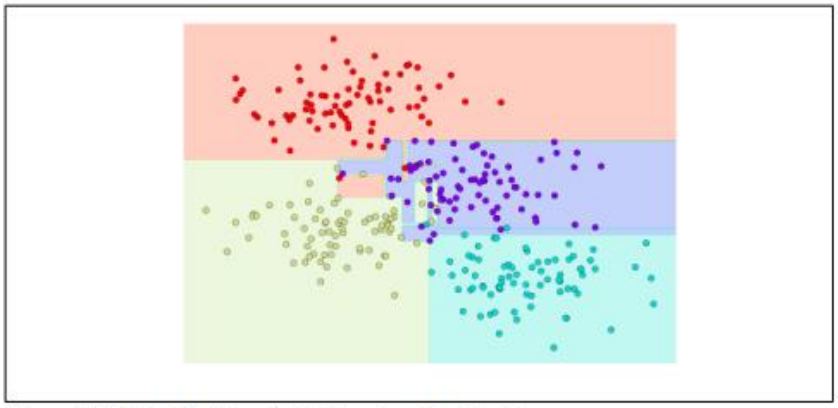

Figure 5-70. Visualization of a decision tree classification

Explanation of Example Code

- **visualize_classifier(DecisionTreeClassifier(), X, y):**
  - This line **executes** the `visualize_classifier` function, **passing** a new instance of `DecisionTreeClassifier` and the dataset `X`, `y`.
  - The function then **generates** and **displays** a plot. This plot **shows** the original data points (colored by their true class) and, crucially, **overlays** the classification regions that the `DecisionTreeClassifier` has learned. These regions visually **demonstrate** how the decision tree segments the 2D feature space to classify points into one of the four classes.

---

3. **How is model validation performed in Scikit-Learn?**

Model validation is the process of evaluating a machine learning model's performance on unseen data. This is critical to ensure the model generalizes well and to avoid overfitting. Scikit-Learn provides essential tools for this, primarily through holdout sets and cross-validation techniques.

Model Validation the Wrong Way (Naive Approach)

A flawed approach to model validation is to train and evaluate the model on the same data. This leads to an overly optimistic and misleading accuracy score.

Example Code

```
In [1]: from sklearn.datasets import load_iris # Import Iris dataset loader
        iris = load_iris()                    # Load the Iris dataset
        X = iris.data                         # Features (e.g., sepal length, petal width)
        y = iris.target                       # Target labels (species)
```

Explanation of Example Code

- This block **loads** the classic Iris dataset, separating its features (`X`) from its corresponding class labels (`y`).

Next, we use a simple K-Neighbors Classifier, train it on the entire dataset, and predict on the same data.

Python

```
In [2]: from sklearn.neighbors import KNeighborsClassifier # Import K-Neighbors Classifier
        model = KNeighborsClassifier(n_neighbors=1)     # Initialize model with 1 neighbor
In [3]: model.fit(X, y)                                   # Train the model on ALL data (X, y)
        y_model = model.predict(X)                      # Predict labels on the SAME training
data
```

Explanation of Example Code

- A `KNeighborsClassifier` **is initialized** and **trained** on the complete dataset.
- It then **makes predictions** using the very same data it was trained on.

Finally, we compute the accuracy.

Example Code

```
In [4]: from sklearn.metrics import accuracy_score # Import accuracy score function
        accuracy_score(y, y_model)           # Compare true labels (y) to predictions (y_model)
Out[4]: 1.0
```

Explanation of Example Code

- The `accuracy_score` **is calculated**. The result of `1.0` (100% accuracy) demonstrates the flaw: training and testing on the same data gives a misleadingly perfect score, especially for an instance-based model like K-Neighbors.

Model Validation the Right Way: Holdout Sets

A correct approach uses a **holdout set**: a portion of the data kept separate from training and used solely for evaluation on unseen data.

Example Code

```
In [5]: from sklearn.model_selection import train_test_split # Import data splitting utility

        # Split data: 50% for training (X1, y1), 50% for testing (X2, y2)
        X1, X2, y1, y2 = train_test_split(X, y, random_state=0, train_size=0.5)

        model.fit(X1, y1)      # Train model ONLY on the training subset
        y2_model = model.predict(X2) # Predict labels on the unseen test subset
        accuracy_score(y2, y2_model) # Calculate accuracy on the test set
Out[5]: 0.90666666666666662
```

Explanation of Example Code

- The dataset **is split** into training (`X1, y1`) and testing (`X2, y2`) subsets.
- The model **is trained** exclusively on the training subset and then **evaluated** on the completely unseen test subset. This yields a more realistic accuracy, demonstrating the model's ability to generalize.

Model Validation via Cross-Validation

Cross-validation is a more robust method that overcomes the data-loss disadvantage of a single holdout set. It performs multiple train-test splits systematically.

Example Code

```
In [6]: from sklearn.model_selection import cross_val_score # Import cross-validation scoring
utility

           # Perform 5-fold cross-validation on the model and full dataset
        cross_val_score(model, X, y, cv=5)
Out[6]: array([ 0.96666667,  0.96666667,  0.93333333,  0.93333333,  1.        ])
```

Explanation of Example Code

- `cross_val_score` **automates** 5-fold cross-validation. The data is divided into 5 "folds." In each iteration, 4 folds **are used** for training and 1 for validation.
- The output is an array of 5 accuracy scores, one for each validation fold, providing a more reliable estimate of performance.

An extreme form of cross-validation is Leave-One-Out (LOO), where each data point serves as a test set once.

Example Code

```
In [7]: from sklearn.model_selection import LeaveOneOut # Import Leave-One-Out CV strategy

           # Perform Leave-One-Out Cross-Validation
        scores = cross_val_score(model, X, y, cv=LeaveOneOut(len(X)))
        scores.mean() # Calculate the average accuracy across all LOO trials
Out[7]: 0.95999999999999996
```

Explanation of Example Code

- `LeaveOneOut` **configures** the cross-validation to use one sample for testing and all remaining samples for training, repeated for every sample.
- The `mean()` of the resulting `scores` array **provides** a very thorough average accuracy estimate from 150 (number of samples) trials.

---

4. Explain the process of splitting datasets into training and testing sets

Splitting a dataset into training and testing sets is a fundamental step in machine learning model validation. This process involves partitioning the available data into two distinct subsets:

- **Training Set:** Used to train (fit) the machine learning model. The model learns patterns and relationships from this data.
- **Testing Set (or Validation Set):** Used to evaluate the trained model's performance on unseen data. This set assesses how well the model generalizes to new, real-world examples, providing an unbiased estimate of its accuracy and avoiding overfitting.

This separation ensures that the model's evaluation is not biased by the data it has already learned from.

Example Code

```
In [1]:    import numpy as np
           from sklearn.datasets import load_iris              # Import a sample dataset
           from sklearn.model_selection import train_test_split # Import the splitting function

           # Load the Iris dataset
           iris = load_iris()
           X = iris.data  # Features
           y = iris.target # Target labels

           # Split the data into training and testing sets
           # test_size=0.3 means 30% of data for testing, 70% for training
           # random_state ensures reproducibility of the split
           # stratify=y ensures equal proportion of classes in train/test sets
           X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                               test_size=0.3,
                                                               random_state=42,
                                                               stratify=y)

           # Print the shapes of the resulting sets
           print(f"Shape of X_train: {X_train.shape}")
           print(f"Shape of X_test: {X_test.shape}")
           print(f"Shape of y_train: {y_train.shape}")
           print(f"Shape of y_test: {y_test.shape}")

Out[1]: Shape of X_train: (105, 4) Shape of X_test: (45, 4) Shape of y_train: (105,) Shape of
y_test: (45,)
```

Explanation of Example Code

- **from sklearn.model_selection import train_test_split:**
  - This line **imports** the `train_test_split` function from Scikit-Learn, which is the primary tool for performing this dataset division.
- **iris = load_iris(); X = iris.data; y = iris.target:**
  - This segment **loads** the Iris dataset, a common dataset for classification. `X` **stores** the features (e.g., sepal/petal measurements), and `y` **stores** the corresponding species labels.
- **X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y):**
  - This line **executes** the dataset splitting.
  - `X` and `y` **are the input** features and labels.
  - `test_size=0.3` **specifies** that 30% of the data will be allocated to the testing set, with the remaining 70% for training.
  - `random_state=42` **sets** a seed for the random number generator, **ensuring** that the split is consistent and reproducible every time the code runs.
  - `stratify=y` **ensures** that the proportion of each class label (`y`) is approximately the same in both the training and testing sets. This is crucial for maintaining representative subsets, especially with imbalanced datasets.
  - The function **returns** four new arrays:
    - `X_train`: Features for training.
    - `X_test`: Features for testing.
    - `y_train`: Labels for training.
    - `y_test`: Labels for testing.
- **print(f"Shape of X_train: {X_train.shape}") ...:**

- These lines **print** the dimensions (shapes) of the resulting arrays, **confirming** how the data has been divided. For the Iris dataset (150 samples), a 30% test size means 45 samples for testing and 105 for training.

---

5. Write a Python program to demonstrate the use of Scikit-Learn for simple linear regression.

Simple Linear Regression is a statistical method that models the relationship between a dependent variable (y) and an independent variable (x) as a straight line. The model takes the form y=ax+b, where 'a' is the slope of the line and 'b' is the y-intercept. Scikit-Learn's `LinearRegression` estimator provides a powerful and easy-to-use interface for fitting such models to data.

Example Code

```
In [1]: import numpy as np                    # Import NumPy for numerical operations
            import matplotlib.pyplot as plt      # Import Matplotlib for plotting
            import seaborn as sns; sns.set()    # Import Seaborn for enhanced plot aesthetics

            # Set a random seed for reproducibility
            rng = np.random.RandomState(1)

            # Generate synthetic data for linear regression
            x = 10 * rng.rand(50)              # 50 random x-values between 0 and 10
            y = 2 * x - 5 + rng.randn(50)      # y-values based on y = 2x - 5, with added noise

            # Plot the generated scattered data
            plt.scatter(x, y)
```

Figure 5-42. Data for linear regression

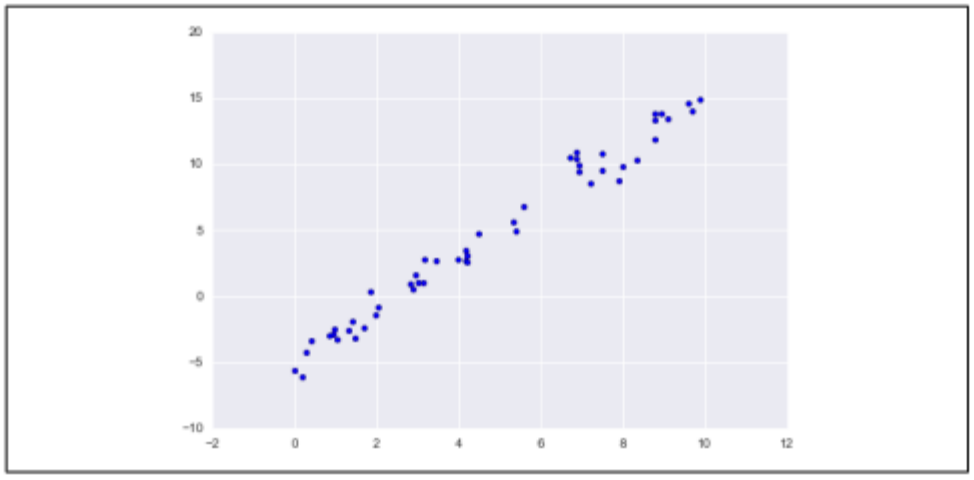

Figure 5-42. Data for linear regression

Explanation of Example Code

- **import numpy as np, matplotlib.pyplot as plt, seaborn as sns; sns.set():**

    - These lines **import** the necessary libraries for numerical operations (NumPy), plotting (Matplotlib), and aesthetic enhancements for plots (Seaborn).
- **rng = np.random.RandomState(1):**
    - This line **initializes** a random number generator with a fixed seed. This **ensures** that the synthetic data generated will be the same every time the code is run, making the demonstration reproducible.
- **x = 10 * rng.rand(50); y = 2 * x - 5 + rng.randn(50):**
    - These lines **generate** the synthetic dataset.
    - `x` **consists** of 50 random numbers uniformly distributed between 0 and 10.
    - `y` **is calculated** based on the linear equation $y=2x-5$, with `rng.randn(50)` adding random Gaussian noise to simulate real-world data imperfections.
- **plt.scatter(x, y):**
    - This line **creates** a scatter plot of the generated `x` and `y` data points. This **visualizes** the linear relationship with added noise, which the regression model will attempt to fit.

Next, we use Scikit-Learn's `LinearRegression` estimator to fit a straight line to this data and then visualize the best-fit line.

Example Code

```
In [2]: from sklearn.linear_model import LinearRegression # Import the LinearRegression model

        model = LinearRegression(fit_intercept=True) # Initialize the model;
                                                     # fit_intercept=True by default

        # Reshape x to a 2D array required by Scikit-Learn's fit method (n_samples, n_features)
        model.fit(x[:, np.newaxis], y)

        # Generate x-values for the fitted line
        xfit = np.linspace(0, 10, 1000)

        # Predict y-values for the fitted line using the trained model
        yfit = model.predict(xfit[:, np.newaxis])

        # Plot the original scattered data
        plt.scatter(x, y)

        # Plot the best-fit regression line
        plt.plot(xfit, yfit, color='red'); # Added color for clarity
```

Figure 5-43. A linear regression model

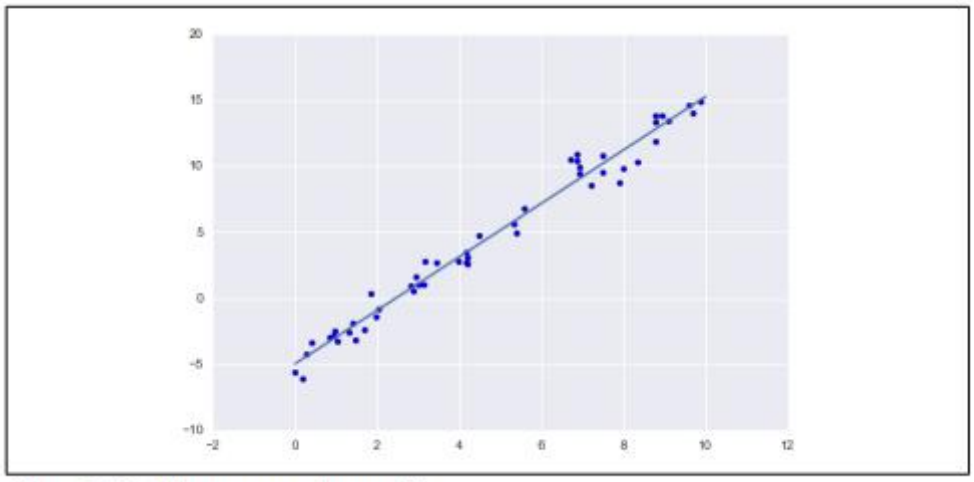

Figure 5-43. A linear regression model

Explanation of Example Code

- **from sklearn.linear_model import LinearRegression:**
  - This line **imports** the `LinearRegression` class from Scikit-Learn's `linear_model` module, which **implements** ordinary least squares Linear Regression.
- **model = LinearRegression(fit_intercept=True):**
  - An instance of the `LinearRegression` model **is created**. `fit_intercept=True` (which is the default) **instructs** the model to calculate an intercept for the regression line.
- **model.fit(x[:, np.newaxis], y):**
  - This is the core training step. The `fit()` method **learns** the optimal slope (a) and intercept (b) that best describe the linear relationship between `x` and `y`.
  - `x[:, np.newaxis]` **reshapes** the 1D `x` array into a 2D array (a column vector) as required by Scikit-Learn's convention for feature matrices (number of samples, number of features).
- **xfit = np.linspace(0, 10, 1000); yfit = model.predict(xfit[:, np.newaxis]):**
  - `xfit` **creates** 1000 evenly spaced points across the range of `x`.
  - `model.predict()` **uses** the trained model to **calculate** the corresponding predicted `y` values (`yfit`) for these `xfit` points, effectively generating the best-fit line.
- **plt.scatter(x, y); plt.plot(xfit, yfit, color='red'):**
  - These lines **plot** the original scattered data points and then **overlay** the straight line (`yfit` versus `xfit`) that the `LinearRegression` model has fitted to the data, visually demonstrating the regression.

After fitting the model, its learned parameters (slope and intercept) can be accessed through its attributes.

Example Code

```
In [3]: print("Model slope: ", model.coef_[0])     # Access the learned slope (coefficient)
           print("Model intercept:", model.intercept_) # Access the learned intercept

Out[3]: Model slope: 2.02720881036 Model intercept: -4.99857708555
```

Explanation of Example Code

- **print("Model slope: ", model.coef_[0]):**

    - `model.coef_` **stores** the learned coefficients (slopes) of the linear model. For simple linear regression (one feature), it returns an array with a single value, so `[0]` **accesses** that specific slope. The printed value is very close to the true slope of 2 used in data generation.
- **print("Model intercept:", model.intercept_):**
    - `model.intercept_` **stores** the learned y-intercept of the linear model. The printed value is very close to the true intercept of -5 used in data generation.
