CBCS SCHEME

USN 1 C R 2 1 M C 0 6 9

20MCA31

Third Semester MCA Degree Examination, Jan./Feb. 2023
Data Analytics using Python

Time: 3 hrs. Max. Marks: 100

Note: Answer any FIVE full questions, choosing ONE full question from each module.

Module-1

1 a. Define keywords, statements, expressions, variables, precedence and associativity with examples and syntax. (10 Marks)

b. Explain with syntax and example different types of Python data types and type( ) function. (10 Marks)

OR

2 a. Discuss different forms of if control statements with necessary examples. (10 Marks)

b. What is a function? Mention its types. Write a python program to add two numbers using function, read input from the user. (10 Marks)

Module-2

3 a. Define string. Explain with necessary coding five basic string operations. Explain string slicing and joining. (10 Marks)

b. Explain List creation, indexing and built in functions used on lists with syntax and examples. (10 Marks)

OR

4 a. Differentiate between sets, tuples and dictionaries. Write a python program to demonstrate encapsulation and overloading. (10 Marks)

b. What is inheritance? Explain different types of inheritance with necessary example. (10 Marks)

Module-3

5 a. Define creating an array from Python lists. Explain numpy array attributes. (06 Marks)

b. Discuss with example numpy array concatenation and splitting. (08 Marks)

c. Explain specialized universal functions:
(i) Trignometric (ii) Exponents and logarithms with necessary coding. (06 Marks)

OR

6 a. Mention Pandas data structures. Create a dataframe with three dimensional list state, year, POP (dictionary). Write necessary coding for retrieving row values and modifying column values. (06 Marks)

b. Explain with example the concept of reindexing and ffill method. (06 Marks)

c. How do we handle missing data in Python using Pandas? Explain with coding. (08 Marks)

Module-4

7 a. Explain reading and writing data in text format in Python with examples. (10 Marks)

b. Explain the following methods with respect to database interaction:
i) Create ii) insert iii) connect iv) execute v) fetch all. (10 Marks)

CR - CR - CR - VTU 27-03-2023 01:33:09pm

OR

8 a. Explain with example the following merge methods:
i) inner ii) left iii) right
Create two dataframes with the following :

df1:

data1	key
0	b
1	b
2	a
3	c
4	a
5	b

df2:

data2	key
0	a
1	b
2	a
3	b
4	d

(10 Marks)

b. Explain Data transforming using a function or mapping. Create a dataframe with the following columns:

Food	Ounce
Bacon	4.0
Pulled pork	3.0
Bacon	12.0
Honeyham	5.0

Add a column indicating the type of animal that each food come from. (10 Marks)

Module-5

9 a. Write a Python program to plot sinusoid and cosine waves using Matplotlib and label them with necessary title and labels. (10 Marks)

b. Explain with necessary coding creating a basic error bars and continuous errors. (10 Marks)

OR

10 a. Write a Python program to plot the histogram as follows (customized histogram). [Refer Fig.Q10(a)].

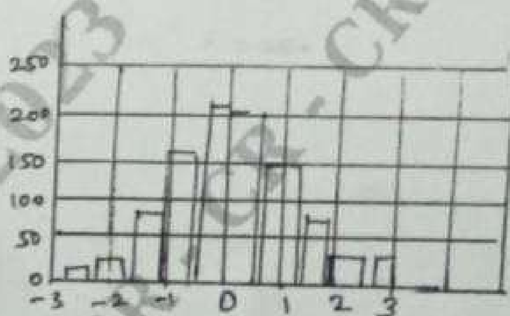

Fig.Q10(a) (10 Marks)

b. What is Seaborn plot? Explain pair plots for 'iris' dataset and kernel density estimation using kdeplot and displot. (10 Marks)

*****

VTU-27-03-2023 01:11:20pm - CR - CR - CR - CR
27-03-2023 01:33:09pm - CR - CR - CR - CR

Third Semester MCA Degree Examination . Jan/Feb 2023

Data Analytics Using Python(20MCA31)
===

(1) Define Keywords, statement, expressions, variables, precedence and associativity with examples and syntax

Keywords:

Keywords in Python are reserved words that have predefined meanings and are part of the language's sy ntax. They cannot be used as identifiers for variables, functions, or other user-defined names. Keywords are essential for defining the structure and behavior of Python programs. Here are some common Python keywords:

Example (1) :-

```
# Using 'if' as a keyword to create a conditional statement
if condition:
    print("This is a conditional statement.")
```

Example (2) :-
def: Used to define functions in Python, allowing you to encapsulate a block of code for reuse.

```
def greet(name):
    print(f"Hello, {name}!")

greet("Alice")
```

=====>
Statements:

Statements in Python are complete instructions or commands that perform actions or tasks. They are typi cally terminated by a newline character, but semicolons can also be used to separate multiple statements on a single line (although this is not a common practice in Python). Python supports several types of state ments:

Assignment Statements: These statements assign values to variables.
such as :-

```
x = 10
```

Conditional Statements (if, elif, else): These statements allow you to execute different code blocks based on specific conditions.

```
if x > 10:
    print("x is greater than 10")
elif x == 10:
    print("x is equal to 10")
else:
    print("x is less than 10")
```

=====>

Expressions:

Expressions in Python are combinations of values, variables, operators, and function calls that can be eva luated to produce a result. Expressions can be simple, like a single value or variable, or complex, involvin g multiple operations. Here are some examples of expressions:

Arithmetic Expressions: These involve arithmetic operators (+, -, *, /, etc.) and numeric values or variables .

Arithmetic Expressions: These involve arithmetic operators (+, -, *, /, etc.) and numeric values or variables .

```
result = 2 * (x + 3)
```

String Expressions: Combining strings using the + operator.

```
full_name = first_name + " " + last_name
```

=====>

Variables:

Variables in Python are used to store and manage data. They are like containers that hold values, and yo u can give them names for easy reference. To create a variable, you simply assign a value to a name usi ng the assignment operator (=). Variables can hold various types of data, including numbers, strings, lists, and more.

example:-

```
x = 10         # Assigning an integer value to the variable ’x’
name = "Alice" # Assigning a string value to the variable ’name’
my_list = [1, 2, 3] # Assigning a list to the variable ’my_list’
```

Variables are essential in programming because they allow you to store and manipulate data dynamically. You can update the value stored in a variable, and that change will be reflected in your program.

Precedence:

Precedence in Python refers to the order in which operators are evaluated in expressions. Operators with higher precedence are evaluated first, followed by those with lower precedence. Understanding operator precedence is crucial because it determines the correct order of operations in complex expressions. Pyth on’s operator precedence is defined in the language’s grammar rules.

Parentheses (): Highest precedence. Operations enclosed in parentheses are evaluated first.

example:-

```
result = (5 + 3) * 2 # Parentheses force addition before multiplication
```

========>

Associativity:

Associativity in Python determines the order in which operators of the same precedence are evaluated wh en they appear consecutively in an expression. Most operators in Python have left-to-right associativity, m eaning they are evaluated from left to right. Here are some examples to illustrate associativity:

example:-

Addition and Subtraction:
result = 5 - 3 + 2

======>
==>
(2) Explain with syntax and example different types of Python data types and type() function

Understanding data types is essential for data analytics using Python as it helps you work with and manip ulate data effectively.

Python supports several built-in data types. Here are some of the most common ones:

(*) Integer (int):

Integers are whole numbers without a decimal point.
They can be positive or negative.
Syntax:
x = 5

(*) Float (float):

Floats are numbers with a decimal point or in exponential form.
They can also be positive or negative.
Syntax:
y = 3.14

String :

Strings are sequences of characters enclosed in single, double, or triple quotes.
They are used to represent text data.
Syntax:
name = "Alice"

Boolean (bool):

Booleans represent either True or False.
They are often used in conditional statements and comparisons.
Syntax:
is_raining = True

List (list):

Lists are ordered collections of items, separated by commas, enclosed in square brackets [ ].
They can hold elements of different data types.
Syntax:

my_list = [1, 2, 3, "apple", True]

Tuple (tuple):

Tuples are similar to lists but enclosed in parentheses ( ).
They are immutable, meaning their elements cannot be changed once defined.

```
my_tuple = (1, 2, 3, "banana")
```

Dictionary (dict):

Dictionaries are unordered collections of key-value pairs enclosed in curly braces { }.
They are used for mapping keys to values.
Syntax:

```
my_dict = {"name": "John", "age": 30, "city": "New York"}
```

(*) Set (set):

Sets are unordered collections of unique elements enclosed in curly braces { }.
They are useful for performing mathematical set operations like union and intersection.
Syntax:

```
my_set = {1, 2, 3, 4, 5}
```

The type() function is used to determine the data type of a variable or an expression.
It returns the type of the object passed as an argument.
Syntax:

```
data_type = type(variable)
```

example :-

```
age = 30
data_type = type(age)
print(data_type) # Output: <class ’int’>
```

In this example, the type() function is used to determine that the age variable is of type int (integer).

Here are examples of using the type() function with different data types:

```
x = 5
print(type(x)) # Output: <class ’int’>

y = 3.14
print(type(y)) # Output: <class ’float’>

name = "Alice"
print(type(name)) # Output: <class ’str’>

is_raining = True
print(type(is_raining)) # Output: <class ’bool’>
```

===>
2(a) Discuss Different forms of if control statements with necessary examples.

The "if" control statement is a fundamental construct in programming that allows you to make decisions b

ased on conditions. In Python, you can use "if," "elif" (else if), and "else" to create various forms of conditi onal statements.

Simple "if" Statement:

The simple "if" statement is the most basic form of conditional statement. It allows you to execute a block of code if a specified condition is true.

Syntax:

```
if condition:
    # Code to execute if the condition is true
```
Example:
```
x = 10
if x > 5:
    print("x is greater than 5")
```
In this example, the code inside the "if" block is executed because the condition x > 5 is true.

"if" and "else" Statement:

The "if" and "else" statement is used to execute one block of code if a condition is true and another block i f the condition is false.

Syntax:

```
if condition:
    # Code to execute if the condition is true
else:
    # Code to execute if the condition is false
```
Example:
```
age = 17
if age >= 18:
    print("You are an adult")
else:
    print("You are not an adult")
```
Here, the output depends on the value of age, and the appropriate message is printed based on whether t he condition is true or false.

"if," "elif," and "else" Statement:

The "if," "elif" (else if), and "else" statement allows you to test multiple conditions and execute different blo cks of code based on which condition is true. You can have multiple "elif" blocks in addition to the initial "if " and "else."

Syntax:

```
if condition1:
    # Code to execute if condition1 is true
elif condition2:
    # Code to execute if condition2 is true
else:
    # Code to execute if neither condition1 nor condition2 is true
```
Example:

```
score = 75
```

```
if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
else:
    grade = 'D'
print(f"Your grade is {grade}")
```

In this example, the code evaluates the value of score and assigns a grade based on the score range.

Nested "if" Statements:

You can nest "if" statements inside other "if," "elif," or "else" statements to create more complex condition al logic.

Syntax:

```
if condition1:
    if condition2:
        # Code to execute if both condition1 and condition2 are true
    else:
        # Code to execute if condition1 is true but condition2 is false
else:
    # Code to execute if condition1 is false
```

Example:

```
age = 25
if age >= 18:
    if age < 21:
        print("You are an adult but not old enough to drink")
    else:
        print("You are an adult and can drink")
else:
    print("You are not an adult")
```

This nested "if" statement checks both age conditions to determine whether someone can drink based on age.

"if" Statement with Logical Operators:

We can use logical operators (e.g., and, or, not) to combine multiple conditions in a single "if" statement.

Syntax:

```
if condition1 and condition2:
    # Code to execute if both condition1 and condition2 are true
```

Example:

```
temperature = 28
time_of_day = "morning"
if temperature > 25 and time_of_day == "morning":
    print("It's a hot morning")
```

In this example, both conditions must be true for the code inside the "if" block to execute.

"if" Statement with Membership Operators:

You can use membership operators (in and not in) to check if an element is present in a sequence, such as a list or a string.

Syntax:

```
if element in sequence:
    # Code to execute if the element is in the sequence
```

Example:

```
fruits = ["apple", "banana", "cherry"]
if "banana" in fruits:
    print("Banana is in the list of fruits")
```

Here, the code checks if "banana" is in the list of fruits before printing the message.

"if" Statement with Multiple Conditions:

You can combine multiple conditions using logical operators to create complex conditionals.

Syntax:

```
if condition1 and (condition2 or condition3):
    # Code to execute if condition1 is true and either condition2 or condition3 is true
```

Example:

```
temperature = 30
humidity = 60
if temperature > 25 and (humidity > 50 or temperature < 35):
    print("Conditions are favorable")
```

This example checks multiple conditions to determine if the conditions are favorable based on temperatur e and humidity.

"if" Statement with the Ternary Operator:

Python supports a concise way to write simple "if" statements known as the ternary operator.

Syntax:

```
result = value_if_true if condition else value_if_false
```

Example:

```
age = 17
status = "minor" if age < 18 else "adult"
print(f"You are a {status}")
```

In this example, the ternary operator assigns the value "minor" if the age is less than 18, and "adult" other wise.

These various forms of "if" control statements allow you to make decisions and control the flow of your Py thon programs based on different conditions. Depending on the complexity of your logic, you can choose t he most suitable form of the "if" statement to achieve your desired outcome in data analytics using Python .

==>

2(b) What is a function? Mention its types. write a python program to add two numbers using function, re ad input from the user.

A function is a self-contained block of code that performs a specific task or a set of related tasks. Function s are used to organize and modularize code, making it more manageable, reusable, and easier to underst and. In Python, functions are defined using the def keyword and can have parameters and return values. There are several types of functions in Python, including built-in functions, user-defined functions, and an onymous functions (lambda functions).

Built-in Functions:

Built-in functions are provided by the Python programming language and are available for use without the need for explicit definition.
Examples of built-in functions include print(), len(), type(), and input().

```
user_input = input("Enter a number: ")
print(f"You entered: {user_input}")
```

In this example, input() is a built-in function used to read user input, and print() is used to display the input .

User-Defined Functions:

User-defined functions are created by the programmer to perform specific tasks.
They are defined using the def keyword, followed by a function name, parameters (if any), and a colon.
The function body contains the code to be executed when the function is called.
User-defined functions are reusable and help in structuring code.
Syntax:

```
def function_name(parameters):
    # Function body
    # Code to perform the task
    return result # Optional
```

Example:

Anonymous Functions (Lambda Functions):

Lambda functions are small, anonymous functions defined using the lambda keyword.
They can take any number of arguments but can only have one expression.
Lambda functions are often used for simple operations and can be used where function objects are requir ed.
Syntax:

```
lambda arguments: expression
```

Example:

```
multiply = lambda x, y: x * y
result = multiply(3, 4)
print(f"The result of multiplication is {result}")
```

In this example, we define a lambda function multiply() that takes two arguments and returns their product . We then call the lambda function to multiply two numbers.

```
# Define a user-defined function to add two numbers
def add_numbers(x, y):
    result = x + y
    return result

# Read input from the user
num1 = float(input("Enter the first number: "))
num2 = float(input("Enter the second number: "))

# Call the user-defined function to add the numbers
sum_of_numbers = add_numbers(num1, num2)

# Display the result
print(f"The sum of {num1} and {num2} is {sum_of_numbers}")
```

.

==>

3(a) Define string, explain with necessary coding five basic string operations. Explain string slicing and joining.

A string is a data type in Python used to represent a sequence of characters. Strings are versatile and commonly used in data analytics for text data processing and manipulation. Python provides several built-in string operations to work with strings effectively. Let’s discuss the definition of strings, demonstrate five basic string operations, explain string slicing, and cover string joining.

Definition of String:
A string in Python is a sequence of characters enclosed within either single (’ ’), double (" "), or triple (’’’ ’’’ or """ """) quotes. Strings can contain letters, numbers, symbols, spaces, and even special characters. Here are some examples of strings:

```
name = "Alice"
```

sentence = "Hello, World!"
email = ’example@email.com’
multiline_text = ’’’This is a
multiline string.’’’
Now, let’s explore five basic string operations:

String Concatenation:

String concatenation is the process of combining two or more strings to create a new string.
It can be achieved using the + operator.

first_name = "John"
last_name = "Doe"
full_name = first_name + " " + last_name
In this example, the + operator is used to concatenate the first name and last name with a space in betwe
en.

String Length:

To find the length (number of characters) of a string, you can use the len() function.

text = "Hello, World!"
length = len(text)
The len() function returns the length of the string, which is 13 in this case.

String Indexing:

String indexing allows you to access individual characters in a string by their position (index).
Python uses 0-based indexing, meaning the first character has an index of 0.

text = "Hello, World!"
first_character = text[0] # Gets the first character ’H’
third_character = text[2] # Gets the third character ’l’
You can also use negative indexing to access characters from the end of the string, e.g., text[-1] returns t
he last character.

String Substring:

Substring extraction involves getting a portion of a string (a substring) from a larger string.
This can be done using string slicing.
String Slicing:

String slicing allows you to extract a portion of a string by specifying a range of indices.
The syntax for slicing is string[start:end], where start is the index of the first character to include, and end i
s the index of the first character to exclude.
If start is omitted, it defaults to 0, and if end is omitted, it defaults to the end of the string.

text = "Hello, World!"
substring = text[0:5] # Gets the substring "Hello"
In this example, we slice the string to extract the substring "Hello."

String Searching:

String searching involves finding the position of a specific substring within a string.
This can be done using the find() method.

```
text = "Hello, World!"
position = text.find("World") # Finds the position of "World" in the string
```
The find() method returns the starting index of the first occurrence of the specified substring. If the substring is not found, it returns -1.

String Joining:
String joining is the process of combining a sequence of strings into a single string. This is often useful when you have a list of strings that you want to concatenate into one. Python provides the join() method for this purpose.

Syntax:

```
separator = "separator_string"
joined_string = separator.join(iterable)
```
separator is the string used to join the elements in the iterable.
iterable is a sequence (e.g., list, tuple) of strings that you want to join.
Example:

```
words = ["Hello", "World", "Python"]
separator = " "
joined_string = separator.join(words)
```
In this example, we join the words in the list with a space as the separator, resulting in the string "Hello World Python".

String operations are essential in data analytics for tasks such as cleaning and processing textual data, extracting information, and generating reports. Understanding basic string operations, including concatenation, length, indexing, substring extraction, and searching, is crucial for manipulating text data effectively. Additionally, string joining is valuable when you need to combine multiple strings into a single, well-structured text for analysis or reporting purposes.

==>
3(b) Explain List creation , indexing and built in functions used on lists with syntax and examples.

Lists are versatile data structures that allow you to store and manipulate collections of items. They are commonly used in data analytics for tasks like data storage, data preprocessing, and data analysis. Here, we'll cover list creation, indexing, and key built-in functions.

List Creation:
Lists in Python are created using square brackets [ ], and they can contain items of various data types, including numbers, strings, or even other lists. You can create an empty list or initialize a list with elements. Here are some examples:

Empty List:

```
empty_list = []
```
This creates an empty list called empty_list.

List with Elements:

```
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "cherry"]
mixed_data = [1, "hello", 3.14, True]
```
These examples demonstrate lists with different types of elements.

List Indexing:
List indexing is the process of accessing individual elements within a list. In Python, indexing is 0-based, meaning the first element has an index of 0, the second has an index of 1, and so on. You can also use n egative indexing, where -1 refers to the last element, -2 to the second-to-last element, and so forth. Here are some examples:

```
fruits = ["apple", "banana", "cherry"]
first_fruit = fruits[0] # Accesses the first element "apple"
second_fruit = fruits[1] # Accesses the second element "banana"
last_fruit = fruits[-1] # Accesses the last element "cherry"
```

Built-In Functions for Lists:

Python provides several built-in functions and methods to perform common operations on lists. Let’s expl ore some of these functions along with their syntax and examples.

len() - Get the Length of a List:

The len() function returns the number of elements in a list.
Syntax:

```
length = len(list)
```

Example:

```
numbers = [1, 2, 3, 4, 5]
length = len(numbers) # Returns 5
```

append() - Add an Item to the End of a List:

The append() method adds an element to the end of a list.
Syntax:

```
list.append(item)
```

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")
```

After the append() method, fruits will contain ["apple", "banana", "cherry", "orange"].

insert() - Insert an Item at a Specific Index:

The insert() method inserts an element at a specified index in the list.
Syntax:

```
list.insert(index, item)
```

Example:

```
numbers = [1, 2, 4, 5]
numbers.insert(2, 3) # Inserts 3 at index 2
```

After the insert() method, numbers will be [1, 2, 3, 4, 5].

remove() - Remove an Item by Value:

The remove() method removes the first occurrence of a specified value from the list.
Syntax:

```
list.remove(value)
```

Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.remove("banana")
```

After the remove() method, fruits will be ["apple", "cherry"].

pop() - Remove and Return an Item by Index:

The pop() method removes an item at a specified index and returns its value.
Syntax:

```
item = list.pop(index)
```

Example:

```
numbers = [1, 2, 3, 4, 5]
item = numbers.pop(2) # Removes and returns the item at index 2 (which is 3)
```

After the pop() method, numbers will be [1, 2, 4, 5], and item will be 3.

index() - Find the Index of an Item:

The index() method returns the index of the first occurrence of a specified item.
Syntax:

```
index = list.index(item)
```

Example:

```
fruits = ["apple", "banana", "cherry"]
index = fruits.index("banana") # Returns 1 (index of "banana")
```

count() - Count the Occurrences of an Item:

The count() method returns the number of times a specified item appears in the list.
Syntax:

```
count = list.count(item)
```

Example:

```
numbers = [1, 2, 2, 3, 4, 2]
count = numbers.count(2) # Returns 3 (2 appears three times)
```

sort() - Sort the List in Ascending Order:

The sort() method arranges the elements of a list in ascending order.
Syntax:

```
list.sort()
```

Example:

```
numbers = [4, 1, 3, 5, 2]
numbers.sort()
```

After the sort() method, numbers will be [1, 2, 3, 4, 5].

reverse() - Reverse the Order of Elements:

The reverse() method reverses the order of elements in the list.
Syntax:

```
list.reverse()
```
Example:

```
fruits = ["apple", "banana", "cherry"]
fruits.reverse()
```
After the reverse() method, fruits will be ["cherry", "banana", "apple"].

These are some of the most commonly used built-in functions and methods for lists in Python. They are e ssential tools for manipulating and working with data in a list format, making them valuable in data analyti cs tasks such as data preprocessing, analysis, and visualization. Understanding how to create, index, and use built-in functions with lists is a fundamental skill for anyone working with data in Python.
===>

4(a) Differentiate between sets, tuples and dictionaries. Write a python program to demonstrate encapsul ation and overloading.

let’s differentiate between sets, tuples, and dictionaries in Python and then discuss encapsulation and ove rloading along with a Python program to demonstrate them.

Differentiating Sets, Tuples, and Dictionaries:

Sets:

A set is an unordered collection of unique elements.
Elements in a set are enclosed within curly braces { } or can be created using the set() constructor.
Sets do not allow duplicate values, and they automatically remove duplicates.
Sets are primarily used for mathematical operations like union, intersection, and difference.
Example:

```
my_set = {1, 2, 3, 3, 4} # Creates a set with unique elements (1, 2, 3, 4)
```
Tuples:

A tuple is an ordered collection of elements, and it is immutable, meaning its elements cannot be changed after creation.
Tuples are created using parentheses ( ) or can be created without parentheses if the elements are comm a-separated.
They are often used for data that should not be modified, like coordinates, database records, and function return values.
Example:

```
my_tuple = (1, 2, 3)
another_tuple = 4, 5, 6
```
Dictionaries:

A dictionary is an unordered collection of key-value pairs.

Each key in a dictionary is unique, and it maps to a specific value.
Dictionaries are created using curly braces { } or the dict() constructor.
They are used to represent data in a structured and efficient manner.
Example:

```
my_dict = {"name": "Alice", "age": 30}
```

Now, let's discuss encapsulation and overloading:

Encapsulation:
Encapsulation is one of the four fundamental OOP (Object-Oriented Programming) principles, which inclu des encapsulation, inheritance, abstraction, and polymorphism. Encapsulation refers to the practice of bu ndling the data (attributes) and the methods (functions) that operate on the data into a single unit called a class. This concept allows you to hide the internal implementation details of an object and expose only th e necessary functionality. In Python, encapsulation is achieved through the use of classes and access mo difiers.

Overloading:
Function overloading refers to the ability to define multiple functions with the same name in a class but wit h different parameters. The choice of which function to call is determined by the number or type of argum ents passed to it. Python does not support traditional function overloading as seen in some other program ming languages, but it allows you to achieve similar behavior through default arguments and variable-leng th arguments.

Let's demonstrate encapsulation and overloading in a Python program:

```
class Calculator:
    def __init__(self):
        self.result = 0 # Initialize the result attribute to 0

    def add(self, *args):
        # Add multiple numbers
        for num in args:
            self.result += num

    def subtract(self, *args):
        # Subtract multiple numbers
        for num in args:
            self.result -= num

    def multiply(self, *args):
        # Multiply multiple numbers
        self.result = 1 # Initialize result to 1 for multiplication
        for num in args:
            self.result *= num

    def get_result(self):
        # Get the result
        return self.result

# Create an instance of the Calculator class
calculator = Calculator()

# Perform addition
calculator.add(5, 3, 2)
print("Result after addition:", calculator.get_result()) # Output: 10
```

```
# Perform subtraction
calculator.subtract(4, 1)
print("Result after subtraction:", calculator.get_result()) # Output: 5

# Perform multiplication
calculator.multiply(2, 3, 4)
print("Result after multiplication:", calculator.get_result()) # Output: 120
```

In this program:

We define a Calculator class that encapsulates the result attribute and various arithmetic operations.
The add(), subtract(), and multiply() methods perform addition, subtraction, and multiplication of multiple n umbers, respectively.
We use variable-length arguments *args to accept any number of arguments for these methods, allowing function overloading-like behavior.
The get_result() method retrieves the result.
By encapsulating the result attribute and the arithmetic operations within the class, we hide the internal im plementation details and expose a clean interface for performing calculations. The use of variable-length arguments demonstrates a form of overloading, allowing the methods to work with different numbers of ar guments.

This program illustrates encapsulation and overloading concepts in Python, showcasing the power of obje ct-oriented programming principles for creating reusable and organized code in data analytics or any othe r application domain.
==========================

4(b) What is inheritance ? Explain different types of inheritance with necessary example

Inheritance in object-oriented programming (OOP) is a mechanism that allows one class to inherit the pro perties and methods of another class. It promotes code reusability and the creation of a hierarchical struct ure among classes. Inheritance models the "is-a" relationship, where a derived (subclass or child) class "i s-a" specialized version of the base (superclass or parent) class. Python supports multiple types of inherit ance, including single inheritance, multiple inheritance, and multilevel inheritance.

1. Single Inheritance:

Single inheritance is the simplest form of inheritance, where a subclass inherits from a single superclass.
It represents a one-to-one relationship between classes.
Single inheritance is commonly used to create a specialized class based on a more general class.
Example:

```
class Animal:
    def__init__(self, name):
        self.name = name

    def speak(self):
        pass

class Dog(Animal):
    def speak(self):
        return f"{self.name} says Woof!"

class Cat(Animal):
    def speak(self):
        return f"{self.name} says Meow!"
```

```
dog = Dog("Buddy")
cat = Cat("Whiskers")

print(dog.speak()) # Output: "Buddy says Woof!"
print(cat.speak())  # Output: "Whiskers says Meow!"
```
In this example, Dog and Cat are subclasses of the Animal superclass. They inherit the name attribute an d override the speak() method to provide their own implementation.

2. Multiple Inheritance:

Multiple inheritance allows a subclass to inherit from multiple superclasses.
It enables the creation of a class that combines features and behaviors from more than one parent class.
Python uses a method resolution order (MRO) to determine the order in which methods are called when t here are conflicts between superclasses.
Example:

```
class A:
    def show(self):
        print("Class A")

class B:
    def show(self):
        print("Class B")

class C(A, B):
    pass

obj = C()
obj.show() # Output: "Class A"
```
In this example, C inherits from both A and B. When the show() method is called on an instance of C, it fol lows the MRO and uses the method from A.

3. Multilevel Inheritance:

Multilevel inheritance involves a chain of inheritance where a subclass inherits from a superclass, and the n another subclass inherits from the first subclass.
It creates a hierarchical relationship between classes.
Example:

```
class Grandparent:
    def show(self):
        print("Grandparent")

class Parent(Grandparent):
    def show(self):
        print("Parent")

class Child(Parent):
    def show(self):
        print("Child")

child = Child()
child.show() # Output: "Child"
```
In this example, Child inherits from Parent, which in turn inherits from Grandparent. Each class can overri

de methods inherited from its parent class.

4. Hierarchical Inheritance:

Hierarchical inheritance involves multiple subclasses inheriting from a single superclass.
It allows for the creation of different specialized classes based on a common parent class.
Example:

```
class Vehicle:
    def __init__(self, name):
        self.name = name

class Car(Vehicle):
    def drive(self):
        return f"{self.name} is driving."

class Bike(Vehicle):
    def ride(self):
        return f"{self.name} is riding."

car = Car("Sedan")
bike = Bike("Mountain Bike")

print(car.drive()) # Output: "Sedan is driving."
print(bike.ride())  # Output: "Mountain Bike is riding."
```

Here, both Car and Bike inherit from the Vehicle superclass, which contains common attributes.

5. Hybrid Inheritance:

Hybrid inheritance is a combination of multiple types of inheritance within a single program.
It allows for a mix of single, multiple, hierarchical, and multilevel inheritance.
Example:

```
class A:
    def method_A(self):
        print("Method A")

class B(A):
    def method_B(self):
        print("Method B")

class C(A):
    def method_C(self):
        print("Method C")

class D(B, C):
    def method_D(self):
        print("Method D")

obj = D()
obj.method_A() # Output: "Method A"
obj.method_B() # Output: "Method B"
obj.method_C() # Output: "Method C"
obj.method_D()  # Output: "Method D"
```

In this example, D inherits from both B and C, and B and C both inherit from A. This demonstrates hybrid i

nheritance with a mix of single and multiple inheritance.

These are the different types of inheritance in Python. Each type serves specific programming needs and helps in creating a more organized and efficient code structure. Understanding inheritance is crucial in obj ect-oriented programming and is often used in data analytics to create specialized classes for data modeli ng and analysis tasks.
===
5(a) Define creating an array from python lists. Explain numpy array attributes.

Creating an array from Python lists and understanding NumPy array attributes are essential concepts in d ata analytics using Python. Let’s cover these topics briefly.

Creating an Array from Python Lists:

In data analytics, you often work with data stored in arrays or matrices. Python provides the NumPy librar y, which is a powerful tool for working with arrays and numerical data. You can create a NumPy array fro m a Python list using the numpy.array() function.

Here’s how to create a NumPy array from a Python list:

```
import numpy as np

# Create a Python list
my_list = [1, 2, 3, 4, 5]

# Convert the Python list to a NumPy array
my_array = np.array(my_list)
```

In this example, we import the NumPy library as np, create a Python list called my_list, and then use np.ar ray() to convert it into a NumPy array called my_array. Now, you can perform various mathematical and d ata manipulation operations on my_array efficiently, which is particularly useful in data analytics.

Understanding NumPy Array Attributes:

NumPy arrays have several important attributes that provide information about the array’s properties. The se attributes are useful for data analysis and manipulation. Let’s discuss some of the key NumPy array att ributes:

shape:

The shape attribute returns a tuple representing the dimensions of the array.
For a one-dimensional array, it shows the number of elements along that dimension.
For a multi-dimensional array, it shows the number of elements along each dimension.

```
import numpy as np

my_array = np.array([[1, 2, 3], [4, 5, 6]])
shape = my_array.shape  # Returns (2, 3)
```

In this example, my_array is a 2x3 array, so its shape attribute returns (2, 3).

dtype:

The dtype attribute specifies the data type of the elements in the array.
NumPy arrays are homogeneous, meaning all elements in the array have the same data type.

```
import numpy as np

my_array = np.array([1, 2, 3], dtype=float)
data_type = my_array.dtype  # Returns float64
```
Here, my_array is explicitly assigned the data type float, so its dtype attribute returns float64.

size:

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

```
import numpy as np

my_array = np.array([[1, 2, 3], [4, 5, 6]])
size = my_array.size # Returns 6
```
The size attribute returns 6 because there are a total of six elements in the array.

ndim:

The ndim attribute returns the number of dimensions (axes) of the array.
A one-dimensional array has ndim equal to 1, a two-dimensional array has ndim equal to 2, and so on.

```
import numpy as np

my_array = np.array([1, 2, 3])
dimensions = my_array.ndim # Returns 1
```
Here, my_array is a one-dimensional array, so its ndim attribute returns 1.

itemsize:

The itemsize attribute returns the size (in bytes) of each element in the array.
It is particularly useful for memory management and optimization.

```
import numpy as np

my_array = np.array([1, 2, 3], dtype=np.int32)
item_size = my_array.itemsize # Returns 4
```
In this example, the itemsize attribute returns 4 because each element in the array is of data type int32, w hich occupies 4 bytes in memory.

Understanding these NumPy array attributes is crucial in data analytics because they provide information about the structure, data types, and memory usage of arrays. This information helps data analysts and sci entists make informed decisions about data processing and analysis techniques. Additionally, NumPy's ar ray attributes enable efficient data manipulation and mathematical operations, making it a fundamental to ol in the field of data analytics using Python.

==
5(b) Discuss with example numpy array concatenation and splitting.

Numpy Array Concatenation:

Concatenation refers to the process of combining two or more arrays into a single array. In NumPy, you c an concatenate arrays along different axes (dimensions) using functions like np.concatenate(), np.vstack( ), and np.hstack(). Here, we'll discuss these methods with examples:

np.concatenate():

The np.concatenate() function is used to concatenate arrays along a specified axis.
It takes a sequence of arrays to be concatenated and an optional axis parameter that specifies the axis al ong which the concatenation will be performed.
If axis is not specified, the default behavior is to concatenate along axis 0 (rows).
Example:

```
import numpy as np

# Create two arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6]])

# Concatenate along axis 0 (rows)
result = np.concatenate((arr1, arr2), axis=0)
```

In this example, arr1 and arr2 are concatenated along axis 0, resulting in a new array result:

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

np.vstack():

The np.vstack() function is used to vertically stack (concatenate along axis 0) arrays.
It is equivalent to using np.concatenate() with axis=0.
Example:

```
import numpy as np

# Create two arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6]])

# Vertically stack the arrays
result = np.vstack((arr1, arr2))
```

The result is the same as the previous example:

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

np.hstack():

The np.hstack() function is used to horizontally stack (concatenate along axis 1) arrays.
It is equivalent to using np.concatenate() with axis=1.
Example:

```
import numpy as np

# Create two arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5], [6]])

# Horizontally stack the arrays
```

```
result = np.hstack((arr1, arr2))
```

The result is a new array result:

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

Numpy Array Splitting:

Splitting refers to the process of dividing a single array into multiple smaller arrays. NumPy provides sever al functions for splitting arrays, including np.split(), np.hsplit(), and np.vsplit(). Here, we'll discuss these m ethods with examples:

np.split():

The np.split() function splits an array into multiple sub-arrays along a specified axis.
It takes the array to be split, the number of equally sized sub-arrays, and the axis along which the split sh ould occur.
Example:

```
import numpy as np

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

# Split the array into three equal parts
sub_arrays = np.split(arr, 3)
```

In this example, the arr array is split into three equal parts along axis 0, resulting in a list of sub-arrays:

```
[array([1, 2]), array([3, 4]), array([5, 6])]
```

np.hsplit():

The np.hsplit() function is used to split an array horizontally (along columns).
It takes the array to be split and the number of equally sized sub-arrays.
Example:

```
import numpy as np

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

# Split the array into two equal parts along columns
sub_arrays = np.hsplit(arr, 2)
```

The arr array is split into two equal parts along columns, resulting in a list of sub-arrays:

```
[array([[1],
       [4]]),
 array([[2, 3],
       [5, 6]])]
```

np.vsplit():

The np.vsplit() function is used to split an array vertically (along rows).
It takes the array to be split and the number of equally sized sub-arrays.

Example:

```
import numpy as np

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

# Split the array into two equal parts along rows
sub_arrays = np.vsplit(arr, 2)
```

The arr array is split into two equal parts along rows, resulting in a list of sub-arrays:

```
[array([[1, 2, 3]]),
 array([[4, 5, 6]])]
```

These are the fundamental concepts of NumPy array concatenation and splitting. Understanding these op erations is essential for data manipulation and analysis, as they allow you to organize and reshape data e fficiently for various analytical tasks in data analytics using Python.

==

5(C) Explain specialized universal function:
 (i) Trignometric (ii) Exponents and logarithms with necessary coding

Specialized Universal Functions in NumPy: Trigonometric and Exponents/Logarithms

NumPy provides a wide range of specialized universal functions (ufuncs) for performing various mathema tical operations on arrays efficiently. Two important categories of specialized ufuncs are trigonometric fun ctions and functions related to exponents and logarithms.

i) Trigonometric Functions:

Trigonometric functions are mathematical functions that deal with the angles and sides of triangles. NumP y provides a set of trigonometric ufuncs that can be applied element-wise to arrays. Here are some comm only used trigonometric functions:

np.sin() - Sine Function:

The np.sin() function computes the sine of each element in the input array.
Example:

```
import numpy as np

# Create an array of angles in radians
angles = np.array([0, np.pi / 2, np.pi])

# Compute the sine of the angles
sine_values = np.sin(angles)
```

The sine_values array will contain the sine values corresponding to the angles [0, 1, 0].

np.cos() - Cosine Function:

The np.cos() function computes the cosine of each element in the input array.
Example:

```
import numpy as np
```

```
# Create an array of angles in radians
angles = np.array([0, np.pi / 4, np.pi / 2])

# Compute the cosine of the angles
cosine_values = np.cos(angles)
```
The cosine_values array will contain the cosine values corresponding to the angles [1, 0.7071, 0].

np.tan() - Tangent Function:

The np.tan() function computes the tangent of each element in the input array.
Example:

```
import numpy as np

# Create an array of angles in radians
angles = np.array([0, np.pi / 4, np.pi / 3])

# Compute the tangent of the angles
tangent_values = np.tan(angles)
```
The tangent_values array will contain the tangent values corresponding to the angles [0, 1, 1.732].

ii) Exponents and Logarithms Functions:

Exponents and logarithms are fundamental mathematical operations used in various data analysis and sci entific computations. NumPy provides ufuncs for exponentiation and logarithmic operations.

np.exp() - Exponential Function:

The np.exp() function computes the exponential of each element in the input array.
Example:

```
import numpy as np

# Create an array of values
values = np.array([1, 2, 3])

# Compute the exponential of the values
exponential_values = np.exp(values)
```
The exponential_values array will contain the exponential values [2.7183, 7.3891, 20.0855].

np.log() - Natural Logarithm Function:

The np.log() function computes the natural logarithm (base e) of each element in the input array.
Example:

```
import numpy as np

# Create an array of values
values = np.array([1, 2, 4])

# Compute the natural logarithm of the values
logarithm_values = np.log(values)
```
The logarithm_values array will contain the natural logarithms [0, 0.6931, 1.3863].

np.log10() - Base-10 Logarithm Function:

The np.log10() function computes the base-10 logarithm of each element in the input array.
Example:

```
import numpy as np

# Create an array of values
values = np.array([1, 10, 100])

# Compute the base-10 logarithm of the values
logarithm10_values = np.log10(values)
```

The logarithm10_values array will contain the base-10 logarithms [0, 1, 2].

These specialized universal functions in NumPy are incredibly useful for performing mathematical operati ons on arrays efficiently. They are essential tools for data analysts and scientists working on data analytic s tasks, as they allow you to apply complex mathematical functions to large datasets with ease and speed . Whether you need to compute trigonometric values, exponentials, or logarithms, NumPy provides the to ols to simplify these calculations for your data analysis needs.

===
6(a) Mention Pandas data structures. Create a dataframe with three dimensional list state, year, POP(dicti onary). Write necessary coding for retrieving row valules and modifying colulmn values.

Pandas Data Structures:

Pandas is a powerful library for data manipulation and analysis in Python. It provides two primary data str uctures: Series and DataFrame.

Series:

A Series is a one-dimensional array-like object.
It is capable of holding data of any type, including integers, floats, and strings.
Each element in a Series has a labeled index, which can be automatically assigned or set explicitly.
You can think of a Series as a single column of data.
DataFrame:

A DataFrame is a two-dimensional tabular data structure.
It is similar to a spreadsheet or SQL table.
A DataFrame consists of rows and columns, where each column can have a different data type.
You can think of a DataFrame as a collection of Series objects, where each Series represents a column.
Now, let's create a DataFrame with a three-dimensional list, where the data includes state, year, and pop ulation information. We'll also write code to retrieve row values and modify column values.

Creating a DataFrame:

To create a DataFrame, we first need to import the Pandas library and then use the pd.DataFrame() const ructor. We can provide the data as a dictionary of lists, where each list represents a column.

```
import pandas as pd

data = {
    'State': ['California', 'New York', 'Texas', 'Florida'],
    'Year': [2010, 2010, 2011, 2011],
    'Population': [37254523, 19378102, 25145561, 18804623]
```

}

```
df = pd.DataFrame(data)
```

In this example, we have created a DataFrame df with three columns: 'State', 'Year', and 'Population', eac h containing the respective data.

Retrieving Row Values:

To retrieve row values from the DataFrame, you can use various indexing and slicing techniques. Here ar e a few examples:

Retrieve the first row:

```
first_row = df.iloc[0] # Using iloc (integer-based location)
```

Retrieve rows with specific conditions, e.g., where the Year is 2011:

```
year_2011_data = df[df['Year'] == 2011]
```

Retrieve a range of rows, e.g., from row 1 to row 2:

```
subset = df.iloc[1:3] # Rows 1 and 2 (exclusive of row 3)
```

Modifying Column Values:

You can modify column values in a DataFrame using various methods and assignments. Here's how you can do it:

Modify values in a specific column, e.g., change the population of California:

```
df.loc[df['State'] == 'California', 'Population'] = 40000000
```

This code locates the row where the 'State' is 'California' and updates the 'Population' column with the ne w value (40,000,000).

Add a new column to the DataFrame, e.g., add a column 'Area' with area values for each state:

```
df['Area'] = [164709, 54555, 268596, 65758]
```

This code adds a new column 'Area' to the DataFrame and assigns values to it.

Drop a column from the DataFrame, e.g., remove the 'Year' column:

```
df = df.drop('Year', axis=1)
```

This code drops the 'Year' column from the DataFrame. Note that we specify axis=1 to indicate that we ar e dropping a column (columns are along axis 1).

In summary, Pandas provides versatile data structures, including Series and DataFrames, for data manip ulation and analysis. You can create DataFrames from various data sources, retrieve row values using in dexing and slicing, and modify column values using assignment and DataFrame manipulation methods. T hese capabilities make Pandas an essential tool for data analytics, data cleaning, and data transformation tasks in Python.

===>

6(b)

Explain with example the concept of reindexing and fill method.

Reindexing and Fill Method in Pandas:

Reindexing is a fundamental operation in Pandas that allows you to change the row and column labels (in dexes) of a DataFrame or Series. It is a crucial data manipulation technique in data analytics as it helps in aligning and reshaping data. Additionally, the fill method in Pandas is used in conjunction with reindexing to handle missing data by specifying how to fill or interpolate missing values. Let's explore the concept of reindexing and the fill method with examples.

Reindexing:

Reindexing involves creating a new object with the data from the original object but with a different index. This is often necessary when you want to change the order of rows, add new rows or columns, or align m ultiple data sources with different indexes.

Here's how you can perform reindexing in Pandas:

```
import pandas as pd

# Create a DataFrame
data = {'A': [1, 2, 3], 'B': [4, 5, 6]}
df = pd.DataFrame(data, index=['row1', 'row2', 'row3'])

# Reindex the DataFrame with a new index
new_index = ['row3', 'row1', 'row4']
df_reindexed = df.reindex(new_index)
```

In this example, df is a DataFrame with an initial index (['row1', 'row2', 'row3']). We reindex it with a new in dex (['row3', 'row1', 'row4']) using the reindex method. The resulting DataFrame, df_reindexed, will have t he rows arranged according to the new index. Any rows with missing data will have NaN values.

The Fill Method:

When you reindex a DataFrame or Series and introduce new labels that did not exist in the original data, Pandas has to decide how to fill in the missing values. This is where the fill method comes into play. The f ill_value parameter specifies the value to use for filling the missing data.

Let's see how the fill method works with an example:

```
import pandas as pd
import numpy as np

# Create a Series with missing values
data = pd.Series([1, 2, np.nan, 4], index=['A', 'B', 'C', 'D'])

# Reindex the Series with a new index and fill missing values with 0
new_index = ['A', 'B', 'E', 'F']
filled_series = data.reindex(new_index, fill_value=0)
```

In this example, data is a Series with the initial index (['A', 'B', 'C', 'D']). When we reindex it with a new ind ex (['A', 'B', 'E', 'F']), we specify fill_value=0. As a result, any missing values in the new index will be filled with 0. The filled_series will look like this:

```
A   1.0
B   2.0
E   0.0
F   0.0
```

dtype: float64
Here, 'C' and 'D' were not present in the new index, so they were filled with 0.

Methods for Fill Values:

Pandas provides various methods to fill missing values when reindexing. You can use these methods inst ead of a specific fill_value. Some of the commonly used methods include:

ffill (or pad): Forward fill, which fills missing values with the previous valid value.

```
filled_series = data.reindex(new_index, method='ffill')
```

bfill (or backfill): Backward fill, which fills missing values with the next valid value.

```
filled_series = data.reindex(new_index, method='bfill')
```

nearest: Fills missing values with the nearest valid value, considering both forward and backward directio ns.

```
filled_series = data.reindex(new_index, method='nearest')
```

These methods can be especially useful when working with time series data, where you want to fill missin g values with data from adjacent time points.

In summary, reindexing is a powerful operation in Pandas for reshaping and aligning data. It allows you to change the index labels and introduces missing values when the new index contains labels not present in the original data. The fill method, along with the fill_value parameter and various fill methods like ffill and bfill, provides flexibility in handling missing data during reindexing. Understanding these concepts is esse ntial for data preprocessing and alignment tasks in data analytics using Pandas.

==>
6(C) How do we handle missing data in Python using Pandas? Explain with coding.

Handling missing data is a crucial step in data analysis and preprocessing. In Python, the Pandas library provides several techniques for handling missing data in a DataFrame or Series efficiently. Let's explore t hese techniques with coding examples:

1. Identifying Missing Data:

Before you can handle missing data, you need to identify where the missing values are located in your dat aset. Pandas uses the isna() and notna() functions to check for missing and non-missing values, respectiv ely.

```
import pandas as pd
import numpy as np

# Create a DataFrame with missing values
data = {'A': [1, 2, np.nan, 4],
       'B': [np.nan, 5, 6, 7],
       'C': [8, 9, 10, 11]}
df = pd.DataFrame(data)

# Check for missing values
missing_values = df.isna()
```

In this example, missing_values will be a DataFrame of the same shape as df, with True in positions wher e values are missing and False where values are not missing.

2. Removing Missing Data:

One way to handle missing data is to remove rows or columns containing missing values. You can use the dropna() method to achieve this.

```
# Remove rows containing any missing values
df_cleaned = df.dropna()

# Remove columns containing any missing values
df_cleaned = df.dropna(axis=1)
```

By default, dropna() removes rows or columns that contain any missing values. You can use the how parameter to specify whether you want to drop rows or columns containing all or any missing values.

3. Filling Missing Data:

Instead of removing missing data, you may want to fill it with specific values. You can use the fillna() method for this purpose.

```
# Fill missing values with a specific value
df_filled = df.fillna(0)
```

In this example, missing values are filled with the value 0. You can replace missing values with any value of your choice.

4. Forward and Backward Filling:

Sometimes, it’s useful to fill missing values using the values from the previous or next row or column. This can be done with the ffill() and bfill() methods.

```
# Forward fill missing values (use values from the previous row)
df_ffill = df.fillna(method='ffill')

# Backward fill missing values (use values from the next row)
df_bfill = df.fillna(method='bfill')
```

5. Interpolation:

Interpolation is a technique for estimating missing values based on the values of neighboring data points. Pandas provides the interpolate() method for this purpose.

```
# Interpolate missing values using linear interpolation
df_interpolated = df.interpolate()
```

By default, interpolate() uses linear interpolation, but you can specify other interpolation methods, such as polynomial or spline interpolation, using the method parameter.

6. Imputation:

Imputation involves replacing missing values with statistically derived estimates. The SimpleImputer class from the sklearn.impute module is commonly used for imputation.

```
from sklearn.impute import SimpleImputer

# Create an imputer object
imputer = SimpleImputer(strategy='mean') # You can use 'mean', 'median', 'most_frequent', etc.

# Fit the imputer to the data and transform it
df_imputed = imputer.fit_transform(df)
```

```
df_imputed = pd.DataFrame(df_imputed, columns=df.columns)
```

In this example, missing values are imputed with the mean of each column. You can choose from various imputation strategies depending on your data and analysis needs.

7. Custom Function for Filling:

You can also define a custom function to fill missing values based on your specific requirements.

```
# Define a custom function to fill missing values with the median of the column
def custom_fill(column):
    median_value = column.median()
    return column.fillna(median_value)

# Apply the custom function to fill missing values
df_custom_filled = df.apply(custom_fill)
```

Here, the custom_fill() function fills missing values with the median of the respective column.

8. Drop Duplicates:

Sometimes, missing data can result from duplicate rows. You can use the drop_duplicates() method to re move duplicate rows from the DataFrame.

```
# Remove duplicate rows
df_no_duplicates = df.drop_duplicates()
```

9. Handling Missing Data in Time Series:

For time series data, handling missing data may require specialized techniques. Pandas provides function s like resample() and asfreq() for working with time-based data.

```
# Resample time series data to fill missing values
df_resampled = df.resample('D').mean() # Resample to daily frequency, filling missing values with mean
```

In this example, we resample the time series data to fill missing values with the mean value for each day.

In summary, Pandas provides a variety of techniques for handling missing data in Python. Depending on t he nature of your data and the analysis you are conducting, you can choose the most appropriate method to either remove or fill missing values. Understanding these techniques is essential for data preprocessin g and ensuring the quality of your data for analysis.

===>

7(A) Explain reading and writing data in text format in Python with examples.

Reading and Writing Data in Text Format in Python

In data analytics using Python, one of the fundamental tasks is reading and writing data in text format. Te xt formats are widely used for data storage and interchange, and Python provides several libraries and m ethods for handling text-based data files, such as CSV, JSON, and plain text files. In this answer, we'll ex plore how to read and write data in text formats using examples.

Reading Data from Text Files:

Python offers several libraries and methods for reading data from text files, but one of the most commonly used libraries is the pandas library. We'll focus on using pandas for reading data from text files.

Example 1: Reading Data from a CSV File

CSV (Comma-Separated Values) files are a common text-based format for storing tabular data. pandas provides the read_csv() function to read data from CSV files.

```
import pandas as pd

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

In this example, data.csv is a CSV file containing tabular data. pd.read_csv() reads the data from the file and stores it in a DataFrame called df. You can then manipulate and analyze the data using pandas.

Example 2: Reading Data from a JSON File

JSON (JavaScript Object Notation) is another widely used text-based format for data storage. pandas can read data from JSON files using the read_json() function.

```
import pandas as pd

# Read data from a JSON file into a DataFrame
df = pd.read_json('data.json')
```

In this example, data.json is a JSON file containing structured data. pd.read_json() reads the data and stores it in a DataFrame.

Example 3: Reading Data from a Text File

Sometimes, you may have data stored in plain text files with custom formatting. You can use Python's built-in file reading capabilities to read data from such files.

```
# Read data from a plain text file
data = []
with open('data.txt', 'r') as file:
    for line in file:
        data.append(line.strip())

# 'data' now contains a list of lines from the text file
```

In this example, data.txt is a plain text file where each line contains a data point. We open the file using a context manager (with open(...) as file) and read the data line by line into a list.

Writing Data to Text Files:

Just as you can read data from text files, you can also write data to text files using Python. Again, pandas is a versatile library for this purpose, but we'll also explore writing data to plain text files.

Example 1: Writing Data to a CSV File

To write data to a CSV file, you can use the to_csv() method of a pandas DataFrame.

```
import pandas as pd

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

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

In this example, we create a DataFrame df and then use the to_csv() method to write it to a CSV file nam ed output.csv. We specify index=False to avoid writing the row index to the file.

Example 2: Writing Data to a JSON File

To write data to a JSON file, you can use the to_json() method of a pandas DataFrame.

```
import pandas as pd

# Create a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}
df = pd.DataFrame(data)

# Write the DataFrame to a JSON file
df.to_json('output.json', orient='records')
```

In this example, we create a DataFrame df and then use the to_json() method to write it to a JSON file na med output.json. We specify orient='records' to format the data in a JSON array of records.

Example 3: Writing Data to a Plain Text File

To write data to a plain text file, you can use Python's built-in file writing capabilities.

```
# Create a list of data
data = ['Line 1', 'Line 2', 'Line 3']

# Write the data to a plain text file
with open('output.txt', 'w') as file:
    for line in data:
        file.write(line + '\n')
```

In this example, we have a list of data, and we write each element of the list as a separate line in the text f ile output.txt.

Summary:

Reading and writing data in text format is a fundamental skill in data analytics using Python. Python provi des various libraries and methods for handling different text-based data formats, including CSV, JSON, a nd plain text files. Depending on your data format and analysis needs, you can choose the appropriate m ethods and libraries to efficiently read and write data.

==>

7(B) Explain reading and writing data in text format in Python with examples.

Reading and Writing Data in Text Format in Python

In data analytics using Python, reading and writing data in text formats are essential tasks. Text formats a re commonly used for data storage and interchange because they are human-readable and versatile. Pyt hon provides several methods and libraries to handle text-based data files, such as CSV, JSON, and plai n text files. In this answer, we'll explore how to read and write data in text formats using examples.

Reading Data from Text Files:

Python offers multiple libraries and methods for reading data from text files. Here, we’ll primarily focus on using the pandas library for reading data.

Example 1: Reading Data from a CSV File

CSV (Comma-Separated Values) files are a prevalent text-based format for storing tabular data. pandas provides the read_csv() function to read data from CSV files.

```
import pandas as pd

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

In this example, data.csv is a CSV file containing tabular data. The pd.read_csv() function reads the data from the file and stores it in a pandas DataFrame called df. This DataFrame can then be manipulated and analyzed using pandas functions.

Example 2: Reading Data from a JSON File

JSON (JavaScript Object Notation) is another commonly used text-based format for data storage. pandas can read data from JSON files using the read_json() function.

```
import pandas as pd

# Read data from a JSON file into a DataFrame
df = pd.read_json('data.json')
```

In this example, data.json is a JSON file containing structured data. The pd.read_json() function reads the data and stores it in a pandas DataFrame.

Example 3: Reading Data from a Text File

Sometimes, data is stored in plain text files with custom formatting. You can use Python’s built-in file reading capabilities to read data from such files.

```
# Read data from a plain text file
data = []
with open('data.txt', 'r') as file:
    for line in file:
        data.append(line.strip())
```

In this example, data.txt is a plain text file where each line contains a data point. We open the file using a context manager (with open(...) as file) and read the data line by line into a Python list.

Writing Data to Text Files:

Just as you can read data from text files, you can also write data to text files using Python. Again, the pandas library is a versatile choice for this purpose, but we’ll also explore writing data to plain text files.

Example 1: Writing Data to a CSV File

To write data to a CSV file, you can use the to_csv() method of a pandas DataFrame.

```
import pandas as pd
```

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

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

In this example, we create a pandas DataFrame df and then use the to_csv() method to write it to a CSV file named output.csv. The index=False parameter is specified to avoid writing the row index to the file.

Example 2: Writing Data to a JSON File

To write data to a JSON file, you can use the to_json() method of a pandas DataFrame.

```
import pandas as pd

# Create a DataFrame
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35]}
df = pd.DataFrame(data)

# Write the DataFrame to a JSON file
df.to_json('output.json', orient='records')
```

In this example, we create a pandas DataFrame df and then use the to_json() method to write it to a JSON file named output.json. We specify orient='records' to format the data as a JSON array of records.

Example 3: Writing Data to a Plain Text File

To write data to a plain text file, you can use Python's built-in file writing capabilities.

```
# Create a list of data
data = ['Line 1', 'Line 2', 'Line 3']

# Write the data to a plain text file
with open('output.txt', 'w') as file:
    for line in data:
        file.write(line + '\n')
```

In this example, we have a list of data, and we write each element of the list as a separate line in the text file output.txt.

Summary:

Reading and writing data in text formats are fundamental tasks in data analytics using Python. Python provides various libraries and methods for handling different text-based data formats, including CSV, JSON, and plain text files. Depending on your data format and analysis needs, you can choose the appropriate methods and libraries to efficiently read and write data.

===>

8(a) Explain with example the following merge methods
(i) inner (ii) left (iii) right

Merge Methods in Pandas: Inner, Left, and Right Joins

In data analytics using Python, merging or joining data frames is a fundamental operation. Pandas provid es various methods for merging data frames, including inner, left, and right joins. These methods allow yo u to combine data from multiple data frames based on a common column (key). Let's explore these merg e methods with examples.

Example DataFrames:

Let's start by creating two data frames, df1 and df2, with the following data:

```
import pandas as pd

# Create df1
df1 = pd.DataFrame({'data1': ['b', 'a', 'c', 'a', 'b'],
                    'key': [1, 2, 3, 4, 5]})

# Create df2
df2 = pd.DataFrame({'data2': ['a', 'b', 'b', 'd'],
                    'key': [0, 1, 3, 4]})
```

Here's what the two data frames look like:

DataFrame df1:

```
  data1 key
0    b   1
1    a   2
2    c   3
3    a   4
4    b   5
```

DataFrame df2:

```
  data2 key
0    a   0
1    b   1
2    b   3
3    d   4
```

Now, let's explore the three merge methods: inner, left, and right joins.

(i) Inner Join:

An inner join returns only the rows that have matching keys in both data frames. In other words, it include s only the common keys between the data frames.

```
# Perform an inner join on 'key'
inner_join = pd.merge(df1, df2, on='key', how='inner')
```

Result of Inner Join:

```
  data1 key data2
0    b   1    b
1    a   4    d
```

In the inner join example, only the rows with common keys '1' and '4' are included in the result. Rows with keys '2', '3', and '5' from df1 and key '0' from df2 are excluded because they do not have matching keys i n both data frames.

(ii) Left Join:

A left join returns all the rows from the left data frame (df1 in this case), along with the matching rows from the right data frame (df2). If there is no match in the right data frame, it fills in missing values with NaN.

python
Copy code
```
# Perform a left join on 'key'
left_join = pd.merge(df1, df2, on='key', how='left')
```
Result of Left Join:

```
  data1  key data2
0     b    1     b
1     a    2   NaN
2     c    3   NaN
3     a    4     d
4     b    5   NaN
```
In the left join example, all the rows from df1 are included in the result, and matching rows from df2 are ad ded based on the common keys ('1' and '4'). Rows with keys '2', '3', and '5' from df1 are included but hav e NaN values in the 'data2' column because there is no matching key in df2.

(iii) Right Join:

A right join is similar to a left join but returns all the rows from the right data frame (df2 in this case) and m atching rows from the left data frame (df1). If there is no match in the left data frame, it fills in missing valu es with NaN.

```
# Perform a right join on 'key'
right_join = pd.merge(df1, df2, on='key', how='right')
```
Result of Right Join:

```
data1 key data2
0     b    1     b
1     a    4     d
2   NaN    0     a
3   NaN    3     b
```
In the right join example, all the rows from df2 are included in the result, and matching rows from df1 are a dded based on the common keys ('1' and '4'). Rows with keys '0' and '3' from df2 are included but have N aN values in the 'data1' column because there is no matching key in df1.

Summary:

Inner Join: Returns only the rows with matching keys in both data frames.
Left Join: Returns all rows from the left data frame and matching rows from the right data frame.
Right Join: Returns all rows from the right data frame and matching rows from the left data frame.
These merge methods are essential tools for combining and analyzing data from multiple sources in data analytics using Python. The choice of which method to use depends on your specific analysis requirement s and the structure of your data.

==
8(b) Explain Data transforming using a function or mapping . Create a dataframe with following columns
Food   Ounce

Bacon 4.0
pulled pork 3.0
bacon 12.0
Honeyham 5.0
add a column indicating the type of animal that each food come from.

Data Transformation Using a Function or Mapping in Pandas

Data transformation is a common task in data analytics using Python, and it often involves applying a func tion or mapping values to create new columns or modify existing ones. In this example, we'll create a Dat aFrame with columns representing food and their respective ounce values and then add a new column in dicating the type of animal each food comes from.

Creating the DataFrame:

Let's start by creating the initial DataFrame with the given columns: "Food" and "Ounce."

```
import pandas as pd

data = {'Food': ['Bacon', 'Pulled Pork', 'Bacon', 'Honey Ham'],
       'Ounce': [4.0, 3.0, 12.0, 5.0]}

df = pd.DataFrame(data)
```

Here's what the DataFrame looks like:

```
        Food Ounce
0       Bacon    4.0
1  Pulled Pork    3.0
2       Bacon  12.0
3   Honey Ham    5.0
```

Adding a Column Indicating the Type of Animal:

To add a new column indicating the type of animal from which each food comes, we can use a mapping d ictionary and the map() function in Pandas.

```
# Define a mapping dictionary for food to animal type
food_to_animal = {'Bacon': 'Pig',
                 'Pulled Pork': 'Pig',
                 'Honey Ham': 'Pig'}

# Map the food names to animal types and create a new 'Animal' column
df['Animal'] = df['Food'].map(food_to_animal)
```

Updated DataFrame:

The updated DataFrame will now have the "Animal" column indicating the type of animal associated with each food.

```
        Food Ounce Animal
0       Bacon    4.0    Pig
1  Pulled Pork    3.0    Pig
2       Bacon   12.0    Pig
3   Honey Ham    5.0    Pig
```

In this example, we used a mapping dictionary (food_to_animal) to associate each food with the type of a

nimal it comes from. Then, we used the map() function to apply this mapping to the "Food" column and create a new "Animal" column in the DataFrame.

This kind of data transformation is useful for adding contextual information to your data and preparing it for various analyses in data analytics using Python. It allows you to enrich your dataset by deriving new columns based on existing information or external knowledge.

===

9(a) Write a Python program to plot sinusoid and cosine waves using matplotlib and label them with necessary titile and labels.

Plotting Sinusoid and Cosine Waves Using Matplotlib

In data analytics using Python, data visualization is a crucial part of understanding and presenting data. Matplotlib is a widely used library for creating various types of plots, including sinusoid and cosine waves. Here's a Python program to plot sinusoid and cosine waves using Matplotlib, complete with titles and labels:

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

# Generate data points for the x-axis (from 0 to 2*pi)
x = np.linspace(0, 2 * np.pi, 100)

# Calculate the sinusoid and cosine values for the y-axis
y_sin = np.sin(x)
y_cos = np.cos(x)

# Create a figure and axis
fig, ax = plt.subplots()

# Plot the sinusoid wave with a blue line
ax.plot(x, y_sin, label='Sinusoid', color='blue')

# Plot the cosine wave with a red dashed line
ax.plot(x, y_cos, label='Cosine', linestyle='--', color='red')

# Add a title
ax.set_title('Sinusoid and Cosine Waves')

# Add x and y axis labels
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')

# Add a legend
ax.legend()

# Show the plot
plt.show()
```

Explanation:

We import the necessary libraries: matplotlib.pyplot for creating plots and numpy for numerical operations.

We generate data points for the x-axis using np.linspace() to create values from 0 to 2*pi (one full cycle).

We calculate the corresponding sinusoid and cosine values for the y-axis using np.sin() and np.cos().

We create a figure and axis using plt.subplots().

We plot the sinusoid wave with a blue solid line and label it as "Sinusoid."

We plot the cosine wave with a red dashed line and label it as "Cosine."

We add a title to the plot using ax.set_title().

We label the x and y axes using ax.set_xlabel() and ax.set_ylabel().

We add a legend to differentiate between the two waves.

Finally, we display the plot using plt.show().

This Python program generates a plot of sinusoid and cosine waves with appropriate titles, labels, and leg ends, making it suitable for data visualization in data analytics using Python.

===
9(b) Explain with necessary coding creating a basic error bars and continuos errors.

Creating Basic Error Bars and Continuous Errors in Matplotlib

Error bars and continuous errors are essential tools in data visualization, allowing us to represent uncertai nty and variation in data. In data analytics using Python, Matplotlib provides functions to create error bars for discrete data points and continuous error bands for continuous data. Let's explore both concepts with necessary coding.

Creating Basic Error Bars:

Error bars are commonly used to represent the uncertainty or variability of data points. In Matplotlib, you c an create error bars for discrete data points using the errorbar() function.

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

# Sample data
x = np.array([1, 2, 3, 4, 5])
y = np.array([5, 7, 8, 7, 6])

# Sample error values (standard deviations)
y_error = np.array([0.5, 0.3, 0.4, 0.2, 0.6])

# Create a figure and axis
fig, ax = plt.subplots()

# Plot the data points with error bars
ax.errorbar(x, y, yerr=y_error, fmt='o', color='blue', label='Data')

# Add labels and title
```

```
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_title('Basic Error Bars')

# Add a legend
ax.legend()

# Show the plot
plt.show()
```

Explanation:

We import the necessary libraries: matplotlib.pyplot for creating plots and numpy for numerical operations.

We define sample data x and y, representing x-values and y-values.

We create sample error values y_error, which represent the standard deviations of the data points.

We create a figure and axis using plt.subplots().

We plot the data points with error bars using ax.errorbar(). The fmt='o' parameter specifies that we want t o plot circles at data points, and color='blue' sets the color of the data points.

We add labels to the x and y axes and set a title for the plot using ax.set_xlabel(), ax.set_ylabel(), and ax.set_title().

We add a legend to the plot to label the data.

Finally, we display the plot using plt.show().

The resulting plot shows data points with error bars representing the uncertainty in the y-values for each d ata point.

Creating Continuous Error Bands:

Continuous error bands are useful when you want to visualize the uncertainty or variability of a continuous dataset. Matplotlib allows you to create continuous error bands using the fill_between() function.

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

# Sample data
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Sample error values (standard deviations)
y_error = 0.2 * np.sin(x)

# Create a figure and axis
fig, ax = plt.subplots()

# Plot the continuous data
ax.plot(x, y, color='blue', label='Data')

# Fill the error band
```

```
ax.fill_between(x, y - y_error, y + y_error, color='gray', alpha=0.5, label='Error Band')

# Add labels and title
ax.set_xlabel('X-axis')
ax.set_ylabel('Y-axis')
ax.set_title('Continuous Error Band')

# Add a legend
ax.legend()

# Show the plot
plt.show()
```

Explanation:

We import the necessary libraries: matplotlib.pyplot for creating plots and numpy for numerical operations.

We generate sample data x and y using np.linspace() and calculate the corresponding sample error value s y_error.

We create a figure and axis using plt.subplots().

We plot the continuous data using ax.plot() with a blue line.

We fill the error band around the data using ax.fill_between(). The color='gray' parameter sets the color of the error band, and alpha=0.5 adjusts the transparency of the fill.

We add labels to the x and y axes and set a title for the plot using ax.set_xlabel(), ax.set_ylabel(), and ax. set_title().

We add a legend to the plot to label the data and the error band.

Finally, we display the plot using plt.show().

The resulting plot shows a continuous dataset with a shaded error band that represents the uncertainty in the data. This is useful for visualizing how data points may vary around the central trend.

Summary:

In data analytics using Python and Matplotlib, error bars and continuous error bands are valuable tools for representing uncertainty and variability in data. You can use errorbar() for discrete data points and fill_bet ween() for continuous data to create informative visualizations. These visualizations help convey the relia bility and uncertainty of data in a clear and concise manner.

===>

10(a) Write a Python program to plot the histogram as follows(customized histogram)

Creating a Customized Histogram Using Matplotlib

In data analytics using Python, histograms are powerful tools for visualizing the distribution of data. You c an create customized histograms in Matplotlib to control various aspects of the plot, such as the number o f bins, colors, and labels. Here's a Python program to create a customized histogram:

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

```
# Sample data (replace with your own dataset)
data = [10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80]

# Create a figure and axis
fig, ax = plt.subplots()

# Define histogram properties
bin_edges = np.arange(0, 90, 10) # Specify bin edges (customized)
hist_color = 'skyblue'            # Specify histogram color
edge_color = 'black'              # Specify edge color
xlabel = 'Value Range'            # Specify x-axis label
ylabel = 'Frequency'              # Specify y-axis label
title = 'Customized Histogram'    # Specify plot title

# Create the histogram
ax.hist(data, bins=bin_edges, color=hist_color, edgecolor=edge_color)

# Add labels and title
ax.set_xlabel(xlabel)
ax.set_ylabel(ylabel)
ax.set_title(title)

# Show the plot
plt.show()
```

Explanation:

We import the necessary libraries: matplotlib.pyplot for creating plots and numpy for numerical operations.

We define sample data in the data variable. In practice, replace this with your own dataset.

We create a figure and axis using plt.subplots().

We customize the histogram properties:

bin_edges: This specifies the bin edges for the histogram. You can customize this to define the bin interva ls that best suit your data.

hist_color: This sets the color of the histogram bars.

edge_color: This sets the color of the edges of the bars.

xlabel and ylabel: These specify the labels for the x and y axes.

title: This sets the title of the plot.

We create the histogram using ax.hist(). We pass the data variable as the dataset, and we customize the bins using the bins parameter with our defined bin_edges. We also specify the histogram and edge colors .

We add labels to the x and y axes and set a title for the plot using ax.set_xlabel(), ax.set_ylabel(), and ax. set_title().

Finally, we display the plot using plt.show().

This program creates a customized histogram with specified bin edges, colors, and labels. You can replac e the data variable with your dataset to visualize the distribution of your data effectively. Customizing histo grams in this way allows you to tailor visualizations to the specific needs of your data analysis in data anal ytics using Python.

==

10(b) What is Seaborn plot? Explain pair plots for 'iris' dataset and kernel density estimation using kdeplot and displot

Seaborn Plots and Pair Plots for the 'Iris' Dataset

Seaborn is a Python data visualization library based on Matplotlib. It provides a high-level interface for cre ating aesthetically pleasing and informative statistical graphics. One of its strengths is its ability to work se amlessly with pandas DataFrames, making it an excellent choice for data analytics using Python. In this a nswer, we'll explore Seaborn's pair plots and kernel density estimation using kdeplot and displot with the ' Iris' dataset.

Understanding the Iris Dataset:

The 'Iris' dataset is a well-known dataset in machine learning and data analytics. It contains measurement s of four features (sepal length, sepal width, petal length, and petal width) for three different species of iris flowers (setosa, versicolor, and virginica). Each species has 50 samples, making a total of 150 samples i n the dataset.

Pair Plots in Seaborn:

Pair plots are a useful way to visualize the relationships between pairs of variables in a dataset. In Seabor n, you can create pair plots using the pairplot() function. It creates a grid of scatterplots for all possible pai rs of numerical columns and histograms for the diagonal.

Here's how you can create a pair plot for the 'Iris' dataset using Seaborn:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the Iris dataset from Seaborn
iris = sns.load_dataset('iris')

# Create a pair plot
sns.pairplot(iris, hue='species', markers=["o", "s", "D"])

# Show the plot
plt.show()
```

Explanation:

We import the necessary libraries: seaborn for creating Seaborn plots and matplotlib.pyplot for displaying plots.

We load the 'Iris' dataset using the sns.load_dataset() function from Seaborn. This dataset is readily avail

able in Seaborn and contains the required data for this demonstration.

We create a pair plot using sns.pairplot(). We pass the 'Iris' dataset (iris) as the data source. Additionally, we specify hue='species' to color the data points based on the species column and markers to customize the marker style for each species.

Finally, we display the pair plot using plt.show().

The resulting pair plot shows scatterplots for all pairs of numerical columns (sepal length, sepal width, pet al length, and petal width) in the 'Iris' dataset. Each species is represented by a different color, and you ca n observe the relationships and distributions between these variables for each species.

Kernel Density Estimation (KDE) with kdeplot:

Kernel Density Estimation (KDE) is a non-parametric way to estimate the probability density function of a continuous random variable. In Seaborn, you can create KDE plots using the kdeplot() function. Let's dem onstrate KDE for the 'Iris' dataset:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the Iris dataset from Seaborn
iris = sns.load_dataset('iris')

# Create a KDE plot for sepal length
sns.kdeplot(data=iris, x='sepal_length', hue='species', common_norm=False)

# Show the plot
plt.show()
```

Explanation:

We import the necessary libraries: seaborn for creating Seaborn plots and matplotlib.pyplot for displaying plots.

We load the 'Iris' dataset using the sns.load_dataset() function from Seaborn.

We create a KDE plot using sns.kdeplot(). We specify the data source (data=iris) and the variable to be pl otted on the x-axis (x='sepal_length'). Additionally, we use hue='species' to color the KDE curves based o n the species column. The common_norm=False parameter ensures that each KDE curve is normalized i ndependently.

Finally, we display the KDE plot using plt.show().

The resulting KDE plot shows the estimated probability density functions for sepal length for each species in the 'Iris' dataset. You can see how the distributions differ among the three species.

Kernel Density Estimation (KDE) with displot:

Seaborn's displot() function allows you to create distribution plots, including KDE plots, in a flexible mann er. Let's create a KDE plot for sepal length in the 'Iris' dataset using displot:

```
import seaborn as sns
import matplotlib.pyplot as plt

# Load the Iris dataset from Seaborn
```

```
iris = sns.load_dataset('iris')

# Create a KDE plot for sepal length using displot
sns.displot(data=iris, x='sepal_length', hue='species', kind='kde', fill=True)

# Show the plot
plt.show()
```

Explanation:

We import the necessary libraries: seaborn for creating Seaborn plots and matplotlib.pyplot for displaying plots.

We load the 'Iris' dataset using the sns.load_dataset() function from Seaborn.

We create a KDE plot using sns.displot(). We specify the data source (data=iris), the variable to be plotted on the x-axis (x='sepal_length'), and hue='species' to color the KDE curves based on the species column. We use kind='kde' to specify that we want a KDE plot, and fill=True to fill the area under the KDE curves.

Finally, we display the KDE plot using plt.show().

The resulting KDE plot using displot shows the estimated probability density functions for sepal length for each species in the 'Iris' dataset, with filled areas under the curves.