CMR
INSTITUTE OF TECHNOLOGY

USN

Third Semester MCA Degree Examination, Dec. 2023/Jan. 2024
Data Analytics Using Python

Time: 3 hrs **Max. Marks:100**

Note :1. Answer FIVE FULL Questions, choosing ONE full question from each Module
2.M:Marks, L:Bloom's level, C:Course Outcomes

		Module 1	M	L	C
1	a.	Describe the following with examples i)Keywords ii)Data types iii) Statements iv) Expressions	8	L2	CO1
	b.	Discuss on various types of operators with examples	8	L2	CO1
	c.	Write a python program to check whether the given year is leap year or not	4	L2	CO1

OR

2	a.	Describe the for statement with an example	5	L2	CO1
	b.	Write a python program to print the following pattern: 1 1 2 1 2 3 1 2 3 4 1 2 3 4 5	5	L2	CO1
	c.	What is function? Discuss on various types of functions with examples.	10	L2	CO1
		Module 2			
3	a.	Explain any five string methods with examples	10	L3	CO2
	b.	Explain the concepts of sets and dictionary with examples	6	L3	CO2
	c.	Illustrate the concepts of list slicing with examples	4	L3	CO2

OR

4	a.	Explain the following methods with an example program. i)inheritance ii)Operator Overloading	5	L3	CO1
	b.	Illustrate the various modes of opening files with examples.	5	L2	CO1
Module 3					
5	a.	Explain the following methods related to database with examples: i)create ii)Insert iii) execute iv) fetchall	8	L3	CO3
	b.	How to handle missing values in python? Explain with examples.	6	L3	CO3
	c.	Explain the reading and writing data in text format in python	6	L3	CO3

OR

6	a.	Explain the following merge methods. i)Outer ii)Left iii) Right	10	L3	CO3
	b.	Explore the following data transformation methods. i) Discretization and binning ii) Detecting and filtering outliers iii) Renaming axis and indexes.	10	L3	CO3
Module 4					
7	a.	What is web Scraping? Explain the various methods available in requests module with an example program	10	L3	CO3
	b.	Discuss the implementation of web scraping in python with beautiful soup	10	L3	CO3

OR

8	a.	Explain the attributes of arrays in numpy with examples	10	L3	CO3
	b.	Write a python program to demonstrate the following operations using numpy array i) Array searching ii) Sorting iii) Splitting iv) Broadcasting v) Concatenation	10	L3	CO3
Module 5					
9	a.	Describe the concepts of matplotlib package in detail	10	L4	CO4
	b.	Write a python program for visualization of lineplot, histogram and scatter plot using matplotlib package	10	L4	CO4

OR

10	a.	Explain the concepts of seaborn package in detail.	10	L4	CO4
	b.	Explain the concepts of time series analysis with pandas	10	L4	CO4

1 a.

i) Keywords:

Keywords are reserved words in Python that have special meanings and cannot be used as variable names or identifiers. They are used to define the syntax and structure of the language. Examples of keywords in Python include `if`, `else`, `for`, `while`, `def`, `import`, `class`, `return`, `True`, `False`, `None`, etc.

```
# Example of using keywords in Python

if True:

    print("This is an example of using the 'if' keyword.")



for i in range(5):

    print(i)
```

ii) Data types:

Data types in Python specify the kind of values that variables can hold. Python supports several built-in data types such as integers, floats, strings, lists, tuples, dictionaries, etc.

```
# Examples of different data types in Python

# Integer data type

num_int = 10



# Float data type

num_float = 3.14



# String data type

str_var = "Hello, world!"



# List data type

list_var = [1, 2, 3, 4, 5]
```

```
# Tuple data type
tuple_var = (10, 20, 30)


# Dictionary data type
dict_var = {"a": 1, "b": 2, "c": 3}
```

iii) Statements:

Statements in Python are instructions that the interpreter can execute. They can be simple or compound. Simple statements are comprised of a single logical line, while compound statements contain multiple logical lines and often use indentation to indicate blocks of code.

```
# Example of simple statement
x = 10
# Example of compound statement (if-else)
if x > 5:
    print("x is greater than 5")
else:
    print("x is less than or equal to 5")
```

iv) Expressions:

Expressions in Python are combinations of values, variables, operators, and function calls that are evaluated to produce a result. Expressions can be simple or complex.

```
# Example of simple expression
result = 10 + 5
# Example of complex expression
complex_result = (5 * 2) + (10 / 2)
```

1b.

1. Arithmetic Operators:

Arithmetic operators are used for mathematical calculations like addition, subtraction, multiplication, division, etc.

```
# Examples of arithmetic operators

a = 10

b = 5


addition = a + b  # Addition

subtraction = a - b  # Subtraction

multiplication = a * b  # Multiplication

division = a / b  # Division

modulo = a % b  # Modulus (remainder of division)

exponentiation = a ** b  # Exponentiation

floor_division = a // b  # Floor division (rounds down to the nearest integer)
```

2. Comparison Operators:

Comparison operators are used to compare values. They return either True or False based on the comparison.

```
# Examples of comparison operators

x = 10

y = 5


# Equal to

print(x == y)  # False

# Not equal to
```

```
print(x != y)  # True


# Greater than

print(x > y)   # True


# Less than

print(x < y)   # False


# Greater than or equal to

print(x >= y)  # True


# Less than or equal to

print(x <= y)  # False
```

3. Logical Operators:

Logical operators are used to combine conditional statements. They return True or False based on the logical condition.

```
# Examples of logical operators

p = True

q = False


# Logical AND

print(p and q)  # False


# Logical OR

print(p or q)   # True
```

```
# Logical NOT
print(not p)    # False
```

4. Assignment Operators:

Assignment operators are used to assign values to variables.

```
# Examples of assignment operators
x = 10  # Assigns 10 to x
x += 5  # Adds 5 to x, equivalent to x = x + 5
x -= 3  # Subtracts 3 from x, equivalent to x = x - 3
x *= 2  # Multiplies x by 2, equivalent to x = x * 2
x /= 4  # Divides x by 4, equivalent to x = x / 4
```

5. Bitwise Operators:

Bitwise operators are used to perform bitwise operations on integers.

```
# Examples of bitwise operators
a = 10  # Binary: 1010
b = 4   # Binary: 0100

# Bitwise AND
print(a & b)  # 0

# Bitwise OR
print(a | b)  # 14
```

```
# Bitwise XOR
print(a ^ b)  # 14


# Bitwise NOT
print(~a)     # -11 (bitwise negation of a)


# Bitwise left shift
print(a << 1)  # 20 (shifts binary representation of a one place to the left)


# Bitwise right shift
print(a >> 1)  # 5 (shifts binary representation of a one place to the right)
```

1 c.

```
def is_leap_year(year):
    """
    Check if the given year is a leap year or not.

    Parameters:
        year (int): The year to be checked.

    Returns:
        bool: True if the year is a leap year, False otherwise.
    """
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    else:
        return False


# Input year from the user
year = int(input("Enter the year: "))


# Check if the input year is a leap year or not
if is_leap_year(year):
    print(f"{year} is a leap year.")
else:
    print(f"{year} is not a leap year.")
```

2 a.

In Python, the `for` statement is used for looping over a sequence (such as a list, tuple, string, or range) or any iterable object. It executes a block of code multiple times, once for each item in the sequence.

Here's the syntax of the `for` statement in Python:

```
for item in sequence:
    # Code block to be executed for each item
```

- `item`: A variable that represents each item in the sequence during each iteration.
- `sequence`: The sequence of items over which the loop iterates.

The `for` loop continues until all items in the sequence have been processed.

Here's an example of using a `for` loop to iterate over a list of numbers:

```
# Example of using the for statement in Python
numbers = [1, 2, 3, 4, 5]

# Iterate over each number in the list and print it
for num in numbers:
    print(num)
```

In this example:

- `numbers` is a list containing integers from 1 to 5.
- The `for` loop iterates over each item in the `numbers` list.
- During each iteration, the variable `num` represents the current item being processed.
- The `print(num)` statement prints each number to the console.

2 b.

```
# Define the number of rows
rows = 5

# Outer loop for each row
for i in range(1, rows + 1):
    # Inner loop for each column in the current row
    for j in range(1, i + 1):
        print(j, end=" ")  # Print the current number followed by a space
    print()  # Move to the next line after each row
```

2c.

In Python, a function is a block of reusable code that performs a specific task. Functions provide modularity and abstraction by allowing you to encapsulate a piece of code and execute it multiple times without rewriting the same code. Functions can accept input parameters, perform operations, and return results.

There are several types of functions in Python:

1. Built-in Functions:

Python provides many built-in functions that are readily available for use without the need for importing any module. Examples include `print()`, `len()`, `sum()`, `max()`, `min()`, etc.

```
  # Example of using built-in functions

  print("Hello, world!")  # prints "Hello, world!"

  nums = [1, 2, 3, 4, 5]

  print(len(nums))  # prints the length of the list `nums`, which is 5
```

2. User-defined Functions:

User-defined functions are created by the user to perform a specific task. They are defined using the `def` keyword followed by the function name, parameters (if any), and the block of code to be executed.

```
  # Example of defining and using a user-defined function

  def add_numbers(x, y):

      return x + y


  result = add_numbers(3, 5)

  print(result)  # prints 8
```

3. Lambda Functions (Anonymous Functions):

Lambda functions are small, anonymous functions defined using the `lambda` keyword. They can have any number of parameters but only one expression, which is evaluated and returned.

```
  # Example of using lambda functions

  square = lambda x: x ** 2

  print(square(5))  # prints 25
```

4. Recursive Functions:

Recursive functions are functions that call themselves within their own definition. They are useful for solving problems that can be broken down into smaller, similar sub-problems.

```
  # Example of a recursive function to calculate factorial

  def factorial(n):

      if n == 0:

          return 1

      else:

          return n * factorial(n - 1)


  print(factorial(5))  # prints 120 (5!)
```

3 a.

Sure, here are explanations of five commonly used string methods in Python along with examples:

1. **`upper()`**:

This method returns a copy of the string with all uppercase letters converted to uppercase.

```
  # Example of using the upper() method

  text = "hello, world!"

  uppercase_text = text.upper()

  print(uppercase_text)  # Output: HELLO, WORLD!
```

2. **`lower()`**:

The `lower()` method returns a copy of the string with all uppercase letters converted to lowercase.

```python
# Example of using the lower() method
text = "Hello, World!"
lowercase_text = text.lower()
print(lowercase_text)  # Output: hello, world!
```

3. **`strip()`**:

The `strip()` method removes any leading (spaces at the beginning) and trailing (spaces at the end) characters (whitespace by default).

```
# Example of using the strip() method
text = "   Hello, World!   "
stripped_text = text.strip()
print(stripped_text)  # Output: Hello, World!
```

4. **`split()`**:

The `split()` method splits a string into a list of substrings based on a delimiter. By default, the delimiter is a whitespace.

```
# Example of using the split() method
text = "apple banana cherry"
words = text.split()
print(words)  # Output: ['apple', 'banana', 'cherry']
```

5. **`join()`**:

The `join()` method joins the elements of an iterable (such as a list) into a single string, using the string as a separator.

```
# Example of using the join() method
```

```
   words = ['apple', 'banana', 'cherry']

   text = ', '.join(words)

   print(text)  # Output: apple, banana, cherry
```

3b.

Sets:

In Python, a set is an unordered collection of unique elements. Sets are mutable, which means you can add or remove elements from them. Sets are commonly used to perform mathematical set operations such as union, intersection, difference, and symmetric difference.

1. **Creating a Set**:

   You can create a set in Python by enclosing a comma-separated list of elements within curly braces `{}`.

```
   # Example of creating a set

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

   print(my_set)  # Output: {1, 2, 3, 4, 5}
```

2. **Unique Elements**:

   Sets contain only unique elements. If you try to add duplicate elements, they are ignored.

```
   # Example of unique elements in a set

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

   print(my_set)  # Output: {1, 2, 3, 4, 5}
```

3. **Accessing Elements**:

   Sets are unordered collections, so you cannot access elements by index. However, you can iterate over the elements of a set.

```
   # Example of accessing elements of a set

   my_set = {1, 2, 3, 4, 5}
   for element in my_set:
       print(element)
```

4. **Adding and Removing Elements**:

You can add elements to a set using the `add()` method, and remove elements using the `remove()` or `discard()` methods.

```
   # Example of adding and removing elements from a set
   my_set = {1, 2, 3}
   my_set.add(4)
   print(my_set)  # Output: {1, 2, 3, 4}

   my_set.remove(2)
   print(my_set)  # Output: {1, 3, 4}
```

5. **Set Operations**:

Sets support various mathematical operations such as union, intersection, difference, and symmetric difference.

```
   # Example of set operations
   set1 = {1, 2, 3}
   set2 = {3, 4, 5}

   # Union
   union_set = set1 | set2
   print(union_set)  # Output: {1, 2, 3, 4, 5}

   # Intersection
   intersection_set = set1 & set2
```

```
  print(intersection_set)  # Output: {3}


  # Difference

  difference_set = set1 - set2

  print(difference_set)  # Output: {1, 2}


  # Symmetric Difference

  symmetric_difference_set = set1 ^ set2

  print(symmetric_difference_set)  # Output: {1, 2, 4, 5
```

Dictionaries:

In Python, a dictionary is an unordered collection of key-value pairs. Each key in a dictionary must be unique, and it is used to access its corresponding value. Dictionaries are mutable, which means you can add, remove, or modify key-value pairs. They are widely used for storing and retrieving data efficiently based on keys.

1. **Creating a Dictionary**:

You can create a dictionary in Python by enclosing a comma-separated list of key-value pairs within curly braces `{}`.

```
  # Example of creating a dictionary

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

  print(my_dict)  # Output: {'name': 'John', 'age': 30, 'city': 'New York'}
```

2. **Accessing Elements**:

You can access the value associated with a key in a dictionary using square brackets `[]`.

```
  # Example of accessing elements of a dictionary

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

  print(my_dict["name"])  # Output: John
```

3. **Adding and Removing Elements**:

You can add new key-value pairs to a dictionary using assignment and remove key-value pairs using the `del` keyword or the `pop()` method.

```
# Example of adding and removing elements from a dictionary

my_dict = {"name": "John", "age": 30}

my_dict["city"] = "New York"  # Adding a new key-value pair

print(my_dict)  # Output: {'name': 'John', 'age': 30, 'city': 'New York'}


del my_dict["age"]  # Removing a key-value pair

print(my_dict)  # Output: {'name': 'John', 'city': 'New York'}
```

4. **Dictionary Methods**:

Python dictionaries provide various methods for performing operations such as accessing keys, values, items, etc.

```
# Example of dictionary methods

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


# Accessing keys

print(my_dict.keys())  # Output: dict_keys(['name', 'age', 'city'])


# Accessing values

print(my_dict.values())  # Output: dict_values(['John', 30, 'New York'])


# Accessing key-value pairs (items)

print(my_dict.items())  # Output: dict_items([('name', 'John'), ('age', 30), ('city', 'New York')])
```

5. **Dictionary Comprehensions**:

Similar to list comprehensions, dictionary comprehensions allow you to create dictionaries using a concise syntax.

```
   # Example of dictionary comprehension

   squares = {x: x ** 2 for x in range(5)}

   print(squares)  # Output: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

3 c.

List slicing in Python allows you to extract a portion of a list by specifying a start index, an end index (exclusive), and an optional step size. It provides a concise way to work with subsequences of lists.

Here's how list slicing works in Python with examples:

1. **Basic List Slicing**:

You can slice a list using the syntax `[start:end]`, where `start` is the index of the first element to include and `end` is the index of the first element to exclude.

```
   # Example of basic list slicing

   my_list = [1, 2, 3, 4, 5]


   # Extract elements from index 1 to index 3 (exclusive)

   sliced_list = my_list[1:4]

   print(sliced_list)  # Output: [2, 3, 4]
```

2. **Slicing with Negative Indices**:

You can use negative indices to slice a list from the end.

```
   # Example of slicing with negative indices

   my_list = [1, 2, 3, 4, 5]
   # Extract the last three elements
   sliced_list = my_list[-3:]
   print(sliced_list)  # Output: [3, 4, 5]
```

3. **Slicing with Step Size**:

   You can specify a step size to skip elements in the list while slicing.

```
   # Example of slicing with step size
   my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9]

   # Extract every second element starting from index 1
   sliced_list = my_list[1::2]
   print(sliced_list)  # Output: [2, 4, 6, 8]
```

4. **Slicing with Negative Step Size**:

   You can use a negative step size to reverse the order of elements in the sliced list.

```
   # Example of slicing with negative step size
   my_list = [1, 2, 3, 4, 5]

   # Reverse the list
   reversed_list = my_list[::-1]
   print(reversed_list)  # Output: [5, 4, 3, 2, 1]
```

4 a.

Sure, let me explain both concepts with examples:

i) **Inheritance**:

Inheritance is a feature in object-oriented programming that allows a class (subclass or derived class) to inherit properties and methods from another class (superclass or base class). This enables code reuse and promotes the creation of hierarchical relationships between classes.

```python
class Animal:
    def __init__(self, species):
        self.species = species

    def sound(self):
        pass

class Dog(Animal):
    def __init__(self, breed):
        super().__init__("Dog")
        self.breed = breed

    def sound(self):
        return "Woof!"

class Cat(Animal):
    def __init__(self, breed):
        super().__init__("Cat")
        self.breed = breed

    def sound(self):
        return "Meow!"



# Creating instances of subclasses
dog = Dog("Labrador")
cat = Cat("Siamese")

# Accessing properties and methods from the superclass
print(f"The {dog.species} of breed {dog.breed} says: {dog.sound()}")  # Output: The Dog of breed Labrador says: Woof!
print(f"The {cat.species} of breed {cat.breed} says: {cat.sound()}")  # Output: The Cat of breed Siamese says: Meow!
```

ii) **Operator Overloading**:

Operator overloading is a feature in Python that allows you to define custom behavior for operators such as `+`, `-`, `*`, `/`, etc., when applied to objects of user-defined classes. This enables you to define how operators should work with instances of your classes.

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"({self.x}, {self.y})"


# Creating instances of the Point class
point1 = Point(1, 2)
point2 = Point(3, 4)

# Adding two Point objects using the '+' operator
result = point1 + point2
print("Result of addition:", result)  # Output: Result of addition: (4, 6)
```

4 b.

In Python, when you work with files, you can specify different modes to control how the file should be opened and manipulated. Here are the various modes of opening files in Python along with examples:

1. **Read Mode (`'r'`)**:

This mode is used to open a file for reading only. It is the default mode when opening a file. If the file does not exist, it will raise a `FileNotFoundError`.

```
   # Example of opening a file in read mode
   with open("example.txt", "r") as file:
       content = file.read()
       print(content)
```

2. **Write Mode (`'w'`)**:

This mode is used to open a file for writing. If the file already exists, it will be truncated. If the file does not exist, a new file will be created.

```
# Example of opening a file in write mode

with open("example.txt", "w") as file:

    file.write("Hello, world!")
```

3. **Append Mode (`'a'`)**:

This mode is used to open a file for appending data. If the file does not exist, it will be created. If the file exists, new data will be added to the end of the file.

```
# Example of opening a file in append mode

with open("example.txt", "a") as file:

    file.write("\nThis is a new line appended.")
```

4. **Read and Write Mode (`'r+'`)**:

This mode is used to open a file for both reading and writing. It does not truncate the file, and the file pointer is placed at the beginning of the file.

```
# Example of opening a file in read and write mode

with open("example.txt", "r+") as file:

    content = file.read()

    file.write("\nAppending data in read and write mode.")
```

5. **Write and Read Mode (`'w+'`)**:

This mode is used to open a file for both reading and writing. It truncates the file and places the file pointer at the beginning of the file.

```
# Example of opening a file in write and read mode

with open("example.txt", "w+") as file:

    file.write("Writing data in write and read mode.")
    file.seek(0)  # Move the file pointer to the beginning of the file

    content = file.read()

    print(content)
```

5. a

i) **create**:

The `create` method typically refers to creating a new database table. It's used to define the structure of the table, including column names, data types, constraints, etc. This method is usually executed as a SQL query.

```
  # Example of creating a new table in a SQLite database using sqlite3 module

  import sqlite3

  # Connect to the database (or create it if it doesn't exist)

  conn = sqlite3.connect('example.db')


  # Create a cursor object to execute SQL queries

  cursor = conn.cursor()


  # Define a SQL query to create a new table

  create_table_query = '''

  CREATE TABLE IF NOT EXISTS students (

      id INTEGER PRIMARY KEY,

      name TEXT NOT NULL,

      age INTEGER

  );


  # Execute the SQL query to create the table

  cursor.execute(create_table_query)
```

```
# Commit the changes and close the connection
conn.commit()
conn.close()
```

ii) **Insert**:

The `insert` method is used to add new records (rows) to a database table. It's typically executed as a SQL `INSERT` statement, specifying the values to be inserted into each column of the table.

```
# Example of inserting data into a table in a SQLite database using sqlite3 module
import sqlite3

# Connect to the database
conn = sqlite3.connect('example.db')
cursor = conn.cursor()

# Define a SQL query to insert data into the table
insert_query = '''
INSERT INTO students (name, age)
VALUES (?, ?);

# Data to be inserted
data = ('Alice', 25)

# Execute the SQL query to insert data into the table
cursor.execute(insert_query, data)
```

```
  # Commit the changes and close the connection

  conn.commit()

  conn.close()
```

iii) **execute**:

The `execute` method is used to execute SQL queries on a database connection. It's a general-purpose method that can be used to execute any valid SQL statement, such as `CREATE TABLE`, `INSERT`, `UPDATE`, `DELETE`, etc.

```
  # Example of executing a SELECT query in a SQLite database using sqlite3 module

  import sqlite3

  # Connect to the database

  conn = sqlite3.connect('example.db')

  cursor = conn.cursor()


  # Define a SQL query to select data from the table

  select_query = '''

  SELECT * FROM students;

  # Execute the SQL query to select data from the table

  cursor.execute(select_query)


  # Fetch the results and print them

  results = cursor.fetchall()

  for row in results:

     print(row)


  # Close the connection
  conn.close()
```

iv) **fetchall**:

The `fetchall` method is used to retrieve all rows of a query result set as a list of tuples. It's typically used after executing a `SELECT` query to fetch the selected data from the database.

```
  # Example of fetching all rows from a query result set in a SQLite database using sqlite3 module
  import sqlite3
  # Connect to the database
  conn = sqlite3.connect('example.db')
  cursor = conn.cursor()
  # Define a SQL query to select data from the table
  select_query = '''
  SELECT * FROM students;
  # Execute the SQL query to select data from the table
  cursor.execute(select_query)
  # Fetch all rows from the query result set
  results = cursor.fetchall()
  # Print the fetched results
  for row in results:
      print(row)


  # Close the connection
  conn.close()
```

5 b.

Handling missing values is an essential task in data analysis and machine learning. Python provides several libraries and techniques to handle missing values effectively. Here are some common approaches along with examples:

1. **Removing Rows or Columns**:

One approach to handling missing values is to remove rows or columns containing missing values entirely. This approach is suitable when missing values are relatively few compared to the size of the dataset.

```
import pandas as pd

# Example of removing rows with missing values using pandas

data = {'A': [1, 2, None, 4],

        'B': [5, None, 7, 8]}

df = pd.DataFrame(data)


# Remove rows with missing values

df_cleaned = df.dropna()

print(df_cleaned)
```

2. **replacing missing values**:

Imputation involves replacing missing values with a suitable substitute, such as the mean, median, mode, or a constant value. This approach allows you to retain the information from the remaining data.

```
import pandas as pd

# Example of imputing missing values with the mean using pandas

data = {'A': [1, 2, None, 4],

        'B': [5, None, 7, 8]}

df = pd.DataFrame(data)
```

```
  # Impute missing values with the mean
  df_imputed = df.fillna(df.mean())
  print(df_imputed)
  `
```

3. **Forward Fill or Backward Fill**:

Forward fill (ffill) and backward fill (bfill) involve filling missing values with the preceding or succeeding non-missing values, respectively. This method is suitable for time series data.

```
  import pandas as pd
  # Example of forward fill and backward fill using pandas
  data = {'A': [1, None, 3, None, 5],
          'B': [None, 2, None, 4, None]}
  df = pd.DataFrame(data)

  # Forward fill missing values
  df_ffilled = df.ffill()
  print("Forward Fill:")
  print(df_ffilled)

  # Backward fill missing values
  df_bfilled = df.bfill()
  print("\nBackward Fill:")
  print(df_bfilled)
```

5 c.

Reading and writing data in text format in Python can be done using built-in functions or libraries such as `open()`, `read()`, `write()`, and `close()` for basic operations, or more specialized libraries like `csv`, `json`, or `pickle` for structured data formats. Here's an explanation of how to perform reading and writing operations in text format:

1. **Reading Text Data**:

To read data from a text file in Python, you can use the `open()` function with the mode `'r'` (read mode). Then you can use methods like `read()`, `readline()`, or `readlines()` to read the content of the file.

```
# Example of reading data from a text file
with open('data.txt', 'r') as file:
    content = file.read()
    print(content)
```

2. **Writing Text Data**:

To write data to a text file in Python, you can use the `open()` function with the mode `'w'` (write mode). Then you can use the `write()` method to write data to the file.

```
# Example of writing data to a text file
with open('output.txt', 'w') as file:
    file.write("Hello, world!")
```

3. **Reading and Writing CSV Files**:

If you are working with CSV (Comma-Separated Values) files, you can use the `csv` module in Python, which provides functionality specifically for reading and writing CSV files.

```
  import csv


  # Reading data from a CSV file
  with open('data.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
      print(row)


  # Writing data to a CSV file
  with open('output.csv', 'w', newline='') as file:
    writer = csv.writer(file)
    writer.writerow(['Name', 'Age', 'City'])
    writer.writerow(['Alice', 25, 'New York'])
    writer.writerow(['Bob', 30, 'Los Angeles'])
```

4. **Reading and Writing JSON Files**:

If you are working with JSON (JavaScript Object Notation) files, you can use the `json` module in Python to read and write JSON data.

```
  import json
  # Reading data from a JSON file
  with open('data.json', 'r') as file:
    data = json.load(file)
    print(data)


  # Writing data to a JSON file
```

```
data = {'name': 'Alice', 'age': 25, 'city': 'New York'}

with open('output.json', 'w') as file:

    json.dump(data, file)
```

6 a.

In Python, when working with data frames or tables, merging refers to combining two or more data frames based on one or more common keys. There are several merge methods available, including outer, left, and right merges. Let's explain each of these merge methods:

i) **Outer Merge**:

An outer merge, also known as a full outer join, combines the rows from both data frames, keeping all rows from both data frames regardless of whether there is a match on the key(s). Missing values are filled with NaN (Not a Number) for columns from the data frame that does not have a match.

![Outer Merge](https://datavizcatalogue.com/methods/images/anatomy/outer_join.png)

```
import pandas as pd

# Example of outer merge in pandas

df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value1': [1, 2, 3]})

df2 = pd.DataFrame({'key': ['B', 'C', 'D'], 'value2': [4, 5, 6]})


merged_df = pd.merge(df1, df2, on='key', how='outer')

print(merged_df)
```

ii) **Left Merge**:

A left merge, also known as a left outer join, includes all rows from the left data frame and matching rows from the right data frame. If there is no match, NaN values are filled for columns from the right data frame.

```
import pandas as pd

# Example of left merge in pandas

df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value1': [1, 2, 3]})

df2 = pd.DataFrame({'key': ['B', 'C', 'D'], 'value2': [4, 5, 6]})


merged_df = pd.merge(df1, df2, on='key', how='left')

print(merged_df)
```

iii) **Right Merge**:

A right merge, also known as a right outer join, includes all rows from the right data frame and matching rows from the left data frame. If there is no match, NaN values are filled for columns from the left data frame.

```
import pandas as pd

# Example of right merge in pandas

df1 = pd.DataFrame({'key': ['A', 'B', 'C'], 'value1': [1, 2, 3]})

df2 = pd.DataFrame({'key': ['B', 'C', 'D'], 'value2': [4, 5, 6]})


merged_df = pd.merge(df1, df2, on='key', how='right')

print(merged_df)
```

6 b.

Certainly! Let's explore each of the data transformation methods:

i) **Discretization and Binning**:

Discretization and binning involve dividing continuous numerical data into discrete intervals or bins. This is useful for converting continuous data into categorical or ordinal data, simplifying the analysis and interpretation of the data.

```
import pandas as pd

# Example of discretization and binning using pandas

# Create a DataFrame with continuous numerical data

data = {'age': [22, 35, 47, 56, 32, 64, 40, 30]}

df = pd.DataFrame(data)


# Define bin edges

bins = [0, 30, 40, 50, 100]


# Define bin labels

labels = ['Young', 'Adult', 'Middle-aged', 'Senior']


# Discretize the age column into bins and assign labels

df['age_group'] = pd.cut(df['age'], bins=bins, labels=labels)


print(df)
```

ii) **Detecting and Filtering Outliers**:

Outliers are data points that significantly differ from other observations in a dataset. Detecting and filtering out outliers is important for ensuring the quality and reliability of the data analysis results.

```
import pandas as pd

# Example of detecting and filtering outliers using pandas

# Create a DataFrame with numerical data

data = {'value': [10, 15, 12, 8, 200, 14, 11, 9]}
```

```
  df = pd.DataFrame(data)


  # Calculate the z-score for each value

  z_scores = (df['value'] - df['value'].mean()) / df['value'].std()


  # Define a threshold for outliers (e.g., z-score > 3)

  threshold = 3


  # Filter out outliers

  df_filtered = df[abs(z_scores) <= threshold]

  print(df_filtered)
```

iii) **Renaming Axis and Indexes**:

Renaming axis and indexes allows you to change the labels of rows (index) and columns (axis) in a DataFrame, making the data more readable and meaningful.

```
  import pandas as pd

  # Example of renaming axis and indexes using pandas

  # Create a DataFrame with numerical data

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

  df = pd.DataFrame(data)

  # Rename the columns

  df.columns = ['Column1', 'Column2', 'Column3']

  # Rename the index

  df.index = ['Row1', 'Row2', 'Row3']


  print(df)
```

7 a.

Web scraping in Python refers to the process of extracting data from websites. It involves fetching HTML content from web pages and parsing it to extract the desired information, such as text, links, images, or structured data. The `requests` module in Python is commonly used for making HTTP requests to web servers and fetching web pages.

Here are some common methods available in the `requests` module along with an example program:

1. **GET Method**:

   The GET method is used to request data from a specified resource. It sends the data in the URL query string.

```
   import requests

   # Example of using the GET method in requests

   response = requests.get('https://www.example.com')

   print(response.text)  # Print the HTML content of the web page
```

2. **POST Method**:

   The POST method is used to submit data to be processed to a specified resource. It sends the data in the body of the request.

```
   import requests

   # Example of using the POST method in requests

   data = {'username': 'john', 'password': 'secret'}

   response = requests.post('https://www.example.com/login', data=data)

   print(response.text)  # Print the response from the server
```

3. **Headers**:

   Headers are used to provide additional information about the request, such as user-agent, content-type, or authorization.

```
   import requests

   # Example of using headers in requests

   headers = {'User-Agent': 'Mozilla/5.0'}

   response = requests.get('https://www.example.com', headers=headers)

   print(response.text)  # Print the HTML content of the web page
```

4. **Query Parameters**:

   Query parameters are used to pass additional information in the URL query string.

```
   import requests

   # Example of using query parameters in requests

   params = {'page': 2, 'limit': 10}

   response = requests.get('https://api.example.com/data', params=params)

   print(response.json())  # Print the JSON response from the server
```

5. **Response Content**:

   The response content can be accessed using the `text` attribute (for HTML content) or the `json()` method (for JSON content).

```
   import requests

   # Example of accessing response content in requests

   response = requests.get('https://www.example.com')

   print(response.text)  # Print the HTML content of the web page
```

7 b.

Web scraping with BeautifulSoup involves extracting data from HTML and XML files. BeautifulSoup is a Python library that provides tools for parsing HTML and XML documents, navigating the parse tree, and extracting data efficiently.

Here's a step-by-step guide to implementing web scraping with BeautifulSoup:

1. **Import Libraries**:

   Import the necessary libraries, including `requests` for fetching web pages and `BeautifulSoup` for parsing HTML content.

```
  import requests

  from bs4 import BeautifulSoup
```

2. **Fetch Web Page**:

   Use the `requests.get()` function to fetch the HTML content of the web page you want to scrape.

```
  url = 'https://example.com'

  response = requests.get(url)
```

3. **Parse HTML**:

   Create a BeautifulSoup object by passing the HTML content and specifying the parser to use (e.g., `'html.parser'`, `'lxml'`, `'html5lib'`).

```
  soup = BeautifulSoup(response.text, 'html.parser')
```

4. **Extract Data**:

   Use BeautifulSoup methods to navigate the HTML structure and extract the desired data. You can use methods like `find()`, `find_all()`, `select()`, etc., along with CSS selectors or XPath expressions.

```
  # Example of extracting all links (<a> tags) from the web page

  links = soup.find_all('a')

  for link in links:

      print(link.get('href'))
```

5. **Data Processing**:

   Process the extracted data as needed. You may need to clean or format the data before further processing or analysis.

```
  # Example of extracting and processing text from <p> tags
```

```
   paragraphs = soup.find_all('p')
   for p in paragraphs:
       print(p.text)
```

6. **Save or Store Data**:

Finally, save or store the extracted data for further analysis or use. You can save the data to a file, database, or data structure (e.g., list, dictionary).

```
   # Example of saving extracted data to a file
   with open('data.txt', 'w') as file:
       for link in links:
           file.write(link.get('href') + '\n')
```

Here's a complete example of web scraping with BeautifulSoup to extract all links from a web page:

```
import requests
from bs4 import BeautifulSoup
# Fetch web page
url = 'https://example.com'
response = requests.get(url)
# Parse HTML
soup = BeautifulSoup(response.text, 'html.parser')
# Extract all links
links = soup.find_all('a')
# Print extracted links
for link in links:
    print(link.get('href'))
```

8 a.

In NumPy, arrays are the fundamental data structure for representing and manipulating numerical data. Arrays in NumPy are homogeneous, meaning that all elements in an array must be of the same data type. They offer efficient storage and operations for large datasets and are widely used in scientific computing, data analysis, and machine learning.

1. **ndarray.shape**:

This attribute returns a tuple representing the shape of the array, i.e., the dimensions (number of rows and columns) of the array.

```
import numpy as np

# Create a 2D array

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


# Get the shape of the array

print(arr.shape)  # Output: (2, 3)
```

2. **ndarray.ndim**:

This attribute returns the number of dimensions (axes) of the array.

```
import numpy as np

# Create a 3D array

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

# Get the number of dimensions of the array

print(arr.ndim)  # Output: 3
```

3. **ndarray.size**:

This attribute returns the total number of elements in the array.

```
import numpy as np
```

```
# Create a 1D array

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

# Get the total number of elements in the array

print(arr.size)  # Output: 5
```

4. **ndarray.dtype**:

   This attribute returns the data type of the elements in the array.

```
import numpy as np

# Create an array with float elements

arr_float = np.array([1.0, 2.0, 3.0])

# Get the data type of the elements in the array

print(arr_float.dtype)  # Output: float64

# Create an array with integer elements

arr_int = np.array([1, 2, 3])

# Get the data type of the elements in the array

print(arr_int.dtype)  # Output: int64
```

5. **ndarray.itemsize**:

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

```
import numpy as np

# Create an array with float elements

arr = np.array([1.0, 2.0, 3.0])


# Get the size of each element in the array

print(arr.itemsize)  # Output: 8 (float64 has 8 bytes)
```

8 b

```
#Array manipulation, Searching,Sorting and splitting

#Array manipulation

#importing numpy package

import numpy as np

#Concatinatin of arrays

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

b=np.array([[5,6]])

print(&quot;concate with axis=0:&quot;,np.concatenate((a,b),axis=0))

print(&quot;concate with axis=1:&quot;,np.concatenate((a,b.T),axis=1))

print(np.concatenate((a,b),axis=None))

a=np.array([[12,4,5],[23,45,66],[45,34,23]])

b=np.array([[1,40,50],[2,4,6],[4,3,2]])

#Verticle stacking

print(np.vstack((a,b)))

#Horizontal stacking

print(np.hstack((a,b)))

print(a.reshape(3,3))

##Sorting sort :Return a sorted copy of an array.

#ndarray.sort : Method to sort an array in-place.

#argsort: Indirect sort. Returns the indices that would sort an array.

#numpy.sort_complex: Sort a complex array using the real part first, then the imaginary part.
```

```
#Sorting

#importing numpy package
```

```
import numpy as np

a=np.array([[1,4],[3,1]])

print(a)

print(&quot;Sorted array:\n&quot;,np.sort(a))

print(&quot;\n sorted flattened array:\n&quot;,np.sort(a,axis=0))

x=np.array([3,1,2])

print(&quot;\n indices that would sort an array&quot;,np.argsort(x))

print(&quot;\n Sorting complex number:&quot;,np.sort_complex([[3 + 4j, 1 - 2j,

5 + 1j, 2 + 2j]]))

#Searching

import numpy as np

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

x=np.where(arr==4)

print(x)

arr=np.array([6,7,8,9])

x=np.searchsorted(arr,5)

print(x)

arr=np.array([1,3,5,7])

x=np.searchsorted(arr,[2,4,6])

print(x)


#Splitting

import numpy as np

x=np.arange(9.0)

print(x)
print(np.split(x,3))

print(np.split(x,[3,5,6,10]))

x=np.arange(9)

print(np.array_split(x,4))

a=np.array([[1,3,5,7,9,11],

 [2,4,6,8,10,12]])

print(&quot;Splitting along horizontal axis into 2 parts:\n&quot;,np.hsplit(a,2))

print(&quot;\n Splitting along vertical axis into 2 parts:\n&quot;,np.vsplit(a,2))


# Broadcasting

import numpy as np

x=np.arange(4)

print(x)

y=np.ones(5)

print(y)

xx=x.reshape(2,2)

print(xx)

z=np.ones((3,4))

print(z)

print(x.shape)

a=np.array([0.0,10.0,20.0,30.0])

b=np.array([1.0,2.0,3.0])

a[:,np.newaxis]+b
```

9 a.

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It provides a MATLAB-like interface and supports a wide range of plots and charts, including line plots, scatter plots, bar plots, histograms, pie charts, 3D plots, and more. Matplotlib is highly customizable, allowing users to fine-tune every aspect of their plots.

Here are the main components and concepts of Matplotlib:

1. **Figure and Axes**:

   - A Figure is the top-level container that holds all elements of a plot.

   - An Axes represents a single plot (e.g., a line plot, scatter plot) within a Figure.

   - A Figure can contain one or more Axes.

2. **Plotting Functions**:

   - Matplotlib provides a wide range of plotting functions, such as `plot()`, `scatter()`, `bar()`, `hist()`, `pie()`, etc., to create different types of plots.

   - These functions generate plots on an Axes within a Figure.

3. **Customization**:

   - Matplotlib allows extensive customization of plots, including setting titles, labels, colors, markers, line styles, gridlines, legends, etc.

   - Users can customize every aspect of their plots using various methods and attributes provided by Matplotlib.

4. **Subplots**:

   - Subplots allow creating multiple plots within the same Figure.

   - Matplotlib provides the `subplot()` function to arrange plots in a grid layout.

5. **Styles and Themes**:

  - Matplotlib supports different styles and themes to customize the appearance of plots.

  - Users can choose from predefined styles (e.g., 'ggplot', 'seaborn', 'bmh') or create custom styles.

9 b.

Line plot;

```
#plotting

import numpy as np

import matplotlib.pyplot as plt

x=np.arange(0,3*np.pi,0.1)

print(&quot;x=&quot;,x)

y_sin=np.sin(x)

y_cos=np.cos(x)

plt.plot(x,y_sin)

plt.plot(x,y_cos)

plt.xlabel(&#39;x values&#39;)

plt.ylabel(&#39;y sine and cosine values&#39;)

plt.title(&#39;Sine and Cosine&#39;)

plt.legend([&#39;Sine&#39;,&#39;Cosine&#39;])

plt.show()
```

histogram:

```
import matplotlib.pyplot as plt

import numpy as np

# Generate random data for the histogram

data = np.random.randn(1000)
```

```
#print(data)

# Plotting a basic histogram

plt.hist(data, bins=30, color=&#39;skyblue&#39;, edgecolor=&#39;black&#39;)

# Adding labels and title

plt.xlabel(&#39;Values&#39;)

plt.ylabel(&#39;Frequency&#39;)

plt.title(&#39;Basic Histogram&#39;)

# Display the plot

plt.show()
```

Scatter plot:

```
import matplotlib.pyplot as plt

x = [5,7,8,7,2,17,2,9,4,11,12,9,6]

y = [99,86,87,88,111,86,103,87,94,78,77,85,86]

plt.scatter(x, y)

plt.show()
```

10 a.

Seaborn is a powerful data visualization library built on top of Matplotlib in Python. It provides a high-level interface for creating attractive and informative statistical graphics. Seaborn simplifies the process of creating complex visualizations by providing easy-to-use functions for common tasks and by integrating seamlessly with Pandas data structures.

Here are the main concepts and features of Seaborn:

1. **Statistical Visualization**:

- Seaborn specializes in statistical visualization, focusing on creating informative plots that convey relationships and patterns in the data.

- It offers a wide range of plots, including univariate and bivariate plots, categorical plots, distribution plots, regression plots, and matrix plots.

2. **Integration with Pandas**:

- Seaborn seamlessly integrates with Pandas DataFrames, allowing users to plot data directly from Pandas data structures.

- This integration simplifies the process of data manipulation and visualization.

3. **High-Level Interface**:

- Seaborn provides a high-level interface for creating complex plots with minimal code.

- It offers easy-to-use functions with sensible default settings, making it accessible to users with varying levels of expertise.

4. **Attractive Aesthetics**:

- Seaborn comes with attractive default styles and color palettes that enhance the visual appeal of plots.

- Users can easily customize the aesthetics of plots using built-in themes and color palettes or by defining custom styles.

5. **Complex Plot Types**:

- Seaborn supports a wide range of complex plot types, such as FacetGrids, PairGrids, and JointGrids, for visualizing relationships between multiple variables.

10 b.

Time series analysis involves analyzing and modeling data that varies over time. This type of data is commonly encountered in various domains, including finance, economics, weather forecasting, and signal processing. Pandas, a powerful library in Python for data manipulation and analysis, provides extensive support for time series data through its `DatetimeIndex` and specialized time series functions.

Here are the main concepts of time series analysis with Pandas:

1. **DatetimeIndex**:

   - Pandas provides the `DatetimeIndex` data structure for representing time series data.

   - A `DatetimeIndex` is a specialized index that stores timestamps as its elements.

   - It allows for efficient manipulation, slicing, and selection of time series data.

2. **Time Resampling**:

   - Time resampling involves changing the frequency of the time series data.

   - Pandas provides the `resample()` method, which allows users to aggregate or downsample time series data to different frequencies, such as daily, weekly, monthly, etc.

   - This is useful for summarizing data over different time periods or aligning data with different reporting frequencies.

3. **Time Shifting**:

   - Time shifting involves shifting the timestamps of the time series data by a specified number of periods.

   - Pandas provides the `shift()` method, which allows users to shift the index of the time series forward or backward by a specified number of periods.

   - This is useful for comparing past and future values or aligning data for analysis.

4. **Rolling Windows**:

   - Rolling window operations involve calculating summary statistics over a moving window of time.

   - Pandas provides the `rolling()` method, which allows users to create a rolling window object and apply aggregation functions, such as mean, sum, min, max, etc., over the window.

   - This is useful for smoothing data, identifying trends, and detecting outliers in time series data.

5. **Time Zone Handling**:

   - Time zone handling involves converting timestamps between different time zones.

   - Pandas provides functions for working with time zones, including `tz_localize()` for localizing timestamps to a specific time zone and `tz_convert()` for converting timestamps between time zones.

   - This is useful for analyzing data collected from different geographic locations or handling daylight saving time changes.