M.C.A. - I (Management)
IT - 21 : PYTHON PROGRAMMING
(2020 Pattern) (Semester - II)

Q1.MCQ

a) If zoo = [‘lion’, tiger’], what will be zoo*2?

i) [‘lion’, ‘tiger’, ‘lion’, ‘tiger’]

b) For dictionary d = {plum : 0.66, pears : 1.25, oranges : 0.49} which ofthe following statement correctly updates the price of oranges to 0.52?i)

i)d [oranges] = 0.52

c) What is the correct syntax for creating single valued tuple?

iii) (a,)

d) The length of sys. argv is

iv)Total number of excluding the file name

e) Which of these definitions correctly describes a module?

Design and implementation of specific functionality tobe incorporated into a program.

f) To include use of functions which are present in random library, we mustuse the option

iv)import random

f) _____is a string literal denoted by triple quote for providing thespecifications of certain program elements.

ii)docstring

g) The correct way of inheriting a derived class from base class is:

i) Class Derived (Base):

h) The + operator is overloaded using the method

ii) --add-- ( )

i) The syntax for using super( ) in derived class__init__( )method definitionlooks like

iv) Super ( ).--init-- (base class parameters)

j) The character___and___ matches the start and end of the string,

respectively.

iv)^ and $

k) Consider a five 21-12-2016. zip. The regular expression pattern to extractdate from filename is

i) ([0-9] {2} \- [0-9] {2} \- [0-9] {4})

l) When will the else part of the try-except-else be executed?
iii)When there is no exception

m) When is the finally block executed?

ii)always

n) Which of the following is not a valid attribute of the file object file-handler.

ii)file-handler.size

o) Which of the following command is used in mango shell to show all thedatabases.

ii)Show dbs

p) The function that returns its arguments with a modified shape and themethod that modifies the arry itself respectively in Numpy are

iv) reshape, resize

q) Which of the following is used for line graph?

ii)plt. plot ( )

r) Which of the following statement is false?

i) ndarray is also known as the axis array

s) The method used in pandas to return top 5 rows

iv)head ( )

Q2) a) Write a program that accept the string from user and display the same string after removing vowels from it. [3]

Here's a Python program that accepts a string from the user and displays the string after removing all the vowels from it:

```python
def remove_vowels(string):
    vowels = ['a', 'e', 'i', 'o', 'u']
    result = ""
    for char in string:
        if char.lower() not in vowels:
            result += char
    return result

user_input = input("Enter a string: ")
result = remove_vowels(user_input)
print("String after removing vowels:", result)
```

In this program, we define a function `remove_vowels()` that takes a string as an argument. It initializes an empty string `result` to store the characters without vowels. Then, it iterates over each character in the input string. If the lowercase character is not found in the list of vowels, it is added to the `result` string. Finally, the modified string is returned.

The program prompts the user to enter a string using the `input()` function and assigns it to the variable `user_input`. Then, it calls the `remove_vowels()` function with `user_input` as the argument and stores the returned result in the `result` variable. Finally, it displays the modified string by printing `result`.

Please note that this program considers the English alphabet and assumes the removal of both lowercase and uppercase vowels.

b) Explain possitional and keyword arguments in function. Use suitable example. [3]

In Python, when defining a function, you can specify parameters that can be passed to the function when it is called. These parameters can be classified as positional arguments and keyword arguments.

1. Positional Arguments:
Positional arguments are the arguments that are passed to a function based on their position or order. The number of positional arguments and their order must match the function definition.

Here's an example:

```python
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet("Alice", 25)
```

In this example, the `greet()` function has two positional arguments: `name` and `age`. When the function is called with `greet("Alice", 25)`, the string `"Alice"` is passed as the first argument (corresponding to `name`), and the integer `25` is passed as the second argument (corresponding to `age`). The function then prints the greeting message using these values.

2. Keyword Arguments:
Keyword arguments are passed to a function with their corresponding parameter names explicitly specified. This allows you to pass arguments out of order, as long as you specify which argument corresponds to which parameter.

Here's an example:

```python
def greet(name, age):
    print(f"Hello, {name}! You are {age} years old.")

greet(age=25, name="Alice")
```

In this example, the function `greet()` is called with keyword arguments: `age=25` and `name="Alice"`. By specifying the parameter names along with the values, the order of the arguments doesn't matter. The function will still correctly assign the values to the corresponding parameters.

Using keyword arguments can make the code more readable and self-explanatory, especially when dealing with functions that have multiple arguments. It also provides flexibility in case you want to omit some arguments or provide default values for certain parameters.

It's worth noting that you can mix positional and keyword arguments in a function call. However, all positional arguments must be placed before keyword arguments.

```python
greet("Alice", age=25)
```

In this example, `"Alice"` is passed as a positional argument, and `age=25` is passed as a keyword argument.

Remember to choose between positional and keyword arguments based on the specific needs of your program and the desired level of flexibility in function calls.

c) Describe instance variable, static variable and local variables in python object oriented programming with example. [4]

In object-oriented programming (OOP) with Python, we have three types of variables: instance variables, static variables, and local variables.

1. Instance Variables:
Instance variables are unique to each instance of a class. They are defined within a class but outside of any methods or functions. Each instance of the class has its own copy of instance variables, and their values can vary between instances.

Here's an example:

```python
class Car:
    def __init__(self, color):
        self.color = color

    def drive(self):
```

```
        print(f"The {self.color} car is driving.")

car1 = Car("blue")
car2 = Car("red")

car1.drive()  # Output: The blue car is driving.
car2.drive()  # Output: The red car is driving.
```

In this example, the `Car` class has an instance variable called `color`. When creating instances of the `Car` class (`car1` and `car2`), we pass different colors as arguments. Each instance has its own `color` attribute, and calling the `drive()` method on each instance prints the corresponding color.

2. Static Variables:
Static variables, also known as class variables, are shared among all instances of a class. They are defined within the class but outside of any methods or functions, just like instance variables. However, static variables are declared using the class name, not `self`, and their values are shared by all instances of the class.

Here's an example:

```python
class Car:
    wheels = 4

    def __init__(self, color):
        self.color = color

    def drive(self):
        print(f"The {self.color} car with {Car.wheels} wheels is driving.")

car1 = Car("blue")
car2 = Car("red")

car1.drive()  # Output: The blue car with 4 wheels is driving.
car2.drive()  # Output: The red car with 4 wheels is driving.
```

In this example, the `Car` class has a static variable called `wheels` with a value of 4. The `wheels` variable is accessed using `Car.wheels` within the class and is shared by all instances of the `Car` class. The `drive()` method uses this static variable to print the number of wheels along with the car's color.

3. Local Variables:
Local variables are defined within a method or function and are accessible only within that specific scope. They are temporary variables that hold values during the execution of the method or function and are discarded once the execution completes.

Here's an example:

```python
def multiply(a, b):
    result = a * b
    return result
```

```
product = multiply(5, 3)
print(product)  # Output: 15
```

In this example, the `multiply()` function takes two parameters `a` and `b`. Inside the function, a local variable `result` is declared to store the product of `a` and `b`. The value of `result` is returned from the function and stored in the `product` variable. The `result` variable is local to the `multiply()` function and is not accessible outside of it.

Local variables are temporary and exist only within the scope of the method or function where they are defined. Instance variables and static variables, on the other hand, have broader scopes and are accessible throughout the class and its instances, with static variables being shared among all instances.

OR

a) Compare List and tuple with suitable example. [3]

In Python, both lists and tuples are used to store collections of items. However, there are some key differences between them. Here's a comparison of lists and tuples with suitable examples:

1. Mutability:
Lists are mutable, which means their elements can be modified after creation. Elements can be added, removed, or changed using various methods and operations.

Example of a list:

```python
my_list = [1, 2, 3, 4]
my_list[2] = 5
print(my_list)  # Output: [1, 2, 5, 4]
```

In this example, we create a list `my_list` with four elements. We then modify the element at index 2 by assigning it a new value. The list is mutable, so the change is reflected in the list.

Tuples, on the other hand, are immutable. Once a tuple is created, its elements cannot be modified. Any attempt to modify a tuple will result in an error.

Example of a tuple:

```python
my_tuple = (1, 2, 3, 4)
my_tuple[2] = 5  # This will raise an error
```

In this example, we create a tuple `my_tuple` with four elements. When trying to modify the element at index 2, an error is raised because tuples are immutable.

2. Syntax:
Lists are defined using square brackets (`[]`), and elements are separated by commas.

Example of a list:

```python
```

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

Tuples, on the other hand, are defined using parentheses (`()`) or without any brackets at all. Elements are also separated by commas.

Example of a tuple:

```python
my_tuple = (1, 2, 3, 4)
```

It's important to note that when creating a tuple with a single element, a trailing comma is required to differentiate it from parentheses used for grouping.

Example of a tuple with a single element:

```python
single_tuple = (1,)  # Note the trailing comma
```

3. Usage:
Lists are commonly used when you have a collection of items that may change over time. They provide flexibility due to their mutability, allowing you to add, remove, or modify elements.

Tuples, on the other hand, are often used when you have a collection of items that should not be modified. They can be useful for representing fixed data, such as coordinates or records, where immutability is desired.

Example usage of lists and tuples:

```python
# List example
fruits = ["apple", "banana", "orange"]
fruits.append("grape")
print(fruits)  # Output: ["apple", "banana", "orange", "grape"]

# Tuple example
coordinates = (3, 4)
x, y = coordinates
print(x, y)  # Output: 3 4
```

In this example, we use a list to store a collection of fruits. We append a new fruit to the list using the `append()` method. On the other hand, we use a tuple to store a pair of coordinates. We can unpack the tuple into individual variables `x` and `y`.

Both lists and tuples have their own strengths and use cases. Lists are more flexible but consume more memory due to their mutability, while tuples provide immutability and can be more efficient in certain scenarios. The choice between them depends on the specific requirements of your program.

b) Explain lambda with example. [3]

In Python, a lambda function is a small, anonymous function that is defined without a name. It is also known as an "anonymous function" because it doesn't require a def statement or a specific name.

Lambda functions are typically used when you need a simple, one-line function without defining a separate named function.

The general syntax of a lambda function is:

```python
lambda arguments: expression
```

Here's an example to demonstrate the usage of lambda functions:

```python
# Regular named function
def add(a, b):
    return a + b

result = add(2, 3)
print(result)  # Output: 5

# Equivalent lambda function
add_lambda = lambda a, b: a + b

result = add_lambda(2, 3)
print(result)  # Output: 5
```

In this example, we have a regular named function called `add()` that takes two arguments `a` and `b` and returns their sum. We can call this function by passing the arguments and assign the returned value to the `result` variable.

We then define an equivalent lambda function using the lambda keyword. The lambda function takes the same two arguments `a` and `b` and returns their sum using the expression `a + b`. We assign this lambda function to the variable `add_lambda`. We can call this lambda function in the same way as the named function, passing the arguments and assigning the result to the `result` variable.

Lambda functions are commonly used in combination with built-in functions like `map()`, `filter()`, and `reduce()`, which expect a function as an argument. Instead of defining a separate named function, you can use lambda functions inline to provide the necessary functionality. This helps in writing more concise and readable code.

Here's an example using the `map()` function with a lambda function:

```python
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)  # Output: [1, 4, 9, 16, 25]
```

In this example, we have a list of numbers. We use the `map()` function to apply a lambda function to each element of the list. The lambda function takes an argument `x` and returns its square using the expression `x ** 2`. The `map()` function returns an iterator, which we convert to a list using `list()` to get the squared numbers. The output is a list of the squared numbers `[1, 4, 9, 16, 25]`.

c) Create class called, library with data attributes like Acc-number publisher, title and

author, the methods of the class should include [4]
i) Read ( ) - Acc- number, title, author, publisher.
ii) Compute ( ) - to accept the number of day late, calculate and display the fine charged at the rate of Rupees 5/- per day.
iii) Display the data

Here's an example of a `Library` class that includes the data attributes mentioned (`AccNumber`, `Publisher`, `Title`, and `Author`) along with the specified methods (`Read()`, `Compute()`, and `Display()`):

```python
class Library:
    def __init__(self, acc_number, publisher, title, author):
        self.AccNumber = acc_number
        self.Publisher = publisher
        self.Title = title
        self.Author = author

    def Read(self):
        print("Enter the following details:")
        self.AccNumber = input("Access Number: ")
        self.Publisher = input("Publisher: ")
        self.Title = input("Title: ")
        self.Author = input("Author: ")

    def Compute(self, days_late):
        fine = days_late * 5
        print(f"Fine charged: {fine} Rupees.")

    def Display(self):
        print("Library Information:")
        print("Access Number:", self.AccNumber)
        print("Publisher:", self.Publisher)
        print("Title:", self.Title)
        print("Author:", self.Author)


# Create an instance of the Library class
book = Library("", "", "", "")

# Read the details of the book
book.Read()

# Display the book information
book.Display()

# Compute and display fine for 3 days late
book.Compute(3)
```

In this example, we define the `Library` class with the specified data attributes (`AccNumber`, `Publisher`, `Title`, and `Author`). The `__init__()` method is the constructor that initializes these attributes when creating an instance of the class.

The `Read()` method prompts the user to enter the details of the book and assigns the input values to the corresponding attributes.

The `Compute()` method takes the number of days late as a parameter and calculates the fine by multiplying the days late by 5. It then displays the calculated fine.

The `Display()` method simply displays the information of the book, including the access number, publisher, title, and author.

Finally, we create an instance of the `Library` class called `book`. We call the `Read()` method to input the details of the book. Then, we call the `Display()` method to display the book information. Lastly, we call the `Compute()` method to calculate and display the fine for 3 days late.

(Note : Feel free to modify the code and add additional methods or functionality as per your requirements.)

Q3) a) Develop a program to print the number of lines, words and characters present in the given file? Accept the file name from user. Handle necessary exceptions. [2]

Here's a program that accepts a file name from the user and prints the number of lines, words, and characters present in the given file. It also handles necessary exceptions:

```python
def count_lines_words_characters(filename):
    try:
        with open(filename, 'r') as file:
            lines = 0
            words = 0
            characters = 0

            for line in file:
                lines += 1
                words += len(line.split())
                characters += len(line)

        print("Number of lines:", lines)
        print("Number of words:", words)
        print("Number of characters:", characters)

    except FileNotFoundError:
        print("File not found!")
    except IOError:
        print("Error reading the file!")

# Prompt the user to enter the file name
filename = input("Enter the file name: ")

# Call the function to count lines, words, and characters
count_lines_words_characters(filename)
```

In this program, the function `count_lines_words_characters()` takes a filename as a parameter. Inside

the function, we use a `with` statement to open the file in read mode. We initialize counters for lines, words, and characters to 0.

We then iterate over each line in the file using a `for` loop. For each line, we increment the lines counter, add the number of words in the line (obtained by splitting the line using whitespace) to the words counter, and add the length of the line to the characters counter.

After processing all the lines, we print the number of lines, words, and characters.

The program also handles exceptions by using a `try-except` block. If the file is not found or there is an error reading the file, the appropriate exception is caught, and an error message is displayed.

Make sure to enter the correct file name or provide the correct path to the file to get the desired results.

b) Illustrate CRUD operations in MongoDB with example [3]

CRUD operations in MongoDB refer to the basic operations performed on data: Create, Read, Update, and Delete. Here's an example illustrating each of these operations using Python and the PyMongo library to interact with MongoDB:

1. Create (Insert):

```python
from pymongo import MongoClient

# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017/")
db = client["mydatabase"]
collection = db["mycollection"]

# Insert a document
data = {"name": "John Doe", "age": 30, "city": "New York"}
result = collection.insert_one(data)
print("Inserted document ID:", result.inserted_id)
```

In this example, we connect to a MongoDB database using `MongoClient`. We define a database `mydatabase` and a collection `mycollection`. We then create a document (`data`) containing information about a person and insert it into the collection using `insert_one()`. The method returns the inserted document's ID, which we print.

2. Read (Query):

```python
# Find documents
query = {"city": "New York"}
results = collection.find(query)

# Iterate and display documents
for doc in results:
    print(doc)
```

Here, we use `find()` to retrieve documents from the collection that match a specified query. In this

case, we want to find documents where the "city" field is "New York". The results are returned as a cursor, which we iterate over and print each document.

3. Update:

```python
# Update a document
query = {"name": "John Doe"}
new_values = {"$set": {"age": 35}}
result = collection.update_one(query, new_values)
print("Modified documents:", result.modified_count)
```

In this example, we use `update_one()` to update a document that matches the given query. We want to update the document where the "name" field is "John Doe" and set the "age" field to 35 using the `$set` operator. The method returns an `UpdateResult` object, and we print the number of modified documents.

4. Delete:

```python
# Delete a document
query = {"name": "John Doe"}
result = collection.delete_one(query)
print("Deleted documents:", result.deleted_count)
```

Finally, we use `delete_one()` to delete a document that matches the specified query. In this case, we want to delete the document where the "name" field is "John Doe". The method returns a `DeleteResult` object, and we print the number of deleted documents.

Remember to install the `pymongo` library using `pip install pymongo` before running the code. Also, ensure that you have MongoDB running locally or provide the appropriate connection details if using a remote MongoDB server.

These examples demonstrate the basic CRUD operations in MongoDB using PyMongo. You can build upon these operations and use more advanced features offered by MongoDB and PyMongo to interact with the database.

c) Compare SQL Database with No SQL Database. [2]

SQL (Structured Query Language) databases and NoSQL (Not Only SQL) databases are two different approaches to storing and managing data. Here's a comparison between the two:

1. Data Model:
   - SQL Database: SQL databases follow a rigid, predefined schema. They use tables with rows and columns to store structured data. Each table has a specific structure defined by the schema, which enforces data consistency and ensures data integrity through primary keys, foreign keys, and constraints.
   - NoSQL Database: NoSQL databases are schema-less or have a flexible schema. They can store unstructured, semi-structured, or structured data. NoSQL databases use various data models such as key-value pairs, documents, graphs, or columnar stores, depending on the specific database type. This flexibility allows for agile development and the handling of complex data structures.

2. Scalability:
   - SQL Database: SQL databases typically use a vertical scaling approach. This means that as the data and workload increase, the database server hardware needs to be upgraded to handle the increased load. Vertical scaling can have limitations in terms of scalability and can be costly.
   - NoSQL Database: NoSQL databases are designed for horizontal scaling. They can handle large amounts of data and high traffic loads by adding more servers to a distributed system. This scalability approach allows for better performance and handling of big data.

3. Data Relationships:
   - SQL Database: SQL databases have a strong emphasis on data relationships. They support complex relationships through the use of foreign keys and join operations. SQL databases are suitable for applications where data consistency and relational integrity are crucial.
   - NoSQL Database: NoSQL databases often prioritize denormalization and do not enforce strict relationships between data. Instead, they typically use embedded documents, key-value pairs, or graph structures to represent relationships. This approach allows for faster data retrieval and can be beneficial in cases where flexibility and high performance are required.

4. ACID Compliance:
   - SQL Database: SQL databases are designed to maintain ACID (Atomicity, Consistency, Isolation, Durability) properties. ACID ensures data integrity and reliability. Transactions in SQL databases are typically atomic and provide strong consistency guarantees.
   - NoSQL Database: NoSQL databases often relax some ACID properties to achieve better performance and scalability. They may prioritize eventual consistency, allowing for higher availability and partition tolerance. However, some NoSQL databases do provide ACID guarantees for specific operations.

5. Use Cases:
   - SQL Database: SQL databases are well-suited for applications with structured data, complex relationships, and the need for strong consistency. They are widely used in traditional enterprise applications, financial systems, and applications requiring complex reporting and analytics.
   - NoSQL Database: NoSQL databases are suitable for applications that handle large amounts of unstructured or semi-structured data, have high scalability and performance requirements, and prioritize flexibility. They are commonly used in web and mobile applications, content management systems, real-time analytics, and IoT applications.

It's important to note that the choice between SQL and NoSQL databases depends on the specific requirements of the application, data structure, scalability needs, and performance considerations. Both types of databases have their strengths and weaknesses, and the optimal choice depends on the specific use case.

d) Write aprogram to check whether entered string & number is palindrome or not. [3]

```
def is_palindrome(input):
    # Convert the input to a string for comparison
    input_str = str(input)

    # Reverse the input string
    reversed_str = input_str[::-1]

    # Compare the input string with the reversed string
    if input_str == reversed_str:
        return True
    else:
        return False

# Prompt the user to enter a string or number
input_value = input("Enter a string or number: ")

# Check if the input is a palindrome
if is_palindrome(input_value):
    print("Palindrome!")
else:
    print("Not a palindrome!")
```

OR

a) **Develop a pythan program to remove the comment character from all the lines in the given file. Accept the file name from user** [2]

Here's a Python program that removes the comment character from all lines in the given file. It prompts the user to enter the file name and removes the comment character ('#') from each line:

```python
def remove_comments(filename):
    try:
        with open(sourcefile.txt, 'r') as file:
            lines = file.readlines()

        with open(destinationfile.txt, 'w') as file:
            for line in lines:
                line = line.split('#')[0]  # Remove comment after '#'
                file.write(line)

        print("Comments removed successfully.")

    except FileNotFoundError:
        print("File not found!")
    except IOError:
        print("Error reading or writing the file!")

# Prompt the user to enter the file name
filename = input("Enter the file name: ")
```

```
# Call the function to remove comments from the file
remove_comments(filename)
```

In this program, the function `remove_comments()` takes a filename as a parameter. Inside the function, we use a `with` statement to open the file in read mode and read all lines into a list using `readlines()`.

We then open the file in write mode and iterate over each line. For each line, we split it using `split('#')` to separate the line content from the comment part after the '#' character. We take only the content before the '#' and write it back to the file using `write()`.

After processing all the lines, we print a success message.

The program handles exceptions using a `try-except` block. If the file is not found or there is an error reading or writing the file, the appropriate exception is caught, and an error message is displayed.

Make sure to enter the correct file name or provide the correct path to the file to remove the comments successfully.

b) Write a pythan program to perform following operations. on MongoDB Database. [5]
i) Create collection "EMP" with fields:
Emp-name, Emp- mobile, Emp, sal, Age
ii) Insert 5 documents.
iii) Find the employees getting salary between 5000 to 10000.
iv) Update mobile number for the employee named as "Riddhi"
v) Display all employees in the order of "Age"

To perform the mentioned operations on a MongoDB database, you can use the PyMongo library in Python. Here's a Python program that demonstrates the required operations:

```python
from pymongo import MongoClient

# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017/")

# Access the database and collection
db = client["mydatabase"]
collection = db["EMP"]

# i) Create collection "EMP" with fields
collection.create_index("Age")  # Creating an index on "Age" field for sorting

# ii) Insert 5 documents
documents = [
    {"Emp-name": "John", "Emp-mobile": "9876543210", "Emp-sal": 8000, "Age": 32},
    {"Emp-name": "Riddhi", "Emp-mobile": "8765432109", "Emp-sal": 9500, "Age": 28},
    {"Emp-name": "Michael", "Emp-mobile": "7654321098", "Emp-sal": 6500, "Age": 35},
    {"Emp-name": "Sarah", "Emp-mobile": "6543210987", "Emp-sal": 7500, "Age": 30},
    {"Emp-name": "David", "Emp-mobile": "5432109876", "Emp-sal": 5500, "Age": 25}
]
```

```
collection.insert_many(documents)
print("Documents inserted successfully.")

# iii) Find employees getting salary between 5000 to 10000
query = {"Emp-sal": {"$gte": 5000, "$lte": 10000}}
result = collection.find(query)
print("Employees with salary between 5000 and 10000:")
for doc in result:
    print(doc)

# iv) Update mobile number for the employee named as "Riddhi"
update_query = {"Emp-name": "Riddhi"}
new_values = {"$set": {"Emp-mobile": "9999999999"}}
collection.update_one(update_query, new_values)
print("Mobile number updated for Riddhi.")

# v) Display all employees in the order of "Age"
sort_query = [("Age", 1)]  # 1 for ascending order, -1 for descending order
result = collection.find().sort(sort_query)
print("Employees sorted by Age:")
for doc in result:
    print(doc)
```

Make sure to have MongoDB running on your local machine on the default port (27017) or provide the appropriate connection details if using a remote MongoDB server.

In this program, we connect to MongoDB using `MongoClient`. We define the database as `mydatabase` and the collection as `EMP`. We create an index on the `Age` field to perform sorting later.

We insert five documents into the collection using `insert_many()`. Each document represents an employee and contains fields such as "Emp-name", "Emp-mobile", "Emp-sal" (salary), and "Age".

For finding employees with salaries between 5000 and 10000, we use a query with `$gte` (greater than or equal to) and `$lte` (less than or equal to) operators.

To update the mobile number for the employee named "Riddhi", we use `update_one()` with a query to find the document and `$set` operator to update the "Emp-mobile" field.

To display all employees in the order of "Age", we use `find().sort()` with a sort query specifying the "Age" field in ascending order (`1`).

Feel free to modify the program according to your specific requirements or add error handling as necessary.

c) Write a python program to find the factorial of a given number using recursion [3]

Here's a Python program that calculates the factorial of a given number using recursion:

```python
def factorial(n):
    if n == 0:
        return 1
```

```
    else:
        return n * factorial(n - 1)

# Prompt the user to enter a number
number = int(input("Enter a number: "))

# Calculate the factorial using recursion
result = factorial(number)
print(f"The factorial of {number} is {result}.")
```

In this program, we define a recursive function called `factorial()`. The function takes a number `n` as an argument. It checks if `n` is equal to 0, in which case it returns 1 as the base case for factorial. Otherwise, it recursively calls `factorial()` with `n - 1` and multiplies it by `n`.

The program prompts the user to enter a number using the `input()` function and converts it to an integer using `int()`. Then, it calls the `factorial()` function with the input number and assigns the returned value to the `result` variable.

Finally, it displays the factorial of the given number using an f-string (formatted string) to include the number and the calculated result.

Please note that recursion can be computationally expensive for large numbers, and there is a limit to the depth of recursion in Python. So, this program is suitable for small numbers to avoid exceeding the recursion limit.

Q4) a) Write a program to demonstrate: [4]
i) Creating a Thread without using any class
ii) Creating a Thread by extending Thread class.

Here's a Python program that demonstrates creating a thread without using any class and creating a thread by extending the Thread class:

```python
import threading

# i) Creating a Thread without using any class
def thread_function():
    print("This is a thread created without using any class.")

thread = threading.Thread(target=thread_function)
thread.start()
thread.join()

# ii) Creating a Thread by extending Thread class
class MyThread(threading.Thread):
    def run(self):
        print("This is a thread created by extending the Thread class.")

my_thread = MyThread()
my_thread.start()
my_thread.join()
```

In this program, we demonstrate two methods of creating threads.

In the first method (i), we define a function called `thread_function()` which will be executed in a separate thread. We create a `Thread` object from the `threading` module, passing the `target` parameter as `thread_function`. Then, we start the thread using the `start()` method and wait for it to finish using the `join()` method.

In the second method (ii), we create a custom thread class called `MyThread` by extending the `Thread` class from the `threading` module. We override the `run()` method, which is called when the thread is started. Inside the `run()` method, we define the actions to be performed by the thread. We create an instance of `MyThread` and start it by calling the `start()` method. Finally, we wait for the thread to finish using the `join()` method.

Both methods demonstrate the creation and execution of threads. The first method uses a function as the target for the thread, while the second method involves creating a custom thread class by extending the `Thread` class.

Note that when using threads, the order of execution between threads may vary, and the output might not always be in the exact order as written in the code.

b) Write a Pythan program to check the validity of a password given by user. [4]
The password should satisfy following criteria:
i) Contain at least 1 letter between a and z
ii) Contain at least 1 number between 0 and 9
iii) Contain at least 1 letter between A and Z
iv) Contain at least 1 character from $, #, @,*
v) Minimum length of password : 8
vi) Maximum length of password : 20

Here's a Python program that checks the validity of a password provided by the user based on the given criteria:

```python
def check_password_validity(password):
    # Criteria 1: Check for at least 1 letter between 'a' and 'z'
    if not any(char.islower() for char in password):
        return False

    # Criteria 2: Check for at least 1 number between 0 and 9
    if not any(char.isdigit() for char in password):
        return False

    # Criteria 3: Check for at least 1 letter between 'A' and 'Z'
    if not any(char.isupper() for char in password):
        return False

    # Criteria 4: Check for at least 1 character from '$', '#', '@', '*'
    special_chars = ['$', '#', '@', '*']
    if not any(char in special_chars for char in password):
        return False

    # Criteria 5: Check for minimum length of 8
    if len(password) < 8:
        return False
```

```
    # Criteria 6: Check for maximum length of 20
    if len(password) > 20:
        return False

    # Password is valid
    return True

# Prompt the user to enter a password
password = input("Enter a password: ")

# Check the validity of the password
if check_password_validity(password):
    print("Password is valid.")
else:
    print("Password is invalid.")
```

In this program, the `check_password_validity()` function takes a password as an argument and checks its validity based on the given criteria.

For each criterion, we use conditional statements and string methods to check if the password satisfies the condition. If any of the conditions are not met, we return `False` indicating that the password is invalid. If all the conditions are satisfied, we return `True` indicating that the password is valid.

The program prompts the user to enter a password using the `input()` function and assigns it to the `password` variable.

Then, it calls the `check_password_validity()` function with the entered password and checks the return value. If the return value is `True`, it prints "Password is valid." Otherwise, it prints "Password is invalid."

Feel free to modify the program according to your specific requirements or add more complexity to the password validation rules.

c) Explain Generators in python with suitable example. [2]

In Python, generators are a type of iterable that generate values on-the-fly, allowing efficient memory usage and lazy evaluation. They are defined using the `yield` keyword instead of the `return` keyword. Generators can be iterated over using a loop, similar to lists or tuples, but they do not store all the values in memory at once.

Here's an example to illustrate the concept of generators:

```python
def fibonacci_generator():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

# Create a Fibonacci sequence generator
fibonacci = fibonacci_generator()

# Generate and print the first 10 Fibonacci numbers
for _ in range(10):
```

```
    print(next(fibonacci))
```

In this example, we define a generator function called `fibonacci_generator()` that generates the Fibonacci sequence. Inside the function, we use an infinite loop with a `yield` statement. On each iteration, we yield the current Fibonacci number (`a`) and update the variables `a` and `b` to calculate the next Fibonacci number.

To create an instance of the generator, we call the `fibonacci_generator()` function, which returns a generator object assigned to the variable `fibonacci`.

We then use a loop to iterate over the generator and print the first 10 Fibonacci numbers. The `next()` function is used to retrieve the next value from the generator on each iteration. The loop terminates after 10 iterations.

The key benefit of using a generator in this example is that it generates Fibonacci numbers on-the-fly, one at a time, without storing the entire sequence in memory. This is particularly useful when dealing with large sequences or infinite sequences, as it allows for efficient memory utilization.

Generators are also useful when working with large data sets or performing time-consuming calculations, as they provide values as needed, reducing memory consumption and improving performance.

Additionally, generators can be used in combination with other iterable operations like filtering, mapping, and reducing to process data efficiently without needing to load the entire dataset into memory at once.

Note that when using generators, each call to `next()` advances the generator to the next yielded value. If there are no more values to yield, a `StopIteration` exception is raised.

OR

a) Write a program for synchronization of threads using RLOCK. Accept the two numbers from user and calculate factorial of both numbers simultaneonly [4]

Here's a Python program that uses thread synchronization with an `RLock` (reentrant lock) to calculate the factorial of two numbers simultaneously:

```python
import threading
from math import factorial

# Create an RLock object for synchronization
lock = threading.RLock()

def calculate_factorial(number):
    # Acquire the lock
    lock.acquire()

    try:
        result = factorial(number)
        print(f"Factorial of {number} is {result}")
    finally:
        # Release the lock
        lock.release()
```

```
# Prompt the user to enter two numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))

# Create threads to calculate factorials
thread1 = threading.Thread(target=calculate_factorial, args=(number1,))
thread2 = threading.Thread(target=calculate_factorial, args=(number2,))

# Start the threads
thread1.start()
thread2.start()

# Wait for both threads to finish
thread1.join()
thread2.join()
```

In this program, we first import the `threading` module for thread-related functionality and the `factorial` function from the `math` module.

We then create an `RLock` object called `lock` for thread synchronization. The `RLock` allows multiple acquisitions of the same lock by the same thread without causing a deadlock.

The `calculate_factorial()` function calculates the factorial of a given number. Within the function, we acquire the lock using `lock.acquire()` to ensure only one thread can execute this section at a time. We calculate the factorial using the `factorial()` function and print the result. Finally, we release the lock using `lock.release()`.

We prompt the user to enter two numbers and create two threads, `thread1` and `thread2`, each targeting the `calculate_factorial()` function with the respective number as an argument.

We start both threads using the `start()` method and wait for them to finish using the `join()` method.

By using the `RLock`, only one thread at a time can execute the critical section of code, ensuring that the factorials of both numbers are calculated one after the other without interference.

Feel free to modify the program according to your specific requirements or add error handling as necessary.

b) Write a pythan program [4]
i) To remove all leading ‘zeros’ from an IP address
ii) To find all 5 character long words in a string Accept string from user.

Here's a Python program that addresses both requirements:

i) To remove all leading 'zeros' from an IP address:

```python
def remove_leading_zeros(ip_address):
    # Split the IP address into its octets
    octets = ip_address.split('.')
```

```
    # Remove leading zeros from each octet
    updated_octets = [str(int(octet)) for octet in octets]

    # Join the octets back into an IP address
    updated_ip_address = '.'.join(updated_octets)

    return updated_ip_address

# Prompt the user to enter an IP address
ip_address = input("Enter an IP address: ")

# Remove leading zeros from the IP address
updated_ip_address = remove_leading_zeros(ip_address)

print("Updated IP address:", updated_ip_address)
```

In this program, the `remove_leading_zeros()` function takes an IP address as input and performs the required operation.

The function first splits the IP address into its octets using the `split()` method with a delimiter of `'.'`. It then creates a new list `updated_octets` where each octet is converted to an integer using `int()` and then back to a string using `str()` to remove any leading zeros.

Finally, it joins the updated octets back into an IP address using the `join()` method with a delimiter of `'.'` and returns the updated IP address.

The program prompts the user to enter an IP address using the `input()` function and assigns it to the `ip_address` variable.

It then calls the `remove_leading_zeros()` function with the entered IP address and assigns the updated IP address to the `updated_ip_address` variable.

The program prints the updated IP address.

ii) To find all 5-character long words in a string:

```python
def find_5_char_words(string):
    # Split the string into words
    words = string.split()

    # Find 5-character long words
    five_char_words = [word for word in words if len(word) == 5]

    return five_char_words

# Prompt the user to enter a string
string = input("Enter a string: ")

# Find all 5-character long words in the string
five_char_words = find_5_char_words(string)

print("5-character long words found:")
for word in five_char_words:
```

```
    print(word)
```

In this program, the `find_5_char_words()` function takes a string as input and performs the required operation.

The function first splits the string into words using the `split()` method, which splits the string based on whitespace by default.

It then creates a new list `five_char_words` using a list comprehension. It iterates over the words and checks if the length of each word is equal to 5. If so, it includes that word in the list.

Finally, the function returns the list of 5-character long words.

The program prompts the user to enter a string using the `input()` function and assigns it to the `string` variable.

It then calls the `find_5_char_words()` function with the entered string and assigns the list of 5-character long words to the `five_char_words` variable.

The program prints each of the 5-character long words found in the string.

Feel free to modify the program according to your specific requirements or add error handling as necessary.

c) Explain decorators in pythan with suitable example [2]

In Python, decorators are a way to modify the behavior of functions or classes without directly changing their source code. Decorators provide a convenient way to add or modify functionality to existing functions or classes by wrapping them with another function.

Here's an example to illustrate the concept of decorators:

```python
def uppercase_decorator(func):
    def wrapper():
        original_result = func()
        modified_result = original_result.upper()
        return modified_result

    return wrapper

@uppercase_decorator
def greet():
    return "Hello, World!"

# Call the decorated function
print(greet())
```

In this example, we define a decorator function called `uppercase_decorator`. This decorator takes a function `func` as input and defines an inner function called `wrapper`. The `wrapper` function wraps the original function and modifies its behavior.

Inside the `wrapper` function, we call the original function `func` and store its result in

`original_result`. We then modify the result by converting it to uppercase and store it in `modified_result`. Finally, we return the modified result.

To apply the decorator to a function, we use the `@` symbol followed by the decorator name (`@uppercase_decorator`) directly above the function definition. This syntax is equivalent to explicitly calling `greet = uppercase_decorator(greet)`.

When we call the decorated function (`greet()`), the decorator function is invoked automatically. The decorator modifies the behavior of the `greet()` function by converting its result to uppercase before returning it.

The output of the program will be "HELLO, WORLD!".

Decorators provide a flexible and reusable way to extend or modify the functionality of functions or classes without modifying their original code. They can be used for a wide range of purposes, such as logging, timing, caching, access control, and more.

It's important to note that decorators can also accept arguments, allowing further customization of the decorated function's behavior. Additionally, there are built-in decorators in Python, such as `@property` and `@classmethod`, which provide specific functionality for properties and class methods, respectively.

Overall, decorators are a powerful feature in Python that promote code reusability, separation of concerns, and enhance the flexibility and extensibility of functions and classes.

Q5) a) Explain numpy integer indexing, array indexing. boolean array indexing and slicing with example. [4]

In NumPy, there are several methods of indexing and slicing arrays, including integer indexing, array indexing, boolean array indexing, and slicing. Let's explore each of these indexing methods with examples:

1. Integer Indexing:
   Integer indexing allows you to access specific elements of an array using integer indices.

   ```python
   import numpy as np

   # Create an array
   arr = np.array([10, 20, 30, 40, 50])

   # Integer indexing
   print(arr[2])  # Output: 30
   ```

   In this example, we have an array `arr` consisting of elements [10, 20, 30, 40, 50]. By using integer indexing `arr[2]`, we access the element at index 2, which is 30.

2. Array Indexing:
   Array indexing allows you to access specific elements of an array using another array of indices.

   ```python
   import numpy as np
   ```

```
# Create an array
arr = np.array([10, 20, 30, 40, 50])

# Array indexing
indices = np.array([1, 3])
print(arr[indices])  # Output: [20 40]
```

In this example, we have an array `arr` and another array `indices` containing the indices [1, 3]. By using array indexing `arr[indices]`, we access the elements at the specified indices, which are [20, 40].

3. Boolean Array Indexing:
Boolean array indexing allows you to access elements of an array based on a boolean condition.

```python
import numpy as np

# Create an array
arr = np.array([10, 20, 30, 40, 50])

# Boolean array indexing
mask = np.array([True, False, True, False, True])
print(arr[mask])  # Output: [10 30 50]
```

In this example, we have an array `arr` and another array `mask` consisting of boolean values. By using boolean array indexing `arr[mask]`, we access the elements in `arr` corresponding to `True` values in the `mask` array. In this case, we get [10, 30, 50].

4. Slicing:
Slicing allows you to extract a portion of an array by specifying a range of indices.

```python
import numpy as np

# Create an array
arr = np.array([10, 20, 30, 40, 50])

# Slicing
print(arr[1:4])  # Output: [20 30 40]
```

In this example, we have an array `arr`. By using slicing `arr[1:4]`, we extract the elements starting from index 1 up to index 4 (exclusive), resulting in [20, 30, 40].

These indexing methods in NumPy provide flexibility and powerful ways to access and manipulate data in arrays. They can be applied to multi-dimensional arrays as well, allowing you to access specific elements, subsets, or sections of the array based on the indexing technique used.

b) Draw bar graph using matplotlib and decorate it by adding various elements

Here's an example of how to create a bar graph using Matplotlib and decorate it by adding various elements:

```python
import matplotlib.pyplot as plt

# Data for the bar graph
categories = ['Category 1', 'Category 2', 'Category 3', 'Category 4', 'Category 5']
values = [15, 24, 18, 30, 22]

# Create the bar graph
plt.bar(categories, values, color='steelblue')

# Decorations
plt.title('Bar Graph Example')
plt.xlabel('Categories')
plt.ylabel('Values')
plt.grid(True, axis='y')
plt.ylim(0, max(values) + 5)

# Adding annotations to the bars
for i, value in enumerate(values):
    plt.text(i, value, str(value), ha='center', va='bottom')

# Show the bar graph
plt.show()
```

In this example, we create a simple bar graph using Matplotlib and then decorate it by adding various elements:

1. We define the data for the bar graph in the `categories` and `values` lists. Each category represents a label for the x-axis, and each value represents the height of the bar.

2. We create the bar graph using `plt.bar(categories, values, color='steelblue')`, where `categories` represents the x-axis labels, `values` represents the height of the bars, and `color` sets the color of the bars.

3. We add decorations to the graph using various methods:
   - `plt.title('Bar Graph Example')` sets the title of the graph.
   - `plt.xlabel('Categories')` and `plt.ylabel('Values')` label the x-axis and y-axis, respectively.
   - `plt.grid(True, axis='y')` adds gridlines along the y-axis.
   - `plt.ylim(0, max(values) + 5)` sets the range of the y-axis to ensure all bars are visible.

4. We add annotations to each bar using a `for` loop and `plt.text()`. The loop iterates over the `values` list, and for each value, it adds a text annotation with the corresponding value at the center of the bar.

5. Finally, we use `plt.show()` to display the bar graph.

You can customize the elements further by adjusting the colors, font sizes, bar widths, and other properties according to your preferences.

c) Prepare the pandas dataframe from csv file. [2]
perform following operations.
i) Fill all 'NaN' values with the mean of respective column.
ii) Display last 5 rows.

To perform the operations on a Pandas DataFrame from a CSV file, you can follow these steps:

1. Import the required libraries:
```python
import pandas as pd
```

2. Read the CSV file into a DataFrame:
```python
df = pd.read_csv('your_file.csv')
```
Replace `'your_file.csv'` with the actual file name or path.

3. Fill 'NaN' values with the mean of respective column:
```python
df.fillna(df.mean(), inplace=True)
```
This line of code fills the missing values (NaN) in the DataFrame with the mean of each respective column. The `inplace=True` parameter modifies the DataFrame in place.

4. Display the last 5 rows:
```python
print(df.tail(5))
```
The `tail(5)` method is used to display the last 5 rows of the DataFrame.

Here's the complete program:

```python
import pandas as pd

# Read the CSV file into a DataFrame
df = pd.read_csv('your_file.csv')

# Fill 'NaN' values with the mean of respective column
df.fillna(df.mean(), inplace=True)

# Display the last 5 rows
print(df.tail(5))
```

Make sure to replace `'your_file.csv'` with the actual file name or path.

This program reads the CSV file into a DataFrame using the `read_csv()` function from Pandas. It then fills the 'NaN' values in the DataFrame with the mean of each respective column using `fillna()`. Finally, it displays the last 5 rows of the updated DataFrame using `tail(5)`.

d) Explain constructors in pythan with example. [2]

In Python, constructors are special methods that are automatically called when an object of a class is created. Constructors are used to initialize the attributes (variables) of an object.

In Python, the constructor method is named `__init__()`. It is defined within a class and is automatically invoked when a new object is created. The `__init__()` method allows you to set the initial state of the object by assigning values to its attributes.

Here's an example that demonstrates the use of constructors in Python:

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def display(self):
        print("Name:", self.name)
        print("Age:", self.age)

# Create an object and initialize it using the constructor
person1 = Person("John", 30)

# Call the display method to print the object's attributes
person1.display()
```

In this example, we define a class called `Person`. The `Person` class has a constructor method named `__init__()` that takes two parameters: `name` and `age`. Inside the constructor, we initialize the `name` and `age` attributes of the object using the values passed to the constructor.

The `display()` method is defined within the class to print the object's attributes (`name` and `age`).

To create an object of the `Person` class, we use the constructor by calling `Person("John", 30)`. This creates a new object and automatically invokes the `__init__()` method with the given arguments. The `self` parameter refers to the newly created object itself.

Finally, we call the `display()` method on the `person1` object to print its attributes.

The output of the program will be:

```
Name: John
Age: 30
```

Constructors are useful for initializing the state of an object, setting its attributes, or performing any other necessary setup operations when creating an object of a class. They allow you to define how the object should be initialized and provide a way to ensure that necessary values are passed during object creation.

OR

a) Write a program to illustrate numpy array attributes/functions. [4]

i) ndarray. shape

ii) np. zeros ( )

iii) np. eye ( )

iv) np. random. random ( )

Certainly! Here's a program that illustrates various attributes and functions of NumPy arrays:

```python
import numpy as np

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

# i) ndarray.shape
print("Shape of the array:", arr.shape)

# ii) np.zeros()
zeros_arr = np.zeros((3, 4))
print("Array filled with zeros:")
print(zeros_arr)

# iii) np.eye()
eye_arr = np.eye(3)
print("Identity matrix:")
print(eye_arr)

# iv) np.random.random()
random_arr = np.random.random((2, 3))
print("Array of random numbers:")
print(random_arr)
```

In this program, we import the NumPy library using `import numpy as np`.

1. `ndarray.shape`:
   The `shape` attribute of a NumPy array returns a tuple that represents the dimensions of the array. In the program, we create an array `arr` and print its shape using `arr.shape`.

2. `np.zeros()`:
   The `np.zeros()` function creates an array of a given shape filled with zeros. In the program, we create a 3x4 array filled with zeros using `np.zeros((3, 4))` and assign it to `zeros_arr`. We then print `zeros_arr` to display the array.

3. `np.eye()`:
   The `np.eye()` function creates an identity matrix, which is a square matrix with ones on the diagonal and zeros elsewhere. In the program, we create a 3x3 identity matrix using `np.eye(3)` and assign it to `eye_arr`. We then print `eye_arr` to display the matrix.

4. `np.random.random()`:
   The `np.random.random()` function generates an array of random numbers from a uniform distribution between 0 and 1. In the program, we create a 2x3 array of random numbers using `np.random.random((2, 3))` and assign it to `random_arr`. We then print `random_arr` to display the array.

b) Read data from csv five and create dataframe. Perform following operations.
i) Display list of all columns.
ii) Display data with last three rows and first three columns [2]

To perform the operations on a CSV file using Pandas, you can follow these steps:

1. Import the required libraries:
```python
import pandas as pd
```

2. Read the CSV file into a DataFrame:
```python
df = pd.read_csv('file.csv')
```
Replace `'file.csv'` with the actual file name or path.

3. Display the list of all columns:
```python
print(df.columns.tolist())
```
The `columns` attribute returns a list of all the column names in the DataFrame. `tolist()` converts the column names to a Python list.

4. Display the data with the last three rows and first three columns:
```python
print(df.iloc[-3:, :3])
```
The `iloc` indexer is used to select specific rows and columns of the DataFrame. `-3:` specifies the last three rows, and `:3` specifies the first three columns.

Here's the complete program:

```python
import pandas as pd

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

# Display the list of all columns
print(df.columns.tolist())

# Display the data with the last three rows and first three columns
print(df.iloc[-3:, :3])
```

Make sure to replace `'file.csv'` with the actual file name or path.

This program reads the CSV file into a DataFrame using the `read_csv()` function from Pandas. It then prints the list of all column names using `df.columns.tolist()`. Finally, it displays the data with the last three rows and first three columns using `df.iloc[-3:, :3]`.

c) Draw line graph using matplot lib and decorate it by adding various elements. Use suitable data.

Here's an example of how to create a line graph using Matplotlib and decorate it by adding various elements:

```python
import matplotlib.pyplot as plt

# Data for the line graph
years = [2015, 2016, 2017, 2018, 2019, 2020]
sales = [100, 150, 200, 180, 220, 250]

# Create the line graph
plt.plot(years, sales, marker='o', linestyle='-', color='blue', linewidth=2)

# Decorations
plt.title('Sales Trend')
plt.xlabel('Year')
plt.ylabel('Sales')
plt.xticks(years)
plt.yticks(range(0, max(sales)+50, 50))
plt.grid(True)

# Adding annotations to the data points
for i, j in zip(years, sales):
    plt.text(i, j, str(j), ha='center', va='bottom')

# Show the line graph
plt.show()
```

In this example, we create a line graph using Matplotlib and then decorate it by adding various elements:

1. We define the data for the line graph in the `years` and `sales` lists. Each year represents the x-axis, and each sales value represents the y-axis.

2. We create the line graph using `plt.plot(years, sales, marker='o', linestyle='-', color='blue', linewidth=2)`. This line of code plots the line graph with circular markers ('o'), solid line style ('-'), blue color, and a linewidth of 2.

3. We add decorations to the graph using various methods:
   - `plt.title('Sales Trend')` sets the title of the graph.
   - `plt.xlabel('Year')` and `plt.ylabel('Sales')` label the x-axis and y-axis, respectively.
   - `plt.xticks(years)` sets the x-axis tick marks to the years.
   - `plt.yticks(range(0, max(sales)+50, 50))` sets the y-axis tick marks from 0 to the maximum sales value with a step of 50.
   - `plt.grid(True)` adds gridlines to the graph.

4. We add annotations to each data point using a `for` loop and `plt.text()`. The loop iterates over the `years` and `sales` lists, and for each pair, it adds a text annotation with the corresponding sales value at the specified x and y coordinates.

5. Finally, we use `plt.show()` to display the line graph.

d) Explain multiple inheritance in python with suitable example. [2]

Multiple inheritance is a feature in Python that allows a class to inherit attributes and methods from multiple parent classes. With multiple inheritance, a derived class can inherit and combine the characteristics of two or more base classes.

Here's an example to illustrate multiple inheritance in Python:

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

    def drive(self):
        print(f"{self.brand} is being driven.")

class Electric:
    def __init__(self, battery_capacity):
        self.battery_capacity = battery_capacity

    def charge(self):
        print(f"Charging the vehicle with {self.battery_capacity} kWh battery.")

class ElectricCar(Vehicle, Electric):
    def __init__(self, brand, battery_capacity):
        Vehicle.__init__(self, brand)
        Electric.__init__(self, battery_capacity)

# Create an instance of ElectricCar
my_car = ElectricCar("Tesla", 75)

# Access attributes and methods from parent classes
print("Brand:", my_car.brand)
print("Battery Capacity:", my_car.battery_capacity)
my_car.drive()
my_car.charge()
```

In this example, we have three classes: `Vehicle`, `Electric`, and `ElectricCar`.

The `Vehicle` class represents a generic vehicle with an `__init__()` method that initializes the `brand` attribute and a `drive()` method.

The `Electric` class represents an electric vehicle with an `__init__()` method that initializes the `battery_capacity` attribute and a `charge()` method.

The `ElectricCar` class is derived from both the `Vehicle` and `Electric` classes using multiple inheritance. It inherits attributes and methods from both parent classes. The `__init__()` method of `ElectricCar` explicitly calls the `__init__()` methods of both parent classes to initialize their respective attributes.

In the program, we create an instance of `ElectricCar` called `my_car` with the brand "Tesla" and a battery capacity of 75 kWh. We then access attributes and methods from both parent classes using `my_car`. We can access the `brand` and `battery_capacity` attributes, as well as call the `drive()` method from the `Vehicle` class and the `charge()` method from the `Electric` class.

Output:
```
Brand: Tesla
Battery Capacity: 75
Tesla is being driven.
Charging the vehicle with 75 kWh battery.
```

Multiple inheritance allows a class to inherit from multiple base classes, providing flexibility in combining and reusing code from different parent classes. However, it's important to be cautious when using multiple inheritance to avoid method name clashes or complications arising from complex class hierarchies.