ASSISTED CODING LAB – 6

HALL TICKET NO : 2403A52403

BATCH NO : 14

TASK : 1

```
class Employee:
  def __init__(self, name, emp_id, salary):
      self.name = name
      self.emp_id = emp_id
      self.salary = salary  # Monthly salary

  def display_details(self):
      print(f"Employee ID: {self.emp_id}")
      print(f"Name: {self.name}")
      print(f"Monthly Salary: ${self.salary:.2f}")

  def calculate_yearly_salary(self):
      return self.salary * 12
  # Create an instance of the Employee class
employee1 = Employee("Alice Smith", "E101", 6000)

# Display employee details
employee1.display_details()
```

```
# Calculate and print yearly salary
yearly_salary = employee1.calculate_yearly_salary()
print(f"Yearly Salary: ${yearly_salary:.2f}")
```

OUTPUT :

```
d-1.py"
Employee ID: E101
Name: Alice Smith
Monthly Salary: $6000.00
Yearly Salary: $72000.00
PS C:\Users\Sravya Reddy>
```

EXPLANATION :

This code defines an Employee class in Python:

- **class Employee:**: This line declares a new class named Employee.
- **__init__(self, name, id, salary):**: This is the constructor method. It's called when you create a new Employee object. It initializes the object's attributes: name, id, and salary.
- **display_details(self):**: This method prints the employee's details (name, ID, and salary) to the console.
- **calculate_yearly_salary(self):**: This method calculates and returns the employee's yearly salary by multiplying the monthly salary by 12.
- **give_bonus(self, bonus_percentage):**: This method calculates a bonus based on a given percentage, adds it to

the employee's salary, and prints the bonus amount and the new salary.

You can create an Employee object like this:

employee1 = Employee("John Doe", "E123", 5000)

And then call its methods:

```
employee1.display_details()
yearly_salary = employee1.calculate_yearly_salary()
print(f"Yearly Salary: ${yearly_salary}")
employee1.give_bonus(10) # Give a 10% bonus
```

TASK : 2

```
def find_automorphic_for(start, end):
    """Finds and displays automorphic numbers in a given range using a for loop."""
    print(f"Automorphic numbers between {start} and {end} (using for loop):")
    for num in range(start, end + 1):
        square = num * num
        if str(square).endswith(str(num)):
            print(num)

# Find automorphic numbers between 1 and 1000 using the function
find_automorphic_for(1, 1000)
```

OUTPUT :

d 2.py"

Automorphic numbers between 1 and 1000 (using for loop):

1

5

6

25

76

376

625

PS C:\Users\Sravya Reddy>

EXPLANATION :

1. **def find_automorphic_for(start, end):**: This line defines a function named find_automorphic_for that takes two arguments: start and end. These arguments represent the beginning and end of the range you want to check for automorphic numbers.
2. **"""Finds and displays automorphic numbers in a given range using a for loop."""**: This is a docstring, which explains what the function does.
3. **print(f"Automorphic numbers between {start} and {end} (using for loop):")**: This line prints a message indicating the range being checked and that a for loop is being used.
4. **for num in range(start, end + 1):**: This is the for loop. It iterates through each number (num) in the specified range,

from start up to and including end. The + 1 in range(start, end + 1) is necessary because range() is exclusive of the end value.

5. **square = num * num**: Inside the loop, this line calculates the square of the current number (num) and stores it in the square variable.
6. **if str(square).endswith(str(num)):**: This is the core logic for checking if a number is automorphic.
    - str(square) converts the square (an integer) into a string.
    - str(num) converts the original number (num) into a string.
    - .endswith(str(num)) is a string method that checks if the string representation of the square (str(square)) ends with the string representation of the original number (str(num)).
7. **print(num)**: If the condition in the if statement is true (meaning the number is automorphic), this line prints the number.
8. **find_automorphic_for(1, 1000)**: This line calls the find_automorphic_for function with the arguments 1 and 1000, initiating the process of finding and displaying automorphic numbers in that range.

In summary, the code iterates through each number in the given range, calculates its square, and then checks if the square ends with the original number (when both are treated as strings). If it does, the number is printed as an automorphic number.

TASK 3 :

```
def classify_feedback_nested_if(rating):
  """
  Classifies online shopping feedback as Positive, Neutral, or Negative
  based on a numerical rating (1-5) using nested if-elif-else.

  Args:
    rating: An integer representing the feedback rating (1-5).

  Returns:
    A string indicating the feedback classification (Positive, Neutral, Negative),
    or an error message for invalid ratings.
  """
  if 1 <= rating <= 5:
    if rating >= 4:
      return "Positive"
    elif rating == 3:
      return "Neutral"
    else:
      return "Negative"
  else:
```

```
    return "Invalid Rating: Please provide a rating between 1 and 5."


# Example usage:
print(f"Rating 5: {classify_feedback_nested_if(5)}")
print(f"Rating 3: {classify_feedback_nested_if(3)}")
print(f"Rating 1: {classify_feedback_nested_if(1)}")
print(f"Rating 0: {classify_feedback_nested_if(0)}")
print(f"Rating 6: {classify_feedback_nested_if(6)}")
```

OUTPUT :

```
d-3.py"
Rating 5: Positive
Rating 3: Neutral
Rating 1: Negative
Rating 0: Invalid Rating: Please provide a rating between 1 and 5.
Rating 6: Invalid Rating: Please provide a rating between 1 and 5.
PS C:\Users\Sravya Reddy>
```

EXPLANATION :

1. **def classify_feedback_nested_if(rating):**: This line defines a function named classify_feedback_nested_if that takes one argument, rating. This argument is expected to be an integer representing the feedback rating.
2. **""" ... """**: This is a docstring that explains the purpose of the function, its arguments (Args), and what it returns (Returns).

3. **if 1 <= rating <= 5:**: This is the outer if condition. It checks if the provided rating is within the valid range of 1 to 5 (inclusive).
4. **if rating >= 4:**: If the outer condition is true (the rating is between 1 and 5), this is the first nested if condition. It checks if the rating is 4 or 5. If it is, the function returns the string "Positive".
5. **elif rating == 3:**: If the previous nested if condition is false (the rating is not 4 or 5), this elif condition checks if the rating is exactly 3. If it is, the function returns the string "Neutral".
6. **else:**: If neither of the previous nested conditions is true (meaning the rating must be 1 or 2, since we've already confirmed it's between 1 and 5), this else block is executed. It returns the string "Negative".
7. **else:**: This else block corresponds to the outer if 1 <= rating <= 5: condition. If the initial rating is *not* between 1 and 5, this block is executed, and the function returns an "Invalid Rating" error message.
8. **# Example usage:**: This is a comment indicating the following lines demonstrate how to use the function.
9. **print(f"Rating 5: {classify_feedback_nested_if(5)}")** and similar lines: These lines call the classify_feedback_nested_if function with different rating values and print the returned classification along with the rating. This shows how the function behaves with various inputs, including valid and invalid ratings.

In essence, the code first validates if the input rating is within the expected range. If it is, it then uses nested conditions to

determine if the feedback is Positive (4 or 5), Neutral (3), or Negative (1 or 2). If the initial rating is outside the 1-5 range, it indicates an invalid input.

TASK 4 :

```
import math


def find_primes_optimized(start, end):
  """Finds and displays prime numbers in a given range using an optimized approach (square root method)."""
  print(f"Prime numbers between {start} and {end} (optimized version):")
  for num in range(start, end + 1):
    # Prime numbers are greater than 1
    if num > 1:
      is_prime = True
      # Check for factors from 2 up to the square root of the number
      # We only need to check up to the square root because if a number n has a factor greater than its square root,
      # it must also have a factor smaller than its square root.
      for i in range(2, int(math.sqrt(num)) + 1):
        if (num % i) == 0:
          is_prime = False
          break  # Found a factor, not prime
```

```
        if is_prime:
            print(num)


# Find prime numbers between 1 and 500 using the optimized function
find_primes_optimized(1, 500)
```

OUTPUT :

d-4.py"

Prime numbers between 1 and 500 (optimized version):

2

3

5

7

11

13

17

19

23

29

31

37

41

43

47

53

59

61

67

71

73

79

83

89

97

101

103

107

109

113

127

131

137

139

149

151

157

163

167

173

179

181

191

193

197

199

211

223

227

229

233

239

241

251

257

263

269

271

277

281

283

293

307

311

313

317

331

337

347

349

353

359

367

373

379

383

389

397

401

409

419

421

431

433

439

443

449

457

461

463

467

479

487

491

499

PS C:\Users\Sravya Reddy>

EXPLANATION :

This version is very similar to the initial one, with a key difference in the nested loop:

1. **import math**: Imports the math module to use the math.sqrt() function.
2. **for i in range(2, int(math.sqrt(num)) + 1):**: This is the crucial optimization. Instead of checking for factors up to num - 1, it only checks for factors from 2 up to the integer part of the square root of num.
    - **Why does this work?** If a number num has a factor i that is greater than its square root (i >

sqrt(num)), then there must be another factor j such that i * j = num. If i > sqrt(num), then j must be less than sqrt(num) (j < sqrt(num)). Therefore, if a number has any factors, it must have at least one factor less than or equal to its square root (excluding 1). By only checking up to the square root, we significantly reduce the number of iterations in the inner loop, especially for larger numbers.

- **int(math.sqrt(num))**: Calculates the square root and converts it to an integer.
- **+ 1**: Includes the square root itself in the range check in case the number is a perfect square of a prime number (though this isn't strictly necessary for primality testing, it's good practice to include the upper bound).

Comparison of Efficiency:

- **Initial Version:** The inner loop runs up to num - 2 times for each number. This approach has a time complexity of approximately $O(n^2)$, where n is the upper limit of the range.
- **Optimized Version:** The inner loop runs up to sqrt(num) times for each number. This significantly reduces the number of checks, especially for larger numbers. The time complexity is approximately O(n * sqrt(n)).

The optimized version is much more efficient, especially for finding prime numbers in larger ranges, because it drastically reduces the number of division checks needed for each number.

TASK 5 :

class Library:

```
    """Represents a simple library system."""

    def __init__(self):
        """Initializes the Library with an empty list to store books."""
        self.books = [] # Stores books as a list of dictionaries, e.g., {'title': 'Book Title', 'available': True}

    def add_book(self, title):
        """Adds a new book to the library."""
        self.books.append({'title': title, 'available': True})
        print(f"'{title}' has been added to the library.")

    def issue_book(self, title):
        """Issues a book if it is available."""
        for book in self.books:
            if book['title'] == title:
                if book['available']:
                    book['available'] = False
                    print(f"'{title}' has been issued.")
                    return
                else:
                    print(f"'{title}' is currently not available.")
                    return
```

```
        print(f"'{title}' not found in the library.")

    def display_books(self):
        """Displays all books in the library and their availability status."""
        if not self.books:
            print("The library is empty.")
            return

        print("\nLibrary Catalog:")
        for book in self.books:
            status = "Available" if book['available'] else "Issued"
            print(f"- {book['title']} ({status})")

# Example usage:
my_library = Library()

my_library.add_book("The Great Gatsby")
my_library.add_book("1984")
my_library.add_book("To Kill a Mockingbird")

my_library.display_books()
```

my_library.issue_book("1984")

my_library.issue_book("The Great Gatsby")

my_library.issue_book("1984") # Attempt to issue again

my_library.display_books()

my_library.issue_book("Nonexistent Book") # Attempt to issue a book not in the library

OUTPUT :

d-5.py"

'The Great Gatsby' has been added to the library.

'1984' has been added to the library.

'To Kill a Mockingbird' has been added to the library.

Library Catalog:

- The Great Gatsby (Available)

- 1984 (Available)

- To Kill a Mockingbird (Available)

'1984' has been issued.

'The Great Gatsby' has been issued.

'1984' is currently not available.

Library Catalog:

- The Great Gatsby (Issued)

- 1984 (Issued)

- To Kill a Mockingbird (Available)

'Nonexistent Book' not found in the library.

PS C:\Users\Sravya Reddy>

EXPLANATION :

1. **class Library:**: This line defines a new class named Library. In object-oriented programming, a class serves as a blueprint for creating objects (instances) that have specific properties (attributes) and behaviors (methods).
2. **"""Represents a simple library system."""**: This is a class-level docstring that briefly describes the purpose of the Library class.
3. **def __init__(self):**: This is the constructor method of the class. It's automatically called when you create a new Library object.
    - **self**: Refers to the instance of the class being created.
    - **self.books = []**: This line initializes an instance variable called books as an empty list. This list will store the books in the library. Each book is represented as a dictionary with keys like 'title' and 'available'.
4. **def add_book(self, title):**: This method is used to add a new book to the library.
    - **self**: Refers to the instance of the Library class.
    - **title**: This argument is the title of the book to be added.

- **self.books.append({'title': title, 'available': True})**: This line creates a dictionary for the new book with the given title and sets its initial available status to True. This dictionary is then added to the self.books list.
- **print(f"'{title}' has been added to the library.")**: Prints a confirmation message.

5. **def issue_book(self, title):**: This method is used to issue a book.
    - **self**: Refers to the instance of the Library class.
    - **title**: The title of the book to be issued.
    - **for book in self.books:**: This loop iterates through each book (which is a dictionary) in the self.books list.
    - **if book['title'] == title:**: Checks if the title of the current book in the loop matches the title of the book being requested.
    - **if book['available']:**: If the titles match, this nested condition checks if the book's available status is True.
    - **book['available'] = False**: If the book is available, its available status is set to False to indicate it has been issued.
    - **print(f"'{title}' has been issued.")**: Prints a confirmation message.
    - **return**: Exits the issue_book method after successfully issuing the book.
    - **else:**: If the book is found but is not available (book['available'] is False), this block is executed.

- **print(f"'{title}' is currently not available.")**: Prints a message indicating the book is not available.
- **return**: Exits the issue_book method.
- **print(f"'{title}' not found in the library.")**: This line is reached only if the for loop completes without finding a book with the matching title. It prints a message indicating the book was not found.

6. **def display_books(self):**: This method displays the list of books in the library and their availability status.
    - **self**: Refers to the instance of the Library class.
    - **if not self.books:**: Checks if the self.books list is empty.
    - **print("The library is empty.")**: If the list is empty, prints a message indicating the library is empty.
    - **return**: Exits the method if the library is empty.
    - **print("\nLibrary Catalog:")**: Prints a header for the library catalog.
    - **for book in self.books:**: Iterates through each book in the self.books list.
    - **status = "Available" if book['available'] else "Issued"**: This is a ternary operator that assigns the string "Available" to the status variable if book['available'] is True, and "Issued" otherwise.
    - **print(f"- {book['title']} ({status})")**: Prints the title of the book and its availability status in a formatted string.

7. **# Example usage:**: This section demonstrates how to create a Library object and use its methods.

This code provides a basic but functional library system with the requested features and handles the edge cases of trying to issue an unavailable book or a book that doesn't exist.