# AI ASSISTED CODING
# LAB ASSIGNMENT-6.1

NAME : K.SUPRIYA

HALL TICKET NO : 2403A52395

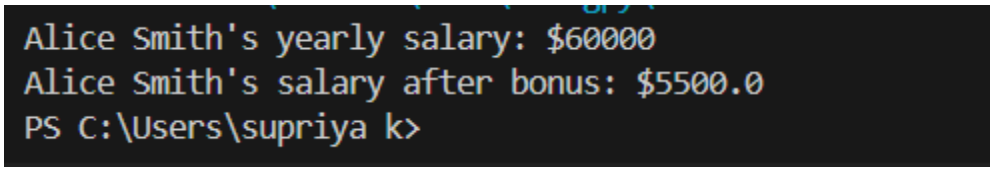
BATCH : 14

## TASK 1:

```
class Employee:
    """Represents an employee with name, ID, and salary."""

    def __init__(self, name, employee_id, salary):
        """Initializes an Employee object.

        Args:
            name: The employee's name.
            employee_id: The employee's ID.
            salary: The employee's monthly salary.
        """
        self.name = name
        self.employee_id = employee = employee_id
        self.salary = salary

    def calculate_yearly_salary(self):
        """Calculates the yearly salary.

        Returns:
            The yearly salary.
```

```
        """
        return self.salary * 12

    def apply_bonus(self, bonus_percentage):
        """Calculates and applies a bonus to the salary.

        Args:
            bonus_percentage: The bonus percentage to apply (as a decimal).
        """
        bonus_amount = self.salary * bonus_percentage
        self.salary += bonus_amount

# Example usage:
employee1 = Employee("Alice Smith", "E123", 5000)
print(f"{employee1.name}'s yearly salary: ${employee1.calculate_yearly_salary()}")
employee1.apply_bonus(0.10) # Apply 10% bonus
print(f"{employee1.name}'s salary after bonus: ${employee1.salary}")
```

OUTPUT:




EXPLANATION:

- **class Employee:**: This line defines a new class named Employee. Think of a class as a blueprint for creating objects (in this case, employee objects).
- **__init__(self, name, employee_id, salary)**: This is the constructor method. It's called when you create a new Employee object.
    - self refers to the instance of the class being created.
    - name, employee_id, and salary are the attributes (data) that each Employee object will have.

    - Inside the constructor, we assign the values passed in when creating the object to the object's attributes using self.attribute_name = value.

- **calculate_yearly_salary(self)**: This is a method (a function within a class) that calculates the employee's yearly salary.
    - It takes self as an argument to access the object's attributes (specifically self.salary).
    - It returns the result of self.salary * 12.
- **apply_bonus(self, bonus_percentage)**: This method calculates and applies a bonus to the employee's salary.
    - It takes self and bonus_percentage as arguments.
    - bonus_percentage is the bonus amount as a decimal (e.g., 0.10 for 10%).
    - It calculates the bonus_amount and then adds it to the employee's current self.salary.

In essence, the Employee class provides a structured way to store and manipulate employee data, allowing you to easily calculate yearly salaries and apply bonuses for individual employees.

## TASK 2:

```
def find_automorphic_while_loop(start, end):
  """
  Finds and displays automorphic numbers within a given range using a while loop.

  Args:
    start: The starting number of the range (inclusive).
    end: The ending number of the range (inclusive).
  """
  print(f"Automorphic numbers between {start} and {end} (using while loop):")
  num = start
  while num <= end:
    square = num * num
    if str(square).endswith(str(num)):
```

```
        print(num)
    num += 1


find_automorphic_while_loop(1, 1000)
```

OUTPUT:

```
Automorphic numbers between 1 and 1000 (using while loop):
1
5
6
25
76
376
625
PS C:\Users\supriya k>
```

EXPLANATION:

- **Automorphic Number**: An automorphic number is a number whose square ends in the same digits as the number itself. For example:
    - 5 is automorphic because 5 * 5 = 25 (ends in 5)
    - 25 is automorphic because 25 * 25 = 625 (ends in 25)
    - 76 is automorphic because 76 * 76 = 5776 (ends in 76)

The code in both the for loop and while loop implementations follows these steps for each number in the range:

1. **Calculate the square**: It calculates the square of the current number (num * num).
2. **Convert to strings**: It converts both the original number and its square to strings. This is done to easily check if the square ends with the digits of the original number.
3. **Check the ending**: It uses the string method .endswith() to check if the string representation of the square ends with the string

1. representation of the original number.
2. **Display if automorphic**: If the condition in step 3 is true, the number is automorphic, and it is printed.

**Comparing the for and while loops:**

- **for loop:** The for loop is typically used when you know the exact number of iterations you need to perform (in this case, iterating through the numbers

from start to end). It's often considered more concise for simple range-based iterations.

- **while loop:** The while loop is used when the number of iterations is not known beforehand and depends on a condition being met. In this case, we are iterating as long as the current number (num) is less than or equal to the end of the range. You need to manually initialize and increment the counter (num) within the loop.

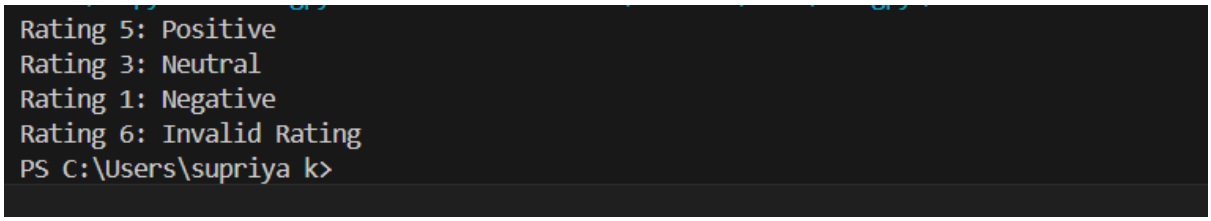
Both implementations achieve the same result for this specific task. The choice between them often comes down to personal preference or the specific nature of the problem. For iterating through a defined range like this, a for loop is generally considered more idiomatic and less prone to errors (like forgetting to increment the counter in a while loop).

## TASK 3:

```
def classify_feedback_if_elif_else(rating):
  """Classifies online shopping feedback based on a numerical rating (1-5)
    using nested if-elif-else conditions.

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

  Returns:
    A string indicating the classification: "Positive", "Neutral", or "Negative".
    Returns "Invalid Rating" for ratings outside the 1-5 range.
  """
  if 1 <= rating <= 5:
    if rating >= 4:
      return "Positive"
    elif rating == 3:
      return "Neutral"
    else:  # rating is 1 or 2
      return "Negative"
```

```
    else:
        return "Invalid Rating"


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

OUTPUT:




EXPLANATION:

This function takes a numerical rating (intended to be between 1 and 5) as input and classifies it as "Positive," "Neutral," or "Negative" using a series of if, elif (else if), and else statements, including nested conditions.

Here's a breakdown of the logic:

1. **if 1 <= rating <= 5:**: This is the first check. It verifies if the input rating is within the valid range of 1 to 5 (inclusive).
    - If the rating is *not* within this range (e.g., 0, 6, or any other number), the code jumps directly to the final else block outside this initial if statement.
2. **if rating >= 4:**: If the rating *is* within the 1-5 range, this nested if statement checks if the rating is 4 or greater (meaning 4 or 5).
    - If true, the function returns the string "Positive".
3. **elif rating == 3:**: If the rating was *not* 4 or 5, this elif block checks if the rating is exactly 3.
    - If true, the function returns the string "Neutral".
4. **else:**: If the rating was *not* 4, 5, or 3 (and we already know it's within the 1-5 range from the first check), this else block catches the remaining possibilities, which are ratings of 1 or 2.

- In this case, the function returns the string "Negative".

5. **else:**: This is the final else block, corresponding to the initial if 1 <= rating <= 5: check.
    - If the initial check was false (the rating was not between 1 and 5), this block is executed, and the function returns the string "Invalid Rating".

In summary, the function first validates the input rating's range and then uses nested conditions to determine the specific classification ("Positive," "Neutral," or "Negative") based on the rating value within that valid range. If the rating is outside the expected range, it's flagged as "Invalid Rating."

## TASK 4:

```
def find_primes_basic(start, end):
  """
  Finds and displays prime numbers within a given range using a basic approach.

  Args:
    start: The starting number of the range (inclusive).
    end: The ending number of the range (inclusive).
  """
  print(f"Prime numbers between {start} and {end} (basic approach):")
  for num in range(max(2, start), end + 1): # Start checking from 2
    is_prime = True
    for i in range(2, num):
      if num % i == 0:
        is_prime = False
        break
    if is_prime:
      print(num)

# Example usage:
find_primes_basic(1, 500)
```

OUTPUT:

```
Prime numbers between 1 and 500 (basic approach):
2
3
5
7
11
13
17
19
23
29
```

```
31
37
41
43
47
53
59
61
67
71
73
79
```

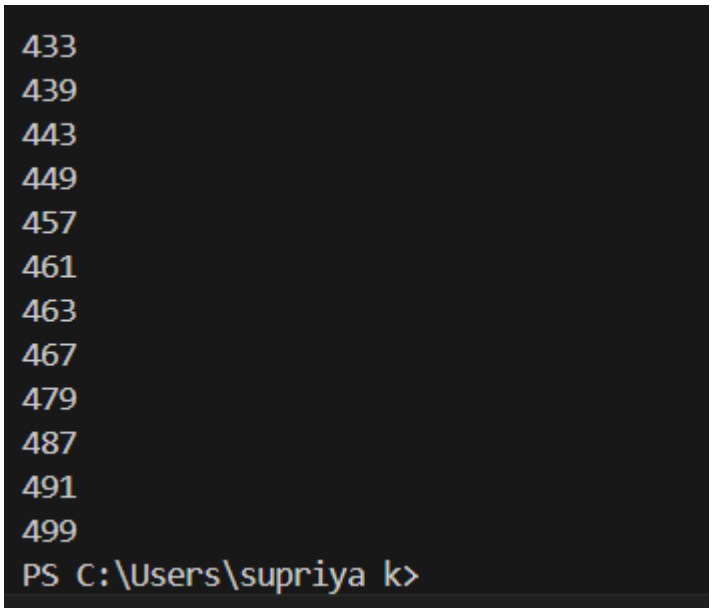
```
83
89
97
101
103
107
109
113
127
131
137
139
```


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






EXPLANATION:

1. **def find_primes_basic(start, end):**: This line defines the function named find_primes_basic that accepts two arguments: start (the beginning of the range) and end (the end of the range).
2. **print(f"Prime numbers between {start} and {end} (basic approach):")**: This line simply prints a header to indicate the range and the approach being used.
3. **for num in range(max(2, start), end + 1):**: This is the outer loop. It iterates through each number (num) in the specified range.
    - range(max(2, start), end + 1): This creates a sequence of numbers. max(2, start) ensures that the checking starts from at least 2, as 1 is not

considered a prime number. end + 1 is used because the range() function's stop value is exclusive.

4. **is_prime = True**: Inside the outer loop, for each num, a boolean variable is_prime is initialized to True. This variable acts as a flag, assuming the current number is prime until proven otherwise.
5. **for i in range(2, num):**: This is the inner loop. For each num, this loop iterates through all possible divisors (i) starting from 2 up to num - 1.
6. **if num % i == 0:**: Inside the inner loop, this condition checks if num is perfectly divisible by i (i.e., the remainder is 0).
    - If num is divisible by any i in this range, it means num has a factor other than 1 and itself, so it's not a prime number.
7. **is_prime = False**: If the condition in step 6 is true, the is_prime flag is set to False.
8. **break**: Once a divisor is found (and is_prime is set to False), there's no need to check further divisors for that num, so the break statement exits the inner loop.
9. **if is_prime:**: After the inner loop finishes (either by checking all possible divisors or by hitting a break), this condition checks if the is_prime flag is still True.
    - If it's True, it means no divisors were found in the inner loop
    - so the number num is prime.
2. **print(num)**: If the number is prime, it is printed to the console.

In essence, this code checks each number in the range by trying to divide it by every number smaller than it (starting from 2). If it finds any number that divides it evenly, it knows the number is not prime and moves on. If it goes through all potential divisors without finding one, it concludes the number is prime.

## TASK 5:

```
class Library:
"""Represents a simple library system."""

def __init__(self):
    """Initializes an empty library."""
    self.books = [] # List to store books. Each book can be a dictionary or object.
```

```
    def add_book(self, book_title, author, quantity=1):
        """Adds a book to the library."""
        # For simplicity, let's represent a book as a dictionary
        book = {"title": book_title, "author": author, "quantity": quantity, "available": quantity}
        self.books.append(book)
        print(f"'{book_title}' by {author} added to the library.")

    def issue_book(self, book_title):
        """Issues a book from the library."""
        for book in self.books:
            if book["title"] == book_title:
                if book["available"] > 0:
                    book["available"] -= 1
                    print(f"'{book_title}' has been issued.")
                    return
                else:
                    print(f"'{book_title}' is currently unavailable.")
                    return
        print(f"'{book_title}' not found in the library.")
    def display_books(self):
        """Displays all books in the library."""
        if not self.books:
            print("The library is empty.")
            return

        print("\nLibrary Catalog:")
```

```
        for book in self.books:
            print(f"Title: {book['title']}, Author: {book['author']}, Available: {book['available']}/{book['quantity']}")


# Example Usage:
my_library = Library()
my_library.add_book("The Lord of the Rings", "J.R.R. Tolkien", 3)
my_library.add_book("Pride and Prejudice", "Jane Austen", 2)
my_library.display_books()
my_library.issue_book("The Lord of the Rings")
my_library.display_books()
my_library.issue_book("The Lord of the Rings")
my_library.issue_book("The Lord of the Rings")
my_library.issue_book("The Lord of the Rings") # Trying to issue an unavailable book
my_library.issue_book("Non-existent Book") # Trying to issue a book not in the library
```

OUTPUT:

```
Library Catalog:
Title: The Lord of the Rings, Author: J.R.R. Tolkien, Available: 3/3
Title: Pride and Prejudice, Author: Jane Austen, Available: 2/2
'The Lord of the Rings' has been issued.

Library Catalog:
Title: The Lord of the Rings, Author: J.R.R. Tolkien, Available: 2/3
Title: Pride and Prejudice, Author: Jane Austen, Available: 2/2
'The Lord of the Rings' has been issued.
'The Lord of the Rings' has been issued.
'The Lord of the Rings' is currently unavailable.
'Non-existent Book' not found in the library.
PS C:\Users\supriya k>
```

EXPLANATION:

1. **class Library:**: This line formally defines our new class named Library. This is the blueprint from which we can create multiple individual library instances if needed.

2. **__init__(self):**: This is the special initialization method (often called the constructor). It runs automatically when you create a new Library object (like my_library = Library()).
    - self is a reference to the specific instance of the Library class that is being created.
    - self.books = []: This is a key part. It creates an empty list and assigns it to the books attribute of the Library instance. This self.books list is where we will store all the information about the books that are added to this particular library object. We've chosen a list to hold our books, and each item in the list will represent a single book.
3. **add_book(self, book_title, author, quantity=1):**: This method handles the process of adding books to our library.
    - self: Again, refers to the specific library instance we are working with.
    - book_title, author: These are required arguments specifying the details of the book.
    - quantity=1: This is an optional argument. If you don't specify a quantity when calling the method (like my_library.add_book("Title", "Author")), it defaults to adding just one copy. If you specify it (like my_library.add_book("Title", "Author", 5)), it will add that many copies.
    - book = {"title": book_title, "author": author, "quantity": quantity, "available": quantity}: This line creates a Python dictionary to represent a single book entry. Using a dictionary allows us to store different pieces of information (title, author, quantity, and available copies) for each book in a structured way, using keys (like "title", "author") to access the corresponding values. We initialize "available" to be the same as the total "quantity" because all copies are available when the book is first added.
    - self.books.append(book): This line adds the newly created book dictionary to the self.books list that belongs to this library instance.
    - print(f"'{book_title}' by {author} added to the library."): This provides user feedback confirming the action.
4. **issue_book(self, book_title):**: This method simulates the action of someone borrowing a book.
    - self: Refers to the library instance.

- book_title: The title of the book the user wants to issue.
- for book in self.books:: The code iterates through every book dictionary currently stored in the self.books list.
- if book["title"] == book_title:: Inside the loop, it checks if the title of the current book dictionary matches the book_title requested by the user.
- **Handling availability (nested if/else):**
    - if book["available"] > 0:: If a matching book is found, this inner condition checks if there is at least one copy marked as "available".
        - If true (book is available), book["available"] -= 1 decreases the count of available copies by one. A success message is printed, and return stops the method's execution since we've found and issued the book.
    - else:: If a matching book is found
      but book["available"] is *not* greater than 0 (meaning all copies are already issued), an "unavailable" message is printed,
      and return stops the method.
- print(f"'{book_title}' not found in the library."): If the for loop finishes iterating through *all* the books in self.books without finding a dictionary where the "title" matches the requested book_title, this line is executed, informing the user that the book wasn't found in the library.

5. **display_books(self):**: This method provides a way to view the current contents of the library catalog.
    - self: Refers to the library instance.
    - if not self.books:: This checks if the self.books list is empty. The boolean value of an empty list is False, so not self.books is True if the list is empty.
        - If the list is empty, it prints "The library is empty." and return exits the method.
    - print("\nLibrary Catalog:"): If the list is not empty, it prints a header before listing the books.
    - for book in self.books:: It then iterates through each book dictionary in the self.books list.
    - print(f"Title: {book['title']}, Author: {book['author']}, Available: {book['available']}/{book['quantity']}"): For each book dictionary, it

accesses the values associated with the "title", "author", "available", and "quantity" keys and prints them in a formatted string, showing the available copies out of the total quantity.

Overall, this Library class demonstrates how to use object-oriented programming to bundle data (the list of books, with details for each) and the operations that can be performed on that data (adding, issuing, displaying). It also includes basic handling for common scenarios like a book not being found or not being available.