

# AI ASSISTED CODING

## LAB ASSIGNMENT – 6.2

NAME : K.SUPRIYA

HALL TICKET NO : 2403A52395

BATCH : 14

### TASK 1:

class Book:

```
def __init__(self, title, author, year):
```

```
    self.title = title
```

```
    self.author = author
```

```
    self.year = year
```

```
def get_summary(self):
```

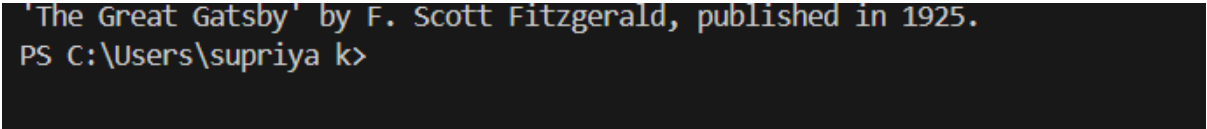
```
    return f'"{self.title}" by {self.author}, published in {self.year}.'
```

# Example usage

```
book1 = Book("The Great Gatsby", "F. Scott Fitzgerald", 1925)
```

```
print(book1.get_summary())
```

OUTPUT:



```
'The Great Gatsby' by F. Scott Fitzgerald, published in 1925.  
PS C:\Users\supriya k>
```

EXPLANATION:

The code defines a Python class named Book.

- **class Book::** This line starts the definition of the class named Book.

- **"""Represents a book with title, author, and publication year."""**: This is a docstring that explains the purpose of the class.
- **def \_\_init\_\_(self, title, author, year):**: This is the constructor method of the class. It's called when you create a new Book object.
  - **self**: Refers to the instance of the class being created.
  - **title, author, year**: These are parameters that you pass when creating a Book object. They represent the title, author, and publication year of the book.
  - **self.title = title**: This line assigns the value passed for title to the title attribute of the Book object.
  - **self.author = author**: This line assigns the value passed for author to the author attribute of the Book object.
  - **self.year = year**: This line assigns the value passed for year to the year attribute of the Book object.
- **def get\_summary(self):**: This defines a method called get\_summary within the Book class.
  - **self**: Again, refers to the instance of the class.
  - **return f"{self.title} by {self.author}, published in {self.year}"**: This line returns a formatted string that combines the title, author, and year attributes of the Book object into a readable summary. The f"" syntax is a f-string, which allows you to embed variables directly within the string.

In essence, this class provides a blueprint for creating objects that represent books, allowing you to store their key information and easily get a formatted summary.

## TASK 2:

```
def print_even_for():
```

```
    """Prints even numbers between 1 and 50 using a for loop."""
```

```
    print("Even numbers (for loop):")
```

```
    for i in range(2, 51, 2):
```

```
        print(i)
```

```
print_even_for()
```

OUTPUT:

```
Even numbers (for loop):
```

```
2  
4  
6  
8  
10  
12  
14
```

```
16  
18  
20  
22  
24  
26
```

```
28  
30  
32  
34  
36  
38  
40  
42  
44  
46  
48  
50
```

```
PS C:\Users\supriya k>
```

```
def print_even_while():
```

```
    """Prints even numbers between 1 and 50 using a while loop."""
```

```
    print("Even numbers (while loop):")
```

```
    i = 2
```

```
    while i <= 50:
```

```
        print(i)
```

```
        i += 2
```

print\_even\_while()

OUTPUT:

```
Even numbers (while loop):  
2  
4  
6  
8  
10  
12  
14  
16  
18  
20  
22
```

```
24  
26
```

```
28  
30  
32  
34  
36  
38  
40  
42  
44  
46  
48  
50  
PS C:\Users\supriya k>
```

EXPLANATION:

**Explanation of the print\_even\_for() function (using a for loop):**

1.
  - **def print\_even\_for():** This line defines a function named print\_even\_for.
  - **"""Prints even numbers between 1 and 50 using a for loop."""**: This is a docstring explaining what the function does.

- **print("Even numbers (for loop):")**: This line simply prints a descriptive header before the numbers.
- **for i in range(2, 51, 2)::** This is the core of the for loop.
  - for i in ...: This iterates through a sequence, assigning each item in the sequence to the variable i in turn.

1.

- **range(2, 51, 2)**: This is a built-in Python function that generates a sequence of numbers.
  - The first argument (2) is the starting number (inclusive).
  - The second argument (51) is the stopping number (exclusive), so the loop will go up to 50.
  - The third argument (2) is the step size, meaning the loop will increment by 2 in each iteration. This ensures that only even numbers are generated (starting from 2).

2. **print(i)**: Inside the loop, this line prints the current value of i (which will be an even number) in each iteration.

### Explanation of the `print_even_while()` function (using a while loop):

```
def print_even_while():
    """Prints even numbers between 1 and 50 using a while loop."""
    print("Even numbers (while loop):")
    i = 2
    while i <= 50:
        print(i)
        i += 2
```

1. **def print\_even\_while()**:: This line defines a function named `print_even_while`.
2. **"""Prints even numbers between 1 and 50 using a while loop."""**: This is a docstring explaining what the function does.
3. **print("Even numbers (while loop):")**: This line prints a descriptive header before the numbers.
4. **i = 2**: This line initializes a variable i to 2. This variable will be used as a counter and will represent the current number being checked.
5. **while i <= 50**:: This is the while loop condition. The code inside the loop will continue to execute as long as the value of i is less than or equal to 50.

6. **print(i):** Inside the loop, this line prints the current value of i.
7. **i += 2:** This line updates the value of i by adding 2 to it in each iteration. This is crucial for two reasons:
  - It ensures that i eventually becomes greater than 50, so the while loop condition becomes false and the loop terminates, preventing an infinite loop.
  - By adding 2 in each step, it ensures that only even numbers are considered and printed.

Both functions achieve the same result of printing even numbers from 2 to 50, but they use different looping constructs to do so. The for loop is often preferred when you know the number of iterations in advance (or can easily determine it from a range), while the while loop is more flexible for conditions that don't necessarily depend on a fixed number of iterations.

### TASK 3:

```
def get_grade(marks):
```

```
    if marks >= 90:
```

```
        return "A"
```

```
    elif marks >= 75:
```

```
        return "B"
```

```
    elif marks >= 50:
```

```
        return "C"
```

```
    else:
```

```
        return "Fail"
```

```
# Testing with sample inputs
```

```
print(get_grade(95)) # A
```

```
print(get_grade(80)) # B
```

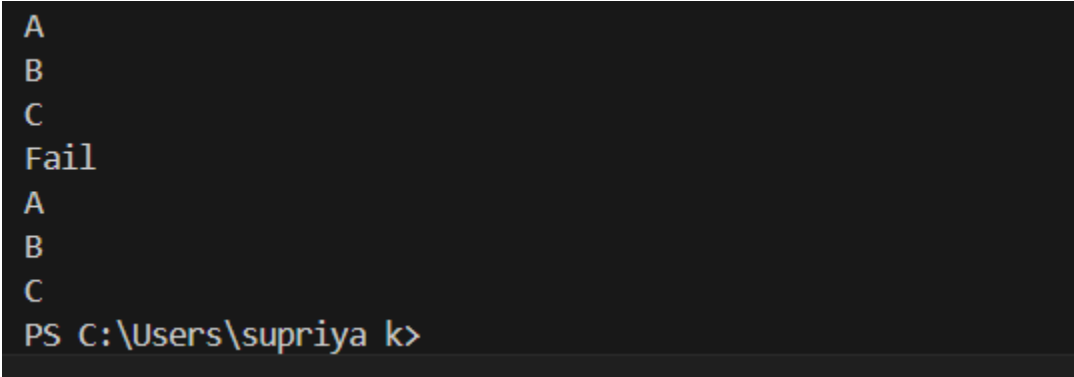
```
print(get_grade(65)) # C
```

```
print(get_grade(45)) # Fail
```

```
print(get_grade(90)) # A (boundary case)
```

```
print(get_grade(75)) # B (boundary case)
print(get_grade(50)) # C (boundary case)
```

OUTPUT:



```
A
B
C
Fail
A
B
C
PS C:\Users\supriya k>
```

EXPLANATION:

1. **def determine\_grade(marks)::** This line defines a function named `determine_grade` that takes one argument, `marks`.
2. **"""Determines the grade category based on marks.""":** This is a docstring explaining the function's purpose.
3. **if marks >= 90::** This is the first conditional check. If the value of `marks` is greater than or equal to 90, the code inside this `if` block is executed.
  - **return "A":** If the condition is true, the function immediately stops and returns the string "A".
4. **elif marks >= 80::** This is an "else if" condition. This check is only performed if the previous `if` condition (`marks >= 90`) was false. If the value of `marks` is greater than or equal to 80 (but less than 90), the code inside this `elif` block is executed.
  - **return "B":** If this condition is true, the function immediately stops and returns the string "B".
5. **elif marks >= 70::** This is another "else if" condition. This check is only performed if the previous `if` and `elif` conditions (`marks >= 90` and `marks >= 80`) were false. If the value of `marks` is greater than or equal to 70 (but less than 80), the code inside this `elif` block is executed.
  - **return "C":** If this condition is true, the function immediately stops and returns the string "C".
6. **else::** This is the final `else` block. The code inside this block is executed only if none of the preceding `if` or `elif` conditions were true (i.e., if `marks` is less than 70).

- **return "Fail":** If none of the above conditions are met, the function returns the string "Fail".

In summary, the `determine_grade` function checks the input marks against a series of thresholds using if-elif-else statements and returns the corresponding grade category ("A", "B", "C", or "Fail")

1. **# Test the function with sample inputs:** This is a comment indicating the purpose of this code block.
2. **print(f"Marks: ..., Grade: {determine\_grade(...)}"):** Each of these lines calls the `determine_grade` function with a specific number as input (e.g., `determine_grade(95)`).
  - The `f"..."` is an f-string used for formatting the output. It includes the input Marks: and the output Grade:.
  - `{determine_grade(...)}` inside the f-string calls the function and embeds its return value (the grade) directly into the string that is printed.
3. **Comments like # Test boundary A/B:** These comments explain which boundary condition or scenario each test case is designed to check. This is good practice for understanding the purpose of your tests.

These test cases are important because they help verify that the `determine_grade` function is working correctly for different inputs, especially at the boundaries between the grade categories.

## TASK 4:

```
def check_eligibility(age, has_id):
```

```
    """
```

Checks if a person is eligible to vote based on age and having an ID.

Args:

age: The age of the person (integer).

has\_id: A boolean indicating if the person has an ID (True or False).

Returns:

A string indicating eligibility ("Eligible to vote", "Not eligible: Must be at least 18",

"Not eligible: Must have an ID", or "Not eligible: Must be at least 18 and have an ID").



```

"""
if age >= 18:
    if has_id:
        return "Eligible to vote"
    else:
        return "Not eligible: Must have an ID"
else:
    if has_id:
        return "Not eligible: Must be at least 18"
    else:
        return "Not eligible: Must be at least 18 and have an ID"

# Test cases
print(f"Age: 20, Has ID: True -> {check_eligibility(20, True)}")
print(f"Age: 17, Has ID: True -> {check_eligibility(17, True)}")
print(f"Age: 20, Has ID: False -> {check_eligibility(20, False)}")
print(f"Age: 17, Has ID: False -> {check_eligibility(17, False)}")
print(f"Age: 18, Has ID: True -> {check_eligibility(18, True)}") # Boundary case

```

OUTPUT:

```

Age: 20, Has ID: True -> Eligible to vote
Age: 17, Has ID: True -> Not eligible: Must be at least 18
Age: 20, Has ID: False -> Not eligible: Must have an ID
Age: 17, Has ID: False -> Not eligible: Must be at least 18 and have an ID
Age: 18, Has ID: True -> Eligible to vote
PS C:\Users\supriya k>

```

EXPLANATION:

The goal of the `check_eligibility` function is to determine if a person meets the criteria to vote, which are being at least 18 years old and having a valid ID. The code uses **nested conditional statements** (an if statement inside another if or else statement) to evaluate these two conditions

- **def check\_eligibility(age, has\_id):** This defines the function that accepts two inputs: age (a number) and has\_id (a boolean value, which is either True or False).

- **The Outer if age >= 18:** This is the first and most important check. It asks, "Is the person's age 18 or greater?"
  - **If this is True:** The code proceeds *inside* this if block to the **nested if has\_id:**. This is because if they are old enough, the next thing to check is if they have an ID.
  - **If this is False:** The code skips the inner if within this block and goes directly to the **outer else block**. This is because if they are not old enough, they are ineligible regardless of whether they have an ID, but we still need to provide a specific reason for ineligibility.
- **The Nested if has\_id: (inside if age >= 18):** This check only happens if the person is 18 or older. It asks, "Does the person have an ID?"
  - **If this is True:** This means both conditions are met (18+ and has ID). The function returns "Eligible to vote".
  - **If this is False:** This means the person is 18+ but *does not* have an ID. The function returns "Not eligible: Must have an ID".
- **The Outer else: (corresponding to if age >= 18):** This block is executed only if the person is *younger* than 18.
  - **The Nested if has\_id: (inside the Outer else):** Even if they are too young, we still check if they have an ID to give a precise reason for ineligibility. It asks, "Does the person have an ID?"
    - **If this is True:** The person has an ID but is too young. The function returns "Not eligible: Must be at least 18".
    - **If this is False:** The person is both too young *and* does not have an ID. The function returns "Not eligible: Must be at least 18 and have an ID"

- **Age: 20, Has ID: True:** Tests the case where both conditions are met (eligible).
- **Age: 17, Has ID: True:** Tests the case where age is below the requirement, but they have an ID.
- **Age: 20, Has ID: False:** Tests the case where age is sufficient, but they lack an ID.
- **Age: 17, Has ID: False:** Tests the case where neither condition is met.
- **Age: 18, Has ID: True:** This is a **boundary case** test.
- It specifically checks the exact age of 18 to ensure that the >= 18 condition correctly includes 18 as eligible (assuming they have an ID).

## TASK 5:

class Rectangle:

"""Represents a rectangle with a width and a height."""

```
def __init__(self, width, height):  
    """Initializes a Rectangle object."""  
  
    if width <= 0 or height <= 0:  
        raise ValueError("Width and height must be positive values.")  
  
    self.width = width  
    self.height = height
```

```
def area(self):  
    """Calculates and returns the area of the rectangle."""  
  
    return self.width * self.height
```

```
def perimeter(self):  
    """Calculates and returns the perimeter of the rectangle."""  
  
    return 2 * (self.width + self.height)
```

# Example Usage:

```
my_rectangle = Rectangle(10, 5)  
print(f"Area: {my_rectangle.area()}")  
print(f"Perimeter: {my_rectangle.perimeter()}")
```

# Example with invalid input (will raise a ValueError):

```
# invalid_rectangle = Rectangle(-5, 10)
```

OUTPUT:

```
Area: 50  
Perimeter: 30  
PS C:\Users\supriya k>
```

EXPLANATION:

class code:

The code defines a Rectangle class, which is like a blueprint for creating rectangle objects.

- **\_\_init\_\_(self, width, height):** This is the setup part. When you create a Rectangle, you give it a width and height. It also checks that these values are positive.
- **area(self):** This calculates the area by multiplying the width and height you set earlier.
- **perimeter(self):** This calculates the perimeter using the formula  $2 * (\text{width} + \text{height})$ .

The example usage shows how to create a Rectangle object with specific dimensions (10 and 5) and then uses the area() and perimeter() methods to get and print their values. The output shows the results of these calculations.