PRACTICE SHEET

What you will learn:

Object-Oriented Programming (OOP), Time & Space Complexity, Matrix Operations (Transpose, Add, Multiply), Turtle Programming, Compiler vs Interpreter, Symbol Table

---

[EASY] Q1) What will be the output of the following Python code?

```
class A:
    def __init__(self):
        self.value = 5
    def get_val(self):
        return self.value * 2
class B(A):
    def __init__(self):
        super().__init__()
        self.value = 10
obj = B()
print(obj.get_val())
```

Options:
(a) 5
(b) 10
(c) 20
(d) 15

[EASY] Q2) Which of the following best describes composition in Python OOP?

(a) Inheriting methods from multiple base classes
(b) Creating objects of one class inside another class
(c) Overriding base methods in derived class
(d) Declaring private variables using __

[EASY] Q3) The Collatz Sequence also called the 3n + 1 problem, is a simple but mysterious numeric sequence that has remained unsolved by mathematicians. It has four rules:

- Begin with a positive, nonzero integer called n.
- If n is 1, the sequence terminates.
- If n is even, the next value of n is n / 2.
- If n is odd, the next value of n is 3n + 1.

For example, if the starting integer is 10, that number is even so the next number is 10 / 2, or 5. 5 is odd, so the next number is 3 × 5 + 1, or 16. 16 is even, so the next number is 8, which is even so the next number is 4, then 2, then 1. At 1, the sequence stops. The entire Collatz Sequence starting at 10 is: 10, 5, 16, 8, 4, 2, 1

Mathematicians have been unable to prove if every starting integer eventually terminates. This gives the Collatz Sequence the description of - the simplest impossible math problem.

Write a function named collatz() with a startingNumber parameter. The function returns a list of integers of the Collatz sequence that startingNumber produces. The first integer in this list must be startingNumber and the last integer must be 1. Your function should check if startingNumber is an integer less than 1, and in that case, return an empty list.

Examples:

collatz(0) == []

collatz(10) == [10, 5, 16, 8, 4, 2, 1]

collatz(11) == [11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]

collatz(12) == [12, 6, 3, 10, 5, 16, 8, 4, 2, 1]

[MEDIUM] Q4) You are designing a **modular neural network system** where:

- Each Layer contains a certain number of neurons.
- A Network is composed of multiple layers.
- The network is **fully connected** (each neuron connects to all neurons in the next layer).

Your task is to compute the **total number of connections** in the network.

Problem Statement:

1. Layer **Class:**
    - Stores the **number of neurons** in the layer.
    - Has a method get_connections(next_layer) that computes connections to the next layer.
2. Network **Class:**
    - Takes a list of Layer objects.
    - Computes the **total connections** in the network.

Twist (Extra Challenge):

- **Validate layer sizes** (must have ≥1 neuron).
- **Handle edge cases** (single-layer networks have zero connections).
- **Add a** __str__ **method** to visualize the network structure.

Hints for Implementation:

1. **Layer Class:**
    - In __init__, validate that neurons is at least 1
    - In get_connections, return 0 if next_layer is None, else multiply neurons
    - In __str__, return something like "Layer(5 neurons)"
2. **Network Class:**
    - In __init__, validate that layers has at least 1 layer
    - In total_connections, sum connections between consecutive layers
    - In __str__, join layer strings with arrows (→)
3. **Edge Cases to Consider:**
    - Single layer network (should return 0 connections)
    - Empty layers list (should raise error)
    - Layer with 0 neurons (should raise error)

[EASY] Q5) What is the output of the code?

```
class A:
    def method(self):
        return "A"


class B(A):
    def method(self):
        return "B"


class C(A):
    def method(self):
        return "C"


class D(B, C):
    pass
print(D().method())   # Output?
```

[EASY] Q6) What is the output?

```
class Material:
    def __init__(self, elements):
        self.elements = elements
    def process(self):
        return [e.upper() for e in self.elements]
class Alloy(Material):
    def process(self):
        return sorted(set(super().process() + ['ZINC']))
a = Alloy(['iron', 'carbon', 'iron'])
```

print(a.process())

(a) ['CARBON', 'IRON', 'ZINC']
(b) ['IRON', 'CARBON', 'ZINC']
(c) ['ZINC']
(d) ['IRON', 'CARBON']

[MEDIUM] Q7)

Type	**What it does**	**Pros**	**Cons**
Compiler	Translates entire code into machine language before execution (e.g., C, C++).	Faster execution (since already compiled).	Needs recompilation after changes.
Interpreter	Executes code line-by-line (e.g., Python, JavaScript).	No waiting for compilation, easier debugging.	Slower since it translates on the fly.

The Tiny Script Interpreter : You're building a micro compiler for a simple scripting language that can handle basic variable assignments like:

x = 5

y = 10

result = x + y

Your task is to create a TokenEvaluator class that processes a stream of tokens and returns the variable bindings (symbol table).

Problem Statement:

Create a class that:

1. Takes a list of tokens in the format (TYPE, VALUE)
2. Evaluates the stream to create variable bindings
3. Handles these token types:
    - VAR: Variable name

    - EQUALS: Assignment operator
    - NUMBER: Numeric value
    - OPERATOR: +, -, *, /
    - SEMICOLON: End of statement

Requirements:

- Implement basic arithmetic operations
- Store variables in a symbol table

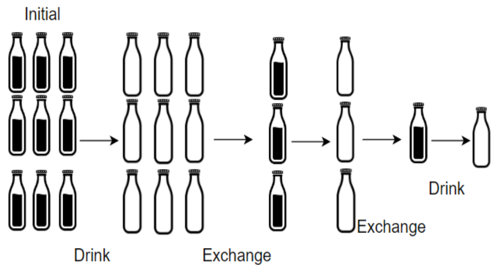
  Symbol Table : A dictionary that stores variable names and their values.

  Example: {"x": 5, "y": 10, "result": 15}

  Used by compilers/interpreters to **track variables** during execution.
- Handle sequential statements
- Validate syntax (e.g., "x = 5 +" should raise error)

[MEDIUM] Q8) There are numBottles water bottles that are initially full of water. You can exchange numExchange empty water bottles from the market with one full water bottle.

The operation of drinking a full water bottle turns it into an empty bottle. Given the two integers numBottles and numExchange, return the **maximum** number of water bottles you can drink.



Input: numBottles = 9, numExchange = 3
Output: 13
Explanation: You can exchange 3 empty bottles to get 1 full water bottle.
Number of water bottles you can drink: 9 + 3 + 1 = 13.

[EASY] Q9) **What is Time Complexity?**

Definition: Time Complexity is a way to describe **how the number of steps** (operations) an algorithm takes **grows** with the size of the input.

It helps us **compare the efficiency** of different algorithms.

Real-Life Analogy:

Imagine two people cleaning a house:

- Person A cleans **1 room per hour**.
- Person B cleans **2 rooms per hour**.

If there are 100 rooms, A takes 100 hours, B takes 50 hours.
But as rooms (input) increase, you see how time **grows with input**.

Code Example:

```
# Task: Print all pairs in an array

arr = [1, 2, 3, 4]

for i in range(len(arr)):

   for j in range(len(arr)):

       print(f"Pair: ({arr[i]}, {arr[j]})")

# Time Complexity: O(n^2)

# Because for each element, we're running another loop of size n.

# So total operations = n * n = n²
```

Breakdown:

If array has:

- 4 elements → 16 operations
- 100 elements → 10,000 operations

That's why we say this algorithm has **quadratic time complexity O(n^2)**.

Takeaway:

- **O(1)** – Constant time (like checking if a light is on)
- **O(n)** – Linear time (walking through a queue of people)
- **O(n²)** – Quadratic time (comparing every student to every other student)

Question: Analyze code and write the time complexity.

```
def print_triplets(arr):
  for i in arr:
    for j in arr:
      for k in arr:
        print(i, j, k)
```

[EASY] Q10) **What is Space Complexity?**

Definition: Space Complexity is the amount of **memory or storage** required by an algorithm to run, **based on input size**.

It helps us understand if our algorithm uses **extra space** or can run in-place.

Real-Life Analogy:

Imagine you're packing for a trip:

- Option 1: Use 1 backpack (in-place, minimal space)
- Option 2: Use 1 backpack + 3 extra bags (extra space)

Which one is more efficient to carry?

Same for programs — using less memory is often better.

Code Example:

```
# Task: Create a new array containing squares of original array

arr = [1, 2, 3, 4]

squares = []
```

```
for i in arr:
    squares.append(i * i)
print(squares)
# Time Complexity: O(n) — we go through the array once
# Space Complexity: O(n) — we create a new array of same size
```

Can we reduce space?

Yes, by modifying in-place:

```
arr = [1, 2, 3, 4]
for i in range(len(arr)):
    arr[i] = arr[i] * arr[i]

print(arr)
# Time Complexity: O(n)
# Space Complexity: O(1) — no new array, just modifying existing one
```

Takeaway:

- **O(1)** – Constant space (in-place modification)
- **O(n)** – Linear space (storing extra array or data)
- **O(n^2)** – Matrix or nested structure space

Question: Analyze code and write the time complexity.

```
def create_matrix(n):
    matrix = []
    for i in range(n):
        row = []
        for j in range(n):
            row.append(i * j)
        matrix.append(row)
    return matrix
```

[HARD] Q11) Welcome to the **Robotics Lab 42X**, where brilliant engineers like you bring machines to life.

Today, you're tasked with building two robotic hands:

- One is **cost-efficient and simple** (Single Motor Hand – "Hand A")
- The other is **precise and flexible** (Dual Motor Hand – "Hand B")

Draw the letter **'T'** using both robotic hands and analyze their performance.

Understanding the Hands:

Hand A: The Single-Motor Explorer

- It has **one extendable arm** (like a telescope).
- It can **rotate its base(**due to a single motor**)** and **change the length** of the arm.
- It has **limited reach**(as motor can only rotate 180 degrees)
- At the end, a pen is attached to draw.

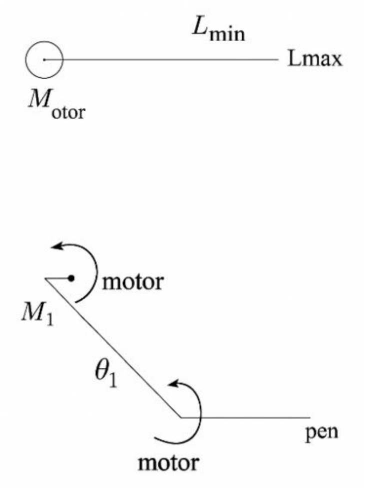
Hand B: The Dual-Motor Artist

- It has **two rotating segments** (like a human arm : shoulder(length a) + elbow(length b)).
- Both segments starts with a motor(has two motors rotating 180 degrees)
- It can **create curves, reach diagonals**, and access more positions.
- At the end, a pen is attached to draw.

Allowed Actions:

- increase_length()
- decrease_length()

- rotate(angle)
- leave_dot()



Turtle Drawing Template – Setup Guide

Here's a handy setup template to use Turtle in Python:

```
import turtle

def setup_turtle():

   screen = turtle.Screen()

   screen.title("Robotic Hand Drawing - Turtle Simulation")

   screen.bgcolor("white")


   pen = turtle.Turtle()

   pen.speed(1)

   pen.pensize(2)

   pen.color("black")

   return pen


def reset_position(pen, x=0, y=0, angle=90):
   pen.penup()
   pen.goto(x, y)
   pen.setheading(angle)
   pen.pendown()

setup_turtle()
pen = turtle.Turtle()
for i in range(4):
   pen.forward(50)
   pen.right(90)

turtle.done()
```

[MEDIUM] Q12) You are building a **matrix computation library** for a cryptography project. Your task is to design a Matrix class that supports:

- **Transposition** (flipping rows and columns)
- **Matrix addition** (element-wise)
- **Matrix multiplication** (dot product)

To make it more interesting, you must **encapsulate** these operations inside a MathOperations class to keep the code modular.

Problem Statement:

1. Matrix **Class:**
    - Stores a 2D list of numbers.
    - Methods:
        - transpose() → Returns a new transposed matrix.
        - add(other) → Adds another matrix (element-wise).
        - multiply(other) → Performs matrix multiplication.

2. MathOperations **Class:**
    - Contains a Matrix **object** as an attribute.
    - Provides methods:
        - apply_transpose() → Returns transposed matrix.
        - apply_addition(other_matrix) → Adds two matrices.
        - apply_multiplication(other_matrix) → Multiplies two matrices.

Challenges:

- **Validate matrix dimensions** before operations (e.g., addition requires same size; multiplication requires cols == other.rows).
- **Raise custom exceptions** for invalid operations (e.g., MatrixSizeMismatchError).

[MEDIUM] Q13) You are designing a security system for a **high-tech vault** that stores sensitive data. To protect the password from being exposed in memory, you need to create a **PasswordMasker** class that:

1. Takes a password and **internally masks it** using a simple ASCII shift (e.g., shifting each character by +1).
2. **Never stores the original password** directly—only the masked version.
3. Provides a method to **check if a guessed password matches** the original.

Your goal is to ensure **encapsulation**—no external code should be able to access the original password, only the masked version.

Problem Statement:

Create a PasswordMasker class with:

1. A constructor (__init__) that takes a password and **immediately masks it**.
2. A method check_password(guess) that returns True if the guess matches the original password.
3. A method get_masked_password() that returns the **masked version** (for debugging).
4. **No direct access** to the original password (encapsulation).

Password Masking Rule:

- Shift each character in the password **forward by 1 in ASCII**.
    - Example: "abc" → "bcd"
    - Special case: "z" → "a" (wraps around).

Hints (Try Solving Before Peeking!):

1. Use **private attributes** (e.g., _masked_password) to store the shifted password.
2. Loop through each character in the input password and apply the shift.
3. For check_password(guess), shift the guess and compare it to the stored masked version.
4. Handle edge cases (empty password, non-alphabetic characters).

[EASY] Q14) There are n people standing in a line labeled from 1 to n. The first person in the line is holding a pillow initially. Every second, the person holding the pillow passes it to the next person standing in the line. Once the pillow reaches the end of the line, the direction changes, and people continue passing the pillow in the opposite direction.

For example, once the pillow reaches the n^{th} person they pass it to the $n - 1^{th}$ person, then to the $n - 2^{th}$ person and so on.

Given the two positive integers n and time, return the index of the person holding the pillow

after time seconds.

Example 1:

Input: n = 4, time = 5

Output: 2

Explanation: People pass the pillow in the following way: 1 -> 2 -> 3 -> 4 -> 3 -> 2.

After five seconds, the 2^{nd} person is holding the pillow.

Example 2:

Input: n = 3, time = 2

Output: 3

Explanation: People pass the pillow in the following way: 1 -> 2 -> 3.

After two seconds, the 3rd person is holding the pillow.

[HARD] Q15) You're working as an intern in an electronics R&D lab. Your mentor has asked you to **simulate the behavior of basic passive components** (Resistors and Capacitors) and **analyze their impedance in an AC circuit**.

You are required to:

1. Build classes for Resistor and Capacitor with impedance computation methods.
2. Implement color-band decoding for resistors (using the standard 3-band code).
3. Create a SeriesCircuit class that calculates the total impedance of connected components at a given frequency.
4. Use object-oriented principles like encapsulation, constructor overloading, and method overriding.

Tasks:

Step 1: Resistor Class

- Create a Resistor class.
- Allow instantiation in **two ways**:
  - Direct value: Resistor(1000)
  - Using 3 color bands: Resistor(bands=['brown', 'black', 'red'])
- Implement a method get_impedance() which returns its resistance.
- Override __str__() to display values as 1000.0 Ω, 1.0 kΩ, or 1.0 MΩ.

Step 2: Capacitor Class

- Create a Capacitor class with value in **farads**.
- Implement get_impedance(frequency) using:

$$Z=1/(2\pi fC)$$

- Override __str__() to display values like 1.0 µF, 220.0 nF, etc.

Step 3: Series Circuit Class

- Create a SeriesCircuit class that accepts multiple components.

- Implement a method get_impedance(frequency) which returns the **sum of individual impedances**.
- Override __str__() to display the components in the series.

Hints:

- Use dictionaries for color band decoding:

  COLOR_CODES = {'black': 0, 'brown': 1, 'red': 2, 'orange': 3, ...}

  MULTIPLIERS = {'black': 1, 'brown': 10, 'red': 100, ...}
- Handle both resistor input types using __init__() by checking value or bands.
- Remember that **impedance of a resistor is constant**, but for a capacitor it **depends on frequency**.

Starter Code (Template):

```
class Resistor:

  # Add color code dictionaries here


  def __init__(self, value=None, bands=None):

    # Initialize using either value or color bands

  def _calculate_value_from_bands(self, bands):

    # Calculate resistance value from color bands

  def get_impedance(self):

    # Return resistance (impedance)

  def __str__(self):

    # Return a formatted string

class Capacitor:

  def __init__(self, farads):

    # Initialize with capacitance value

  def get_impedance(self, frequency):

    # Return impedance using formula
    def __str__(self):
        # Format the capacitor value in µF or nF
class SeriesCircuit:
    def __init__(self, *components):
        # Store all components
    def get_impedance(self, frequency):
        # Add impedance of all components
    def __str__(self):
        # Return string listing all components
```

Expected Output:

```
=== Resistor Examples ===
Resistor 1: 1.0 kΩ
Resistor 2: 1.0 kΩ

=== Capacitor Example ===
Capacitor: 1.0 µF
Impedance at 1000Hz: 159.2 ohms

=== Series Circuit Example ===
Series circuit: [1.0 kΩ, 1.0 µF]
Total impedance at 1000Hz: 1159.2 ohms
```

Follow-up Challenges:

After completing the above:

1. **Inductor Class**
   Add a new class Inductor, which calculates impedance using:

   $$Z=2\pi fL$$

2. **Parallel Circuit Class**
   Implement a ParallelCircuit class where total impedance is calculated as:

   $$1/Z_{\mathbf{total}} = \sum 1/Z_{\mathbf{i}}$$

3. **Plotting Impedance vs Frequency**
   Using matplotlib, plot how the impedance of a capacitor/inductor changes with frequency.