# Python Coding Task

Time: 30 Minutes

Level: Intermediate

## Q1. Understanding Access Specifiers

Create a class `Student` with the following properties:  
  
Class Requirements:  
1. `name` → Public attribute   
2. `\_roll\_number` → Protected attribute   
3. `\_\_marks` → Private attribute   
  
Implement the following methods:  
- Constructor to initialize all attributes.  
- `display\_details()` → Public method to display all attribute values.  
- `\_update\_roll\_number(new\_roll)` → Protected method to update roll number.  
- `\_\_update\_marks(new\_marks)` → Private method to update marks.  
- `access\_private\_method(new\_marks)` → Public method that uses the private method `\_\_update\_marks`.

## Q2. Demonstrate Access

In the main section:  
- Create an object of the `Student` class.  
- Modify and print the `name` directly.  
- Modify and print the `\_roll\_number` directly.  
- Try accessing `\_\_marks` directly and observe the result.

## Q3. Inheritance and Access Control

Create a subclass `Topper` that inherits from `Student` and includes:  
- A method `try\_access()` that attempts to access `\_roll\_number` and `\_\_marks` from the subclass.  
- Show what works and what doesn't.

## Q4. Use of Name Mangling

Demonstrate how to access the private attribute `\_\_marks` using name mangling technique from outside the class.

## Q5. Reflection

Answer the following short questions:  
1. Why can’t private members be accessed directly?  
2. What is the purpose of using protected members in class design?  
3. How does name mangling help with private members in Python?

**Answers:**

1. class Student:

def \_\_init\_\_(self, name, roll\_number, marks):

self.name = name

self.\_roll\_number = roll\_number

self.\_\_marks = marks

def display\_details(self):

print(f"Name: {self.name}")

print(f"Roll Number: {self.\_roll\_number}")

print(f"Marks: {self.\_\_marks}")

def \_update\_roll\_number(self, new\_roll):

self.\_roll\_number = new\_roll

print(f"Roll number updated to: {self.\_roll\_number}")

def \_\_update\_marks(self, new\_marks):

self.\_\_marks = new\_marks

print(f"Marks updated to: {self.\_\_marks}")

def access\_private\_method(self, new\_marks):

self.\_\_update\_marks(new\_marks)

1. student1 = Student("Arun", 101, 85)

student1.name = "Bhuvan"

print("Updated Name:", student1.name)

student1.\_roll\_number = 102

print("Updated Roll Number:", student1.\_roll\_number)

try:

print("Marks (direct):", student1.\_\_marks)

except AttributeError as e:

print("Error:", e)

student1.access\_private\_method(95)

student1.display\_details()

1. class Topper(Student):

def \_\_init\_\_(self, name, roll\_number, marks):

super().\_\_init\_\_(name, roll\_number, marks)

def try\_access(self):

print("Accessing protected \_roll\_number:", self.\_roll\_number)

try:

print("Trying to access private \_\_marks:", self.\_\_marks)

except AttributeError as e:

print("Error:", e)

topper1 = Topper("Divya", 201, 98)

topper1.try\_access()

1. print("Accessing \_\_marks via name mangling:", student1.\_Student\_\_marks)
2. **1. Why can’t private members be accessed directly?**

In Python, private members (prefixed with \_\_) are **name-mangled** to prevent accidental access and to protect internal class data. This promotes **encapsulation**, one of the pillars of OOP.

**2. What is the purpose of using protected members in class design?**

Protected members (prefixed with \_) indicate that they are intended for **internal use** or **subclass access only**, though they **can still be accessed** from outside. It’s more of a **convention than enforcement**.

**3. How does name mangling help with private members in Python?**

Name mangling changes \_\_marks to \_ClassName\_\_marks, making it **harder to access directly**. It helps prevent **accidental overrides** and preserves data integrity, especially in inheritance.