Encapsulation---->>> Encapsulation is one of the fundamental concepts in object-oriented programming (OOP). It describes the idea of wrapping data and the methods that work on data within one unit. This puts restrictions on accessing variables and methods directly and can prevent the accidental modification of data. To prevent accidental change, an object's variable can only be changed by an object's method. Those types of variables are known as private variables.

A class is an example of encapsulation as it encapsulates all the data that is member functions, variables, etc. The goal of information hiding is to ensure that an object's state is always valid by controlling access to attributes that are hidden from the outside world.

Protected member---->>>

Protected members (in C++ and JAVA) are those members of the class that cannot be accessed outside the class but can be accessed from within the class and its subclasses. To accomplish this in Python, just follow the convention by prefixing the name of the member by a single underscore "_".

class Tree:

def **init**(self, height):

self._height = height

pine = Tree(20)

pine._height

Private members---->>>

Private members are similar to protected members, the difference is that the class members declared private should neither be accessed outside the class nor by any base class. In Python, there is no existence of Private instance variables that cannot be accessed except inside a class.

However, to define a private member prefix the member name with double underscore "__".

class Tree:

def **init**(self, height):

self.__height = height

pine = Tree(20)

pine.__height

Practice Question: Question 1: Library Management System Scenario: You are tasked with developing a library management system where users can borrow and return books.

Requirements:

Create a Book class that has the following private attributes: title, author, and isbn. Implement methods to get and set the values of these attributes (getters and setters). Create a Library class that has a list of Book objects. Implement methods to add a new book, remove a book by its ISBN, and list all available books. Use encapsulation to ensure that the books can only be manipulated through the Library class. Task:

Implement the Book and Library classes with proper encapsulation

```
# prompt: Practice Question:
Question 1: Library Management System
# Scenario: You are tasked with developing a library management system
where users can borrow and return books.
# Requirements:
# Create a Book class that has the following private attributes: ti

class Book:
  """
  A class representing a book in a library.
  """

  def __init__(self, title, author, isbn):
    self.__title = title
    self.__author = author
    self.__isbn = isbn

  def get_title(self):
    return self.__title

  def set_title(self, title):
    self.__title = title

  def get_author(self):
    return self.__author

  def set_author(self, author):
    self.__author = author

  def get_isbn(self):
    return self.__isbn

  def set_isbn(self, isbn):
    self.__isbn = isbn

class Library:
  """
  A class representing a library.
  """

  def __init__(self):
    self.__books = []
```

```
  def add_book(self, book):
    self.__books.append(book)

  def remove_book(self, isbn):
    for i, book in enumerate(self.__books):
      if book.get_isbn() == isbn:
        del self.__books[i]
        return
    raise ValueError(f"Book with ISBN {isbn} not found.")

  def list_books(self):
    for book in self.__books:
      print("Title:", book.get_title())
      print("Author:", book.get_author())
      print("ISBN:", book.get_isbn())
      print()

# Create a library
library = Library()

# Add some books
library.add_book(Book("The Great Gatsby", "F. Scott Fitzgerald",
"9780743273565"))
library.add_book(Book("To Kill a Mockingbird", "Harper Lee",
"9780062420701"))

# List all books
library.list_books()

# Remove a book
library.remove_book("9780062420701")

# List all books again
library.list_books()
```

```
Title: The Great Gatsby
Author: F. Scott Fitzgerald
ISBN: 9780743273565

Title: To Kill a Mockingbird
Author: Harper Lee
ISBN: 9780062420701

Title: The Great Gatsby
Author: F. Scott Fitzgerald
ISBN: 9780743273565
```

Question 2: Bank Account Management Scenario: Develop a bank account management system where users can create accounts, deposit, and withdraw money.

Requirements:

Create a BankAccount class with private attributes: account_number, account_holder, and balance. Implement methods to get the account details and balance (getters). Implement methods to deposit and withdraw money, ensuring that the balance cannot go negative. Use encapsulation to ensure that the balance cannot be directly modified.

```
class bank:
  def __init__(self,account_number,account_holder,balance):
    self.__account_number=account_number
    self.__account_holder=account_holder
    self.__balance=balance
  def account_no_get(self):
    return self.__account_number
  def account_holder_get(self):
    return self.__account_holder
  def balance_get(self):
    return self.__balance
  def deposit(self,amount):
    if amount>0:
      self.__balance+=amount
    else:
      print("invalid")
  def withdraw(self,amount):
    if(amount>0 and amount<=self.__balance):
      self.__balance-=amount
    else:
      print("invalid")

  def display(self):
    print("account_number",self.__account_number)
    print("account_holder",self.__account_holder)
    print("balance",self.__balance)
mob1=bank(123,"ram",1000)
mob1.account_no_get()
mob1.account_holder_get()
mob1.balance_get()
mob1.deposit(500)
mob1.withdraw(200)
mob1.display()
```

```
account_number 123
account_holder ram
balance 1300
```

Question 3: Employee Management System Scenario: You are developing an employee management system for a company.

Requirements:

Create an Employee class with private attributes: employee_id, name, position, and salary. Implement methods to get and set the values of these attributes, with validation on the salary (e.g., salary cannot be negative). Create a Department class that has a list of Employee objects.

```
class Employee:
    def __init__(self, employee_id, name, position, salary):
        self.__employee_id = employee_id
        self.__name = name
        self.__position = position
        self.__salary = salary
    def employee_id_get(self):
      return self.__employee_id
    def employee_id_set(self,id):
      self.__employee_id=id
    def employee_name_get(self):
      return self.__name
    def employee_name_set(self,name):
      self.__name=name
    def employee_position_get(self):
      return self.__position
    def employee_position_set(self,position):
      self.__position=position
    def employee_salary_get(self):
      return self.__salary
    def employee_salary_set(self,salary):
      if salary>0:
        self.__salary=salary
      else:
        print("invalid")

class department:
  def __init__(self):
    self.__employee=[]
  def add_employee(self,employee):
    self.__employee.append(employee)
  def remove_employee(self,employee_id):
    for i,emp in enumerate(self.__employee):
      if emp.employee_id_get()==employee_id:
        del self.__employee[i]
        return
    raise ValueError(f"Employee with employee_id {employee_id} not
found.")
  def list_employee(self):
    for emp in self.__employee:
      print("Employee_id is:",emp.employee_id_get())
      print("Employee_name is:",emp.employee_name_get())
      print("Employee_position is:",emp.employee_position_get())
      print("Employee_salary is:",emp.employee_salary_get())
mob1=department()
mob1.add_employee(Employee(123,"Mehak",10000,"CEO"))
```

```
mob1.add_employee(Employee(234,"jacky",10000,"CEO"))
mob1.list_employee()
mob1.remove_employee(234)
mob1.list_employee()
```

```
Employee_id is: 123
Employee_name is: Mehak
Employee_position is: 10000
Employee_salary is: CEO
Employee_id is: 234
Employee_name is: jacky
Employee_position is: 10000
Employee_salary is: CEO
Employee_id is: 123
Employee_name is: Mehak
Employee_position is: 10000
Employee_salary is: CEO
```

Question 4: Online Shopping System Scenario: Develop an online shopping system where users can add products to a shopping cart and view the total cost.

Requirements:

Create a Product class with private attributes: product_id, name, and price. Implement methods to get the product details (getters). Create a ShoppingCart class that has a list of Product objects. Implement methods to add a product to the cart, remove a product by its ID, and calculate the total cost of all products in the cart. Use encapsulation to ensure that products in the cart can only be manipulated through the ShoppingCart class. Task:

```
class product:
  def __init__(self,product_id,name,price):
    self.__product_id=product_id
    self.__name=name
    self.__price=price
  def product_id_get(self):
    return self.__product_id
  def product_name_get(self):
    return self.__name
  def product_price_get(self):
    return self.__price
  def product_id_set(self,id):
    self.__product_id=id
  def product_name_set(self,name):
    self.__name=name
  def product_price_set(self,price):
    self.__price=price
class shoppingcart:
  def __init__(self):
    self.__product=[]
  def add_product(self,product):
```

```
    self.__product.append(product)
  def remove_product(self,product_id):
    for i,prod in enumerate(self.__product):
      if prod.product_id_get()==product_id:
        del self.__product[i]
        return
    raise ValueError(f"Product with product_id {product_id} not
found.")
  def amount_product(self):
    total=0
    for prod in self.__product:
      total+=prod.product_price_get()
    print("total amount is:",total)

  def display(self):
    for prod in self.__product:
      print("product_id is:",prod.product_id_get())
      print("product_name is:",prod.product_name_get())
      print("product_price is:",prod.product_price_get())
mob1=shoppingcart()
mob1.add_product(product(123,"laptop",10000))
mob1.add_product(product(234,"mobile",10000))
mob1.display()
# mob1.remove_product(234)
mob1.display()
mob1.amount_product()
```

```
product_id is: 123
product_name is: laptop
product_price is: 10000
product_id is: 234
product_name is: mobile
product_price is: 10000
product_id is: 123
product_name is: laptop
product_price is: 10000
product_id is: 234
product_name is: mobile
product_price is: 10000
total amount is: 20000
```

Question 5: School Management System Scenario: Develop a school management system where you can manage students and their courses.

Requirements:

Create a Student class with private attributes: student_id, name, and courses (a list of course names). Implement methods to get and set the student's details and courses (getters and setters). Create a School class that has a list of Student objects. Implement methods to add a

new student, remove a student by their ID, and list all students and their courses. Use encapsulation to ensure that students can only be manipulated through the School class. Task:

Implement the Student and School classes with proper encapsulation. Add methods to add, remove, and list students and their courses.

```
# prompt: Question 5: School Management System
# Scenario: Develop a school management system where you can manage
students and their courses.
# Requirements:
# Create a Student class with private attributes: student_id, name,
and courses (a list of course names).
# Implement methods to get and set the student's details and courses
(getters and setters).
# Create a School class that has a list of Student objects.
# I

class Student:
  """
  A class representing a student in a school.
  """

  def __init__(self, student_id, name, courses):
    self.__student_id = student_id
    self.__name = name
    self.__courses = courses

  def get_student_id(self):
    return self.__student_id

  def set_student_id(self, student_id):
    self.__student_id = student_id

  def get_name(self):
    return self.__name

  def set_name(self, name):
    self.__name = name

  def get_courses(self):
    return self.__courses

  def set_courses(self, courses):
    self.__courses = courses

class School:
  """
  A class representing a school.
  """

  def __init__(self):
```

```
    self.__students = []

  def add_student(self, student):
    self.__students.append(student)

  def remove_student(self, student_id):
    for i, student in enumerate(self.__students):
      if student.get_student_id() == student_id:
        del self.__students[i]
        return
    raise ValueError(f"Student with student_id {student_id} not
found.")

  def list_students(self):
    for student in self.__students:
      print("Student ID:", student.get_student_id())
      print("Name:", student.get_name())
      print("Courses:", student.get_courses())
      print()

# Create a school
school = School()

# Add some students
school.add_student(Student(123, "John Doe", ["Math", "Science"]))
school.add_student(Student(456, "Jane Doe", ["English", "History"]))

# List all students
school.list_students()

# Remove a student
school.remove_student(456)

# List all students again
school.list_students()
```

```
Student ID: 123
Name: John Doe
Courses: ['Math', 'Science']

Student ID: 456
Name: Jane Doe
Courses: ['English', 'History']

Student ID: 123
Name: John Doe
Courses: ['Math', 'Science']
```

Question 6: Car Rental System Scenario: Develop a car rental system where users can rent and return cars.

Requirements:

Create a Car class with private attributes: car_id, make, model, and availability (boolean). Implement methods to get and set the car's details and availability (getters and setters). Create a CarRental class that has a list of Car objects. Implement methods to add a new car, remove a car by its ID, and list all available cars. Implement methods to rent a car (set its availability to False) and return a car (set its availability to True). Use encapsulation to ensure that cars can only be manipulated through the CarRental class. Task:

Implement the Car and CarRental classes with proper encapsulation. Add methods to add, remove, list, rent, and return cars.

```
# prompt: Question 6: Car Rental System
# Scenario: Develop a car rental system where users can rent and
return cars.
# Requirements:
# Create a Car class with private attributes: car_id, make, model, and
availability (boolean).
# Implement methods to get and set the car's details and availability
(getters and setters).
# Create a CarRental class that has a list of Car objects.
# Implement methods to add a new car, r

class Car:
  """
  A class representing a car in a rental system.
  """

  def __init__(self, car_id, make, model, availability):
    self.__car_id = car_id
    self.__make = make
    self.__model = model
    self.__availability = availability

  def get_car_id(self):
    return self.__car_id

  def set_car_id(self, car_id):
    self.__car_id = car_id

  def get_make(self):
    return self.__make

  def set_make(self, make):
    self.__make = make

  def get_model(self):
    return self.__model

  def set_model(self, model):
    self.__model = model
```

```
  def get_availability(self):
    return self.__availability

  def set_availability(self, availability):
    self.__availability = availability

class CarRental:
  """
  A class representing a car rental service.
  """

  def __init__(self):
    self.__cars = []

  def add_car(self, car):
    self.__cars.append(car)

  def remove_car(self, car_id):
    for i, car in enumerate(self.__cars):
      if car.get_car_id() == car_id:
        del self.__cars[i]
        return
    raise ValueError(f"Car with car_id {car_id} not found.")

  def list_available_cars(self):
    for car in self.__cars:
      if car.get_availability():
        print("Car ID:", car.get_car_id())
        print("Make:", car.get_make())
        print("Model:", car.get_model())
        print()

  def rent_car(self, car_id):
    for car in self.__cars:
      if car.get_car_id() == car_id:
        car.set_availability(False)
        return
    raise ValueError(f"Car with car_id {car_id} not found or not
available.")

  def return_car(self, car_id):
    for car in self.__cars:
      if car.get_car_id() == car_id:
        car.set_availability(True)
        return
    raise ValueError(f"Car with car_id {car_id} not found.")

# Create a car rental service
car_rental = CarRental()
```

```
# Add some cars
car_rental.add_car(Car(123, "Toyota", "Corolla", True))
car_rental.add_car(Car(456, "Honda", "Civic", True))

# List available cars
car_rental.list_available_cars()

# Rent a car
car_rental.rent_car(123)

# List available cars again
car_rental.list_available_cars()

# Return the car
car_rental.return_car(123)

# List available cars again
car_rental.list_available_cars()
```

Practice Questions on Getter and Setter Methods 1. Bank Account Management System • Create a BankAccount class with private attributes account_number, account_holder, and balance. • Implement getter and setter methods for account_number and account_holder. • Implement a setter method for balance that checks if the balance being set is non-negative.

```
# prompt: Practice Questions on Getter and Setter Methods
# 1. Bank Account Management System
# •  Create a BankAccount class with private attributes
account_number, account_holder, and balance.
# •  Implement getter and setter methods for account_number and
account_holder.
# •  Implement a setter method for balance that checks if the balance
being set is non-negative.

class BankAccount:
    def __init__(self, account_number, account_holder, balance):
        self.__account_number = account_number
        self.__account_holder = account_holder
        self.__balance = balance

    # Getter methods
    def get_account_number(self):
        return self.__account_number

    def get_account_holder(self):
        return self.__account_holder
    def get_balance(self):
        return self.__balance

    # Setter methods
```

```
    def set_account_number(self, account_number):
        self.__account_number = account_number

    def set_account_holder(self, account_holder):
        self.__account_holder = account_holder

    def set_balance(self, balance):
        if balance >= 0:
            self.__balance = balance
        else:
            print("Invalid balance. Balance cannot be negative.")

# Create a bank account
account = BankAccount("123456789", "John Doe", 1000)

# Get account details
print("Account Number:", account.get_account_number())
print("Account Holder:", account.get_account_holder())
print("Balance:", account.get_balance())

# Set new account number and holder
account.set_account_number("987654321")
account.set_account_holder("Jane Doe")

# Set new balance (valid)
account.set_balance(2000)

# Set new balance (invalid)
account.set_balance(-500)

# Get updated account details
print("Account Number:", account.get_account_number())
print("Account Holder:", account.get_account_holder())
print("Balance:", account.get_balance())
```

```
Account Number: 123456789
Account Holder: John Doe
Balance: 1000
Invalid balance. Balance cannot be negative.
Account Number: 987654321
Account Holder: Jane Doe
Balance: 2000
```

2. Student Record System • Create a student class with private attributes student_id, name, age, and grades (a list of integers) • Implement getter and setter methods for name and age. • Implement a setter method for grades that ensures all grades are within a valid range (e.g., 0-100).

```
# prompt: 2. Student Record System
# •  Create a student class with private attributes student_id, name,
```

```
age, and grades (a list of integers)
# •  Implement getter and setter methods for name and age.
# •  Implement a setter method for grades that ensures all grades are
within a valid range (e.g., 0-100).

class Student:
    def __init__(self, student_id, name, age, grades):
        self.__student_id = student_id
        self.__name = name
        self.__age = age
        self.__grades = grades

    # Getter methods
    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age
    def student_id(self):
        return self.__student_id
    def grades(self):
        return self.__grades

    # Setter methods
    def set_name(self, name):
        self.__name = name

    def set_age(self, age):
        self.__age = age

    def set_grades(self, grades):
        valid_grades = True
        for grade in grades:
            if grade < 0 or grade > 100:
                valid_grades = False
                break
        if valid_grades:
            self.__grades = grades
        else:
            print("Invalid grades. Grades must be between 0 and 100.")

# Create a student
student = Student("123456789", "John Doe", 18, [90, 85, 95])

# Get student details
print("Student ID:", student.student_id())
print("Name:", student.get_name())
print("Age:", student.get_age())
print("Grades:", student.grades())
```

```
# Set new name and age
student.set_name("Jane Doe")
student.set_age(19)

# Set new grades (valid)
student.set_grades([95, 90, 85])

# Set new grades (invalid)
student.set_grades([-5, 105, 70])

# Get updated student details
print("Student ID:", student.student_id())
print("Name:", student.get_name())
print("Age:", student.get_age())
print("Grades:", student.grades())
```

```
Student ID: 123456789
Name: John Doe
Age: 18
Grades: [90, 85, 95]
Invalid grades. Grades must be between 0 and 100.
Student ID: 123456789
Name: Jane Doe
Age: 19
Grades: [95, 90, 85]
```

3. Employee Management System • Create an Employee class with private attributes employee_id, name, position, and salary. • Implement getter and setter methods for position. • Implement a setter method for salary that ensures the salary being set is positive.

```
# prompt: 3. Employee Management System
# •  Create an Employee class with private attributes employee_id,
name, position, and salary.
# •  Implement getter and setter methods for position.
# •  Implement a setter method for salary that ensures the salary
being set is positive.

class Employee:
    def __init__(self, employee_id, name, position, salary):
        self.__employee_id = employee_id
        self.__name = name
        self.__position = position
        self.__salary = salary

    # Getter methods
    def get_position(self):
        return self.__position
    def employee_id(self):
        return self.__employee_id
    def salary(self):
```

```
        return self.__salary
    # Setter methods
    def set_position(self, position):
        self.__position = position


    def set_salary(self, salary):
        if salary > 0:
            self.__salary = salary
        else:
            print("Invalid salary. Salary must be positive.")

# Create an employee
employee = Employee("123456789", "John Doe", "Software Engineer",
100000)

# Get employee details
print("Employee ID:", employee.employee_id())

print("Position:", employee.get_position())
print("Salary:", employee.salary())

# Set new position and salary (valid)
employee.set_position("Senior Software Engineer")
employee.set_salary(120000)

# Set new salary (invalid)
employee.set_salary(-5000)

# Get updated employee details
print("Employee ID:", employee.employee_id())
print("Position:", employee.get_position())
print("Salary:", employee.salary())
```

```
Employee ID: 123456789
Position: Software Engineer
Salary: 100000
Invalid salary. Salary must be positive.
Employee ID: 123456789
Position: Senior Software Engineer
Salary: 120000
```

5. Online Shopping System • Create a Product class with private attributes product_id, name, and price. • Implement getter and setter methods for product_id. • Implement setter methods for name and price that perform validation (e.g., ensure name is not empty and price is positive).

```
class Product:
    def __init__(self, product_id, name, price):
        self.__product_id = product_id
        self.__name = name
```

```
        self.__price = price

    # Getter methods
    def get_product_id(self):
        return self.__product_id

    def get_name(self):
        return self.__name

    def get_price(self):
        return self.__price

    # Setter methods
    def set_product_id(self, product_id):
        self.__product_id = product_id

    def set_name(self, name):
        if name.strip():  # Check if name is not empty
            self.__name = name
        else:
            print("Invalid name. Name cannot be empty.")

    def set_price(self, price):
        if price > 0:
            self.__price = price
        else:
            print("Invalid price. Price must be positive.")

# Create a product
product = Product("123456789", "iPhone 13", 1000)

# Get product details
print("Product ID:", product.get_product_id())
print("Name:", product.get_name())
print("Price:", product.get_price())

# Set new product ID (valid)
product.set_product_id("987654321")

# Set new name (valid)
product.set_name("Samsung Galaxy S22")

# Set new name (invalid)
product.set_name("")

# Set new price (valid)
product.set_price(1200)

# Set new price (invalid)
product.set_price(-500)
```

```python
# Get updated product details
print("Product ID:", product.get_product_id())
print("Name:", product.get_name())
print("Price:", product.get_price())
```

```
Product ID: 123456789
Name: iPhone 13
Price: 1000
Invalid name. Name cannot be empty.
Invalid price. Price must be positive.
Product ID: 987654321
Name: Samsung Galaxy S22
Price: 1200
```