1. Identify any four problems in procedure Oriented Approach
Lack of Data Encapsulation: This can lead to issues where data can be easily modified or accessed from different parts of the code, making it harder to control data integrity and maintainability.
Limited Code Reusability:functions are typically written for specific tasks, and there is often limited code reuse. This can result in code duplication and make it harder to maintain a DRY (Don't Repeat Yourself) coding practice.
Difficulty in Managing State:managing the state of the program can be challenging. As the codebase grows, it becomes increasingly complex to track and manage data flow, leading to potential errors and difficulties in debugging.
Global Namespace Pollution: potentially causing unexpected behaviour or errors. This can be particularly problematic in larger projects where naming conflicts become more likely.

2. What is mean by encapsulation? Give an example.
Encapsulation involves bundling data and the methods that operate on that data into a single unit, called a class. Encapsulation allows to restrict access to some of an object's components while exposing others. encapsulation is achieved through the use of access control modifiers like private and protected attributes and methods.
Example:

```
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number  # Private attribute
        self.__balance = balance                # Private attribute
    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
    def withdraw(self, amount):
        if 0 < amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")
    def get_balance(self):
        return self.__balance
account = BankAccount("12345", 1000)
print("Account Balance:", account.get_balance())
account.deposit(500)
print("Account Balance after deposit:", account.get_balance())
account.withdraw(300)
print("Account Balance after withdrawal:", account.get_balance())
```

Output:

```
Account Balance: 1000
Account Balance after deposit: 1500
Account Balance after withdrawal: 1200
```

3. Identify the need of self variable.
The self variable is used to represent the instance of the class which is often used in object-oriented programming. It works as a reference to the object. Python uses the self parameter to refer to instance attributes and methods of the class.

4. Write a python code to make a private variable.
```
class MyClass:
    def __init__(self):
        self.__private_variable = 42  # Private variable
    def get_private_variable(self):
        return self.__private_variable
    def set_private_variable(self, value):
        if value >= 0:
            self.__private_variable = value
obj = MyClass()
print("Private Variable:", obj.get_private_variable())
obj.set_private_variable(100)
print("Modified Private Variable:", obj.get_private_variable())
```
Output:
```
Private Variable: 42
Modified Private Variable: 100
```

5. Write a python code to calculate power value with the help of static method.
```
class PowerCalculator:
    @staticmethod
    def calculate_power(base, exponent):
        return base ** exponent
# Using the static method to calculate power
result = PowerCalculator.calculate_power(2, 3)
print("2^3 =", result)
```
Output:
```
2^3 = 8
```

Part-B

1. Construct a python code to differentiate three types of methods and constructors.
```
class MyClass:
    class_variable = 0  # Class variable
    def __init__(self, instance_variable):
        self.instance_variable = instance_variable  # Instance variable
    def instance_method(self):
        return f"This is an instance method. Instance variable: {self.instance_variable}, 
Class variable: {MyClass.class_variable}"
    @classmethod
    def class_method(cls):
        return f"This is a class method. Class variable: {cls.class_variable}"
    @staticmethod
    def static_method():
        return "This is a static method."
# Creating an instance of MyClass
obj = MyClass("Hello")
# Instance method
print(obj.instance_method())
# Class method
print(MyClass.class_method())
# Static method
print(MyClass.static_method())
```
Output:
```
This is an instance method. Instance variable: Hello, Class variable: 0
This is a class method. Class variable: 0
This is a static method.
```

i)instance_method is an instance method. It can access both instance variables and class variables but must be called on an instance of the class.
ii)class_method is a class method. It can access and modify class variables, and it's defined with the @classmethod decorator. It takes the class (cls) as its first argument and is often used for operations that apply to the class as a whole.
iii)static_method is a static method. It does not have access to instance variables or class variables, and it's defined with the @staticmethod decorator. It can be called on the class itself and is often used for utility functions that are related to the class but do not depend on instance-specific or class-specific data.

2. Implement a python code to create Bank class where deposits and withdrawals can be handled by using instance methods.
```
class Bank:
    def __init__(self, account_number, balance=0):
        self.account_number = account_number
        self.balance = balance
    def deposit(self, amount):
        if amount > 0:
            self.balance += amount
            return f"Deposited ${amount}. New balance: ${self.balance}"
        else:
            return "Invalid deposit amount."
    def withdraw(self, amount):
        if amount > 0 and amount <= self.balance:
            self.balance -= amount
            return f"Withdrew ${amount}. New balance: ${self.balance}"
        else:
            return "Invalid withdrawal amount or insufficient funds."
    def check_balance(self):
        return f"Account #{self.account_number} has a balance of ${self.balance}"
# Create a bank account
account1 = Bank("12345", 1000)
# Make deposits and withdrawals
print(account1.check_balance())
print(account1.deposit(500))
print(account1.withdraw(200))
print(account1.withdraw(1500))
```

Output:

```
Account #12345 has a balance of $1000
Deposited $500. New balance: $1500
Withdrew $200. New balance: $1300
Invalid withdrawal amount or insufficient funds.
```

i)The Bank class is defined with an __init__ constructor that initializes the account number and an optional initial balance.
ii)The deposit method allows you to deposit a specified amount into the account, and it updates the account's balance.
iii)The withdraw method lets you withdraw a specified amount from the account, given that the withdrawal amount is valid and doesn't exceed the account balance.
iv)The check_balance method displays the current balance for the account.

3. i) Develop a python program to create Emp class and make all the members of the Emp class available to another class.

```
class Emp:
    def __init__(self, emp_id, emp_name):
        self.emp_id = emp_id
        self.emp_name = emp_name
    def display_details(self):
        return f"Employee ID: {self.emp_id}, Employee Name: {self.emp_name}"
```

```
class AnotherClass(Emp):
    def __init__(self, emp_id, emp_name, additional_info):
        super().__init__(emp_id, emp_name)
        self.additional_info = additional_info
    def display_additional_info(self):
        return f"Additional Info: {self.additional_info}"
# Create an instance of AnotherClass
employee = AnotherClass("E123", "John Doe", "Department: Sales")
# Access members of the Emp class via the AnotherClass instance
print(employee.display_details())           # Accessing the display_details method
print(employee.display_additional_info())     # Accessing the additional_info attribute
```

Output:

```
Employee ID: E123, Employee Name: John Doe
Additional Info: Department: Sales
```

i)Emp class, which has two members: emp_id and emp_name, along with a method display_details to display employee details.
ii)The AnotherClass class is created, which inherits from Emp. It has an additional member called additional_info and a method display_additional_info to display the additional information.
iii)In the AnotherClass constructor, we use super().__init__(emp_id, emp_name) to call the constructor of the base class (Emp) and initialize the inherited members (emp_id and emp_name).

ii) Develop a python program to create Dob class within Person class.

```
class Dob:
    def __init__(self, day, month, year):
        self.day = day
        self.month = month
        self.year = year
    def display_dob(self):
        return f"Date of Birth: {self.day:02d}/{self.month:02d}/{self.year}"
class Person:
    def __init__(self, name, dob):
        self.name = name
        self.dob = dob
    def display_person_info(self):
        return f"Name: {self.name}\n{self.dob.display_dob()}"
# Create a Dob instance
dob1 = Dob(10, 5, 1990)
# Create a Person instance with the Dob instance
```

```
person1 = Person("Alice", dob1)
# Display person's information, including date of birth
print(person1.display_person_info())
```

Output:

```
Name: Alice
Date of Birth: 10/05/1990
```

i)Dob class to represent a person's date of birth. It has attributes day, month, and year, along with a display_dob method to display the date of birth in a formatted manner.
ii)The Person class is created, which has a name attribute and a dob attribute that stores an instance of the Dob class.
iii)We create an instance of the Dob class (dob1) and pass it as an argument when creating a Person instance (person1).
iv)The display_person_info method in the Person class displays the person's name along with their date of birth using the display_dob method from the Dob class.