Abstraction

Abstraction in programming is the concept of hiding unnecessary details and showing only the essential features of an object or process. It helps simplify complex systems by focusing on what an object does, rather than how it does it.

Key Features of Abstraction:

1. Hides Complexity: It allows you to hide complex logic and provide a simple interface to users.

2. Simplifies Code Maintenance: By separating the details and the interface, it becomes easier to manage and modify the code.

3. Provides Flexibility: Users of a class don't need to know the detailed implementation but can still use the features provided.

Simplified Example of Abstraction:

```
class Payment:
    # General method to be implemented by subclasses
    def process_payment(self, amount):
        raise NotImplementedError("This method should be overridden by
subclasses")

class CreditCardPayment(Payment):
    # Implementation of the process for credit card payments
    def process_payment(self, amount):
        print(f"Processing credit card payment of ${amount}")

class PayPalPayment(Payment):
    # Implementation of the process for PayPal payments
```

```
    def process_payment(self, amount):
        print(f"\nProcessing PayPal payment of ${amount}")

# Now you can use these payment methods without worrying about the
details.
payment1 = CreditCardPayment()
payment1.process_payment(100)  # Outputs: Processing credit card
payment of $100

payment2 = PayPalPayment()
payment2.process_payment(150)  # Outputs: Processing PayPal payment of
$150
```

```
Processing credit card payment of $100

Processing PayPal payment of $150
```

Another Example

```
print("Using Abstraction\n")

# Base class
class Car():
    def __init__(self, make, model, year):
        # Initialize the attributes of the Car class
        self.make = make   # The manufacturer of the car (e.g.,
Toyota)
        self.model = model # The specific model of the car (e.g.,
Crown)
        self.year = year   # The year the car was manufactured (e.g.,
2023)



    def detail_info(self): # This method prints the details of the car
        # Print the car's make, model, and year
        print(f"Car Make: {self.make}")
        print(f"Car Model: {self.model}")
        print(f"Car Year: {self.year}")



# Example of creating a Car object and displaying its details
car1 = Car("Toyota", "Crown", 2023) # Create an instance of Car with
specified attributes
car1.detail_info()  # Call the detail_info method to display the car's
details
```

```
Using Abstraction

Car Make: Toyota
Car Model: Crown
Car Year: 2023
```

Explanation:

Class and Object:

We created a Car class with attributes like make, model, and year.

The Car class has a method called detail_info() that shows the car's essential information.

Abstraction in Action:

When we create an object car1 = Car("Toyota", "Crown", 2023), we don't need to know the internal workings of the Car class.

The detail_info() method provides us with what we need: the car's make, model, and year. It hides unnecessary details and focuses on essential features.

Real-World Example:

Think of a car itself — you don’t need to understand the mechanics of the engine to drive it. You only need the steering wheel, pedals, and essential controls — that’s abstraction!

Problem 1: Shape Area Calculator

Objective: Create an abstract class to represent different shapes and calculate their area.

Instructions:

Define an Abstract Class:

Create an abstract class named Shape.

It should have an abstract method area() that will be overridden by subclasses.

Implement Subclasses:

Create a subclass Circle that takes the radius as an argument and implements the area() method using the formula:

Area=π×r^2

Create another subclass Rectangle that takes width and height as arguments and implements the area() method using the formula:

Area =width × height

Area=width×height.

Create Instances:

Instantiate objects of Circle and Rectangle.

Print the area of each shape.

```
class Shape():
    def area(self):
        pass


class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 *(self.radius ** 2) # Pi = π = 3.14
```

```python
class Rectangle(Shape):
    def __init__(self, width: int, height: int):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height


cir = Circle(5)
print(f"Area of Circle: {cir.area()}") # Output: 78.5

rect = Rectangle(10,3)
print(f"\nArea of Rectangle is: {rect.area()}") # Output: 30
```

```
Area of Circle: 78.5

Area of Rectangle is: 30
```

Problem 2: Employee Management System

Objective: Create an abstract class to represent employees and calculate their salaries based on different employment types.

Instructions:

Define an Abstract Class:

Create an abstract class named Employee.

It should have an abstract method calculate_salary().

Implement Subclasses:

Create a subclass FullTimeEmployee that takes a fixed salary as an argument and implements the calculate_salary() method to return the salary.

Create another subclass PartTimeEmployee that takes an hourly wage and the number of hours worked as arguments and implements the calculate_salary() method to return the total salary for the hours worked.

Create Instances:

Instantiate objects of FullTimeEmployee and PartTimeEmployee.

Print the salary for each employee type.

```
class Employee():
    def calculate_salary(self):
        pass


class FullTimeEmployee(Employee):
    def __init__(self, fix_salary: int):
        self.fix_salary = fix_salary
```

```
    def calculate_salary(self):
        return self.fix_salary


class PartTimeEmployee(Employee):
    def __init__(self, hourly_wage: int, hours_worked: int):
        self.hourly_wage = hourly_wage
        self.hours_worked = hours_worked

    def calculate_salary(self):
        return self.hours_worked * self.hourly_wage

full_time = FullTimeEmployee(50000)
print(f"Full-Time-Employee Salary is: {full_time.calculate_salary()}")

part_time = PartTimeEmployee(650,10)
print(f"\nPart-Time-Employee Salary is:
{part_time.calculate_salary()}")
```

```
Full-Time-Employee Salary is: 50000

Part-Time-Employee Salary is: 6500
```

Where is Abstraction Used in Real Life?

1. ATM Machines: Users interact with a simple interface (withdraw, deposit, check balance), but the internal processes (like validating the card, checking the account balance, etc.) are hidden.

2. Car Interfaces: When driving a car, the driver interacts with controls (steering wheel, pedals, gear) without worrying about how the engine or transmission works.

3. Web Applications: In frameworks like Django, you write high-level Python code that interacts with the database without worrying about raw SQL queries or database connection handling.