Answer to the question no-01

Class method	Static method
1. A class method is a method bound to the class and has access over class variables.	1. A static method is a method that belongs to the class but does not have the access over class variable.
2. It is defined as @classmethod as a decorator.	2. It is defined as @staticmethod as a decorator.
3. It can modify the class variable	3. It cannot modify the class variable.
4. We have to write __init__ function for it.	4. We do not need to write __init__ function for it.

class anonymous:
count = 0 # class attribute
 	
def __init__(self):
anonymous.count += 1
 	
@classmethod
def total_count():
return count
 	
obj1 = anonymous()
obj2 = anonymous()
 	
print(anonymous.total_count())
 	
# Output: 2 	
	class Maths:
	
	@staticmethod
	def modulus(a, b):
	return a%b
	
	result = Maths.modulus(5, 3)
	print(result)
	
	# Output: 2

Answer to the question no-02

Polymorphism is a fundamental concept in object-oriented programming that allows objects of different classes to be treated as objects of a common superclass. It enables

different objects to respond to the same method or function call based on their specific implementation.

```
class Animal:
    def __init__(self,name) -> None:
        self.name=name

    def Animal_sound(self):
        print('Animal making some sound')

class Cat(Animal):
    def __init__(self, name) -> None:
        super().__init__(name)

    def Animal_sound(self):
        print('meow meow')

class Dog(Animal):
    def __init__(self, name) -> None:
        super().__init__(name)

    def Animal_sound(self):
        print('ghew ghew')

Tommy=Dog('Tommy')
Tommy.Animal_sound()
```

Here Cat, Dog inherit Animal class. But for a specific function the task is different. In Cat class its sound and in Dog class dog's sound is totally different but their method name is same. It is also called the process of overriding.

We can also do overload in the class which is also a polymorphism.

```
class Animal:
    def __init__(self,name) -> None:
        self.name=name

    def Animal_sound(self):
        print('Animal making some sound')
```

```
class Cat(Animal):
    def __init__(self, name) -> None:
        super().__init__(name)

    def Animal_sound(self):
        print('meow meow')

    def likes(self):
        print('loves to eat fish and milk')

class Dog(Animal):
    def __init__(self, name) -> None:
        super().__init__(name)

    def Animal_sound(self):
        print('ghew ghew')

    def likes(self):
        print('loves to eat meat and bones')

Tommy=Dog('Tommy')
Tommy.Animal_sound()
Tommy.likes()
```

Here the like method is added as an overload method. It is called overload due to there being no section of that name in Animal class but it exists in inherited class. It is also a polymorphism.

Answer to the question no-03

```
import math
class Math:
    def __init__(self,a,b,c) -> None:
        self.a=a
        self.b=b
        self.c=c

    def sum(self):
        return self.a+self.b+self.c
```

```
    def factorial(self):
        return get_factorial(self.b)
```

Answer to the question no-04

Multilevel inheritance in Python refers to a situation where a class inherits from another class, which itself inherits from another class. This creates a hierarchical structure of inheritance, where each class extends the functionality of the previous class.

Here is an example:

```
class Vehicle:
    def __init__(self) -> None:
        pass

class Bus(Vehicle):
    def __init__(self) -> None:
        super().__init__()

class Truck(Vehicle):
    def __init__(self) -> None:
        super().__init__()

class PickupTruck(Truck):
    def __init__(self) -> None:
        super().__init__()

class ACBus(Bus):
    def __init__(self) -> None:
        super().__init__()
```

Here are 5 classes.
Vehicle, Bus, Truck, PickupTruck, ACBus. These are inherited by each of them. Bus, Truck class inherit Vehicle class, PickupTruck inherits Truck class, ACBus inherits Bus class. Inheritance is important to relate the same matter or features without writing duplicates.

Answer to the question no-05

An inner function is a function that is defined inside of a function. It is also called the nested Function.
The advantages are:

1. Encapsulation: It helps to encapsulate a function within another function. It cannot be accessible from outside. It helps to hide implementation details. It helps to keep related functions together.
2. Readability and Maintainability: Inner functions can improve the readability and maintainability of the code by keeping related functions together. It allows defining auxiliary methods within the context where they are most relevant, making the code easier to understand and modify.
3. Function Reusability: Inner functions can be designed to perform specific tasks that are reusable within the scope of the outer function or class. They can be called multiple times within the outer function, providing code reuse and reducing redundancy. This promotes modular programming and helps in writing clean and concise code.

Helper Functions: Inner functions are often used to define helper functions that are specific to a particular task or algorithm. By encapsulating these helper functions within the outer function, you can improve code readability by keeping related functionality together.

```
def avg(numbers):
    def total_sum():
        return sum(numbers)

    def Count():
        return len(numbers)

    return total_sum()/Count()
```

When it is called it directly goes to the last line of the return and then called the total_sum function and then Count function. After returning from them it calculates the value and returns the value to the outside.

Decorators: Inner functions are commonly used in decorators, which are functions that modify the behavior of other functions. Inner functions in decorators can be used to wrap the original function and add additional functionality.

```python
import math
def timer(func):
    def inner(*args, **kwargs):
        print('time started')
        func(*args,**kwargs)
        print('time ended')
    return inner
@timer
def get_factorial(n):
    result=math.factorial(n)
    print(f'factorial of {n} is: {result}')

timer(get_factorial(10))
```

When the 'timer' function is called with get_factorial(10) ,the timer function receives it as a function as a func variable then when the func is called then it goes to the get_factorial function to do the function's work.

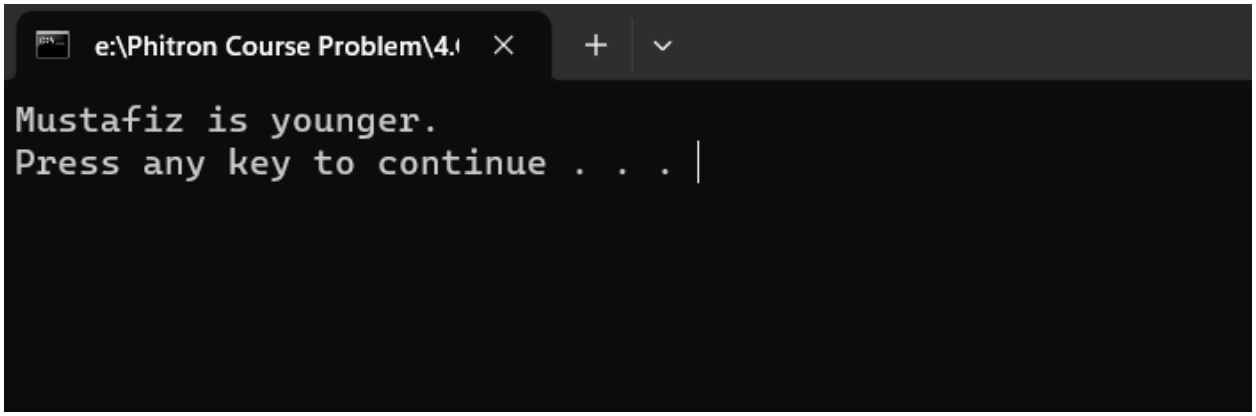
Answer to the question no-06

```python
from collections import Counter
n,a=map(int,input().split())
A=list(map(int,input().split()))
Cnt=Counter(A)

for i in range(1,a+1):
    print(Cnt[i])
```

Answer to the question no-07

```
`
```


e:\Phitron Course Problem\4.
Mustafiz is younger.
Press any key to continue . . .