

Question 1.1: Write the Answer to these questions. Note: Give at least one example for each of the questions

#Q1 What is the difference between static and dynamic variables in Python?

```
"""Static variables are also called as class variables and therefore they have fixed memory locations,
limited scope, and persistent values across function calls, while dynamic variables have runtime memory allocation,
broader scope, and object-specific data"""
```

#Q2 Explain the purpose of "pop", "popitem", "clear()" in a dictionary with suitable examples

```
dict1={"a":1,"b":2,"c":3,"d":4,"e":5}
print(dict1) #original dictionary
dict1.pop("b")
print(dict1) #dictionary after popping one item
dict1.popitem()
print(dict1) #dictionary after popping last item
dict1.clear()
print(dict1) #dictionary after clearing all items
```

```
→ {'a': 1, 'b': 2, 'c': 3, 'd': 4, 'e': 5}
   {'a': 1, 'c': 3, 'd': 4, 'e': 5}
   {'a': 1, 'c': 3, 'd': 4}
   {}
```

#What do you mean by FrozenSet? Explain it with suitable examples

```
""" Frozen sets are built-in data type in python which are like sets but are IMMUTABLE and can be used as a key in dictionary unlike normal sets.
They can perform union, difference and other operations of set by creating a new frozen set """
```

Creating a frozenset from a list

```
fs1 = frozenset([1, 2, 3, 4])
print(fs1)
```

Creating a frozenset from a set

```
fs2 = frozenset({5, 6, 7, 8})
print(fs2)
```

Creating a frozenset from a string

```
fs3 = frozenset('hello')
print(fs3)
```

```
→ frozenset({1, 2, 3, 4})
   frozenset({8, 5, 6, 7})
   frozenset({'h', 'o', 'e', 'l'})
```

#Differentiate between mutable and immutable data types in Python and give examples of mutable and immutable data types

```
"""Mutable Data types: They can be changed/modified after creation eg. list, set, dictionary
Immutable Data types: They cannot be changed after creation eg. int, float, string, tuple, frozenset"""
```

#What is __init__? Explain with an example

```
"""The __init__ method in Python is a special method (Dunder method) used in classes for initializing new instances of the class.
It's known as the constructor method and is automatically called when a new object of the class is created.
```

The purpose of __init__ is to set up the initial state of the object by assigning values to its attributes.

Initialization: __init__ initializes the newly created object. It's where you define the attributes that the object will have and set their initial values.

Self Parameter: The first parameter of __init__ is always self, which refers to the instance being created.

It allows you to access and set attributes on the instance.

Other Parameters: You can add more parameters to __init__ to pass initial values when creating an instance."""

```
class Person:
    def __init__(self, name, age):
        self.name = name # Initialize the name attribute
        self.age = age # Initialize the age attribute

    def greet(self):
        return f"Hello, my name is {self.name} and I am {self.age} years old."
```

Creating an instance of Person

```
person1 = Person("Ankita", 32)
```

Accessing attributes and methods

```
print(person1.name)
print(person1.age)
print(person1.greet())
```

```
→ Ankita
   32
   Hello, my name is Ankita and I am 32 years old.
```

#What is docstring in Python? Explain with an example

```
"""A docstring in Python is a string literal that appears right after the definition of a function, method, class, or module.
```

It is used to document what the function, method, class, or module does.
 A docstring is written within triple quotes (""" or ''').
 This allows the docstring to span multiple lines, which is useful for providing detailed documentation.
 the above codes are the example for the same"""

#What are unit tests in Python?

#What is break, continue and pass in Python?

"""break:it used to break an iterative loop or switch statement if the condition is fulfilled
 continue: is used to continue an iterative loop if the above condition is done
 pass; it simply passes to the next iteration """

#eg of break statement

```
for i in range (11): #0 to 10
    print(i)
    if i==5: #when i attains value 5 the loop will break and comeout of the loop
        break
```

```
print("\n")
```

#eg of continue statement

```
for j in range (11):
    if j%2==0:
        print(j) #for even values of j it will print and for odd values it will continue the next iteration
    else:
        continue
```

```
print("\n")
```

#eg of pass statement

```
for k in range (6):
    if k==3:
        pass #it will pass when k=3
    else:
        print(k)
```

```
0
1
2
3
4
5
```

```
0
2
4
6
8
10
```

```
0
1
2
4
5
```

#What is the use of self in Python?

"""self is NOT a keyword of Python, it is used to initialise the function to call itself
 Self Parameter: The first parameter of any function is always self, which refers to the instance being created.
 It allows you to access and set attributes on the instance."""

#What are global, protected and private attributes in Python?

"""Global Attributes: Defined outside any class or function and accessible from anywhere in the module.
 Protected Attributes: Indicated by a single underscore (_). Intended for internal use within the class and subclasses, but can still be a
 Private Attributes: Indicated by double underscores (__). Intended to be accessible only within the class due to name mangling, but not s

class Student:

```
def __init__(self, name, age, std):
    self.__name=name    #private
    self._age=age       #protected
    self.std=std        #public
def display(self):
    print(f"{self.__name}, {self._age} years old is studying in standard {self.std}")
```

```
stud=Student("Ankita",16,"Tenth")
stud.display()
```

```
Ankita, 16 years old is studying in standard Tenth
```

#What are modules and packages in Python?

"""A module is a single file containing Python code, whereas a package is a collection of modules that are organized in a directory hierarchy"""

#What are lists and tuples? What is the key difference between the two?

```
"""Lists and Tuples are both in built data types of python that contain a group of data
list is mutable data type whereas tuples are immutable
list is ordered and tuples are unordered
list is enclosed by square brackets whereas tuples are enclosed by round brackets
tuples are more memory efficient than lists"""
lst=list(range(10))
print(lst, type(lst))
tup=tuple(range(10))
print(tup, type(tup))
```

```
lst2=list([1,"a",3.4,True])
lst2[2]="b"
print(lst2)
tup2=(1,"a",3.4,True)
tup2[2]="b" #tuples are immutable it will throw type error
print(tup2)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9] <class 'list'>
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9) <class 'tuple'>
[1, 'a', 'b', True]
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-25-6e9b3c1817e9> in <cell line: 16>()
    14 print(lst2)
    15 tup2=(1,"a",3.4,True)
--> 16 tup2[2]="b" #tuples are immutable it will throw type error
    17 print(tup2)
```

TypeError: 'tuple' object does not support item assignment

#What is an Interpreted language & dynamically typed language? Write 5 differences between them.

"""An interpreted language is a type of programming language for which most of the instructions are executed directly by an interpreter rather than being compiled into machine code beforehand. The interpreter reads and executes the code line-by-line or statement-by-statement, which can make development and debugging easier. Examples: Python, Ruby, JavaScript

A dynamically typed language is a type of programming language in which the type of a variable is determined at runtime rather than at compile-time. This means that variables do not have fixed types and can change their type as the program executes. Examples: Python, Ruby, JavaScript

Differences Between Interpreted and Dynamically Typed Languages

Definition:

Interpreted Language: Refers to how the code is executed (interpreted line-by-line).

Dynamically Typed Language: Refers to how types are managed (types are assigned at runtime).

Compilation:

Interpreted Language: Does not require a separate compilation step; the interpreter executes the code directly.

Dynamically Typed Language: Focuses on variable types being resolved at runtime, regardless of whether the language is interpreted or compiled.

Type Checking:

Interpreted Language: May or may not be dynamically typed; the focus is on the execution method (e.g., Python is both interpreted and dynamically typed).

Dynamically Typed Language: Emphasizes type checking at runtime, not at compile-time, which can apply to both interpreted and compiled languages.

Error Detection:

Interpreted Language: Errors are often detected at runtime during execution. Errors in syntax or logic can be found while running the code.

Dynamically Typed Language: Type-related errors are detected at runtime, but syntax errors may be caught by the interpreter.

Flexibility:

Interpreted Language: Offers flexibility in development and debugging because code is executed line-by-line, which allows for rapid testing.

Dynamically Typed Language: Provides flexibility in terms of variable types and can lead to more versatile code, but also requires careful management of types.

#What are Dict and List comprehensions?

```
lst=[1,2,3,4,5,6,7,8,9,10]
lst_comp=[i for i in lst if i%2==0]
print(lst_comp)
```

```
dict1={"a":1,"b":2,"c":3,"d":4,"e":5}
dict_comp={k:v for k,v in dict1.items() if v%2==0}
print(dict_comp)
```

```
[2, 4, 6, 8, 10]
{'b': 2, 'd': 4}
```

```
#What are decorators in Python? Explain it with an example. Write down its use cases.
"""A decorator is a higher-order function that takes another function or method as an argument and extends or alters its behavior."""
```

```
# Define the decorator function
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

# Use the decorator
@my_decorator #syntax of decorator function
def say_hello():
    print("Hello!")

# Call the decorated function
say_hello()
```

```
Something is happening before the function is called.
Hello!
Something is happening after the function is called.
```

```
#How is memory managed in Python?
"""1. Memory Allocation
Object Creation: Memory is allocated dynamically for objects when they are created. Python handles this allocation internally and adapts to varying object sizes.

2. Reference Counting
Counting References: Each object maintains a count of how many references point to it. When the reference count drops to zero, the memory for the object can be reclaimed.

3. Garbage Collection
Cycle Detection: To handle reference cycles that reference counting alone can't manage, Python uses a cyclic garbage collector that detects and collects objects involved in cycles.

4. Memory Pools and Object Interning
Memory Pools: Small objects are managed using memory pools to improve efficiency and reduce fragmentation.
Object Interning: Python interns small immutable objects (e.g., small integers and strings) to save memory and improve performance by reusing objects.

5. Automatic Memory Management
Automatic Allocation/Deallocation: Python automatically handles memory allocation and deallocation for objects, using the garbage collector to clean up unused memory.

6. Profiling and Leak Detection
Tools: Tools like gc, tracemalloc, and memory_profiler help analyze memory usage, detect leaks, and optimize performance."""
```

```
#What is lambda in Python? Why is it used?
"""lambda is a one line anonymous function of python"""
```

```
x=lambda a,b:a+b
print(x(2,3))
```

```
5
```

```
#Explain split() and join() functions in Python?
str1="Apple, Oranges, Kiwi, pineapple, berry"
str1.split(",") #splitting string wherever comma is there
print(str1)
s1=" ".join(str1.split(","))
print(s1)
```

```
Apple, Oranges, Kiwi, pineapple, berry
Apple Oranges Kiwi pineapple berry
```

```

#What are iterators , iterable & generators in Python?
"""An iterable is any Python object capable of returning its members one at a time, allowing it to be iterated over.
This means the object must implement the __iter__() method, which returns an iterator,
or the __getitem__() method, which allows indexed access."""

# List is an iterable
my_list = [1, 2, 3]

# Iterating over the list
for item in my_list:
    print(item)

print("\n")
"""An iterator is an object that represents a stream of data. It is used to iterate over a sequence of values.
An iterator must implement two methods:
__iter__(): Returns the iterator object itself. This method is required to make the object an iterable.
__next__(): Returns the next value from the stream. When there are no more items, it raises a StopIteration exception."""

# Creating an iterator from a list
my_list = [1, 2, 3]
iterator = iter(my_list)

# Using the iterator
print(next(iterator)) # Output: 1
print(next(iterator)) # Output: 2
print(next(iterator)) # Output: 3
# print(next(iterator)) # This would raise StopIteration
print("\n")

"""Generators are a special type of iterator created using functions with the yield keyword. They provide a convenient way to produce a
of values which are not stored in memory all at once."""

def my_generator():
    yield 1
    yield 2
    yield 3

# Using the generator
generator = my_generator()
for value in generator:
    print(value)

1
2
3

1
2
3

1
2
3

#What is the difference between xrange and range in Python?
"""Python 2 range: Stores the entire list of numbers in memory.
Python 2 xrange: Uses an iterator and generates numbers on demand.
Python 3 range: Similar to Python 2 xrange; generates numbers on demand and is memory-efficient.

Python 2 range: Returns a list.
Python 2 xrange: Returns an xrange object (iterator-like).
Python 3 range: Returns a range object (iterator-like).

Python 2 range and xrange: Both support three arguments (start, stop, step), but range returns a list and xrange returns an iterator.
Python 3 range: Supports three arguments as well (start, stop, step), and behaves like the old xrange in Python 2."""

'Python 2 range: Stores the entire list of numbers in memory.\nPython 2 xrange: Uses an iterator and generates numbers on demand.\n
Python 3 range: Similar to Python 2 xrange; generates numbers on demand and is memory-efficient.\n\nPython 2 range: Returns a lis
t.\nPython 2 xrange: Returns an xrange object (iterator-like).\nPython 3 range: Returns a range object (iterator-like).\n\nPython 2
range and xrange: Both support three arguments (start, stop, step), but range returns a list and xrange returns an iterator.\nPytho

#Pillars of OOps
"""Pillars of OOPs are Inheritance, Polymorphism, Encapsulation and Abstraction"""

```

```
#How will you check if a class is a child of another class?
"""The isinstance() function checks if a class is a subclass of another class. It returns True if the first class is a subclass (directly or indirectly) of the second class, otherwise, it returns False."""
class Animal:
    pass

class Mammal(Animal):
    pass

class Dog(Mammal):
    pass

print(isinstance(Dog, Mammal))
print(isinstance(Dog, Animal))
print(isinstance(Mammal, Dog))
```

```
True
True
False
```

#How does inheritance work in python? Explain all types of inheritance with an example

"""Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called a subclass or derived class) to inherit attributes and methods from another class (called a superclass or base class). In Python, inheritance facilitates code reuse and helps in building a hierarchy of classes.

Working of Inheritance:

Base Class: The class whose properties and methods are inherited.

Derived Class: The class that inherits from the base class and can extend or override its behavior.

Types of Inheritance:

1. Single Inheritance: In single inheritance, a class (derived class) inherits from a single base class.
2. Multiple Inheritance: In multiple inheritance, a class inherits from more than one base class.
3. Multilevel Inheritance: In multilevel inheritance, a class is derived from another derived class, forming a chain.
4. Hierarchical Inheritance: In hierarchical inheritance, multiple classes are derived from a single base class.
5. Hybrid Inheritance: It is a combination of two or more types of inheritance. It may involve multiple inheritance in a complex hierarchy.

#single inheritance

```
class Animal:
    def speak(self):
        return "Animal speaks"
```

```
class Dog(Animal):
    def bark(self):
        return "Dog barks"
```

Usage

```
dog = Dog()
print(dog.speak()) # Output: Animal speaks
print(dog.bark()) # Output: Dog barks
```

```
Animal speaks
Dog barks
```

#multiple inheritance

```
class Flyable:
    def fly(self):
        return "Can fly"
```

```
class Swimmable:
    def swim(self):
        return "Can swim"
```

```
class Duck(Flyable, Swimmable):
    pass
```

Usage

```
duck = Duck()
print(duck.fly()) # Output: Can fly
print(duck.swim()) # Output: Can swim
```


```
Can fly
Can swim
```

```
#multilevel inheritance
class Animal:
    def speak(self):
        return "Animal speaks"

class Mammal(Animal):
    def walk(self):
        return "Mammal walks"

class Dog(Mammal):
    def bark(self):
        return "Dog barks"

# Usage
dog = Dog()
print(dog.speak()) # Output: Animal speaks
print(dog.walk())  # Output: Mammal walks
print(dog.bark())  # Output: Dog barks
```


 Animal speaks
Mammal walks
Dog barks

```
#hierarchical inheritance
class Animal:
    def speak(self):
        return "Animal speaks"

class Dog(Animal):
    def bark(self):
        return "Dog barks"

class Cat(Animal):
    def meow(self):
        return "Cat meows"

# Usage
dog = Dog()
cat = Cat()
print(dog.speak()) # Output: Animal speaks
print(dog.bark())  # Output: Dog barks
print(cat.speak()) # Output: Animal speaks
print(cat.meow())  # Output: Cat meows
```

 Animal speaks
Dog barks
Animal speaks
Cat meows

```
#hybrid inheritance
class Animal:
    def speak(self):
        return "Animal speaks"

class Flyable:
    def fly(self):
        return "Can fly"


class Swimmable:
    def swim(self):
        return "Can swim"

class Bird(Flyable, Animal):
    pass

class Fish(Swimmable, Animal):
    pass


class Duck(Bird, Fish):
    pass

# Usage
duck = Duck()
print(duck.speak()) # Output: Animal speaks
print(duck.fly())   # Output: Can fly
print(duck.swim())  # Output: Can swim
```

 Animal speaks
Can fly

Can swim

```
#What is encapsulation? Explain it with an example
"""Encapsulation is an pillar of OOPS. It refers to the bundling of data (attributes) and methods (functions)
that operate on the data into a single unit, known as a class, and restricting direct access to some of the object's components."""
class Student:
    def __init__(self, name, age, std):
        self.__name=name      #private
        self._age=age         #protected
        self.std=std          #public
    def display(self):
        print(f"{self.__name}, {self._age} years old is studying in standard {self.std}")
stud=Student("Ankita", 16, "Tenth")
stud.display()
```

 Ankita, 16 years old is studying in standard Tenth

```
#What is polymorphism? Explain it with an example.
"""Polymorphism means "many shapes".It is the ability of different objects to respond in its own way to the method call.
Polymorphism lets us treat objects of different classes that have a common
interface as if they were objects of a common superclass."""
```


```
class Animal:
    def speaks(self):
        print("Animal speaks")
class Dog(Animal):
    def speaks(self):
        print("Dog barks")
class Cat(Animal):
    def speaks(self):
        print("Cat meows")
def animalspeak(animal):
    animal.speaks()

d=Dog()
c=Cat()
animalspeak(d)
animalspeak(c)
```

 Dog barks
Cat meows


Question 1. 2. Which of the following identifier names are invalid and why? a) Serial_no. b) 1st_Room c) Hundred\$ d) Total_Marks e) total-Marks f) Total Marks g) True h) _Percentag

```
"""Invalid identifiers are as bellow:
a)Serial_no.: Due to .
b)1st_Room: identifier name cannot start with number
c)Hundred$: $ special symbols are not allowed
e)total-Marks: - is not allowed
f)Total Marks: space is not allowed
g) True: true is a Python keyword"""
```

 'Invalid identifiers are as bellow:\n a)Serial_no.: Due to .\n b)1st_Room: identifier name cannot start with number\n c)Hundred\$: \$ sp

Question 1.3. name = ["Mohan", "dash", "karam", "chandra","gandhi","Bapu"] Do the following operations in this list;

```
#a) add an element "freedom_fighter" in this list at the 0th index.
name = ["Mohan", "dash", "karam", "chandra", "gandhi", "Bapu"]
name.insert(0,"freedom_fighter")
print(name)
```

 ['freedom_fighter', 'Mohan', 'dash', 'karam', 'chandra', 'gandhi', 'Bapu']

```
#b) find the output of the following ,and explain how?
name = ["freedom_fighter","Bapuji","Mohan","dash", "karam","chandra","gandhi"]
length1=len((name[-len(name)+1:-1:2]))
"""len(name)=7, then list comprehension name[-7+1:-1:2] i.e. name[-6:-1:2] which means from -6 to -1th
element every 2nd element is chosen and then finding its length which is equal to 3"""
length2=len((name[-len(name)+1:-1]))
"""len(name)=7, then list comprehension name[-7+1:-1] i.e. name[-6:-1] which means from -6 to -1th
every element is chosen and then finding its length which is equal to 5"""
print(length1+length2) #3+5=8 output
```

 8


```
#c) add two more elements in the name ["NetaJi","Bose"] at the end of the list.
name.extend(["NetaJi","Bose"])
print(name)
```

```
↵ ['freedom_fighter', 'Bapuji', 'Mohan', 'dash', 'karam', 'chandra', 'gandhi', 'NetaJi', 'Bose']
```

```
#d) what will be the value of temp:
name = ["Bapuji", "dash", "karam", "chandra","gandhi","Mohan"]
temp=name[-1] #temp is "Mohan"
name[-1]=name[0] #"Mohan" is "Bapuji"
name[0]=temp #"Bapuji" is "Mohan"
print(name) #Output: ["Mohan", "dash", "karam", "chandra","gandhi","Bapuji"]
```

```
↵ ['Mohan', 'dash', 'karam', 'chandra', 'gandhi', 'Bapuji']
```

```
#Question 1.4.Find the output of the following.
animal = ['Human','cat','mat','cat','rat','Human', 'Lion']
print(animal.count('Human')) #Human is presnt 2 times
print(animal.index('rat')) #rat is at 4th index
print(len(animal)) #length of list is 7
```

```
↵ 2
4
7
```

```
#Question 1.5. tuple1=(10,20,"Apple",3.4,'a',["master","ji"],("sita","geeta",22),[{"roll_no":1},{"name":"Navneet"}])
tuple1=(10,20,"Apple",3.4,'a',["master","ji"],("sita","geeta",22),[{"roll_no":1},{"name":"Navneet"}])
print(len(tuple1)) #length of tuple is 8
print(tuple1[-1][-1]["name"]) #Output: "Navneet", last element of tuple which is a dictionary, last element of the dictionary having in
#fetch the value of roll_no from this tuple.
print(tuple1[-1][0]["roll_no"])
print(tuple1[-3][1]) #Output: "ji"
#fetch the element "22" from this tuple
print(tuple1[-2][-1])
```

```
↵ 8
Navneet
1
ji
22
```

```
#1.6. Write a program to display the appropriate message as per the color of signal(RED-Stop/Yellow-Stay/Green-Go) at the road crossing.
def signal_message(color):
```

```
    # Define messages for each signal color
    if color.lower() == "red":
        return "Stop"
    elif color.lower() == "yellow":
        return "Stay"
    elif color.lower() == "green":
        return "Go"
    else:
        return "Invalid signal color"
```

```
# Input color from user
signal_color = input("Enter the traffic signal color (Red/Yellow/Green): ")
```

```
# Display the appropriate message
message = signal_message(signal_color)
print(message)
```

```
↵ Enter the traffic signal color (Red/Yellow/Green): Red
Stop
```

#1.7. Write a program to create a simple calculator performing only four basic operations(+,-,/,*) .

```
def calculator(num1, num2, operation):
    if operation == '+':
        return num1 + num2
    elif operation == '-':
        return num1 - num2
    elif operation == '*':
        return num1 * num2
    elif operation == '/':
        if num2 != 0:
            return num1 / num2
        else:
            return "Cannot divide by zero"
    else:
        return "Invalid operation"
```

```
print(calculator(4,5,"+"))
print(calculator(10,5,"/"))
print(calculator(8,5,"-"))
print(calculator(2,3,"*"))
print(calculator(10,0,"/"))
```

```
9
2.0
3
6
Cannot divide by zero
```

#1.8 Write a program to find the larger of the three pre-specified numbers using ternary operators.

```
a = 5
b = 10
c = 7
```

```
# Find the largest of the three numbers
largest = a if (a > b and a > c) else (b if (b > c) else c)
```

```
# Output the largest number
print("The largest number is:", largest)
```

```
The largest number is: 10
```

#1.9 Write a program to find the factors of a whole number using a while loop.

```
num = int(input("Enter a number: "))
```

```
print("Factors of", num, "are:")
i = 1
while i <= num:
    if num % i == 0:
        print(i)
    i += 1
```

```
Enter a number: 40
Factors of 40 are:
1
2
4
5
8
10
20
40
```

1.10. Write a program to find the sum of all the positive numbers entered by the user. As soon as the user enters a negative number, stop.

```
sum = 0
while True:
    num = int(input("Enter a positive number (enter a negative number to stop): "))
    if num < 0:
        break
    sum += num
print("The sum of all positive numbers entered is:", sum)
```

```
Enter a positive number (enter a negative number to stop): 4
Enter a positive number (enter a negative number to stop): 5
Enter a positive number (enter a negative number to stop): 7
Enter a positive number (enter a negative number to stop): 3
Enter a positive number (enter a negative number to stop): -2
The sum of all positive numbers entered is: 19
```

```
#1.11. Write a program to find prime numbers between 2 to 100 using nested for loops.
print("Prime numbers between 2 and 100 are:")
lst_prime=[]
for num in range(2, 101):
    for i in range(2, num):
        if num % i == 0:
            break
    else:
        lst_prime.append(num)
print(lst_prime)
```

→ Prime numbers between 2 and 100 are:
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]

1.12. Write the programs for the followingp Accept the marks of the student in five major subjects and display the same? Calculate the sum of the marks of all subjects.Divide the total marks by number of subjects (i.e. 5), calculate percentage = total marks/5 and display the percentage? Find the grade of the student as per the following criteria . Hint: Use Match & case for this.:

Criteria	Grade
percentage > 85	A
percentage < 85 && percentage >= 75	B
percentage < 75 && percentage >= 50	C
percentage > 30 && percentage <= 50	D
percentage <30	Reappear

```
marks=[]
subjects=['English', 'Hindi', 'Maths','Science','Social Science']
for i in range(len(subjects)):
    marks.append(int(input(f"Enter the marks of {subjects[i]} out of 100 : ")))

sum=0
for j in range(len(marks)):
    sum=sum+marks[j]
print(f"Total marks: {sum}")
percentage=sum/5
print(f"The percentage obtained is: {percentage}%")
match percentage:
    case percentage if percentage>=85:
        print("Grade A")
    case percentage if percentage<85 and percentage>=75:
        print("Grade B")
    case percentage if percentage<75 and percentage>=50:
        print("Grade C")
    case percentage if percentage<50 and percentage>=30:
        print("Grade D")
    case percentage if percentage<30:
        print("Reappear")
```

→ Enter the marks of English out of 100 : 73
Enter the marks of Hindi out of 100 : 66
Enter the marks of Maths out of 100 : 75
Enter the marks of Science out of 100 : 80
Enter the marks of Social Science out of 100 : 64
Total marks: 358
The percentage obtained is: 71.6%
Grade C

COLOR	WAVELENGTH (nm)
Violet	400.0-440.0
Indigo	440.0-460.0
Blue	460.0-500.0
Green	500.0-570.0
Yellow	570.0-590.0
Orange	590.0-620.0
Red	620.0-720.0

1.13. Write a program for VIBGYOR Spectrum base+ on their Wavelength using.

Wavelength Range:

```
wavelength=float(input("Enter the wavelength of signal:"))
match wavelength:
    case wavelength if wavelength>=400 and wavelength<=440:
        print("violet")
    case wavelength if wavelength>=441 and wavelength<=460:
        print("Indigo")
    case wavelength if wavelength>=461 and wavelength<=500:
        print("Blue")
    case wavelength if wavelength>=501 and wavelength<=570:
        print("Green")
    case wavelength if wavelength>=571 and wavelength<=590:
        print("Yellow")
    case wavelength if wavelength>=591 and wavelength<=620:
        print("Orange")
    case wavelength if wavelength>=621 and wavelength<=720:
        print("Red")
    case wavelength if wavelength>=721 and wavelength <400:
        print("Invalid")
```

Enter the wavelength of signal:476
Blue


1.14.Consi+er the gravitational interactions between the Earth, Moon, and Sun in our solar system. Given: mass_earth = 5A972e24 # Mass of Earth in kilograms mass_moon = 7A347 7309e22 # Mass of Moon in kilograms mass_sun = 1A989e30 # Mass of Sun in kilograms distance_earth_sun = 1.496 e11 # Average distance between Earth and Sun in meters distance_moon_earth = 3.844e8 # Average distance between Moon and Earth in meters Tasks

- Calculate the gravitational force between the Earth and the Sun.
- Calculate the gravitational force between the Moon and the Earth.
- Compare the calculated forces to determine which gravitational force is stronger.
- Explain which celestial body (Earth or Moon) is more attracted to the other based on the comparison.

```

mass_earth = 5.972e24 # Mass of Earth in kilograms
mass_moon = 7.34767309e22 # Mass of Moon in kilograms
mass_sun = 1.989e30 # Mass of Sun in kilograms
distance_earth_sun = 1.496e11 # Average distance between Earth and Sun in meters
distance_moon_earth = 3.844e8 # Average distance between Moon and Earth in meters
#Calculate the gravitational force between the Earth and the Sun.
#From the universal law of gravitation,  $F=G*m_1*m_2/r^2$ 
F1=(6.67408e-11)*mass_earth*mass_sun/(distance_earth_sun**2)
print(f"The gravitational force between the Earth and the Sun is {F1}")
#Calculate the gravitational force between the Moon and the Earth.
F2=(6.67408e-11)*mass_earth*mass_moon/(distance_moon_earth**2)
print(f"The gravitational force between the Moon and the Earth is {F2}")
if F1>F2:
    print("The Earth is more attracted to the Sun")
else:
    print("The Moon is more attracted to the Earth")

```

 The gravitational force between the Earth and the Sun is 3.5422793159941665e+22
 The gravitational force between the Moon and the Earth is 1.9819572137137452e+20
 The Earth is more attracted to the Sun

4. Design and implement a Python program for managing student information using object-oriented principles. Create a class called `Student` with encapsulated attributes for name, age, and roll number. Implement getter and setter methods for these attributes. Additionally, provide methods to display student information and update student details. Tasks: . Define the `Student` class with encapsulated attributes. . Implement getter and setter methods for the attributes. . Write methods to display student information and update details. . Create instances of the `Student` class and test the implemented functionality.

```


class Student:
    def __init__(self,name,age,roll_number):
        self.__name=name
        self.__age=age
        self.__roll_number=roll_number

    def get_details(self):
        print(self.__name)
        print(self.__age)
        print(self.__roll_number)

    def set_details(self,name,age,roll_number):
        self.__name=name
        self.__age=age
        self.__roll_number=roll_number

stud1=Student("Ankita", 16, 105)
stud1.get_details()
stud1.set_details("Ankita", 18, 105)
stud1.get_details()

```

 Ankita
 16
 105
 Ankita
 18
 105

3. Develop a Python program for managing library resources efficiently. Design a class named `LibraryBook` with attributes like book name, author, and availability status. Implement methods for borrowing and returning books while ensuring proper encapsulation of attributes. Tasks:

1. Create the `LibraryBook` class with encapsulated attributes.
2. Implement methods for borrowing and returning books.
3. Ensure proper encapsulation to protect book details.
4. Test the borrowing and returning functionality with simple data.

```

class LibraryBook:
    def __init__(self,book_name,author,availability_status=True):
        self.__book_name=book_name
        self.__author=author
        self.__availability_status=True

    def get_details(self):
        print(self.__book_name)
        print(self.__author)

    def borrow(self):
        if self.__availability_status:
            self.__availability_status = False
            print(f"You have borrowed '{self.__book_name}' by {self.__author}.")
        else:
            print(f"Sorry, '{self.__book_name}' is currently not available.")

    def return_book(self):
        if not self.__availability_status:
            self.__availability_status = True
            print(f"Thank you for returning '{self.__book_name}' by {self.__author}.")
        else:
            print(f"'{self.__book_name}' was not borrowed.")

book1=LibraryBook("Learning Python", "Mark Lutz")
book2=LibraryBook("The Alchemist", "Paulo Coelho")
book3=LibraryBook("The Da Vinci Code", "Dan Brown")
book1.get_details()
book1.borrow()
book2.return_book()
book1.return_book()

```

```

↔ Learning Python
Mark Lutz
You have borrowed 'Learning Python' by Mark Lutz.
'The Alchemist' was not borrowed.
Thank you for returning 'Learning Python' by Mark Lutz.

```

4. Create a simple banking system using object-oriented concepts in Python. Design classes representing different types of bank accounts such as savings and checking. Implement methods for deposit, withdraw, and balance inquiry. Utilize inheritance to manage different account types efficiently. Tasks:

1. Define base class(es) for bank accounts with common attributes and methods.
2. Implement subclasses for specific account types (e.g., SavingsAccount, CheckingAccount)
3. Provide methods for deposit, withdraw, and balance inquiry in each subclass.
4. Test the banking system by creating instances of different account types and performing transactions.

```
class BankAccount:
    def __init__(self, account_number, holder_name, balance=0):
        self.__account_number = account_number
        self.__holder_name = holder_name
        self.__balance = balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited {amount}. New balance is {self.__balance}.")
        else:
            print("Deposit amount must be positive.")

    def withdraw(self, amount):
        if amount > 0:
            if amount <= self.__balance:
                self.__balance -= amount
                print(f"Withdrew {amount}. New balance is {self.__balance}.")
            else:
                print("Insufficient funds.")
        else:
            print("Withdrawal amount must be positive.")

    def get_balance(self):
        return self.__balance

    def account_info(self):
        return f"Account Number: {self.__account_number}\nHolder Name: {self.__holder_name}\nBalance: ${self.__balance}"

class SavingsAccount(BankAccount):
    def __init__(self, account_number, holder_name, balance=0, interest_rate=0.02):
        super().__init__(account_number, holder_name, balance)
        self.__interest_rate = interest_rate

    def apply_interest(self):
        interest = self.get_balance() * self.__interest_rate
        self.deposit(interest)
        print(f"Applied interest: {interest}")

    def account_info(self):
        return super().account_info() + f"\nInterest Rate: {self.__interest_rate * 100}%"

class CheckingAccount(BankAccount):
    def __init__(self, account_number, holder_name, balance=0, overdraft_limit=500):
        super().__init__(account_number, holder_name, balance)
        self.__overdraft_limit = overdraft_limit

    def withdraw(self, amount):
        if amount > 0:
            if amount <= (self.get_balance() + self.__overdraft_limit):
                self.__balance -= amount
                print(f"Withdrew {amount}. New balance is {self.get_balance()}.")
            else:
                print(f"Insufficient funds. Your overdraft limit is {self.__overdraft_limit}.")
        else:
            print("Withdrawal amount must be positive.")

    def account_info(self):
        return super().account_info() + f"\nOverdraft Limit: {self.__overdraft_limit}"

# Testing the Banking System
if __name__ == "__main__":
    # Create instances of SavingsAccount and CheckingAccount
    savings = SavingsAccount(account_number="SA123456", holder_name="Ankita", balance=10000)
    checking = CheckingAccount(account_number="CA123456", holder_name="Deepak", balance=1000, overdraft_limit=200)

    # Perform transactions on SavingsAccount
    print("\nSavings Account:")
    print(savings.account_info())
    savings.deposit(200)
    savings.withdraw(150)
    savings.apply_interest()
    print(savings.account_info())

    # Perform transactions on CheckingAccount
    print("\nChecking Account:")
    print(checking.account_info())
    checking.withdraw(1300) # Should exceed overdraft limit
```



```

Savings Account:
Account Number: SA123456
Holder Name: Ankita
Balance: $10000
Interest Rate: 2.0%
Deposited 200. New balance is 10200.
Withdrew 150. New balance is 10050.
Deposited 201.0. New balance is 10251.0.
Applied interest: 201.0
Account Number: SA123456
Holder Name: Ankita
Balance: $10251.0
Interest Rate: 2.0%

Checking Account:
Account Number: CA123456
Holder Name: Deepak
Balance: $1000
Overdraft Limit: 200
Insufficient funds. Your overdraft limit is 200.

```

5. Write a Python program that models different animals and their sounds. Design a base class called `Animal` with a method `make_sound()`. Create subclasses like `Dog` and `Cat` that override the `make_sound()` method to produce appropriate sounds. Tasks:

1. Define the `Animal` class with a method `make_sound()`.
2. Create subclasses `Dog` and `Cat` that override the `make_sound()` method.
3. Implement the sound generation logic for each subclass.
4. Test the program by creating instances of `Dog` and `Cat` and calling the `make_sound()` method.

```

class Animal:
    def make_sound(self):
        print("Animal makes a sound")

class Dog(Animal):
    def make_sound(self):
        print("Dog barks")

class Cat(Animal):
    def make_sound(self):
        print("Cat meows")

dog=Dog()
dog.make_sound()
cat=Cat()
cat.make_sound()

```



```

Dog barks
Cat meows

```

6. Write a code for Restaurant Management System Using OOPS/

- Create a `MenuItem` class that has attributes such as name, description, price, and category.
- Implement methods to add a new menu item, update menu item information, and remove a menu item from the menu
- Use encapsulation to hide the menu item's unique identification number.
- Inherit from the `MenuItem` class to create a `FoodItem` class and a `BeverageItem` class, each with their own specific attributes and methods.


```

import itertools

class MenuItem:
    _id_counter = itertools.count(1)

    def __init__(self, name, description, price, category):
        self._id = next(MenuItem._id_counter)
        self.name = name
        self.description = description
        self.price = price
        self.category = category

    @property
    def id(self):
        return self._id

    def update_info(self, name=None, description=None, price=None, category=None):
        if name:
            self.name = name
        if description:
            self.description = description
        if price is not None:
            self.price = price
        if category:
            self.category = category

    def __str__(self):
        return f"ID: {self._id}, Name: {self.name}, Description: {self.description}, Price: {self.price:.2f}, Category: {self.category}"

class FoodItem(MenuItem):
    def __init__(self, name, description, price, category, cuisine_type):
        super().__init__(name, description, price, category)
        self.cuisine_type = cuisine_type

    def update_info(self, name=None, description=None, price=None, category=None, cuisine_type=None):
        super().update_info(name, description, price, category)
        if cuisine_type:
            self.cuisine_type = cuisine_type

    def __str__(self):
        return f"{super().__str__()}, Cuisine Type: {self.cuisine_type}"

class BeverageItem(MenuItem):
    def __init__(self, name, description, price, category, volume):
        super().__init__(name, description, price, category)
        self.volume = volume

    def update_info(self, name=None, description=None, price=None, category=None, volume=None):
        super().update_info(name, description, price, category)
        if volume is not None:
            self.volume = volume

    def __str__(self):
        return f"{super().__str__()}, Volume: {self.volume}ml"

class Menu:
    def __init__(self):
        self.items = {}

    def add_item(self, item):
        if item.id in self.items:
            print("Item with this ID already exists.")
        else:
            self.items[item.id] = item
            print(f"Item '{item.name}' added to the menu.")

    def remove_item(self, item_id):
        if item_id in self.items:
            removed_item = self.items.pop(item_id)
            print(f"Item '{removed_item.name}' removed from the menu.")
        else:
            print("Item not found.")

    def update_item(self, item_id, **kwargs):
        if item_id in self.items:
            self.items[item_id].update_info(**kwargs)
            print(f"Item '{self.items[item_id].name}' updated.")
        else:
            print("Item not found.")

    def display_menu(self):
        for item in self.items.values():

```

```

        print(item)

# Example usage:
menu = Menu()

# Adding items
pizza = FoodItem(name="Margherita Pizza", description="Classic pizza with tomato and cheese", price=299, category="Main Course", cuisine="Italian")
coffee = BeverageItem(name="Espresso", description="Strong coffee", price=80, category="Beverage", volume=300)

menu.add_item(pizza)
menu.add_item(coffee)

# Display menu
print("\nMenu:")
menu.display_menu()

# Updating an item
menu.update_item(pizza.id, price=349, cuisine_type="Mediterranean")

# Display menu again
print("\nUpdated Menu:")
menu.display_menu()

# Removing an item
menu.remove_item(coffee.id)

# Display menu again
print("\nFinal Menu:")
menu.display_menu()

```

Item 'Margherita Pizza' added to the menu.
Item 'Espresso' added to the menu.

Menu:
ID: 1, Name: Margherita Pizza, Description: Classic pizza with tomato and cheese, Price: 299.00, Category: Main Course, Cuisine Type: Italian
ID: 2, Name: Espresso, Description: Strong coffee, Price: 80.00, Category: Beverage, Volume: 300ml
Item 'Margherita Pizza' updated.

Updated Menu:
ID: 1, Name: Margherita Pizza, Description: Classic pizza with tomato and cheese, Price: 349.00, Category: Main Course, Cuisine Type: Mediterranean
ID: 2, Name: Espresso, Description: Strong coffee, Price: 80.00, Category: Beverage, Volume: 300ml
Item 'Espresso' removed from the menu.

Final Menu:
ID: 1, Name: Margherita Pizza, Description: Classic pizza with tomato and cheese, Price: 349.00, Category: Main Course, Cuisine Type: Mediterranean

7. Write a code for Hotel Management System using OOPS /

- Create a Room class that has attributes such as room number, room type, rate, and availability (private)
- Implement methods to book a room, check in a guest, and check out a guest
- Use encapsulation to hide the room's unique identification number
- Inherit from the Room class to create a SuiteRoom class and a StandardRoom class, each with their own specific attributes and methods.

```

import itertools

class Room:
    _id_counter = itertools.count(1)

    def __init__(self, room_number, room_type, rate):
        self._id = next(Room._id_counter) # Unique ID for each room
        self.room_number = room_number
        self.room_type = room_type
        self.rate = rate
        self._availability = True # Room is available by default
        self.current_guest = None

    @property
    def id(self):
        return self._id

    @property
    def availability(self):
        return self._availability

    def book_room(self):
        if self._availability:
            self._availability = False
            print(f"Room {self.room_number} booked.")
        else:
            print(f"Room {self.room_number} is already booked.")

    def check_in(self, guest_name):
        if not self._availability:
            self.current_guest = guest_name
            print(f"Guest {guest_name} checked into room {self.room_number}.")
        else:
            print(f"Room {self.room_number} is available. Cannot check in.")

    def check_out(self):
        if not self._availability and self.current_guest:
            print(f"Guest {self.current_guest} checked out of room {self.room_number}.")
            self.current_guest = None
            self._availability = True
        else:
            print(f"Room {self.room_number} is already available or no guest to check out.")

    def __str__(self):
        availability_status = "Available" if self._availability else "Occupied"
        return f"Room {self.room_number} ({self.room_type}) - Rate:Rs {self.rate:.2f}, Status: {availability_status}"

class SuiteRoom(Room):
    def __init__(self, room_number, rate, amenities):
        super().__init__(room_number, "Suite", rate)
        self.amenities = amenities

    def __str__(self):
        return f"{super().__str__()}, Amenities: {'', '.join(self.amenities)}"

class StandardRoom(Room):
    def __init__(self, room_number, rate, bed_type):
        super().__init__(room_number, "Standard", rate)
        self.bed_type = bed_type

    def __str__(self):
        return f"{super().__str__()}, Bed Type: {self.bed_type}"

class Hotel:
    def __init__(self):
        self.rooms = {}

    def add_room(self, room):
        if room.id in self.rooms:
            print("Room with this ID already exists.")
        else:
            self.rooms[room.id] = room
            print(f"Room {room.room_number} added to the hotel.")

    def book_room(self, room_id):
        if room_id in self.rooms:
            self.rooms[room_id].book_room()
        else:
            print("Room not found.")

```

```

def check_in(self, room_id, guest_name):
    if room_id in self.rooms:
        self.rooms[room_id].check_in(guest_name)
    else:
        print("Room not found.")

def check_out(self, room_id):
    if room_id in self.rooms:
        self.rooms[room_id].check_out()
    else:
        print("Room not found.")

def display_rooms(self):
    for room in self.rooms.values():
        print(room)

# Example usage:
hotel = Hotel()

# Adding rooms
suite = SuiteRoom(room_number=101, rate=4500.00, amenities=["Jacuzzi", "King-sized bed", "Ocean view"])
standard = StandardRoom(room_number=102, rate=2500.00, bed_type="Queen-sized")

hotel.add_room(suite)
hotel.add_room(standard)

# Display rooms
print("\nAvailable Rooms:")
hotel.display_rooms()


# Booking and checking in
hotel.book_room(suite.id)
hotel.check_in(suite.id, "Deepak Roy")

# Display rooms again
print("\nRooms after check-in:")
hotel.display_rooms()

# Checking out
hotel.check_out(suite.id)

# Display rooms after checkout
print("\nRooms after checkout:")
hotel.display_rooms()


```

 Room 101 added to the hotel.
 Room 102 added to the hotel.

Available Rooms:
 Room 101 (Suite) - Rate: 4500.00, Status: Available, Amenities: Jacuzzi, King-sized bed, Ocean view
 Room 102 (Standard) - Rate: 2500.00, Status: Available, Bed Type: Queen-sized
 Room 101 booked.
 Guest Deepak Roy checked into room 101.

Rooms after check-in:
 Room 101 (Suite) - Rate: 4500.00, Status: Occupied, Amenities: Jacuzzi, King-sized bed, Ocean view
 Room 102 (Standard) - Rate: 2500.00, Status: Available, Bed Type: Queen-sized
 Guest Deepak Roy checked out of room 101.

Rooms after checkout:
 Room 101 (Suite) - Rate: 4500.00, Status: Available, Amenities: Jacuzzi, King-sized bed, Ocean view
 Room 102 (Standard) - Rate: 2500.00, Status: Available, Bed Type: Queen-sized

8. Write a code for Fitness Club Management System using OOPS/

- Create a Member class that has attributes such as name, age, membership type, and membership status(private)
- Implement methods to register a new member, renew a membership, and cancel a membership.
- Use encapsulation to hide the member's unique identification number
- Inherit from the Member class to create a FamilyMember class and an IndividualMember class, each with their own specific attributes and methods.

```

class Member:
    def __init__(self, name, age, membership_type):
        self.name = name
        self.age = age
        self.membership_type = membership_type
        self.__membership_status = 'Active' # Private attribute for membership status
        self.__unique_id = id(self) # Private unique ID for each member

    def register_member(self):
        print(f"Member '{self.name}' registered with ID {self.__unique_id}.")

    def renew_membership(self):
        if self.__membership_status == 'Inactive':
            self.__membership_status = 'Active'
            print(f"Membership for '{self.name}' has been renewed.")
        else:
            print(f"Membership for '{self.name}' is already active.")

    def cancel_membership(self):
        if self.__membership_status == 'Active':
            self.__membership_status = 'Inactive'
            print(f"Membership for '{self.name}' has been canceled.")
        else:
            print(f"Membership for '{self.name}' is already inactive.")

    def get_status(self):
        return self.__membership_status

    def get_unique_id(self):
        return self.__unique_id

class FamilyMember(Member):
    def __init__(self, name, age, membership_type, family_size):
        super().__init__(name, age, membership_type)
        self.family_size = family_size

    def add_family_member(self):
        self.family_size += 1
        print(f"Family size updated to {self.family_size} for '{self.name}'.")

    def remove_family_member(self):
        if self.family_size > 1:
            self.family_size -= 1
            print(f"Family size updated to {self.family_size} for '{self.name}'.")
        else:
            print(f"Cannot remove family member. Only one member left in the family.")

class IndividualMember(Member):
    def __init__(self, name, age, membership_type, personal_training_required=False):
        super().__init__(name, age, membership_type)
        self.personal_training_required = personal_training_required


    def request_personal_training(self):
        if self.personal_training_required:
            print(f"Personal training requested for '{self.name}'.")
        else:
            print(f"Personal training is not requested by '{self.name}'.")

# Example usage:

# Register a family member
family_member = FamilyMember(name="Deepak", age=35, membership_type="Full time", family_size=2)
family_member.register_member()
family_member.add_family_member()
family_member.remove_family_member()
family_member.cancel_membership()
print(f"Membership status: {family_member.get_status()}")
print(f"Unique ID: {family_member.get_unique_id()}")

# Register an individual member
individual_member = IndividualMember(name="Harsh", age=25, membership_type="Part time", personal_training_required=True)
individual_member.register_member()
individual_member.request_personal_training()
individual_member.renew_membership()
print(f"Membership status: {individual_member.get_status()}")
print(f"Unique ID: {individual_member.get_unique_id()}")

```

 Member 'Deepak' registered with ID 132277477011328.
 Family size updated to 3 for 'Deepak'.
 Family size updated to 2 for 'Deepak'.
 Membership for 'Deepak' has been canceled.
 Membership status: Inactive

```
Unique ID: 132277477011328
Member 'Harsh' registered with ID 132277477011424.
Personal training requested for 'Harsh'.
Membership for 'Harsh' is already active.
Membership status: Active
Unique ID: 132277477011424
```

9. Write a code for Event Management System using OOPS/

- Create an Event class that has attributes such as name, date, time, location, and list of attendees (private)
- Implement methods to create a new event, add or remove attendees, and get the total number of attendees.
- Use encapsulation to hide the event's unique identification number
- Inherit from the Event class to create a PrivateEvent class and a PublicEvent class, each with their own specific attributes and methods.

```

class Event:
    def __init__(self, name, date, time, location):
        self.name = name
        self.date = date
        self.time = time
        self.location = location
        self.__attendees = [] # Private list of attendees
        self.__unique_id = id(self) # Private unique ID for each event

    def create_event(self):
        print(f"Event '{self.name}' created with ID {self.__unique_id} on {self.date} at {self.time} in {self.location}.")

    def add_attendee(self, attendee):
        if attendee not in self.__attendees:
            self.__attendees.append(attendee)
            print(f"Attendee '{attendee}' added to event '{self.name}'.")
        else:
            print(f"Attendee '{attendee}' is already on the list for event '{self.name}'.")

    def remove_attendee(self, attendee):
        if attendee in self.__attendees:
            self.__attendees.remove(attendee)
            print(f"Attendee '{attendee}' removed from event '{self.name}'.")
        else:
            print(f"Attendee '{attendee}' was not found in the list for event '{self.name}'.")

    def get_total_attendees(self):
        return len(self.__attendees)

    def get_unique_id(self):
        return self.__unique_id

class PrivateEvent(Event):
    def __init__(self, name, date, time, location, invite_only):
        super().__init__(name, date, time, location)
        self.invite_only = invite_only

    def set_invite_only(self, status):
        self.invite_only = status
        print(f"Invite-only status set to {self.invite_only} for private event '{self.name}'.")

class PublicEvent(Event):
    def __init__(self, name, date, time, location, is_free_entry):
        super().__init__(name, date, time, location)
        self.is_free_entry = is_free_entry

    def set_free_entry(self, status):
        self.is_free_entry = status
        print(f"Free entry status set to {self.is_free_entry} for public event '{self.name}'.")

# Example usage:

# Create a private event
private_event = PrivateEvent(name="VIP Gala", date="2024-08-15", time="19:00", location="AB Auditorium", invite_only=True)
private_event.create_event()
private_event.add_attendee("Ankita")
private_event.add_attendee("Asmita")
private_event.remove_attendee("Asmita")
private_event.set_invite_only(False)
print(f"Total attendees for '{private_event.name}': {private_event.get_total_attendees()}")
print(f"Unique ID: {private_event.get_unique_id()}")

# Create a public event
public_event = PublicEvent(name="Community Fair", date="2024-09-01", time="10:00", location="Central Park", is_free_entry=True)
public_event.create_event()
public_event.add_attendee("Deepak")
public_event.add_attendee("Harsh")
public_event.add_attendee("Ankita")
public_event.add_attendee("Asmita")
public_event.remove_attendee("Asmita")
public_event.set_free_entry(False)
print(f"Total attendees for '{public_event.name}': {public_event.get_total_attendees()}")
print(f"Unique ID: {public_event.get_unique_id()}")

```



```

Event 'VIP Gala' created with ID 132277529285184 on 2024-08-15 at 19:00 in AB Auditorium.
Attendee 'Ankita' added to event 'VIP Gala'.
Attendee 'Asmita' added to event 'VIP Gala'.
Attendee 'Asmita' removed from event 'VIP Gala'.
Invite-only status set to False for private event 'VIP Gala'.
Total attendees for 'VIP Gala': 1
Unique ID: 132277529285184
Event 'Community Fair' created with ID 132277529285232 on 2024-09-01 at 10:00 in Central Park.

```

```

Attendee 'Deepak' added to event 'Community Fair'.
Attendee 'Harsh' added to event 'Community Fair'.
Attendee 'Ankita' added to event 'Community Fair'.
Attendee 'Asmita' added to event 'Community Fair'.
Attendee 'Asmita' removed from event 'Community Fair'.
Free entry status set to False for public event 'Community Fair'.
Total attendees for 'Community Fair': 3
Unique ID: 132277529285232

```

10. Write a code for Airline Reservation System using OOPS/

- Create a Flight class that has attributes such as flight number, departure and arrival airports, departure and arrival times, and available seats (private)
- Implement methods to book a seat, cancel a reservation, and get the remaining available seats.
- Use encapsulation to hide the flight's unique identification number
- Inherit from the Flight class to create a DomesticFlight class and an InternationalFlight class, each with their own specific attributes and methods.

10. Write a code for Airline Reservation System using OOPS/

```

+ Create a Flight class that has attributes such as flight number, departure and arrival airports, departure and arrival times, and available seats (private)
+ Implement methods to book a seat, cancel a reservation, and get the remaining available seats
+ Use encapsulation to hide the flight's unique identification number
+ Inherit from the Flight class to create a DomesticFlight class and an InternationalFlight class, each with their own specific attributes and methods.

```



```

class Flight:
    def __init__(self, flight_number, departure_airport, arrival_airport, departure_time, arrival_time, total_seats):
        self.flight_number = flight_number
        self.departure_airport = departure_airport
        self.arrival_airport = arrival_airport
        self.departure_time = departure_time
        self.arrival_time = arrival_time
        self.__available_seats = total_seats # Private attribute for available seats
        self.__unique_id = id(self) # Private unique ID for each flight

    def book_seat(self, number_of_seats):
        if number_of_seats <= self.__available_seats:
            self.__available_seats -= number_of_seats
            print(f"Successfully booked {number_of_seats} seat(s).")
        else:
            print("No seats available.")

    def cancel_reservation(self, number_of_seats):
        self.__available_seats += number_of_seats
        print(f"Successfully canceled {number_of_seats} seat(s).")

    def get_remaining_seats(self):
        return self.__available_seats

    def get_unique_id(self):
        return self.__unique_id

class DomesticFlight(Flight):
    def __init__(self, flight_number, departure_airport, arrival_airport, departure_time, arrival_time, total_seats, state):
        super().__init__(flight_number, departure_airport, arrival_airport, departure_time, arrival_time, total_seats)
        self.state = state # Additional attribute for domestic flights

    def display_flight_info(self):
        print(f"Domestic Flight {self.flight_number} from {self.departure_airport} to {self.arrival_airport} "
              f"on {self.departure_time} arriving at {self.arrival_time}.")
        print(f"State: {self.state}")
        print(f"Remaining Seats: {self.get_remaining_seats()}")

class InternationalFlight(Flight):
    def __init__(self, flight_number, departure_airport, arrival_airport, departure_time, arrival_time, total_seats, country):
        super().__init__(flight_number, departure_airport, arrival_airport, departure_time, arrival_time, total_seats)
        self.country = country # Additional attribute for international flights

    def display_flight_info(self):
        print(f"International Flight {self.flight_number} from {self.departure_airport} to {self.arrival_airport} "
              f"on {self.departure_time} arriving at {self.arrival_time}.")
        print(f"Country: {self.country}")
        print(f"Remaining Seats: {self.get_remaining_seats()}")

# Example usage:

# Create a domestic flight
domestic_flight = DomesticFlight(
    flight_number="AI123",
    departure_airport="RPR",
    arrival_airport="DUM",
    departure_time="2024-08-10 08:00",
    arrival_time="2024-08-10 11:00",
    total_seats=150,
    state="West Bengal"
)
domestic_flight.display_flight_info()
domestic_flight.book_seat(10)
domestic_flight.cancel_reservation(5)
print(f"Remaining seats: {domestic_flight.get_remaining_seats()}")
print(f"Unique ID: {domestic_flight.get_unique_id()}")

# Create an international flight
international_flight = InternationalFlight(
    flight_number="AI456",
    departure_airport="BOM",
    arrival_airport="MAU",
    departure_time="2024-08-15 16:00",
    arrival_time="2024-08-16 08:00",
    total_seats=200,
    country="Mauritis"
)
international_flight.display_flight_info()
international_flight.book_seat(20)
international_flight.cancel_reservation(10)
print(f"Remaining seats: {international_flight.get_remaining_seats()}")

```

```
print(f"Unique ID: {international_flight.get_unique_id()}")
```

```
Domestic Flight AI123 from RPR to DUM on 2024-08-10 08:00 arriving at 2024-08-10 11:00.
State: West Bengal
Remaining Seats: 150
Successfully booked 10 seat(s).
Successfully canceled 5 seat(s).
Remaining seats: 145
Unique ID: 132277477150432
International Flight AI456 from BOM to MAU on 2024-08-15 16:00 arriving at 2024-08-16 08:00.
Country: Mauritis
Remaining Seats: 200
Successfully booked 20 seat(s).
Successfully canceled 10 seat(s).
Remaining seats: 190
Unique ID: 132277477156624
```

11. Define a Python module named constants.py containing constants like pi and the speed of light.

```
import constant_py as con
print(con.pi)
print(con.speed_of_light)
```

```
3.142
3000000000.0
```

12. Write a Python module named calculator.py containing functions for addition, subtraction, multiplication, and division.

```
import calculator as cal

# Access the function within the module
print(cal.Calc(5, 4, "+"))
print(cal.Calc(5, 4, "-"))
print(cal.Calc(5, 4, "*"))
print(cal.Calc(5, 4, "/"))
```

```
9
1
20
1.25
```

13. Implement a Python package structure for a project named ecommerce, containing modules for product management and order processing.

```
ecommerce/
├── ecommerce/
│   ├── __init__.py
│   ├── product_management/
│   │   ├── __init__.py
│   │   ├── product.py
│   │   └── inventory.py
│   ├── order_processing/
│   │   ├── __init__.py
│   │   ├── order.py
│   │   └── payment.py
│   └── utils.py
├── tests/
│   ├── __init__.py
│   ├── test_product_management.py
│   └── test_order_processing.py
├── setup.py
├── README.md
└── requirements.txt
```

14. Implement a Python module named string_utils.py containing functions for string manipulation, such as reversing and capitalizing strings.

```
import string_utils as su

print(su.string_manipulation("deepak"))
```

↗ The reversed string is: kapedd
The capitalized string is: DEEPAK
None

15. Write a Python module named file_operations.py with functions for reading, writing, and appending data to a file.

```
import file_operation_py as fo
fp="/content/sample_data/newfile.txt"
fo.read_file(fp)
content="This is a new file. I am Ankita Chandrakar, enthusiastically diving deep into the world of data science. "
fo.write_file(fp, content )
fo.append_file(fp, "I am from engineering background.")
```

↗ Content written to file succesfully
Content appended to file succesfully

16. Write a Python program to create a text file named "employees.txt" and write the details of employees, including their name, age, and salary, into the file.

```
def write_employee_details(filename, employees):
    """
    Writes the details of employees to a text file.

    :param filename: Name of the file to write to.
    :param employees: List of tuples, each containing employee details (name, age, salary).
    """
    with open(filename, 'w') as file:
        for name, age, salary in employees:
            file.write(f"Name: {name}\n")
            file.write(f"Age: {age}\n")
            file.write(f"Salary: ${salary:.2f}\n")
            file.write("\n") # Add a blank line between employee records

def main():
    # List of employee details: (name, age, salary)
    employees = [
        ("Ankita", 30, 50000),
        ("Asmita", 25, 55000),
        ("Harsh", 28, 60000),
        ("Deepak", 35, 70000)
    ]

    filename = 'employees.txt'

    write_employee_details(filename, employees)
    print(f"Employee details have been written to {filename}")

if __name__ == "__main__":
    main()
```

↗ Employee details have been written to employees.txt

17. Develop a Python script that opens an existing text file named "inventory.txt" in read mode and displays the contents of the file line by line.

```
import read_inventory as ri
ri.read_and_display_file("inventory.txt")
```

↗ This is an inventory file.
Product ID: 101
Product Name: Widget A
Quantity: 50
Price: 1299

Product ID: 102
Product Name: Widget B
Quantity: 30
Price: 1499

Product ID: 103
Product Name: Gadget C
Quantity: 20

Price: 1799

Product ID: 104
 Product Name: Gadget D
 Quantity: 15
 Price: 1999

18. Create a Python script that reads a text file named "expenses.txt" and calculates the total amount spent on various expenses listed in the file.

```
import calc_expenses as ce
ce.calculate_total_expenses("expenses.txt")
```

34200.0

19. Create a Python program that reads a text file named "paragraph.txt" and counts the occurrences of each word in the paragraph, displaying the results in alphabetical order.

```
import count_words as cw
cw.count_words("paragraph.txt")
```

```
'background': 1,
'and': 5,
'passion': 1,
'for': 1,
'data': 3,
'science': 2,
'gate': 1,
'qualified': 1,
'professional': 1,
'three': 1,
'years': 1,
'of': 2,
'teaching': 1,
'experience': 2,
'in': 2,
'world': 1,
'bank': 1,
'funded': 1,
'project': 1,
'teqip': 1,
'under': 1,
'npiu': 1,
'unit': 1,
'moe': 1,
'goi': 1,
'formerly': 1,
'mhrd': 1,
'currently': 1,
'expanding': 1,
'my': 1,
'expertise': 1,
'python': 1,
'machine': 1,
'learning': 2,
'to': 5,
'transition': 1,
'into': 1,
'the': 1,
'industry': 1,
'committed': 1,
'leveraging': 1,
'analytical': 1,
'skills': 1,
'research': 1,
'solve': 1,
'complex': 1,
'problems': 1,
'drive': 1,
'driven': 1,
'decision': 1,
'making': 1,
'enthusiastic': 1,
'about': 1,
'continuous': 1,
'eager': 1,
'contribute': 1,
'innovative': 1,
'projects': 1})
```

20. What do you mean by Measure of Central Tendency and Measures of Dispersion .How it can be calculated.

Measures of Central Tendency are: 1) Mean : Arithmetic Average of data Sum of all values divided by the number of values 2) Median: The middle value in a dataset when the values are sorted in ascending or descending order. If there is an even number of values, it is the average of the two middle values. 3) Mode: The value that appears most frequently in a dataset.

Measures of Dispersion are: 1) Range: The difference between minimum and maximum value of data 2) Variance: The average of the squared differences between each data point and the mean. 3) standard deviation: Square root of variance 4) IQR Inter Quartile Range: Range between Q_3 (75%) and Q_1 (25%). $IQR = Q_3 - Q_1$

21. What do you mean by skewness. Explain its types. Use graph to show.

Skewness is a statistical term that describes the asymmetry of the distribution of data around its mean. It provides insight into the direction and extent of the deviation of the distribution from the normal distribution.

Types of Skewness Positive Skewness (Right Skewness):

Definition: When the tail on the right side of the distribution is longer or fatter than the left side. This indicates that most data points are concentrated on the left side, with a few larger values stretching the tail to the right. Characteristics: $Mean > Median > Mode$ Example: Income distribution where most people earn below the median income but a few high earners stretch the distribution to the right. Negative Skewness (Left Skewness):

Definition: When the tail on the left side of the distribution is longer or fatter than the right side. This indicates that most data points are concentrated on the right side, with a few smaller values stretching the tail to the left. Characteristics: $Mean < Median < Mode$ Example: Test scores where most students score high but a few students score very low. Zero Skewness (Symmetric Distribution):

Definition: When the distribution is symmetrical around the mean, meaning both tails are of equal length and the distribution resembles a normal distribution. Characteristics: $Mean = Median = Mode$ Example: Height distribution of adults where most values are around the average height with fewer values at the extremes.

22. Explain PROBABILITY MASS FUNCTION (PMF) and PROBABILITY DENSITY FUNCTION (PDF). and what is the difference between them?

Probability Mass Function (PMF) and Probability Density Function (PDF) are tools used to describe the distribution of random variables. Here's a brief overview:

Probability Mass Function (PMF) Used For: Discrete random variables. Definition: Provides the probability that a discrete random variable is exactly equal to a specific value. Formula: $P(X = x)$ Properties: Sum of probabilities for all possible values equals 1. Example: Rolling a die (each face has a probability of $\frac{1}{6}$). Probability Density Function (PDF) Used For: Continuous random variables. Definition: Describes the relative likelihood of a continuous random variable falling within a specific range. Probabilities are found by integrating the PDF over an interval. Formula: $P(a \leq X \leq b) = \int_a^b f(x) dx$ Properties: The total area under the curve of the PDF equals 1. Example: Height of people (where probabilities are computed over intervals, not exact values). Key Differences Type of Random Variable:

PMF: Discrete. PDF: Continuous. Probability Calculation:

PMF: Gives exact probabilities for discrete values. PDF: Gives density; probabilities are found over ranges. Summation vs. Integration:

PMF: Summed over all values. PDF: Integrated over ranges.

23. What is correlation. Explain its type in details. what are the methods of determining correlation

Correlation is a statistical measure that describes the strength and direction of a relationship between two variables. It indicates how changes in one variable are associated with changes in another.

Types of Correlation 1) Positive Correlation: Definition: When two variables move in the same direction. As one variable increases, the other also increases. Example: Height and weight of individuals; generally, as height increases, weight tends to increase as well. Correlation Coefficient: Between 0 and +1.

2) Negative Correlation: Definition: When two variables move in opposite directions. As one variable increases, the other decreases. Example: Speed and travel time; as speed increases, the time required to travel a fixed distance decreases. Correlation Coefficient: Between 0 and -1. No Correlation:

Definition: When there is no predictable relationship between the two variables. Changes in one variable do not correspond to predictable changes in the other. Example: Shoe size and intelligence; generally, there is no systematic relationship between these two variables.

Correlation Coefficient: Close to 0. Methods of Determining Correlation 1) Pearson Correlation Coefficient: 2) Spearman's Rank Correlation Coefficient

```

import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt

# Create a sample dataframe
data = {
    'X': np.random.rand(100),
    'Y': np.random.rand(100) * 0.5
}
df = pd.DataFrame(data)

# Compute Pearson correlation
pearson_corr = df.corr(method='pearson')
print(f'Pearson Correlation:\n{pearson_corr}')

# Compute Spearman correlation
spearman_corr = df.corr(method='spearman')
print(f'\nSpearman Correlation:\n{spearman_corr}')

# Plotting correlation heatmap
plt.figure(figsize=(10, 7))
sns.heatmap(df.corr(), annot=True, cmap='coolwarm', fmt='.2f')
plt.title('Correlation Heatmap')
plt.show()

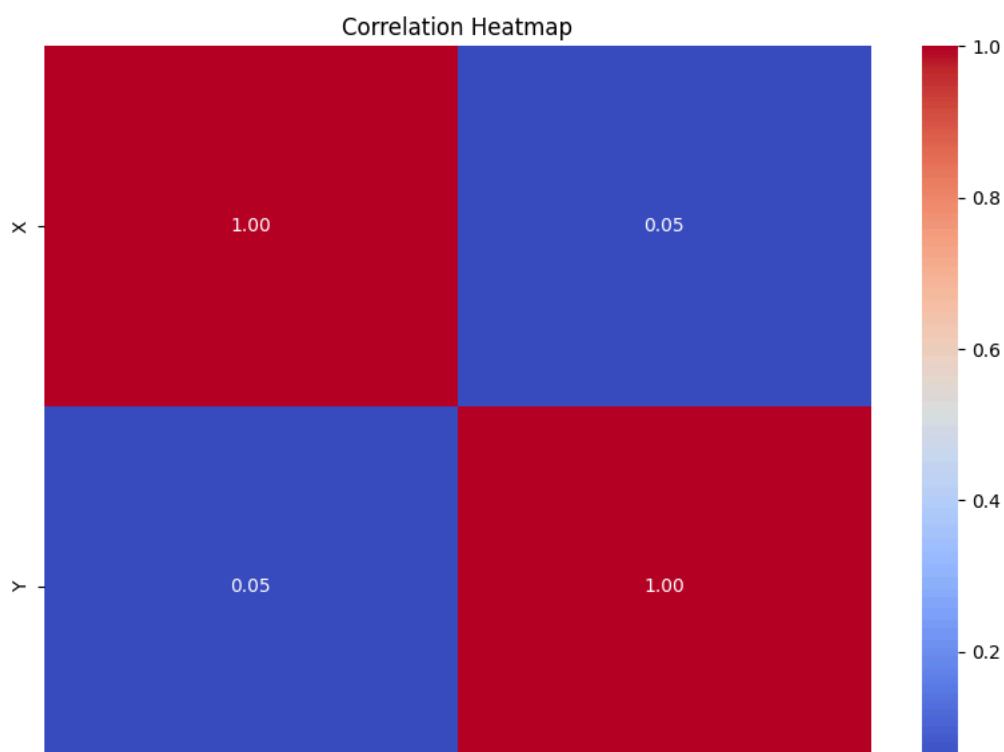
```

➡ Pearson Correlation:

	X	Y
X	1.000000	0.049347
Y	0.049347	1.000000

Spearman Correlation:

	X	Y
X	1.000000	0.032691
Y	0.032691	1.000000



24. Calculate coefficient of correlation between the marks obtained by 10 students in Accountancy and statistics:

Use Karl Pearson's Coefficient of Correlation Method to find it.

Student	1	2	3	4	5	6	7	8	9	10
Accountancy	45	70	65	30	90	40	50	75	85	60
Statistics	35	80	70	40	85	40	60	80	90	50