Question 1.1: Write the Answer to these questions.

• What is the difference between static and dynamic variables in python?

Key Differences

Association: Static variables are associated with the class itself, while dynamic variables are associated with instances of the class.

Lifetime: Static variables persist for the lifetime of the class, while dynamic variables persist for the lifetime of the instance.

Access: Static variables are accessed using the class name, whereas dynamic variables are accessed using the instance reference.

Modification: Changing a static variable affects all instances of the class, while changing a dynamic variable only affects the particular instance.

```python
''' static variables Example'''
class MyClass:
    static_variable = 42  # This is a static variable

    def __init__(self, value):
        self.instance_variable = value  # This is an instance variable

    def show(self):
        print(f'Static variable: {MyClass.static_variable}')
        print(f'Instance variable: {self.instance_variable}')

obj1 = MyClass(1)
obj2 = MyClass(2)

obj1.show()
obj2.show()

MyClass.static_variable = 100  # Modifying the static variable

obj1.show()
obj2.show()
```

```
Static variable: 42
Instance variable: 1
Static variable: 42
Instance variable: 2
Static variable: 100
Instance variable: 1
Static variable: 100
Instance variable: 2
```

```python
''' Dynamic varible Example'''
class MyClass:
    def __init__(self, value):
        self.instance_variable = value  # This is a dynamic variable

    def show(self):
        print(f'Instance variable: {self.instance_variable}')

obj1 = MyClass(1)
obj2 = MyClass(2)

obj1.show()
obj2.show()

obj1.instance_variable = 10  # Modifying the instance variable of obj1

obj1.show()
obj2.show()
```

```
Instance variable: 1
Instance variable: 2
Instance variable: 10
Instance variable: 2
```

• Explain the purpose of 'pop','popitem','clear()' in a dictionary with suitable examples.

```python
'''pop()
Purpose: Removes the specified key and returns the corresponding
value. If the key is not found, it raises a KeyError unless a default
value is provided.
'''
my_dict = {'a': 1, 'b': 2, 'c': 3}
# Removing an item by key
value = my_dict.pop('b')
print(value)  # Output: 2
print(my_dict)  # Output: {'a': 1, 'c': 3}
# Removing a key that doesn't exist, with a default value
value = my_dict.pop('d', 'Not Found')
print(value)  # Output: Not Found

'''popitem()
Purpose: Removes and returns an arbitrary (key, value) pair from the
dictionary. This method is useful for destructively iterating over a
dictionary.
'''
my_dict = {'a': 1, 'b': 2, 'c': 3}
# Removing an arbitrary item
key, value = my_dict.popitem()
print((key, value))  # Output could be: ('c', 3)
print(my_dict)  # Output could be: {'a': 1, 'b': 2}
# Removing another arbitrary item
key, value = my_dict.popitem()
print((key, value))  # Output could be: ('b', 2)
print(my_dict)  # Output could be: {'a': 1

'''clear()
Purpose: Removes all items from the dictionary, leaving it empty.
'''
my_dict = {'a': 1, 'b': 2, 'c': 3}
# Clearing the dictionary
my_dict.clear()
print(my_dict)  # Output: {}
```

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

In Python, a frozenset is an immutable version of a set. This means that once a frozenset is created, its elements cannot be changed (i.e., no additions or removals are allowed). frozenset can be useful when you need a set that should not be modified, for instance, as keys in a dictionary or as elements in another set.

```
# Creating a frozenset from a list
fs = frozenset([1, 2, 3, 4])
print(fs)
# Creating a frozenset from a set
fs_from_set = frozenset({1, 2, 2, 3})
print(fs_from_set)
# Creating a frozenset from a set
fs_from_tuple = frozenset((4, 5, 6, 6))
print(fs_from_tuple)
my_dict = {fs: "Immutable set", fs_from_set: "Another frozenset"}
print(my_dict)
set_of_frozensets = {fs, fs_from_set}
print(set_of_frozensets)
```

```
frozenset({1, 2, 3, 4})
frozenset({1, 2, 3})
frozenset({4, 5, 6})
{frozenset({1, 2, 3, 4}): 'Immutable set', frozenset({1, 2, 3}):
'Another frozenset'}
{frozenset({1, 2, 3}), frozenset({1, 2, 3, 4})}
```

• Differentiate between mutable and immutable data types in python and give examples of mutable and immutable data types.

Mutable Data Types

Mutable data types are those whose state can be changed after they are created. This means that you can modify their contents without creating a new object.

Immutable Data Types

Immutable data types are those whose state cannot be changed after they are created. Any modification to an immutable object results in the creation of a new object.

```python
''' Example of Mutable Data Types:'''
# List example
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)
# Dictionary example
my_dict = {'a': 1, 'b': 2}
my_dict['c'] = 3
print(my_dict)
# Set example
my_set = {1, 2, 3}
my_set.add(4)
print(my_set)
```

```
[1, 2, 3, 4]
{'a': 1, 'b': 2, 'c': 3}
{1, 2, 3, 4}
```

```python
''' Example of Imutable Data Types'''
# String example
my_string = "hello"
new_string = my_string.upper()
print(my_string)
print(new_string)

# Tuple example
my_tuple = (1, 2, 3)
# Number example
a = 10
b = a + 5
print(a)
print(b)

# frozenset example
my_frozenset = frozenset([1, 2, 3])
```

• What is '**init**'?Explain with an example.

The **init** method in Python is a special method that is called when an instance (object) of a class is created. It is commonly known as the constructor method in Python. The primary purpose of **init** is to initialize the attributes of the object. It can be used to set the initial state of an object by assigning values to the instance variables.

```
''' __init__ method Example '''
class Person:
    def __init__(self, name, age):
        self.name = name  # Initialize the instance variable 'name'
        self.age = age    # Initialize the instance variable 'age'

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age}
years old.")
# Creating an instance of the Person class
person1 = Person("Alice", 30)
# Accessing instance variables
print(person1.name)
print(person1.age)

# Calling a method of the Person class
person1.greet()
```

```
Alice
30
Hello, my name is Alice and I am 30 years old.
```

• What is docstring in Python?Explain with an example.

Docstrings are a crucial part of writing readable and maintainable code, as they help others and your future self understand the purpose and usage of various parts of your code. Purpose: To provide a convenient way to associate documentation with Python code.

```
def add(a, b):
    """
    Adds two numbers and returns the result.

    Parameters:
    a (int or float): The first number to add.
    b (int or float): The second number to add.

    Returns:
    int or float: The sum of the two numbers.
    """
    return a + b

# Accessing the docstring
print(add.__doc__)
```

```
    Adds two numbers and returns the result.

    Parameters:
    a (int or float): The first number to add.
    b (int or float): The second number to add.

    Returns:
    int or float: The sum of the two numbers.
```

• What are unit tests in Python?

Unit tests in Python are a method of testing individual units or components of a program to ensure that they are functioning correctly. A unit is the smallest testable part of an application, such as a function, method, or class. Unit testing is crucial for verifying that each part of the software works as expected and helps in identifying and fixing bugs early in the development process.

```python
# math_operations.py

def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero!")
    return a / b
```

• What is break, continue and pass in python?

break: Exits the loop immediately.

continue: Skips the rest of the current loop iteration and moves to the next iteration.

pass: Does nothing and is used as a placeholder.

```python
# Examle of break
for i in range(10):
    if i == 5:
        break
    print(i)
```

```
0
1
2
3
4

#Example of continue
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)

1
3
5
7
9

# Example of pass
for i in range(10):
    if i % 2 == 0:
        pass
    else:
        print(i)

1
3
5
7
9
```

• What is the use of self in Python?

In Python, self is a conventional name for the first parameter of instance methods in a class. It is a reference to the current instance of the class and is used to access variables and methods associated with that instance. By using self, you can access attributes and methods of the class in object-oriented programming.

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

    def area(self):
        return 3.14 * self.radius ** 2

    def circumference(self):
        return 2 * 3.14 * self.radius

# Creating an instance of the Circle class
circle = Circle(5)
```

```
print(f"Area: {circle.area()}")
print(f"Circumference: {circle.circumference()}")

Area: 78.5
Circumference: 31.400000000000002
```

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

Global Variables: Defined outside of functions and classes, accessible throughout the module or program.

Protected Attributes: Conventionally indicated by a single leading underscore (_), implying internal use within a module or class hierarchy.

Private Attributes: Achieved by name mangling with double leading underscores (__), intended to avoid name clashes in subclasses and to signal internal use only.

```
# Global variable
global_var = 10
def func():
    print(global_var)

func()

10

# protected Attributes
class Base:
    def __init__(self):
        self._protected_var = 20

class Derived(Base):
    def __init__(self):
        super().__init__()
        print(self._protected_var)

obj = Derived()

20

# Private Attributes
class MyClass:
    def __init__(self):
        self.__private_var = 30

    def get_private_var(self):
        return self.__private_var

obj = MyClass()
# Accessing private variable indirectly through a method
print(obj.get_private_var())
```

```
30
```

• What are modules and packages in python?

Modules: Single files containing Python code. They help in organizing code by defining functions, classes, and variables.

Packages: Directories containing multiple modules and an **init**.py file. They help in organizing modules hierarchically and managing large codebases.

```
'''Creating a Module'''
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

PI = 3.14159

import mymodule

print(mymodule.greet("Alice"))  # Output: Hello, Alice!
print(mymodule.PI)              # Output: 3.14159


# mypackage/__init__.py

def package_greet(name):
    return f"Hello from the package, {name}!"

# main.py

import mypackage.module1
import mypackage.module2

print(mypackage.module1.some_function())
```

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

Lists

Definition: A list is an ordered collection of items that is mutable, meaning its elements can be changed after the list is created. Syntax: Lists are defined using square brackets []. Mutable: You can add, remove, or modify elements in a list. Dynamic: Lists can grow and shrink in size as needed. Usage: Suitable for collections of items that need to be modified.

Tuple

Definition: A tuple is an ordered collection of items that is immutable, meaning its elements cannot be changed after the tuple is created. Syntax: Tuples are defined using parentheses (). Immutable: Once a tuple is created, you cannot add, remove, or modify its elements. Static:

Tuples have a fixed size and content. Usage: Suitable for collections of items that should not be modified, such as fixed data sets or constants.

```
# List Example
fruits = ["apple", "banana", "cherry"]
print(fruits[0])
fruits[1] = "blueberry"
print(fruits)
fruits.append("date")
print(fruits)
fruits.remove("apple")
print(fruits)
```

```
apple
['apple', 'blueberry', 'cherry']
['apple', 'blueberry', 'cherry', 'date']
['blueberry', 'cherry', 'date']
```

```
# TUPLE EXAMPLE
fruits = ("apple", "banana", "cherry")
print(fruits[0])
new_fruits = fruits + ("date",)
print(new_fruits)

a, b, c = ("apple", "banana", "cherry")
print(a)
print(b)
print(c)
```

```
apple
('apple', 'banana', 'cherry', 'date')
apple
banana
cherry
```

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

Execution vs. Type Determination:

Interpreted: Code is executed directly line by line. Dynamically Typed: Variable types are determined at runtime.

Compilation:

Interpreted: No compilation needed. Dynamically Typed: No type declarations needed.

Type Checking:

Interpreted: Performed at runtime by the interpreter. Dynamically Typed: Performed at runtime, allowing variables to change type.

Error Detection:

Interpreted: Errors are found at runtime. Dynamically Typed: Type-related errors are found at runtime.

Performance:

Interpreted: Generally slower due to interpretation overhead. Dynamically Typed: Variable performance; dynamic typing adds flexibility but may impact speed.

• What are Dict and List comprehensions?

List and dictionary comprehensions are concise ways to create lists and dictionaries in Python. They offer a more readable and expressive way to generate these collections by embedding loops and conditional logic within a single line of code.

```python
''' Creating a list of squares of even numbers from 0 to 9'''
squares = [x**2 for x in range(10) if x % 2 == 0]
print(squares)
```

```
[0, 4, 16, 36, 64]
```

```python
'''Creating a dictionary with numbers as keys and their squares as
values for even numbers from 0 to 9:'''
squares_dict = {x: x**2 for x in range(10) if x % 2 == 0}
print(squares_dict)
```

```
{0: 0, 2: 4, 4: 16, 6: 36, 8: 64}
```

• What are decorators in Python? Explain it with an example.Write down its use cases.

Decorators in Python are a powerful and elegant way to modify the behavior of a function or a method. They allow you to wrap another function in order to extend its behavior without explicitly modifying it. Decorators are commonly used for logging, access control, memoization, and other cross-cutting concerns.

Use Cases for Decorators

Logging: Automatically log information about function calls. Authentication and Authorization: Check user permissions before allowing access to certain functions. Memoization: Cache results of expensive function calls to improve performance. Input Validation: Check and validate the inputs to a function. Timing: Measure the time a function takes to execute.

```python
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
```

• How is memory managed in Python?

Memory management in Python involves a combination of techniques to efficiently allocate, use, and reclaim memory during the execution of a program. Here's an overview of how memory is managed in Python:

• What is lambda in python? Why is it used?

In Python, a lambda function is a small anonymous function defined using the lambda keyword. Lambda functions can have any number of arguments but only one expression. They are syntactically restricted to a single expression.

Concise Function Definitions: Lambda functions are a way to define small, throwaway functions without formally defining a def function. This can make the code more concise.

Higher-Order Functions: Lambda functions are often used in conjunction with higher-order functions, such as map(), filter(), and reduce(), where a simple function is required for a short period.

Inline Functionality: They are useful for short-lived operations that are not reused elsewhere in the code.

```python
numbers = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x ** 2, numbers))
print(squared)
```

```
[1, 4, 9, 16, 25]
```

• Explain split() and join() functions in Python?

In Python, the split() and join() functions are used for string manipulation, specifically for breaking down and combining strings, respectively

```python
# Replacing spaces with hyphens in a string
text = "This is a test"
words = text.split()
hyphenated = '-'.join(words)
print(hyphenated)
```

```
This-is-a-test
```

• What are iterators , iterable & generators in Python?

Iterables

An iterable is any Python object capable of returning its members one at a time. Common examples include lists, tuples, dictionaries, sets, and strings. An object is considered iterable if it implements the **iter**() method, which returns an iterator.

Iterators

An iterator is an object that represents a stream of data. It is the object returned by calling the **iter**() method on an iterable. An iterator implements the **next**() method, which returns the next item from the stream and raises the StopIteration exception when there are no more items.

Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the yield statement to return data one piece at a time, suspending the function's state between yields. Generators are more memory efficient than lists, especially for large datasets, because they generate items on the fly and do not store the entire dataset in memory.

```
#Iterables
# Lists
my_list = [1, 2, 3]

# Strings
my_string = "Hello"

# Both can be iterated over using a for loop
for item in my_list:
    print(item)

for char in my_string:
    print(char)
```

```
1
2
3
H
e
l
l
o
```

```
# Iterator
my_list = [1, 2, 3]
my_iterator = iter(my_list)

print(next(my_iterator))
print(next(my_iterator))
print(next(my_iterator))
```

```
1
2
3
```

```
# Generator
def my_generator():
    yield 1
    yield 2
    yield 3

gen = my_generator()

print(next(gen))
print(next(gen))
print(next(gen))

1
2
3
```

• What is the difference between xrange and range in python?

```
for i in xrange(1, 10):
    print(i)

----------------------------------------------------------------------
-----
NameError                                 Traceback (most recent call
last)
Cell In[6], line 1
----> 1 for i in xrange(1, 10):
      2     print(i)

NameError: name 'xrange' is not defined
```

• Pillars of Oops.

the four pillars of OOP are implemented and used in Python:

Encapsulation:

Definition: Encapsulation in Python is achieved by using classes to bundle data (attributes) and methods (functions) that operate on that data into a single unit. Python provides access control mechanisms through the use of public, protected, and private attributes.

Abstraction:

Definition: Abstraction in Python is implemented using abstract classes and interfaces, allowing the hiding of complex implementation details and exposing only the necessary parts.

Inheritance:

Definition: Inheritance in Python allows a class (child class) to inherit attributes and methods from another class (parent class), promoting code reuse and the creation of hierarchical relationships.

```python
# Encapsulation:
class Car:
    def __init__(self, speed, fuel):
        self.__speed = speed   # Private attribute
        self.__fuel = fuel     # Private attribute

    def get_speed(self):
        return self.__speed

    def refuel(self, amount):
        self.__fuel += amount

# Abstraction
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Square(Shape):
    def draw(self):
        print("Drawing a square")

# Inheritance
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start(self):
        print(f"{self.make} {self.model} is starting.")

class Car(Vehicle):
    def __init__(self, make, model, num_doors):
        super().__init__(make, model)
        self.num_doors = num_doors

    def open_doors(self):
        print(f"Opening {self.num_doors} doors.")
```

• How will you check if a class is a child Of another class?

if a class is a subclass of another class using the built-in issubclass() function. This function returns True if the first class is a subclass (directly or indirectly) of the second class, and False otherwise.

```
class Parent:
    pass

class Child(Parent):
    pass

class GrandChild(Child):
    pass

# Check if Child is a subclass of Parent
print(issubclass(Child, Parent))

# Check if GrandChild is a subclass of Parent
print(issubclass(GrandChild, Parent))

# Check if Parent is a subclass of Child
print(issubclass(Parent, Child))
```

```
True
True
False
```

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

Inheritance in Python allows a class to inherit attributes and methods from another class. This promotes code reuse and establishes a natural hierarchy between classes. There are several types of inheritance in Python:

Single Inheritance

Multiple Inheritance

Multilevel Inheritance

Hierarchical Inheritance

Hybrid Inheritance

```
# Single Inheritance
class Parent:
    def parent_method(self):
        print("This is a method in the Parent class.")

class Child(Parent):
    def child_method(self):
        print("This is a method in the Child class.")

# Usage
child = Child()
```

```python
child.parent_method()
child.child_method()
```

```
This is a method in the Parent class.
This is a method in the Child class.
```

```python
# Multiple Inheritance
class Parent1:
    def parent1_method(self):
        print("This is a method in Parent1 class.")

class Parent2:
    def parent2_method(self):
        print("This is a method in Parent2 class.")

class Child(Parent1, Parent2):
    def child_method(self):
        print("This is a method in the Child class.")

# Usage
child = Child()
child.parent1_method()
child.parent2_method()
child.child_method()
```

```
This is a method in Parent1 class.
This is a method in Parent2 class.
This is a method in the Child class.
```

```python
# Multilevel Inheritance
class Grandparent:
    def grandparent_method(self):
        print("This is a method in the Grandparent class.")

class Parent(Grandparent):
    def parent_method(self):
        print("This is a method in the Parent class.")

class Child(Parent):
    def child_method(self):
        print("This is a method in the Child class.")

# Usage
child = Child()
child.grandparent_method()
child.parent_method()
child.child_method()
```

```
This is a method in the Grandparent class.
This is a method in the Parent class.
This is a method in the Child class.
```

```python
# Hierarchical Inheritance
class Parent:
    def parent_method(self):
        print("This is a method in the Parent class.")

class Child1(Parent):
    def child1_method(self):
        print("This is a method in the Child1 class.")

class Child2(Parent):
    def child2_method(self):
        print("This is a method in the Child2 class.")

# Usage
child1 = Child1()
child1.parent_method()
child1.child1_method()

child2 = Child2()
child2.parent_method()
child2.child2_method()
```

```
This is a method in the Parent class.
This is a method in the Child1 class.
This is a method in the Parent class.
This is a method in the Child2 class.
```

```python
# Hybrid Inheritance
class Parent:
    def parent_method(self):
        print("This is a method in the Parent class.")

class Child1(Parent):
    def child1_method(self):
        print("This is a method in the Child1 class.")

class Child2(Parent):
    def child2_method(self):
        print("This is a method in the Child2 class.")

class Grandchild(Child1, Child2):
    def grandchild_method(self):
        print("This is a method in the Grandchild class.")

# Usage
grandchild = Grandchild()
grandchild.parent_method()
grandchild.child1_method()
grandchild.child2_method()
grandchild.grandchild_method()
```

```
This is a method in the Parent class.
This is a method in the Child1 class.
This is a method in the Child2 class.
This is a method in the Grandchild class.
```

• What is encapsulation? Explain it with an example.

Encapsulation is one of the fundamental principles of Object-Oriented Programming (OOP). It involves bundling the data (attributes) and the methods (functions) that operate on the data into a single unit or class, and restricting direct access to some of an object's components. This is typically done to protect the integrity of the data and to prevent unauthorized or unintended interference.

```python
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        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 0 < amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew {amount}. New balance is 
{self.__balance}.")
        else:
            print("Invalid withdraw amount.")

    def get_balance(self):
        return self.__balance

# Usage
account = BankAccount("Alice", 1000)

# Accessing public methods
account.deposit(500)
account.withdraw(200)
print(f"Balance: {account.get_balance()}")

print(account._BankAccount__balance)
```

```
Deposited 500. New balance is 1500.
Withdrew 200. New balance is 1300.
```

```
Balance: 1300
1300
```

• What is polymorphism? Explain it with an example

Polymorphism is one of the core concepts in Object-Oriented Programming (OOP). It refers to the ability of different objects to be treated as instances of the same class through a common interface. Polymorphism allows methods to be used interchangeably between objects of different types, enabling a single interface to handle a variety of underlying forms (data types).

```
class Shape:
    def draw(self):
        raise NotImplementedError("Subclass must implement abstract
method")

class Circle(Shape):
    def draw(self):
        print("Drawing a circle")

class Square(Shape):
    def draw(self):
        print("Drawing a square")

class Triangle(Shape):
    def draw(self):
        print("Drawing a triangle")


def draw_shape(shape):
    shape.draw()

# Usage
shapes = [Circle(), Square(), Triangle()]

for shape in shapes:
    draw_shape(shape)

Drawing a circle
Drawing a square
Drawing a triangle
```

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

Anawer

Serial_no.: Invalid

Reason: An identifier cannot end with a period (.). 1st_Room: Invalid

Reason: Identifiers cannot start with a digit. They must begin with a letter (a-z, A-Z) or an underscore (_).

Hundred$: Invalid

Reason: Identifiers can only contain letters, digits, and underscores (_). The dollar sign ($) is not allowed.

Total_Marks: Valid

Reason: This identifier follows all the rules. It starts with a letter and contains only letters and underscores.

total-Marks: Invalid

Reason: Identifiers cannot contain hyphens (-). Only letters, digits, and underscores are allowed.

Total Marks: Invalid

Reason: Identifiers cannot contain spaces. They must be continuous strings of letters, digits, and underscores.

True: Invalid

Reason: True is a reserved keyword in Python, used to represent the boolean value True. Keywords cannot be used as identifiers.

_percentag: Valid

Reason: This identifier starts with an underscore and contains only letters. It follows all the naming rules.

11. Define a python module named constants.py containing constants like pi and the speed Of light.

```python
import constants

print(constants.PI)
print(constants.SPEED_OF_LIGHT)
```

```
3.14
299792458
```

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

```
import calculater as cal
cal.subtract(22,33)

-11
```

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

```
# ecommerce/main.py

from product_management_project.product_management import Product,
add_product, remove_product
from product_management_project.order_processing import Order,
process_order

def main():
    # Product management
    products = []

    product1 = Product(1, "Laptop", 1000.0, 10)
    product2 = Product(2, "Smartphone", 500.0, 20)

    add_product(products, product1)
    add_product(products, product2)

    print("Products after addition:")
    for product in products:
        print(product)

    remove_product(products, 1)

    print("Products after removal of product with id 1:")
    for product in products:
        print(product)

    # Order processing
    order = Order(1, "John Doe", [product2])
    print("Order before processing:")
    print(order)

    process_order(order)
    print("Order after processing:")
    print(order)

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

```
Products after addition:
Product(id=1, name=Laptop, price=1000.0, stock=10)
Product(id=2, name=Smartphone, price=500.0, stock=20)
Products after removal of product with id 1:
Product(id=2, name=Smartphone, price=500.0, stock=20)
Order before processing:
Order(id=1, customer=John Doe, products=[Product(id=2,
name=Smartphone, price=500.0, stock=20)], status=Pending)
Order after processing:
Order(id=1, customer=John Doe, products=[Product(id=2,
name=Smartphone, price=500.0, stock=20)], status=Processed)
```

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

```
s="implement a python MODULE named string_utils.py containing
functions for string manipulation, such as reversing and capitalizing
strings."
import string_utils as su
su.capitalize_string(s)
su.reverse_string(s)
```

```
'.sgnirts gnizilatipac dna gnisrever sa hcus ,noitalupinam gnirts rof
snoitcnuf gniniatnoc yp.slitu_gnirts deman ELUDOM nohtyp a tnemelpmi'
```

```
su.capitalize_string(s)
```

```
'Implement a python module named string_utils.py containing functions
for string manipulation, such as reversing and capitalizing strings.'
```

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

```
import file_operations as fo
fo.read_file(f"C:\\Users\\anand\\OneDrive\\Desktop\\FILE\\pwassiment\\
file.txt" )
```

```
'THIS IS SAURAV HERE\nI AM FROM JHARKHAND RANCHI\nTHERE ARE FOUR
MAMBERS IN MY FAMAILY\nI AM A DATA SCIENTIST'
```

```
fo.write_file(r"C:\Users\anand\OneDrive\Desktop\FILE\pwassiment\
file.txt","this is again saurav")
```

```
'Data successfully written to C:\\Users\\anand\\OneDrive\\Desktop\\
FILE\\pwassiment\\file.txt'

fo.append_to_file(r"C:\Users\anand\OneDrive\Desktop\FILE\pwassiment\
file.txt","this is saurav here last time")

'Data successfully appended to C:\\Users\\anand\\OneDrive\\Desktop\\
FILE\\pwassiment\\file.txt'
```

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.

```
# write_employees.py

def write_employees_to_file(file_path, employees):
    """Write the details of employees to a text file."""
    try:
        with open(file_path, 'w') as file:
            for employee in employees:
                name, age, salary = employee
                file.write(f"Name: {name}, Age: {age}, Salary:
{salary}\n")
        print(f"Employee details successfully written to {file_path}")
    except Exception as e:
        print(f"Error: {e}")

def main():
    employees = [
        ("John Doe", 28, 50000),
        ("Jane Smith", 34, 65000),
        ("Emily Johnson", 45, 75000),
        ("Michael Brown", 50, 80000)
    ]

    file_path = "employees.txt"
    write_employees_to_file(file_path, employees)

if __name__ == "__main__":
    main()

Employee details successfully 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.

```
# read_inventory.py
```

```python
def read_inventory(file_path):
    """Read and display the contents of the inventory file line by
line."""
    try:
        with open(file_path, 'r') as file:
            for line in file:
                print(line.strip())
    except FileNotFoundError:
        print(f"Error: The file at {file_path} was not found.")
    except Exception as e:
        print(f"Error: {e}")

def main():
    file_path = "inventory.txt"
    read_inventory(file_path)

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

```
Item1: 10
Item2: 20
Item3: 15
Item4: 5
```

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.

```python
# calculate_expenses.py

def read_expenses(file_path):
    """Read the expenses from the file and return a list of tuples
(description, amount)."""
    expenses = []
    try:
        with open(file_path, 'r') as file:
            for line in file:
                try:
                    description, amount = line.strip().split(':')
                    amount = float(amount)
                    expenses.append((description, amount))
                except ValueError:
                    print(f"Skipping invalid line: {line.strip()}")
    except FileNotFoundError:
        print(f"Error: The file at {file_path} was not found.")
    except Exception as e:
        print(f"Error: {e}")
    return expenses
```

```
def calculate_total_expenses(expenses):
    """Calculate the total amount spent based on the expenses list."""
    total = sum(amount for description, amount in expenses)
    return total

def main():
    file_path = "expenses.txt"
    expenses = read_expenses(file_path)
    total = calculate_total_expenses(expenses)
    print(f"Total amount spent: ${total:.2f}")

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

```
Total amount spent: $1531.55
```

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.

```
# count_words.py

import string

def read_paragraph(file_path):
    """Read the content of the file and return it as a string."""
    try:
        with open(file_path, 'r') as file:
            return file.read()
    except FileNotFoundError:
        print(f"Error: The file at {file_path} was not found.")
    except Exception as e:
        print(f"Error: {e}")
    return ""

def count_words(paragraph):
    """Count the occurrences of each word in the paragraph and return
a dictionary."""
    word_counts = {}
    # Remove punctuation and convert to lower case
    translator = str.maketrans('', '', string.punctuation)
    paragraph = paragraph.translate(translator).lower()
    words = paragraph.split()

    for word in words:
        if word in word_counts:
            word_counts[word] += 1
        else:
            word_counts[word] = 1
    return word_counts

def main():
    file_path = "paragraph.txt"
    paragraph = read_paragraph(file_path)
    word_counts = count_words(paragraph)

    # Display results in alphabetical order
    for word in sorted(word_counts):
        print(f"{word}: {word_counts[word]}")

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

```
across: 1
algorithms: 1
an: 1
analyze: 1
and: 8
are: 1
as: 1
by: 1
can: 1
combines: 1
complex: 1
computer: 1
data: 6
decisions: 1
domain: 1
drive: 1
elements: 1
expertise: 1
extract: 1
field: 1
finance: 1
from: 3
gained: 1
healthcare: 1
improve: 1
industries: 1
informed: 1
innovation: 1
insights: 2
interdisciplinary: 1
interpret: 1
is: 1
it: 1
knowledge: 1
learning: 1
```

```
leveraging: 1
machine: 1
make: 1
methods: 1
mining: 1
modeling: 1
of: 1
outcomes: 1
patterns: 1
predictive: 1
science: 3
scientific: 1
scientists: 1
sets: 1
statistics: 1
structured: 1
such: 1
systems: 1
techniques: 1
that: 1
the: 1
to: 4
uncover: 1
unstructured: 1
used: 1
uses: 1
various: 1
```

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

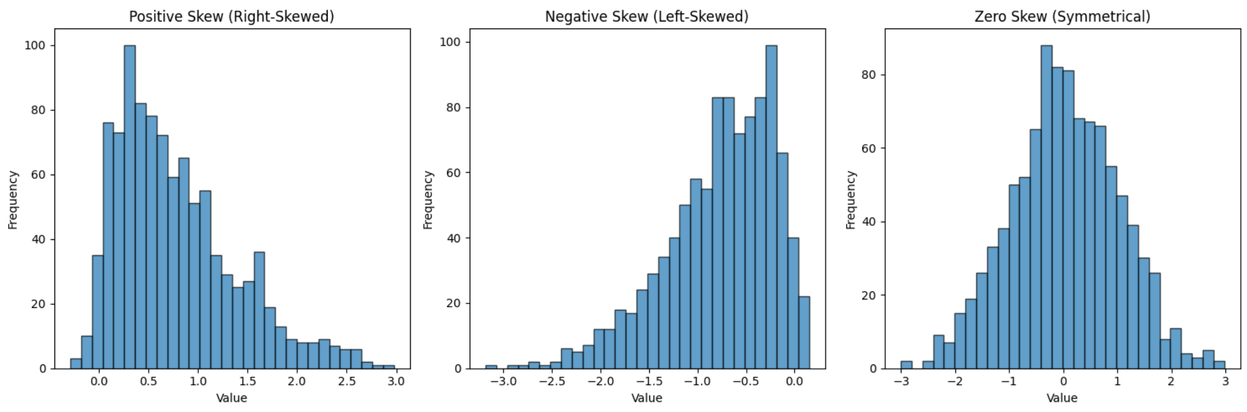
```python
import numpy as np
from scipy import stats

data = [10, 20, 20, 30, 40, 50, 60, 70, 80, 90]

# Measures of Central Tendency
mean = np.mean(data)
median = np.median(data)
mode = stats.mode(data)[0][0]

# Measures of Dispersion
range_ = np.ptp(data)  # Range
variance = np.var(data, ddof=1)  # Sample Variance
std_dev = np.std(data, ddof=1)  # Sample Standard Deviation
q1 = np.percentile(data, 25)  # First Quartile (Q1)
q3 = np.percentile(data, 75)  # Third Quartile (Q3)
iqr = q3 - q1  # Interquartile Range (IQR)
```

```
print(f"Mean: {mean}")
print(f"Median: {median}")
print(f"Mode: {mode}")
print(f"Range: {range_}")
print(f"Variance: {variance}")
print(f"Standard Deviation: {std_dev}")
print(f"Interquartile Range (IQR): {iqr}")

----------------------------------------------------------------------
-----
IndexError                                Traceback (most recent call
last)
Cell In[4], line 9
      7 mean = np.mean(data)
      8 median = np.median(data)
----> 9 mode = stats.mode(data)[0][0]
     11 # Measures of Dispersion
     12 range_ = np.ptp(data)  # Range

IndexError: invalid index to scalar variable.
```

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

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.stats import skewnorm

# Generate data
np.random.seed(0)

# Positive Skew
data_pos_skew = skewnorm.rvs(a=10, size=1000)  # 'a' parameter > 0

# Negative Skew
data_neg_skew = skewnorm.rvs(a=-10, size=1000)  # 'a' parameter < 0

# Zero Skew (Normal Distribution)
data_zero_skew = np.random.normal(loc=0, scale=1, size=1000)

# Plotting
fig, axes = plt.subplots(1, 3, figsize=(15, 5))

# Positive Skew
axes[0].hist(data_pos_skew, bins=30, edgecolor='k', alpha=0.7)
axes[0].set_title('Positive Skew (Right-Skewed)')
axes[0].set_xlabel('Value')
axes[0].set_ylabel('Frequency')

# Negative Skew
```

```
axes[1].hist(data_neg_skew, bins=30, edgecolor='k', alpha=0.7)
axes[1].set_title('Negative Skew (Left-Skewed)')
axes[1].set_xlabel('Value')
axes[1].set_ylabel('Frequency')

# Zero Skew
axes[2].hist(data_zero_skew, bins=30, edgecolor='k', alpha=0.7)
axes[2].set_title('Zero Skew (Symmetrical)')
axes[2].set_xlabel('Value')
axes[2].set_ylabel('Frequency')

plt.tight_layout()
plt.show()
```



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

39.How would you create a basic Flask route that displays "Hello, World!" on the homepage?

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == '__main__':
    app.run(debug=True)
```

40.Explain how to set up a Flask application to handle form submissions using POST requests.