CMR
INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | | |

Internal Assessment Test I – January 2024

Sub:	Data Analytics using Python						Sub Code:	22MCA31
Date:	17/01/2024	Duration:	90 mins	Max Marks:	50	Sem: III	Branch:	MCA

Note : Answer FIVE FULL Questions, choosing ONE full question from each Module

	PART I	MARKS	OBE	
			CO	RBT
1	Write the features of Python. Give the advantages & disadvantages of it. **OR**	6+2+2	CO1	L1
2	Explain the operators of python in detail with an example program.	10	CO2	L2
	PART II			
3	Explain functions and its types in python with an example program **OR**	4+6	CO2	L3
4	Develop a python program to calculate the area of square, rectangle, and circle using function	[10]	CO3	L3
	PART III			
5	Discuss the precedence and associativity of the operators in python with a program example **OR**	5+5	CO2	L3
6	Illustrate args and kwargs parameters in python with example programs	5+5	CO3	L3
	PART IV			
7	What is a list and how to create lists in python. Explain five list methods with a brief description and example **OR**	5+5	CO2	L3
8	Discuss the following tasks on tuple with syntax and example i) append ii) Add two tuples iii) remove item from tuple iv) multiply tuples by 3 v) Count method in tuple	2+2+2+2+2	CO2	L2
	PART V			
9	Write a python program to demonstrate Constructors, inheritance and operator overloading. **OR**	10	CO3	L4
10	Write Python program to count the words and store in a dictionary from a file.	10	CO3	L4

1. <u>Features of python:</u>

1. Easy Language

Python is an easy language. It is easy to read, write, learn and understand.

• Python has a smooth learning curve. It is easy to learn.

• Python has a simple syntax and Python code is easy to understand.

• Since it's easy to understand, you can easily read and understand someone else's code.

• Python is also easy to write because of its simple syntax.

Because it is an easy language, it is used in schools and universities to introduce students to programming. Python is for both startups and big companies.

2. Readable

The Python language is designed to make developers life easy. Reading a Python code is like reading an English sentence. This is one of the key reason that makes Python best for beginners.

Python uses indentation instead of curly braces, unlike other programming languages. This makes the code look clean and easier to understand.

3. Interpreted Language

Python is an interpreted language. It comes with the IDLE (Interactive Development Environment). This is an interpreter and follows the REPL structure (Read-Evaluate Print-Loop). It executes and displays the output of one line at a time.

So it displays errors while you're running a line and displays the entire stack trace for the error.

4. Dynamically-Typed Language

Python is not statically-typed like Java. You don't need to declare data type while defining a variable. The interpreter determines this at runtime based on the types of the parts of the expression. This is easy for programmers but can create runtime errors.

Python follows duck-typing. It means, "If it looks like a duck, swims like a duck and quacks like a duck, it must be a duck."

5. Object-Oriented

Python is object-oriented but supports both functional and object-oriented programming. Everything in Python is an object.

It has the OOP (Object-oriented programming) concepts like inheritance and polymorphism.

issubclass(str,object)

6. Popular and Large Community Support

Python has one of the largest communities on StackOverflow and Meetup. If you need help, the community will answer your questions.

They also already have many answered questions about Python.

7. Open-Source

Python is open-source and the community is always contributing to it to improve it. It is free and its source code is freely available to the public. You can download Python from the official Python Website.

8. Large Standard Library

The standard library is large and has many packages and modules with common and important functionality. If you need something that is available in this standard library, you don't need to write it from scratch. Because of this, you can focus on more important things.

You can also install packages from the PyPI (Python Package Index) if you want even more functionality.

9. Platform-Independent

Python is platform-independent. If you write a program, it will run on different platforms like Windows, Mac and Linux. You don't need to write them separately for each platform.

10. Extensible and Embeddable
Python is extensible. You can use code from other languages like C++ in your Python code.
It is also embeddable. You can embed your Python code in other languages like C++.
11. GUI Support
You can use Python to create GUI (Graphical User Interfaces). You can use tkinter, PyQt, wxPython or Pyside for this.
Python features a huge number of GUI frameworks available for it to variety of other cross-platform solutions. It binds to platform-specific technologies.
12. High-level Language
Python is a high-level language and C++ is mid-level. It is easy to understand and closer to the user. You don't need to remember system architecture or manage the memory.

Advantages of python:
1. Easy to Read, Learn and Write

Python is a high-level programming language that has English-like syntax. This makes it easier to read and understand the code.

Python is really easy to pick up and learn, that is why a lot of people recommend Python to beginners. You need less lines of code to perform the same task as compared to other major languages like C/C++ and Java.

2. Improved Productivity

Python is a very productive language. Due to the simplicity of Python, developers can focus on solving the problem. They don't need to spend too much time in understanding the syntax or behavior of the programming language. You write less code and get more things done.

3. Interpreted Language

Python is an interpreted language which means that Python directly executes the code line by line. In case of any error, it stops further execution and reports back the error which has occurred.

Python shows only one error even if the program has multiple errors. This makes debugging easier.

4. Dynamically Typed

Python doesn't know the type of variable until we run the code. It automatically assigns the data type during execution. The programmer doesn't need to worry about declaring variables and their data types.

5. Free and Open-Source

Python comes under the OSI approved open-source license. This makes it free to use and distribute. You can download the source code, modify it and even distribute your version of Python. This is useful for organizations that want to modify some specific behavior and use their version for development.

Disadvantages of Python:
1. Slow Speed

We discussed above that Python is an interpreted language and dynamically-typed language. The line by line execution of code often leads to slow execution.

The dynamic nature of Python is also responsible for the slow speed of Python because it has to do the extra work while executing code. So, Python is not used for purposes where speed is an important aspect of the project.

2. Not Memory Efficient

To provide simplicity to the developer, Python has to do a little tradeoff. The Python programming language uses a large amount of memory. This can be a disadvantage while building applications when we prefer memory optimization.

3. Weak in Mobile Computing

Python is generally used in server-side programming. We don't get to see Python on the client-side or mobile applications because of the following reasons. Python is not memory efficient and it has slow processing power as compared to other languages.

4. Database Access

Programming in Python is easy and stress-free. But when we are interacting with the database, it lacks behind.

The Python's database access layer is primitive and underdeveloped in comparison to the popular technologies like JDBC and ODBC.

5. Runtime Errors

As we know Python is a dynamically typed language so the data type of a variable can change anytime. A variable containing integer number may hold a string in the future, which can lead to Runtime Errors.

2. <u>Operators in python:</u>

Arithmetic Operators
Arithmetic operators are used to performing mathematical operations like
addition, subtraction, multiplication, and division.
Operator Description
Syntax
+
Addition: adds two operands
x + y
–
Subtraction: subtracts two operands
x – y
*
Multiplication: multiplies two operands
x * y
/
Division (float): divides the first operand by the second
x / y
//
Division (floor): divides the first operand by the second
x // y

Operator Description
Syntax
%
Modulus: returns the remainder when the first operand is divided
by the second
x % y
**
Power: Returns first raised to power second
x ** y
Example: Arithmetic operators in Python
Examples of Arithmetic Operator

```
a = 9
b = 4
# Addition of numbers
add = a + b
# Subtraction of numbers
sub = a - b
# Multiplication of number
mul = a * b
# Division(float) of number
div1 = a / b
# Division(floor) of number
div2 = a // b
# Modulo of both number
mod = a % b
# Power
p = a ** b
# print results
print(add)
print(sub)
print(mul)
print(div1)
print(div2)
print(mod)
print(p)
```

Output

```
5
36
2.25
2
1
6561
```

Note: Refer to Differences between / and // for some interesting facts about these two operators.

Comparison Operators

Comparison of Relational operators compares the values. It either returns True or False according to the condition.

Operator	Description	Syntax
>	Greater than: True if the left operand is greater than the right	x > y
<	Less than: True if the left operand is less than the right	x < y
==	Equal to: True if both operands are equal	x == y
!=	Not equal to – True if operands are not equal	x != y
>=	Greater than or equal to True if the left operand is greater than or equal to the right	x >= y
<=		

Less than or equal to True if the left operand is less than or equal to the right
x <= y

Example: Comparison Operators in Python

```
# Examples of Relational Operators
a = 13
b = 33
# a > b is False
print(a > b)
# a < b is True
print(a < b)
# a == b is False
print(a == b)
# a != b is True
print(a != b)
# a >= b is False
print(a >= b)
# a <= b is True
print(a <= b)
```

Output

```
False
True
False
True
False
True
```

Logical Operators

Logical operators perform Logical AND, Logical OR, and Logical NOT operations. It is used to combine conditional statements.

Operator	Description	Syntax
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if the operand is false	not x

Example: Logical Operators in Python

```
# Examples of Logical Operator
a = True
b = False
# Print a and b is False
print(a and b)
# Print a or b is True
print(a or b)
# Print not a is False
print(not a)
```

Output

```
False
True
False
```

Bitwise Operators

Bitwise operators act on bits and perform the bit-by-bit operations. These are

used to operate on binary numbers.
Operator Description
Syntax
&
Bitwise AND
x & y
|
Bitwise OR
x | y
~
Bitwise NOT
~x
^
Bitwise XOR
x ^ y
>>
Bitwise right shift x>>
<<
Bitwise left shift
x<<
Example: Bitwise Operators in Python

```
# Examples of Bitwise operators
a = 10
b = 4
# Print bitwise AND operation
print(a & b)
# Print bitwise OR operation
print(a | b)
# Print bitwise NOT operation
print(~a)
# print bitwise XOR operation
print(a ^ b)
# print bitwise right shift operation
print(a >> 2)
# print bitwise left shift operation
print(a << 2)
```

Output

```
0
14
-11
14
2
40
```

Assignment Operators
Assignment operators are used to assigning values to the variables.
Operator Description
Syntax
=
Assign value of right side of expression to left side operand
x = y + z
+=
Add AND: Add right-side operand with left side operand and
then assign to left operand
a+=b a=a+b
-=

Subtract AND: Subtract right operand from left operand and then assign to left operand
a-=b a=a-b
*=
Multiply AND: Multiply right operand with left operand and then assign to left operand
a*=b a=a*b
/=
Divide AND: Divide left operand with right operand and then assign to left operand
a/=b a=a/b
Operator Description
Syntax
%=
Modulus AND: Takes modulus using left and right operands and assign the result to left operand
a%=b
a=a%b
//=
Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand
a//=b
a=a//b
**=
Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand
a**=b
a=a**b
&=
Performs Bitwise AND on operands and assign value to left operand
a&=b
a=a&b
|=
Performs Bitwise OR on operands and assign value to left operand
a|=b a=a|b
^=
Performs Bitwise xOR on operands and assign value to left operand
a^=b a=a^b
>>=
Performs Bitwise right shift on operands and assign value to left operand
a>>=b
a=a>>b
<<=
Performs Bitwise left shift on operands and assign value to left operand
a <<= b a=
a << b
Example: Assignment Operators in Python
# Examples of Assignment Operators
a = 10
# Assign value

```
b = a
print(b)
# Add and assign value
b += a
print(b)
# Subtract and assign value
b -= a
print(b)
# multiply and assign
b *= a
print(b)
# bitwise lishift operator
b <<= a
print(b)
```

Output

```
10
20
10
100
102400
```

3. Functions:

In Python, a function is a block of organized, reusable code that performs a specific task or set of tasks. Functions provide a way to structure code and make it modular, promoting code reuse, readability, and maintainability. Functions in Python are defined using the `def` keyword, followed by the function name, parameters (if any), a colon, and the function body.

Syntax of a Function:

```
def function_name(parameters):
    """docstring"""
    # function body
    return result  # optional
```

Example:

```
def factorial(n):
    if n == 0 or n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Types of Functions in Python:

Built-in function
User defined function
Built-in Functions
Python comes with many *built-in functions* that perform common operations.
One example is abs, which produces the absolute value of a
number:

```
>>> abs(-9)
9
```

4. We'll see later how to create functions that take any number of arguments.
STYLE NOTES 34
Another is round, which rounds a floating-point number to the nearest
integer:

```
>>> round(3.8)
4.0
```

```
>>> round(3.3)
3.0
>>> round(3.5)
4.0
```
Just like user-defined functions, Python's built-in functions can take more than one argument. For example, we can calculate 24 using the power function pow:
```
>>> pow(2, 4)
16
```
Some of the most useful built-in functions are ones that convert from one type to another. The type names int and float can be used as if they were functions:
```
>>> int(34.6)
34
>>> float(21)
21.0
```
In this example, we see that when a floating-point number is converted to an integer and truncated, not rounded.

User-defined functions:

User-defined functions in Python are functions created by the user to perform specific tasks or operations. They provide a way to structure code, promote reusability, and enhance code readability. Here's the basic syntax for defining a user-defined function in Python:

```
def function_name(parameters):
    """docstring"""
    # function body
    return result  # optional
```

Here's an example of a simple user-defined function:

```
def greet(name):
    """This function greets the person passed in as a parameter."""
    return f"Hello, {name}!"

# Calling the function
result = greet("Alice")
print(result)
```

4. <u>Program:</u>

```
import math

def calculate_circle_area(radius):
    return math.pi * radius**2

def calculate_rectangle_area(length, width):
    return length * width

def calculate_triangle_area(base, height):
    return 0.5 * base * height

# Get user input for shapes
shape = input("Enter the shape (circle, rectangle, or triangle): ").lower()

# Calculate and print the area based on the selected shape
```

```
if shape == "circle":
    radius = float(input("Enter the radius of the circle: "))
    area = calculate_circle_area(radius)
    print(f"The area of the circle is: {area}")
elif shape == "rectangle":
    length = float(input("Enter the length of the rectangle: "))
    width = float(input("Enter the width of the rectangle: "))
    area = calculate_rectangle_area(length, width)
    print(f"The area of the rectangle is: {area}")
elif shape == "triangle":
    base = float(input("Enter the base of the triangle: "))
    height = float(input("Enter the height of the triangle: "))
    area = calculate_triangle_area(base, height)
    print(f"The area of the triangle is: {area}")
else:
    print("Invalid shape. Please enter 'circle', 'rectangle', or 'triangle'.")
```

Output:

```
Enter the shape (circle, rectangle, or triangle): rectangle
Enter the length of the rectangle: 4
Enter the width of the rectangle: 2
The area of the rectangle is: 8.0
```

5.

Operator Precedence

This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

Example: Operator Precedence

```
# Examples of Operator Precedence
# Precedence of '+' & '*'
expr = 10 + 20 * 30
print(expr)
# Precedence of 'or' & 'and'
name = "Alex"
age = 0
if name == "Alex" or name == "John" and age >= 2:
print("Hello! Welcome.")
else:
print("Good Bye!!")
```

Output

```
610
Hello! Welcome.
```

Operator Associativity

If an expression contains two or more operators with the same precedence then Operator Associativity is used to determine. It can either be Left to Right or from Right to Left.

Example: Operator Associativity

```
# Examples of Operator Associativity
# Left-right associativity
# 100 / 10 * 10 is calculated as
# (100 / 10) * 10 and not
# as 100 / (10 * 10)
print(100 / 10 * 10)
# Left-right associativity
```

```
# 5 - 2 + 3 is calculated as
# (5 - 2) + 3 and not
# as 5 - (2 + 3)
print(5 - 2 + 3)
# left-right associativity
print(5 - (2 + 3))
# right-left associativity
# 2 ** 3 ** 2 is calculated as
# 2 ** (3 ** 2) and not
# as (2 ** 3) ** 2
print(2 ** 3 ** 2)
```

Output

```
100.0
6
0
512
```

6. *args and **kwargs

*args and **kwargs

I have come to see that most new python programmers have a hard time figuring out the *args and **kwargs magic variables. So what are they ? First of all, let me tell you that it is not necessary to write *args or **kwargs. Only the * (asterisk) is necessary. You could have also written *var and **vars. Writing *args and **kwargs is just a convention. So now let's take a look at *args first.

1.1. Usage of *args

*args and **kwargs are mostly used in function definitions. *args and **kwargs allow you to pass an unspecified number of arguments to a function, so when writing the function definition, you do not need to know how many arguments will be passed to your function. *args is used to send a non-keyworded variable length argument list to the function. Here's an example to help you get a clear idea:

```
def test_var_args(f_arg, *argv):
print("first normal arg:", f_arg)
for arg in argv:
print("another arg through *argv:", arg)
test_var_args('yasoob', 'python', 'eggs', 'test')
```

This produces the following result:

```
first normal arg: yasoob
another arg through *argv: python
another arg through *argv: eggs
another arg through *argv: test
```

I hope this cleared away any confusion that you had. So now let's talk about **kwargs

1.2. Usage of **kwargs

**kwargs allows you to pass keyworded variable length of arguments to a function. You should use **kwargs if you want to handle named arguments in a function. Here is an example to get you going with it:

```
def greet_me(**kwargs):
for key, value in kwargs.items():
print("{0} = {1}".format(key, value))
>>> greet_me(name="yasoob")
name = yasoob
```

So you can see how we handled a keyworded argument list in our function. This is just the basics of **kwargs and you can see how useful it is. Now let's talk about how you can use *args and **kwargs to call a function with a list or dictionary of arguments.

1.3. Using *args and **kwargs to call a function

So here we will see how to call a function using *args and **kwargs. Just consider that you have this little function:

```
def test_args_kwargs(arg1, arg2, arg3):
print("arg1:", arg1)
print("arg2:", arg2)
print("arg3:", arg3)
```

Now you can use *args or **kwargs to pass arguments to this little function. Here's how to do it:

```
# first with *args
>>> args = ("two", 3, 5)
>>> test_args_kwargs(*args)
arg1: two
arg2: 3
arg3: 5
# now with **kwargs:
>>> kwargs = {"arg3": 3, "arg2": "two", "arg1": 5}
>>> test_args_kwargs(**kwargs)
arg1: 5
arg2: two
arg3: 3
```

68

Order of using *args **kwargs and formal args

So if you want to use all three of these in functions then the order is

```
some_func(fargs, *args, **kwargs)
```

7. <u>Lists:</u>

LISTS

Like a string, a *list* is a sequence of values. In a string, the values are characters; in a list, they can be any type. The values in list are called *elements* or sometimes *items*.

CREATING A LIST

➢ A list is created by placing all the items (elements) inside a square bracket [ ], separated by commas.

For example,

I.

num = [10, 20, 30, 40, 50]→ List of five integers

II.

string = [‘Virat Kohli’, ‘MS Dhoni’, ‘Hardik Pandya’, ‘Sachin Tendulkar’]→ List of four strings

➢ The elements of a list don’t have to be the same type.

For example,

```
mix = ['awesome', 19, 27.5, [14, 32]]
```

The above list contains a string, an integer, a float and a nested list. A list within a list is called a nested list

➢ A list that contains no elements is called an empty list; we can create one with empty brackets, [].

For example,

```
list = []
```

➢ We can print the list as follows:

```
>>> print(list,mix,string,num)
```

[] ['awesome', 19, 27.5, [14, 32]] ['Virat Kohli', 'MS Dhoni', 'Hardik Pandya', 'Sachin Tendulkar'] [10, 20, 30, 40, 50]

LIST METHODS

➢ Python provides methods that operate on lists.

1. append()

✓ The append() method adds a single item to the existing list. It doesn't return a new list; rather it modifies the original list.
✓ The syntax is:
list_name.append(item)
✓ The append() method takes a single *item* and adds it to the end of the list. The *item* can be numbers, strings, another list, dictionary etc.
✓ For example:
I.
```
>>> n = [1, 2, 3, 4]
>>> n.append('end')
>>> print(n)
[1, 2, 3, 4, 'end']
```
II.
```
>>> n = [1, 2, 3, 4]
>>> n.append([5, 6])
>>> print(n)
[1, 2, 3, 4, [5, 6]]
```
2. extend()
✓ The extend() extends the list by adding all items of a list (passed as an argument) to the end.
✓ The extend() method takes a single argument (a list) and adds it to the end.
✓ The syntax is:
list1_name.extend(list2_name)
✓ For example:
```
>>> branch = ['ise', 'cse', 'tce','ece']
>>> branch1 = ['mech', 'civil']
>>> branch.extend(branch1)
>>> print(branch)
['ise', 'cse', 'tce', 'ece', 'mech', 'civil']
```
3. sort()
✓ The sort() method sorts the elements of a given list.
✓ The sort() method sorts the elements of a given list in a specific order - Ascending or Descending.
✓ The syntax is:
list_name.sort()
✓ By default, sort() doesn't require any extra parameters. However an optional parameter: reverse - If true, the sorted list is reversed (or sorted in Descending order)
✓ For example:
I.
```
>>> vowels = ['u', 'i', 'a', 'e', 'o']
>>> vowels.sort()
>>> print(vowels)
['a', 'e', 'i', 'o', 'u']
```
II.
```
>>> vowels = ['u', 'i', 'a', 'e', 'o']
>>> vowels.sort(reverse=True)
>>> print(vowels)
['u', 'o', 'i', 'e', 'a']
```
4. insert()
✓ The insert() method inserts the element to the list at the given index.
✓ The syntax is:
list_name.insert(index, element)
✓ The insert() function takes two parameters:
• index - position where element needs to be inserted

• element - this is the element to be inserted in the list
✓ For example:
I.
```
>>> vowels.insert(2, 'i')
>>> print(vowels)
['a', 'e', 'i', 'o', 'u']
```
II.
```
>>> num = [34, 56, 67, 89]
>>> num.insert(6, 23)
>>> print(num)
[34, 56, 67, 89, 23]
```
III.
```
>>> num = [34, 56, 67, 89]
>>> num.insert(0, 23)
>>> print(num)
[23, 34, 56, 67, 89]
>>> num.insert(-1, 23)
>>> print(num)
[23, 34, 56, 67, 23, 89]
```
5. index()
✓ The index() method searches an element in the list and returns its index.
✓ The syntax is:
list_name.index(element)
✓ The index method takes a single argument: element - element that is to be searched.
✓ If the same element is present more than once, index() method returns its smallest/first position.
✓ The index() method returns the index of the element in the list.
✓ If not found, it raises a ValueError exception indicating the element is not in the list.
✓ For example:
```
>>> list = [85, -5, 546, 43, -532, 32]
>>> i = list.index(43)
>>> print(i)
3
>>> i = list.index(9)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: 9 is not in list
```
6. reverse()
✓ The reverse() method reverses the elements of a given list.
✓ The syntax is:
list_name.reverse()
✓ For example:
```
>>> os = ['iOS', 'Android', 'Windows', 'Linux']
>>> os.reverse()
>>> print(os)
['Linux', 'Windows', 'Android', 'iOS']
>>> os[::-1]
['iOS', 'Android', 'Windows', 'Linux']
```
✓ Accessing individual elements in reversed order:
```
>>> os = ['iOS', 'Android', 'Windows', 'Linux']
>>> for i in reversed(os):
... print(i)
...
```

Linux
Windows
Android
iOS
7. count()
✓ The count() method returns the number of occurrences of an element in a list.
✓ The syntax is:
list_name.count(element)
✓ The count() method takes a single argument: element - element whose count is to be found.
✓ The count() method returns the number of occurrences of an element in a list.
✓ For example:
```
>>> n = [1, 2, 2, 3, 4, 5, 5]
>>> n.count(2)
2
>>> n.count(4)
1
>>> n.count(8)
0
```
8. clear()
✓ The clear() method removes all items from the list.
✓ The syntax is:
list_name.clear()
✓ For example:
```
>>> os = ['iOS', 'Android', 'Windows', 'Linux']
>>> os.clear()
>>> print(os)
[]
```
9. copy()
✓ The copy() method returns a shallow copy of the list.
✓ A list can be copied with = operator.
For example:
```
>>> n = [1, 2, 3]
>>> new = n
>>> print(new)
[1, 2, 3]
>>> print(n)
[1, 2, 3]
```
✓ The problem with copying the list in this way is that if you modify the *new*, the *n* is also modified.
For example:
```
>>> n = [1, 2, 3]
>>> new = n
>>> new.append(4)
>>> print(new)
[1, 2, 3, 4]
>>> print(n)
[1, 2, 3, 4]
```
✓ If you need the original list unchanged when the new list is modified, you can use copy() method. This is called shallow copy.
✓ The syntax is:
new_list = list.copy()
✓ The copy() function returns a list. It doesn't modify the original list.
✓ For example:
```
>>> mix = ['awesome', 19, 27.5]
```

```
>>> new = mix.copy()
>>> new.append([34, -143])
>>> print(mix)
['awesome', 19, 27.5]
>>> print(new)
['awesome', 19, 27.5, [34, -143]]
```

8.Tuple methods:

i) Append to a Tuple:

Tuples are immutable in Python, meaning you cannot directly add or remove elements from an existing tuple. However, you can create a new tuple by combining the elements of two tuples.

Example:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Append elements using concatenation
appended_tuple = tuple1 + tuple2
print(appended_tuple)
```

ii) Add Two Tuples:

Example:

```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

# Add two tuples element-wise
added_tuple = tuple(map(lambda x, y: x + y, tuple1, tuple2))
print(added_tuple)
```

iii) Remove Item from Tuple:

Since tuples are immutable, you cannot directly remove items. You can create a new tuple excluding the item you want to "remove."

Example:

```
original_tuple = (1, 2, 3, 4, 5)

# Remove item by creating a new tuple
removed_tuple = tuple(item for item in original_tuple if item != 3)
print(removed_tuple)
```

iv) Multiply Tuples by 3:

Multiplying a tuple by a scalar repeats its elements.

Example:

```
original_tuple = (1, 2, 3)
```

```
# Multiply tuple elements by 3
multiplied_tuple = original_tuple * 3
print(multiplied_tuple)
```

v) Count Method in Tuple:

The `count()` method returns the number of occurrences of a specified value in the tuple.

Example:

```
tuple_with_duplicates = (1, 2, 2, 3, 4, 2, 5)

# Count occurrences of 2
count_of_2 = tuple_with_duplicates.count(2)
print(count_of_2)
```

9.**Program**:

```
class Base:
    def __init__(self):
        self.a=10
        self._b=20
    def display(self):
        print("the values are:")
        print(f"a={self.a} b={self._b}")

class Derived(Base):
    def __init__(self):
        Base.__init__(self)
        self.d=30

    def display(self):
        Base.display(self)
        print(f"d={self.d}")

    def __add__(self,ob):
        return self.a + ob.a + self.d + ob.d

obj1=Base()
obj2=Derived()
obj3=Derived()

obj2.display()
obj3.display()

print("\n Sum of two objects:", obj2 + obj3)
```

Output:

```
the values are:
a=10 b=20
d=30
the values are:
a=10 b=20
d=30
```

Sum of two objects: 80

10.Program:

Poem.txt:

```
When to the session of sweet silent thought
I summon up remembrance of things past
I sigh the lack : of many a thing I sought
And with old woes new wail my dear time is waste
The sad account of fore-bemoaned moan ,
Which I new pay as if not paid before
But if the while I think on thee dear friend
All losses are restored and sorrows end !!
```

Dictionary.py

```
#To find frequency of words in a file.
fname = input('Enter the file name: ')
try:
    fhand = open(fname)
except:
    print('File cannot be opened:', fhand)
    exit()
counts = dict()
for line in fhand:
    words = line.split()
    for word in words:
        if word not in counts:
            counts[word] = 1
        else:
            counts[word] += 1
print(counts)
```

Output:

```
Enter the file name: poem.txt
{'When': 1, 'to': 1, 'the': 3, 'session': 1, 'of': 4, 'sweet': 1, 'silent': 1, 'thought': 1, 'I': 5, 'summon': 1, 'up': 1, 'rem
embrance': 1, 'things': 1, 'past': 1, 'sigh': 1, 'lack': 1, ':': 1, 'many': 1, 'a': 1, 'thing': 1, 'sought': 1, 'And': 1, 'with':
 1, 'old': 1, 'woes': 1, 'new': 2, 'wail': 1, 'my': 1, 'dear': 2, 'time': 1, 'is': 1, 'waste': 1, 'The': 1, 'sad': 1, 'account':
1, 'fore-bemoaned': 1, 'moan': 1, ',': 1, 'Which': 1, 'pay': 1, 'as': 1, 'if': 2, 'not': 1, 'paid': 1, 'before': 1, 'But': 1, '
while': 1, 'think': 1, 'on': 1, 'thee': 1, 'friend': 1, 'All': 1, 'losses': 1, 'are': 1, 'restored': 1, 'and': 1, 'sorrows': 1, 'e
nd': 1, '!!': 1}
```