ANSWER KEY
Internal Assessment Test 1 – Nov, 2021

Sub:	**Data Analytics using Python**							**Sub Code:**	**20MCA31**
Date:	**11/11//2021**	**Duration:**	**90 min's**	**Date:**	**11/11//2021**	**Duration:**	**90 min's**	**Date:**	**11/11//2021**

1. **(a) Write the features of Python. Give the advantages & disadvantages of it. (5 Marks)**

Features

- **Object oriented language : .** Python supports object-oriented language and concepts of classes, objects encapsulation, etc.
- **Interpreted language** : Python code is executed line by line at a time.
- **Supports dynamic data type :** A variable is decided at run time not in advance. hence, we don't need to specify the type of variable. (for example- int, double, long, etc.)
- **Simple and easy to code :** Python is very easy code
- **High-level Language**
- **Automatic memory management**
- **open source:** Python language is freely available

advantages & disadvantages

advantages

- Free availability (like Perl, Python is open source).
- Stability (Python is in release 2.6 at this point and, as I noted earlier, is older than Java).
- Very easy to learn and use
- Good support for objects, modules, and other reusability mechanisms.
- Easy integration with and extensibility using C and Java.

disadvantages

- Smaller pool of Python developers compared to other languages, such as Java
- Lack of true multiprocessor support
- Absence of a commercial support point, even for an Open Source project (though this situation is changing)
- Software performance slow, not suitable for high performance applications

(b) Write a Python function to sum of the numbers in a list (5 Marks)

```
def sum(numbers):
    total = 0
    for x in numbers:
        total += x
    return total
print(sum((8, 2, 3, 0, 7)))
```

2. **Discuss the Looping Statements with an example. (10 Marks)**
(i) while (ii) for (iii) range

The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.
We generally use this loop when we don't know the number of times to iterate beforehand.

Syntax of while Loop in Python

```
while test_expression:
    Body of while
```

In the while loop, test expression is checked first. The body of the loop is entered only if the test_expression evaluates to True. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to False.

In Python, the body of the while loop is determined through indentation.

The body starts with indentation and the first unindented line marks the end.

Python interprets any non-zero value as True. None and 0 are interpreted as False.

Flowchart of while Loop

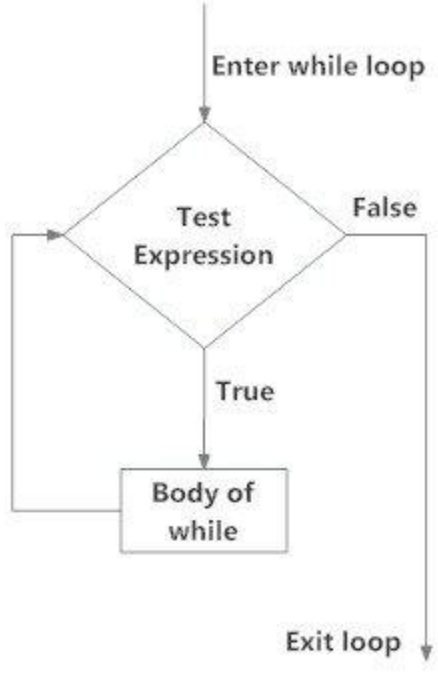


Fig: operation of while loop

Flowchart for while loop in Python

Example: Python while Loop

```
# Program to add natural
# numbers up to
# sum = 1+2+3+...+n

# To take input from the user,
# n = int(input("Enter n: "))

n = 10

# initialize sum and counter
sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1    # update counter

# print the sum
print("The sum is", sum)
```

When you run the program, the output will be:

```
Enter n: 10
The sum is 55
```

In the above program, the test expression will be `True` as long as our counter variable `i` is less than or equal to `n` (10 in our program).

We need to increase the value of the counter variable in the body of the loop. This is very important (and mostly forgotten). Failing to do so will result in an infinite loop (never-ending loop).

Finally, the result is displayed.

While loop with else

Same as with for loops, while loops can also have an optional `else` block.

The `else` part is executed if the condition in the while loop evaluates to `False`.

The while loop can be terminated with a break statement. In such cases, the `else` part is ignored. Hence, a while loop's `else` part runs if no break occurs and the condition is false.

Here is an example to illustrate this.

```
'''Example to illustrate
the use of else statement
with the while loop'''

counter = 0

while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

Output

```
Inside loop
Inside loop
Inside loop
Inside else
```

Here, we use a counter variable to print the string `Inside loop` three times.

On the fourth iteration, the condition in `while` becomes `False`. Hence, the `else` part is executed.

What is for loop in Python?

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

Syntax of for Loop

```
for val in sequence:
    loop body
```

Here, `val` is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Flowchart of for Loop

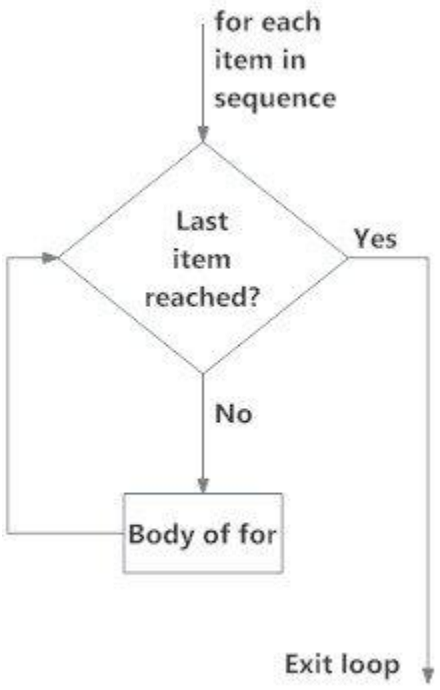


Fig: operation of for loop

Flowchart of for Loop in Python

Example: Python for Loop

```
# Program to find the sum of all numbers stored in a list

# List of numbers
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum
sum = 0

# iterate over the list
for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

When you run the program, the output will be:

```
The sum is 48
```

RANGE()

Range generates a list of integers and there are 3 ways to use it.
The function takes 1 to 3 arguments. Note I've wrapped each usage in list comprehension so we can see the values generated.

i) **range(end) :** generate integers from 0 to the "end" integer.

```
[i for i in range(10)]
#=> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

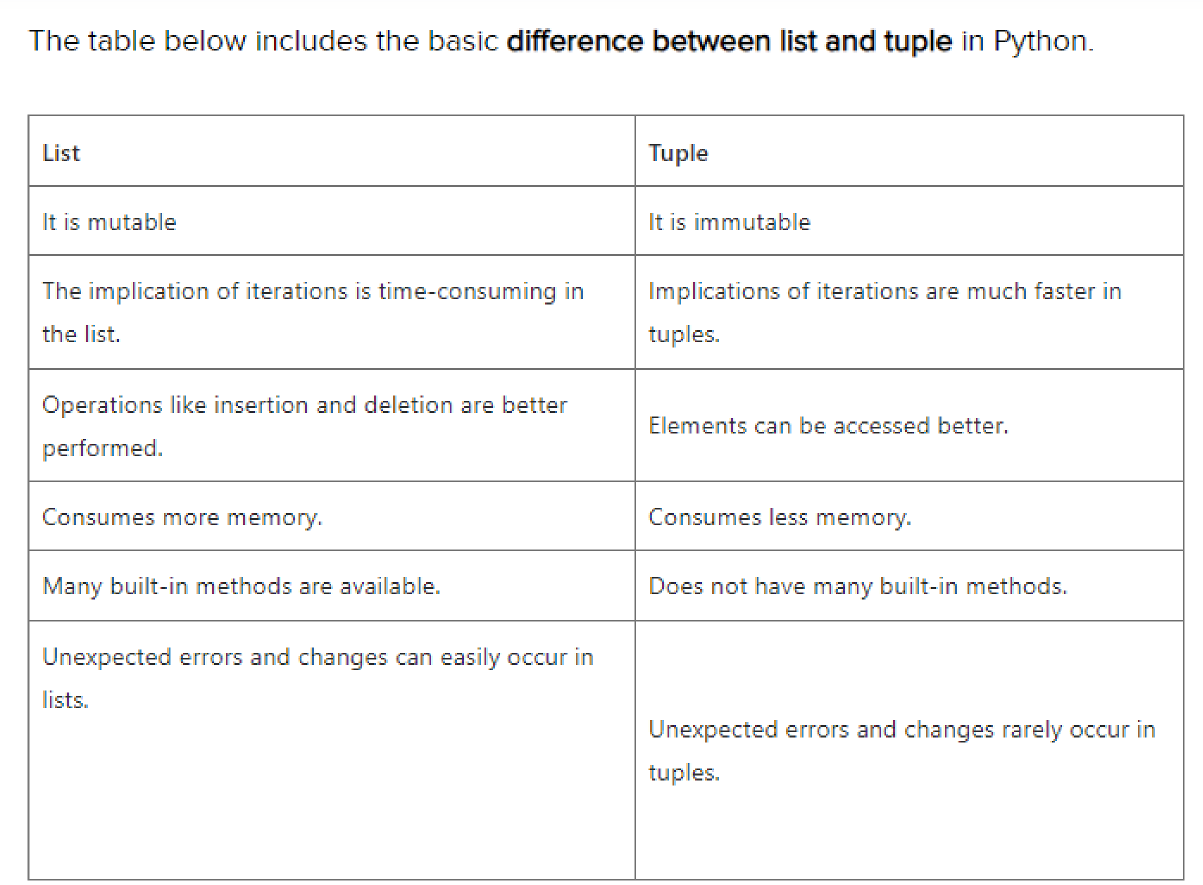
ii) **range(start, end) :** generate integers from the "start" to the "end" integer.

```
[i for i in range(2,10)]
#=> [2, 3, 4, 5, 6, 7, 8, 9]
```

iii) **range(start, end, step) :** generate integers from “start” to “end” at intervals of “step”.

```
[i for i in range(2,10,2)]
#=> [2, 4, 6, 8]
```

3. (a) What is the difference between a list and a tuple? Give an example (5 Marks)



The table below includes the basic **difference between list and tuple** in Python.

List	Tuple
It is mutable	It is immutable
The implication of iterations is time-consuming in the list.	Implications of iterations are much faster in tuples.
Operations like insertion and deletion are better performed.	Elements can be accessed better.
Consumes more memory.	Consumes less memory.
Many built-in methods are available.	Does not have many built-in methods.
Unexpected errors and changes can easily occur in lists.	Unexpected errors and changes rarely occur in tuples.

(b) What is the difference between a module and a package? (5 Marks)

Module

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement.

Items are imported using from or import.

Syntax :

```
 from module import function
 function()
```

Example :

```
      from math import sqrt
      print sqrt(2.0)
```

Package

The collections of modules organized together and kept into a directory. That directory is known as Package. That is , a package is a directory of modules.

Inside this directory there will be **__init__.py** file. This file is the one which will always be recognized and run by the compiler. Packages are modules, but not all modules are packages.

Example

```
From sklearn import cross_validation
```

4. **(a) Evaluate the following expression (i) 5//3*2-6/3*5%3 (ii) 5%8 *3+8%3*5 ((5 Marks)**

(i) 5//3*2-6/3*5%3
=1*2-6/3*5%3
=2-6/3*5%3
=2-2*5%3
=2-10%3
=2-1

Ans=1

(ii) 5%8*3+8%3*5
=5*3+8%3*5
=15+8%3*5
=15+2*5
=15+10

Ans=25

(b) Write the following in Python (5 Marks)

(ii) a is greater than any one of x,y,z (ii) (logx+sin45)/xy

(i) (a>x) or (a>y) or (a>z)
(ii) math.log2 (x)+math.sin(45)/(x*y)

5. **Explain in detail about python operators (10 Marks)**

Python Operators in general are used to perform operations on values and variables. These are standard symbols used for the purpose of logical and arithmetic operations. In this article, we will look into different types of Python operators.

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
%	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

```
13
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

Operator	Description	Syntax
<	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

Operator	Description	Syntax
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
%=	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

Operator	Description	Syntax
**=	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
```

Identity Operators

is and **is not** are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

is True if the operands are identical
is not True if the operands are not identical
Example: Identity Operator

```
a = 10
b = 20
c = a

print(a is not b)
print(a is c)
```

Output

```
True
True
```

Membership Operators
in and **not in** are the membership operators; used to test whether a value or variable is in a sequence.
in True if value is found in the sequence
not in True if value is not found in the sequence

Example: **Membership** Operator

```
# Python program to illustrate
# not 'in' operator
x = 24
y = 20
list = [10, 20, 30, 40, 50]

if (x not in list):
print("x is NOT present in given list")
else:
print("x is present in given list")

if (y in list):
print("y is present in given list")
else:
print("y is NOT present in given list")
```

Output

```
x is NOT present in given list
y is present in given list
```

Precedence and Associativity of Operators

6. **(a) Write python program to illustrate variable length keyword arguments (5 Marks)**
   *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
```

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)
```

(b) Write python program to perform linear search (5 Marks)

```
def search(arr, n, x):

    for i in range(0, n):
        if (arr[i] == x):
            return i
    return -1
```

```
# Driver Code
arr = [10,50,30,70, 80, 60, 20, 90,40]
x = 20
n = len(arr)

# Function call
result = search(arr, n, x)
if(result == -1):
    print("Element is not present in array")
else:
    print("Element is present at index", result)
```

7. **Explain any 5 string functions with examples (10 Marks)**

```
swapcase(...)
 |      S.swapcase() -> string
 |
 |      Return a copy of the string S with uppercase characters
 |      converted to lowercase and vice versa

strip(...)
 |      S.strip([chars]) -> string or unicode
 |
 |      Return a copy of the string S with leading and trailing
 |      whitespace removed.
 |      If chars is given and not None, remove characters in chars instead.

| startswith(...)
 |      S.startswith(prefix[, start[, end]]) -> bool
 |
 |      Return True if S starts with the specified prefix, False otherwise.
 |      With optional start, test S beginning at that position.
 |      With optional end, stop comparing S at that position.
 |      prefix can also be a tuple of strings to try.

| split(...)
 |      S.split([sep [,maxsplit]]) -> list of strings
 |
 |      Return a list of the words in the string S, using sep as the
 |      delimiter string.  If maxsplit is given, at most maxsplit
 |      splits are done. If sep is not specified or is None, any
 |      whitespace string is a separator and empty strings are removed
 |      from the result.

  format(...)
 |      S.format(*args, **kwargs) -> string
 |
 |      Return a formatted version of S, using substitutions from args and kwargs.
 |      The substitutions are identified by braces ('{' and '}').
```

8. **(a) Write short note on slice operator (5 Marks)**

Python slicing is about obtaining a sub-string from the given string by slicing it respectively from start to end.
Python slicing can be done in two ways.

- slice() Constructor
- Extending Indexing

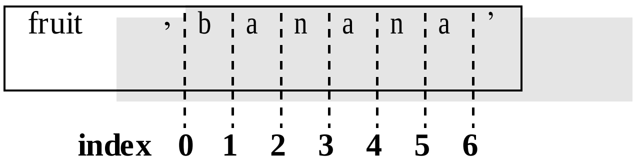
slice(stop)
slice(start, stop, step)
Parameters:
start: Starting index where the slicing of object starts.
stop: Ending index where the slicing of object stops.
step: It is an optional argument that determines the increment between each index for slicing.
Return Type: Returns a sliced object containing elements in the given range only.

slice() Constructor
The slice() constructor creates a slice object representing the set of indices specified by range(start, stop, step).

A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

```
>>> s = 'MontyPython'
>>>prints[0:5] Monty
>>>prints[6:13] Python
```

The operator [n:m]returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last. This behavior is counterintuitive, but it might help to imagine the indices pointing *between* the characters, as in the following diagram:



If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string:

```
>>> fruit = 'banana'
>>> fruit[:3] 'ban'
>>> fruit[3:] 'ana'
```

If the first index is greater than or equal to the second the result is an **empty string**, represented by two quotation marks:

```
>>> fruit = 'banana'
>>> fruit[3:3] ''
```

An empty string contains no characters and has length 0, but other than that, it is the same as any other string.

Extending indexing

In Python, indexing syntax can be used as a substitute for the slice object. This is an easy and convenient way to slice a string both syntax wise and execution wise.

Syntax

string[start:end:step]
start, end and step have the same mechanism as slice() constructor.

Example

```
# String slicing
String ='ASTRING'

# Using indexing sequence
print(String[:3])
print(String[1:5:2])
print(String[-1:-12:-2])

# Prints string in reverse
print("\nReverse String")
print(String[::-1])
```

```
Output:
AST
SR
GITA

Reverse String
GNIRTSA
```

(b) nums = [10, 20, 30, 40, 50, 60, 70, 80, 90]

Write the output (i)nums[2:7] (ii) nums[:5] (iii) nums[-3:] (5 Marks)

(i) [30, 40, 50, 60, 70]

(ii) [10, 20, 30, 40, 50]

(iii) [70, 80, 90]

9. Write a python program using object oriented programming to demonstrate encapsulation, overloading and inheritance (10 Marks)

```
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):                          # Creating a derived class
   def __init__(self):
      Base.__init__(self)                     # Calling constructor of Base class
      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
      #return self.a + ob.a+self.d + ob.d+self.b + ob.b

obj1 = Base()
```

```
obj2 = Derived()
obj3 = Derived()

obj2.display()
obj3.display()

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

10. Write Python program to count words and store in dictionary for the given input text
Input Text : the clown ran after the car and the car ran into the tent and the tent fell down on the clown and the car
Output : word count : {'and': 3, 'on': 1, 'ran': 2, 'car': 3, 'into': 1, 'after': 1, 'clown': 2, 'down': 1, 'fell': 1, 'the': 7, 'tent': 2}

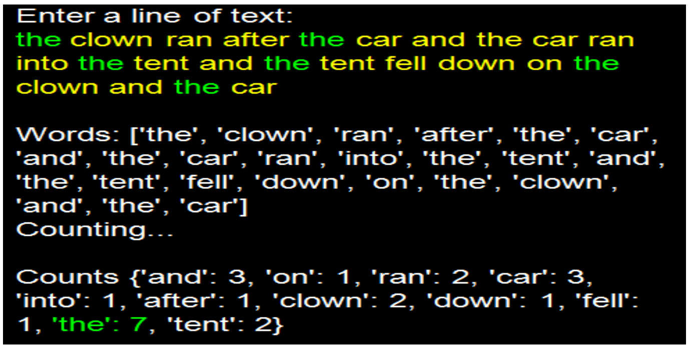
Method 1

```
counts = dict()
line = input('Enter a line of text:')
words = line.split()
print('Words:', words)
print('Counting...')

for word in words:
    counts[word] = counts.get(word,0) + 1
print('Counts', counts)
```

Method 2

```
def word  count(str):
   counts = dict()
   words = str.split()
   for word in words:
       if word in counts:
           counts[word] += 1
       else:
           counts[word] = 1
   return counts

#Driver Code
print( word_count(' the clown ran after the car and the car ran into the tent and the tent fell down on the clown and the car'))
```

Output:



```
Enter a line of text:
the clown ran after the car and the car ran
into the tent and the tent fell down on the
clown and the car

Words: ['the', 'clown', 'ran', 'after', 'the', 'car',
'and', 'the', 'car', 'ran', 'into', 'the', 'tent', 'and',
'the', 'tent', 'fell', 'down', 'on', 'the', 'clown',
'and', 'the', 'car']
Counting...

Counts {'and': 3, 'on': 1, 'ran': 2, 'car': 3,
'into': 1, 'after': 1, 'clown': 2, 'down': 1, 'fell':
1, 'the': 7, 'tent': 2}
```