

EXPERIMENT NO. - 01

AIM: Edit/compile/run a program to display the statements on two different lines.

THEORY:

Python string functions are very popular. There are two ways to represent strings in python. String is enclosed either with single quotes or double quotes. Both the ways (single or double quotes) are correct depending upon the requirement. Sometimes we have to use quotes (single or double quotes) together in the same string, in such cases, we use single and double quotes alternatively so that they can be distinguished.

Example #1:

Check below example and analyze the error –

#Gives Error

```
print('It's python')
```

Explanation –

It gives an invalid syntax error. Because single quote after “it” is considered as the end of the string and rest part is not the part of a string.

It can be corrected as:

```
print("It's Python !")
```

Output:

It's Python!

Example #2:

If you want to print ‘*WithQuotes*’ in python, this can’t be done with only single (or double) quotes alone, it requires simultaneous use of both.

```
# this code prints the output within quotes.
```

```
# print WithQuotes within single quotes
```

```
print("WithQuotes")
```

```
print("Hello 'Python'")
```

```
# print WithQuotes within single quotes
```

```
print("WithQuotes")
```

```
print('Hello "Python"')
```

Output –

'WithQuotes'

EXPERIMENT NO. - 01

Hello 'Python'

"WithQuotes"

Hello "Python"

Summary –

The choice between both the types (single quotes and double quotes) depends on the programmer's choice. Generally, double quotes are used for string representation and single quotes are used for regular expressions, dict keys or SQL. Hence both single quote and double quotes depict string in python but it's sometimes our need to use one type over the other.

PROGRAM:

EXPT 1) A:

```
a="Hello World!"  
b="Welcome to Python Programming!"  
print(a)  
print(b)
```

EXPT 1) B:

```
print("Hello World!" + "\n Welcome to Python Programming!")
```

EXPERIMENT NO. - 02

AIM: Edit/compile/run a program to display the statements on two different lines.

THEORY:

Many scripting and programming languages, such as JScript, C#, and C++, make no attempt to match the code that is run with the actual physical lines typed into the text editor. This is because they not recognize the end of a line of code until it sees the termination character (in these cases, the semicolon). Thus, the actual physical lines of type taken up by the code are irrelevant.

Unlike other languages, Python does not use an end of line character. Most of the time a simple Enter will do. Yet, Python is very particular about indentation, spaces and lines in certain cases. This document is to help understand Python formatting.

It is important to understand how Python interprets:

2. End of Statement
3. Names and Capitalization
4. Comments
5. Block Structures
6. Tabs and Spaces

End of Statements

To end a statement in Python, you do not have to type in a semicolon or other special character; you simply press Enter. For example, this code will generate a syntax error:

```
message  
  
=  
  
'Hello World!'
```

This will not:

```
message = 'Hello World!'
```

In general, the lack of a required statement termination character simplifies script writing in Python. There is, however, one complication: To enhance readability, it is recommended that you limit the length of any single line of code to 79 characters. What happens, then, if you have a line of code that contains 100 characters?

Although it might seem like the obvious solution, you cannot split a statement into multiple lines simply by entering a carriage return. For example, the following code snippet returns a run-time error in Python because a statement was split by using Enter.

```
message= 'This message will generate an error because  
it was split by using the enter button on your  
keyboard'
```

You cannot split a statement into multiple lines in Python by pressing Enter. Instead, use the backslash (\) to indicate that a statement is continued on the next line. In the revised version of the script, a blank space and an underscore indicate that the statement that was started on line 1 is continued on line 2. To make it more apparent that line 2 is a continuation of line 1, line 2 is also indented four spaces. (This was done for the sake of readability, but you do not have to indent continued lines.)

EXPERIMENT NO. - 02

```
message\  
=  
"This \  
back slash \  
acts \  
like \  
enter'  
print\  
message  
message\  
=  
"""triple  
quotes  
will  
span  
multiple lines  
without  
errors"""  
print\  
message
```

Line continuation is automatic when the split comes while a statement is inside parenthesis ((), brackets ([]) or braces ({ }). This is convenient, but can also lead to errors if there is no closing Parenthesis, bracket or brace. Python would interpret the rest of the script as one statement in that case.

Python uses single quotes (') double quotes (") and triple quotes (""") to denote literal strings. Only the triple quoted strings (""") also will automatically continue across the end of line statement.

Sometimes, more than one statement may be put on a single line. In Python a semicolon (;) can be used to separate multiple statements on the same line. For instance three statements can be written:

```
y = 3; x = 5; print(x+y)
```

To the Python interpreter, this would be the same set of statements:

```
y = 3  
x = 5  
print(x+y)
```

EXPERIMENT NO. - 02

PROGRAM:

EXPT 2) A:

```
print('Hello World!')
print('Welcome to Python Programming!')

print(""" This is the Strangest
way to print over
multiple lines I know!""")
```

EXPT 2) B:

```
# Python multiline string with newlines example using brackets
multiline_str = ("Hello World! \n"
"Welcome to Python Programming! \n"
"I'm learning Python.\n"
"I refer to Lectures, Notes & Tutorials given by \n Subject Teacher: Prof. K. R. Korpe.\n"
"If you want a line break in the STRING, then you can use" "\n" " as a string literal wherever you
need it.")
print(multiline_str)
```

EXPERIMENT NO. - 03

AIM: Edit/compile/run a program to initialize the string “hello world!” to a variable Str1 and convert the string into uppercase.

THEORY:

Python String upper() method

The Python string **upper()** method is used to convert all the lowercase characters present in a string into uppercase. However, if there are elements in the string that are already uppercased, the method will skip through those elements.

The upper() method can be used in applications where case-sensitivity is not considered. It works cohesively with the lower() method; where the string is converted into lowercased letters.

Note – This method will not show affect any non-casebased characters like digits and symbols.

Syntax

Following is the syntax for Python String **upper()** method –

`str.upper()`

Parameters

This method does not accept any parameters.

Return Value

This method returns a copy of the string in which all case-based characters have been uppercased.

Example

If the given string contains all lowercased letters, the method will return the string with all uppercased letters.

The following example shows the usage of Python String upper() method. We will create a string containing all lowercase letters, say "this is string example". The upper() method is called on this string and the return value obtained will be the uppercased string of the input.

```
str = "this is string example";  
print(str.upper())
```

When we run above program, it produces following result –

THIS IS STRING EXAMPLE

Example

If the given string contains all uppercased letters, the method will return the original string.

In this example, we will create a string containing all uppercase letters, "THIS IS STRING EXAMPLE". The upper() method is called on this string and the return value will be the original string.

```
str = "THIS IS STRING EXAMPLE";  
print(str.upper())
```

When we run above program, it produces following result –

THIS IS STRING EXAMPLE

Example

Suppose the given string contains digits or symbols, the upper() method will not throw an error but returns the original string.

In the following example, a string "This is a digit/symbol string: 781261&*(*&&", containing digits and symbols is created. The upper() method is invoked on this input string as follows –

```
str = "This is a digit/symbol string: 781261&*(*&&";  
print(str.upper())
```

On executing the above program, the output will be displayed as given below –

THIS IS A DIGIT/SYMBOL STRING: 781261&*(*&&

Example

In a non case sensitive environment, two strings are compared to see if they are equal using the upper() method. The return value will be either true or false.

In this example program, we take two input strings, "string example", "STRING example". Then, we call the upper() method on both these strings. Using the conditional statements, we check if both strings are equal after they are uppercased. The result is printed for either case.

```
str1 = "string example"  
str2 = "STRING example"  
if(str1.lower() == str2.lower()):  
    print("Both strings are equal")  
else:  
    print("Both strings are not equal")
```

The output for the program above is displayed as –

Both strings are equal

PROGRAM:

The `upper()` method converts all lowercase characters in a string into uppercase characters and returns it.

Example

```
message = 'python is fun'  
  
# convert message to uppercase  
print(message.upper())  
  
# Output: PYTHON IS FUN
```

upper() Return Value

`upper()` method returns the uppercase string from the given string. It converts all lowercase characters to uppercase.

If no lowercase characters exist, it returns the original string.

Example 1: Convert a string to uppercase

```
# example string
string = "this should be uppercase!"
print(string.upper())

# string with numbers
# all alphabets should be lowercase
string = "Th!s Sh0uLd B3 uPp3rCas3!"
print(string.upper())
```

Output

```
THIS SHOULD BE UPPERCASE!
TH!S SH0ULD B3 UPP3RCAS3!
```

Example 2: How upper() is used in a program?

```
# first string
firstString = "python is awesome!"

# second string
secondString = "PyThOn Is AwEsOmE!"

if(firstString.upper() == secondString.upper()):

    print("The strings are same.")
else:
    print("The strings are not same.")
```

Output

```
The strings are same.
```

Note: If you want to convert to lowercase string, use [lower\(\)](#). You can also use [swapcase\(\)](#) to swap between lowercase to uppercase.

Conclusion: Hence, we have successfully studied about initializing the string “hello world!” to a variable Str1 (or any String) and Converted the string into uppercase.

EXPERIMENT NO. - 04

AIM: Edit/compile/run a program to read the radius of a circle and print the area of the circle.

THEORY:

Calculate the Area of the Circle using Python

INPUT FORMAT:

The input of the code consists of the integer "R", which represents the radius of the circle.

OUTPUT FORMAT:

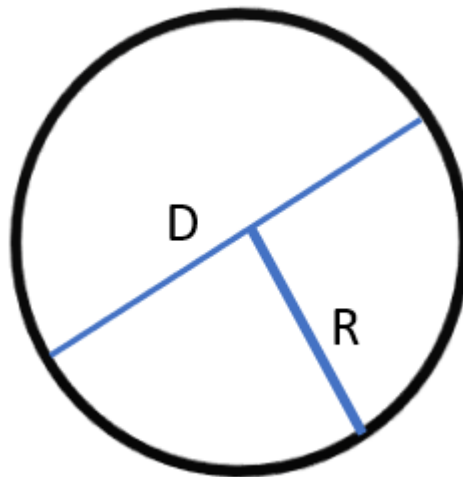
The output of the code will print the area of the circle.

Algorithm for Calculating the Area of the Given Circle

Following are the steps we will use for calculating the area of the given circle:

- **Step 1:** We have to pass the input using the input () function. The input will be corresponding to the radius of the given circle.
- **Step 2:** The Area of the circle will be calculated by using the formula of the $\text{Area} = \pi R^2$.

Area of Circle = $\pi * R * R$.



Where, π (PI) = 3.14

R = Radius of circle.

D or (2R) = Diameter of Circle, (R + R).

- **Step 3:** Print the output of the code, that is, the area of the given circle.

EXPERIMENT NO. - 04

Methods for finding the Area of the Given Circle using Python

Conclusion

In this experiment, we have shown three methods for calculating the area of the given circle. To calculate the area of the given circle, the user must know the radius or the diameter of the circle. Among the three methods, the first one is the easiest and most direct method.

Hence, we have successfully studied about program to read the radius of a circle and print the area of the circle.

PROGRAM:

EXPT 4) A:

Method 1: Find the area of the given circle using the math module.

1. **import** math as M
2. Radius = **float** (input ("Please enter the radius of the given circle: "))
3. area_of_the_circle = M.pi* Radius * Radius
4. print (" The area of the given circle is: ", area_of_the_circle)

Output:

```
Please enter the radius of the given circle: 3
The area of the given circle is: 28.274333882308138
```

EXPT 4) B:

Method 2: Calculate the area of the given circle using π

1. $\pi = 3.14$
2. Radius = **float** (input ("Please enter the radius of the given circle: "))
3. area_of_the_circle = π * Radius * Radius
4. print (" The area of the given circle is: ", area_of_the_circle)

Output:

```
Please enter the radius of the given circle: 3
The area of the given circle is: 28.259999999999998
```

EXPERIMENT NO. - 04

EXPT 4) C:

Method 3: Calculate the area of the given circle by using function

1. **import** math
- 2.
3. **def** area_of_the_circle (Radius):
4. area = Radius** 2 * math.pi
5. **return** area
- 6.
7. Radius = **float** (input ("Please enter the radius of the given circle: "))
8. **print** (" The area of the given circle is: ", area_of_the_circle (Radius))

Output:

```
Please enter the radius of the given circle: 3
The area of the given circle is: 28.274333882308138
```

EXPERIMENT NO. - 05

AIM: Edit/compile/run a program to read a four digit number through the keyboard and calculate the sum of its digit.

THEORY:

Introduction to Sum of Digits of a Number

The goal is to determine the **sum of digits** of a number given as an input in Python. **Example:**

Input:

```
n = 54
```

Output:

```
9
```

Input:

```
n = 121
```

Output:

```
4
```

We begin by dividing the number into digits and then adding all of the digits to the sum variable. To break down the string, we use the following operators:

The modulo operator % is used to extract the digits from a number. After removing the digit, we apply the divide operator to shorten the number.

Different Methods to Find Sum of Digits of a Number in Python

Using str() and int() methods

To convert a number to a string, use the [str\(\) function](#). To convert a string digit to an integer, use the [int\(\) function](#).

Convert the number to a string, iterate over each digit in the string, and add to the sum of the digits in each iteration.

Algorithm Flow:

- Step 1: Gather user input.
- Step 2: Create a variable to hold the result.
- Step 3: Convert the number to a string.
- Step 4: Write a loop for each digit in a number.
- Step 5: Convert the digit to an integer and add it to the sum.
- Step 6: Invoke the function and print the result.

Program A:

```
# Function to get sum of digits
def getSum(n):

    sum = 0
    for digit in str(n):
        sum += int(digit)
    return sum

n = 569
print(getSum(n))
```

Output:

```
20
```

Using iteration

We shall use loops to calculate the sum of digits of a number. Loops are used to execute a specific piece of code continually. Some looping statements are for loop, while, and do-while.

To find the rightmost digit of an integer, divide the integer by 10 until it equals 0. Finally, the remaining will be the rightmost digit. Use the remaining operator " percent " to receive the reminder. Divide the obtained quotient by 10 to get all the digits of a number. To find the number's quotient, we use “//”.

Algorithm Flow:

- Step 1: Create a function for finding the sum using the parameter n.
- Step 2: Declare a variable sum to store the digit sum.
- Step 3: Create a loop that will run until n is greater than zero.
- Step 4: To the remainder returned by, add the sum variable (n percent 10)
- Step 5: Change n to n/10.
- Step 6: Collect feedback from the user.
- Step 7: Invoke the function defined earlier and pass the input as an argument.
- Step 8: Print the sum of the values returned by the function.

Program B:

```
# Function to get the sum of digits
def getSum(n):

    sum = 0
    while (n != 0):

        sum = sum + (n % 10)
        n = n//10

    return sum

n = 569
print(getSum(n))
```

Output:

Using Recursion

Recursion is the process of defining a problem or the solution to a problem in terms of a simpler version of itself. The corresponding function is called the recursive function. The use of recursion eliminates the requirement for loops in the programming.

Follow the algorithm for a thorough description of how the software works.

Algorithm Flow:

- Step 1: Create a function for finding the sum of digits with the parameter n to compute the sum.
- Step 2: Determine whether n is less than 10; return n.
- Step 3: If not, divide the number by 10 and find the residual (n percent 10)
- Step 4: Recursively call the function and pass (n/10) as a parameter.
- Step 5: Add the remainder and the value returned by the function.
- Step 6: Collect user input.
- Step 7: Invoke the sum of digits function for finding the sum of digits of a number, passing input as a parameter.

Program C:

```
# sum of digits in number.

def sumDigits(no):
    return 0 if no == 0 else int(no % 10) + sumDigits(int(no / 10))

# Driver code
n = 569
print(sumDigits(n))
```

Output:

20

Using Sum() method

The [sum\(\) method](#) is used to compute the sum of digits of a number in python in a list.

Convert the number to a string using str(), then strip the string and convert it to a list of numbers with the strip() and map() methods, respectively. Then, compute the total using the sum() method.

Algorithm Flow:

- Step 1: Create a function for finding the sum of digits with the parameter n to compute the sum.
- Step 2: The number is converted to a string via the str() method.
- Step 3: Then, the string is striped and converted to list digits of the given number via strip() and map() method, respectively.
- Step 4: The sum() method is invoked to compute the total sum of digits.

Program D:

```
# Function to get sum of digits
```

```
def getSum(n):  
    strr = str(n)  
    list_of_number = list(map(int, strr.strip()))  
    return sum(list_of_number)  
  
n = 569  
print(getSum(n))
```

Output:

```
20
```

Conclusion

- A sum of digits of a number in python in a given number base is the sum of all its digits. For example, the digit sum of the decimal number 9045 would be $9+0+4+5=18$.
- sum of digits of a number in Python can be calculated in many ways, such as using inbuilt functions, iteration, and recursion, as discussed earlier.

Conclusion: Hence, we have successfully studied about initializing the program to read a four digit number through the keyboard and calculate the sum of its digit.

EXPERIMENT NO. - 06

AIM: Edit/compile/run a program to display the Floyd Triangle.

THEORY:

Floyd's triangle

A Floyd's triangle is a consecutive series of natural numbers in a right-angled triangle. This is what Floyd's triangle looks like-

```
1
2 3
4 5 6
7 8 9 10
```

Approach

The way we can approach to this problem could be-

1. Input n (number of rows).
2. Initialize count as 1 which will be incremented later.
3. Use a for loop initialize i from 1 to n+1
4. Use a nested for loop initialize j from 1 to i+1.
5. Print count with end=" ".
6. Increment the value of count by 1.
7. print() will change the line.

Implementation in Python

As we have discussed approach, now we can implement it using a for loop in Python as shown below-

Program A)

```
n = int(input("Enter the total Number of Rows : "))
count= 1
for i in range(1, n + 1):
    for j in range(1, i + 1):
        print(count, end = ' ')
        count = count + 1
    print()
```

Enter the total Number of Rows : 5

```
1
2 3
```



```
4 5 6
7 8 9 10
11 12 13 14 15
```

So we have printed our expected output. We can also use a similar approach using a while loop. The approach is quite similar with a different syntax as-

Program B)

```
n = int(input("Enter the total Number of Rows : "))
count= 1
i = 1
while(i <= n):
    j = 1
    while(j <= i):
        print(count, end = ' ')
        count = count + 1
        j = j + 1
    i = i + 1
    print()
```

```
Enter the total Number of Rows : 7
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
22 23 24 25 26 27 28
```

A Floyd's triangle is a right-angled triangle formed with natural numbers.

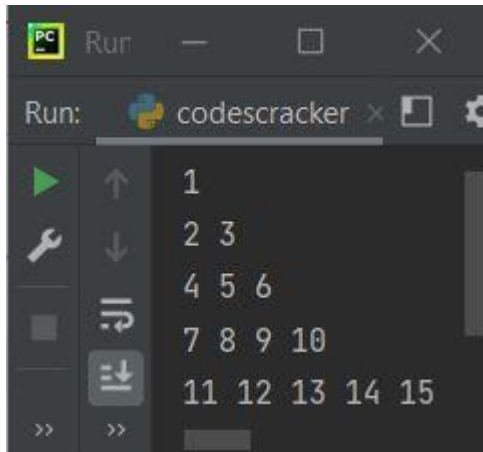
Print Floyd's Triangle in Python

The question is, *write a Python program to print Floyd's triangle*. The program given below is its answer:

```
num = 1
for i in range(5):
    for j in range(i+1):
        print(num, end=" ")
        num = num+1
```

`print()`

The snapshot given below shows the sample output produced by above Python program, that prints Floyd's triangle of 5 rows:



Print Floyd's Triangle of n Rows in Python

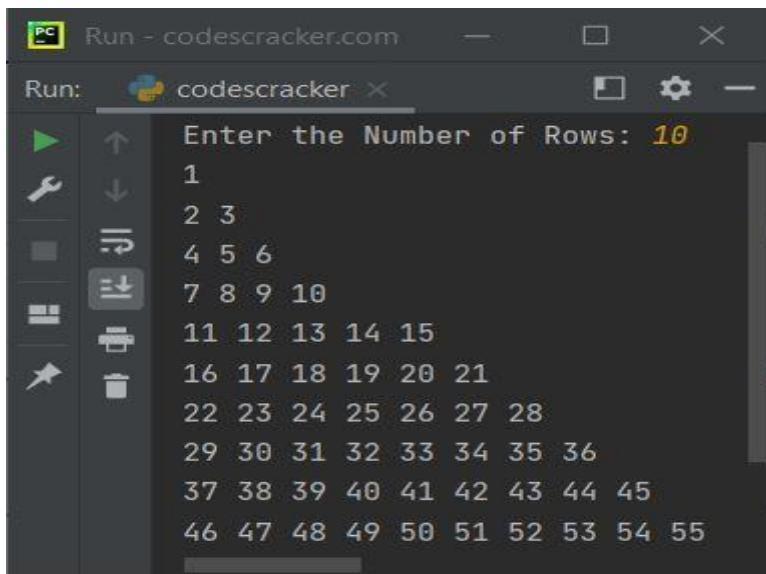
To print Floyd's triangle of **n** rows in Python, you need to ask from user to enter the number of rows or lines up to which, he/she wants to print the desired Floyd's triangle as shown in the program given below.

Program C)

```
print("Enter the Number of Rows: ", end="")
row = int(input())

num = 1
for i in range(row):
    for j in range(i+1):
        print(num, end=" ")
        num = num+1
    print()
```

Sample run of above program, with user input **10** as number of rows, is shown in the snapshot given below:



```
Run - codescracker.com
Run: codescracker x
Enter the Number of Rows: 10
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15
16 17 18 19 20 21
22 23 24 25 26 27 28
29 30 31 32 33 34 35 36
37 38 39 40 41 42 43 44 45
46 47 48 49 50 51 52 53 54 55
```

Print Floyd's Triangle using while Loop in Python

Let me create the same program as of previous, using **while** loop, instead of **for** loop.

Program D)

```
print("Enter the Number of Rows: ", end="")
row = int(input())

num = 1
i = 0
while i < row:
    j = 0
    while j < i+1:
        print(num, end=" ")
        num = num+1
        j = j+1
    print()
    i = i+1
```

Conclusion: Hence, we have successfully studied about program to display the Floyd Triangle.

EXPERIMENT NO. - 07

AIM: Edit/compile/run a program to read a string and display the total number of uppercase and lowercase letters.

THEORY:

Problem Description

The program takes a string and counts the number of lowercase letters and uppercase letters in the string.

Problem Solution

1. Take a string from the user and store it in a variable.
2. Initialize the two count variables to 0.
3. Use a for loop to traverse through the characters in the string and increment the first count variable each time a lowercase character is encountered and increment the second count variable each time an uppercase character is encountered.
4. Print the total count of both the variables.
5. Exit.

Program/Source Code

Here is source code of the Python Program to count the number of lowercase characters and uppercase characters in a string. The program output is also shown below.

```
string=raw_input("Enter string:")
count1=0
count2=0
for i in string:
    if(i.islower()):
        count1=count1+1
    elif(i.isupper()):
        count2=count2+1
print("The number of lowercase characters is:")
print(count1)
print("The number of uppercase characters is:")
print(count2)
```

Program Explanation

1. User must enter a string and store it in a variable.
2. Both the count variables are initialized to zero.
3. The for loop is used to traverse through the characters in the string.
4. The first count variable is incremented each time a lowercase character is encountered and the second count variable is incremented each time an uppercase character is encountered.
5. The total count of lowercase characters and uppercase characters in the string are printed.

Runtime Test Cases

```
Case 1:
Enter string:HeLlO
The number of lowercase characters is:
2
The number of uppercase characters is:
3

Case 2:
Enter string: San Francisco
The number of lowercase characters is:
10
The number of uppercase characters is:
3
```

PROGRAM A)

Method 1: Using the built-in methods

```
Str="InformationTechnology"

lower=0

upper=0

for i in Str:

    if(i.islower()):

        lower+=1

    else:

        upper+=1

print("The number of lowercase characters is:",lower)

print("The number of uppercase characters is:",upper)
```

Output

The number of lowercase characters is: 19

The number of uppercase characters is: 2

Explanation:

Here we are simply using the built-in method [islower\(\)](#) and checking for lower case characters and counting them and in the else condition we are counting the number of upper case characters provided that the string only consists of alphabets.

PROGRAM B)

Method 2: Using the [ascii](#) values, Naive Method

```
# Python3 program to count upper and

# lower case characters without using

# inbuilt functions
```

```
def upperlower(string):

    upper = 0

    lower = 0

    for i in range(len(string)):

        # For lower letters

        if (ord(string[i]) >= 97 and

            ord(string[i]) <= 122):

            lower += 1

        # For upper letters

        elif (ord(string[i]) >= 65 and

            ord(string[i]) <= 90):

            upper += 1

    print('Lower case characters = %s' %lower,

          'Upper case characters = %s' %upper)

# Driver Code

string = 'Google Classroom for Python is a Study-portal for Students'

upperlower(string)
```

Output

Lower case characters = 44 Upper case characters = 5

Explanation:

Here we are using the [ord\(\)](#) method to get the ascii value of that particular character and then calculating it in the particular range.

PROGRAM C)

Method 3: Calculating the characters within the given range of ascii code

```
s = "The Knowledge King"

l,u = 0,0

for i in s:

    if (i>='a'and i<='z'):

        # counting lower case

        l=l+1

    if (i>='A'and i<='Z'):

        #counting upper case

        u=u+1

print('Lower case characters: ',l)

print('Upper case characters: ',u)
```

Output

Lower case characters: 13

Upper case characters: 3

Explanation:

Here we are iterating through the string and calculating upper and lower case characters using the ascii code range.

PROGRAM D)

Method 4: Using 'in' keyword

```
# Python3 program to count upper and  
  
# lower case characters without using  
  
# inbuilt functions  
  
string = 'Google Classroom for Python is a Study-portal for Students'  
  
upper = 0  
  
lower = 0  
  
up="ABCDEFGHIJKLMNOPQRSTUVWXYZ"  
  
lo="abcdefghijklmnopqrstuvwxyz"  
  
for i in string:  
  
    if i in up:  
  
        upper+=1  
  
    elif i in lo:  
  
        lower+=1  
  
print('Lower case characters = %s' %lower)  
  
print('Upper case characters = %s' %upper)
```

Output

Lower case characters = 44

Upper case characters = 5

Explanation:

Here we have taken all the upper and lower case characters in separate strings and then count how many characters are present in individual strings.

PROGRAM E)

Method #5 : Using operator.countOf() method

```
import operator as op

Str = "ConsistencyIsNewGrowth"

lower = "abcdefghijklmnopqrstuvwxyz"

l = 0

u = 0

for i in Str:

    if op.countOf(lower, i) > 0:

        l += 1

    else:

        u += 1

print("The number of lowercase characters is:", l)

print("The number of uppercase characters is:", u)
```

Output

The number of lowercase characters is: 18

The number of uppercase characters is: 4

Conclusion: Hence, we have successfully studied executing the program to read a string and display the total number of uppercase and lowercase letters.

EXPERIMENT NO. - 08

AIM: Edit/compile/run a program to duplicate all the elements of a list.

THEORY:

Python Program to print duplicates from a list of integers:

In this experiment, we will see how we can print duplicates from a list of integers in Python. The List is an ordered set of values enclosed in square brackets []. List stores some values called elements in it, which can be accessed by their particular index.

A duplicate element in a list is the element that has occurred more than once in the list. We can print duplicates by following various approaches.

Given a list of integers, we have to print all those integers which have occurred more than once in the list.

Input: [10, 2, 2, 3, 1, 6, -3, -2, 3, 4, 8, 10]

Output: [10, 2, 3]

For printing duplicate integers from a list in Python, we can follow these approaches:

Brute force approach
using Counter() method

Approach 1: Brute force approach

Also known as exhaustive search, in this, we will use two for loops which will count the occurrence of each integer in the list and display the duplicate elements based on the count of their occurrence.

Algorithm

Follow the algorithm to understand the approach better.

Step 1- Define a function to find duplicates

Step 2- In the function declare a list that will store all the duplicate integers

Step 3- Run a loop for each element in the list

Step 4- For each integer, run another loop that will check if the same integer is repeated

Step 5- If found the repeated integer will be added to another list using append()

Step 6- Return the list which stores repeated integers

Python Program 1

Look at the program to understand the implementation of the above-mentioned approach. In this program, we have defined a function that will check for each integer in the list if it is repeated in the list or not. Integers that are repeated will be stored in another list and returned by the function.

PROGRAM A:

```
# print duplicates from a list of integers
```

```
#function
```

```
def duplicate(li):
```

```
    n=len(li)
```

```
    dup=[]
```

```
    for i in range(n):
```

```
        k = i + 1
```

```
        for j in range(k,n):
```

```
            if li[i] == li[j] and li[i] not in dup:
```

```
                dup.append(li[i])
```

```
    return dup
```

```
#test
```

```
li=[ 10, 20, 30, -10, -20, 10, 80, -10, -20, 30]
```

```
print("Duplicate integeres: ",duplicate(li))
```

OUTPUT:

Duplicate integeres: [10, 30, -10, -20]

Approach 2: Counter() function

Counter() is a built-in function that returns a dictionary of all the elements and their occurrences in a list. A dictionary has keys and values corresponding to it, the elements in the list and their occurrences are returned in a similar way.

After getting occurrences of each integer, we will check which integers occurred more than once and then store them in another list.

Algorithm

Follow the algorithm to understand the approach better.

Step 1- Import Counter function

Step 2- Take input of list with integer values

Step 3- Use Counter() to calculate occurrences of all the numbers in the list

Step 4- Declare another list

Step 5- Use a list comprehension to store integers with occurrence more than 1 in the list

Step 6- Print the list

Python Program 2

To use the Counter() function in your program, import it from the collections module. We will add all the numbers which are repeated in the list into another list with the help of list comprehension.

PROGRAM B:

```
from collections import Counter
```

```
li=[1, 3, 2, 6, 2, 3, 5, 6, 4, 5]
```

```
d = Counter(li)
```

```
repeated_list = list([num for num in d if d[num]>1])
```

```
print("Duplicate integers: ",repeated_list)
```

OUTPUT:

```
Duplicate integers: [3, 2, 6, 5]
```

PROGRAM C:

Approach 3: Using a single for loop

```
# Python program to print duplicates from
```

```
# a list of integers
```

```
lis = [1, 2, 1, 2, 3, 4, 5, 1, 1, 2, 5, 6, 7, 8, 9, 9]
```

```
uniqueList = []  
duplicateList = []
```

```
for i in lis:  
    if i not in uniqueList:  
        uniqueList.append(i)  
    elif i not in duplicateList:  
        duplicateList.append(i)
```

```
print(duplicateList)
```

OUTPUT:

```
[1, 2, 5, 9]
```

Conclusion: Hence, we have successfully studied about program for duplicating all the elements of a list.

EXPERIMENT NO. - 09

AIM: Edit/compile/run a program to implement quick sort/ merge sort/ bubble sort.

THEORY:

Python Program for QuickSort:

QuickSort is a **divide and conquer algorithm**. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick the first element as a pivot
2. Always pick the last element as a pivot
3. Pick a random element as a pivot
4. Pick median as a pivot

Here we will be picking the last element as a pivot. The key process in quickSort is partition(). Target of partitions is, given an array and an element 'x' of array as a pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code: Recursive QuickSort function

PROGRAM A) QUICK SORT ALGORITHM

```
# Python program for implementation of Quicksort Sort

# This implementation utilizes pivot as the last element in the nums list
# It has a pointer to keep track of the elements smaller than the pivot
# At the very end of partition() function, the pointer is swapped with
the pivot
# to come up with a "sorted" nums relative to the pivot


# Function to find the partition position
def partition(array, low, high):

    # choose the rightmost element as pivot
    pivot = array[high]

    # pointer for greater element
    i = low - 1

    # traverse through all elements
    # compare each element with pivot
    for j in range(low, high):
        if array[j] <= pivot:

            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with the greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])
```

```

        # Return the position from where partition is done
        return i + 1

# function to perform quicksort

def quickSort(array, low, high):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quickSort(array, low, pi - 1)

        # Recursive call on the right of pivot
        quickSort(array, pi + 1, high)

data = [1, 7, 4, 1, 10, 9, -2]
print("Unsorted Array")
print(data)

size = len(data)

quickSort(data, 0, size - 1)

print('Sorted Array in Ascending Order:')
print(data)

```

OUTPUT:

Unsorted Array

[1, 7, 4, 1, 10, 9, -2]

Sorted Array in Ascending Order:

[-2, 1, 1, 4, 7, 9, 10]

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

Need for Merge Sort

One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of $O(n \log n)$, which means it can

sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

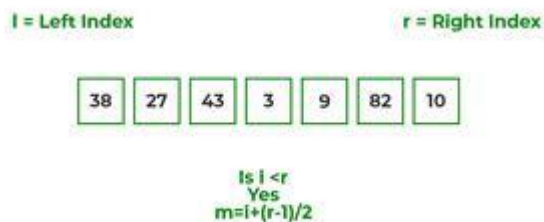
Merge Sort Working Process:

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

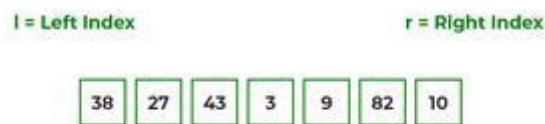
Illustration:

To know the functioning of merge sort lets consider an array `arr[] = {38, 27, 43, 3, 9, 82, 10}`

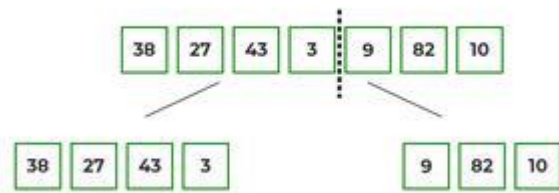
- *At first, check if the left index of array is less than the right index, if yes then calculate its mid point*



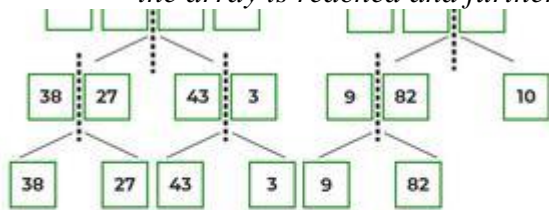
- *Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.*
- *Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.*



- *Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.*

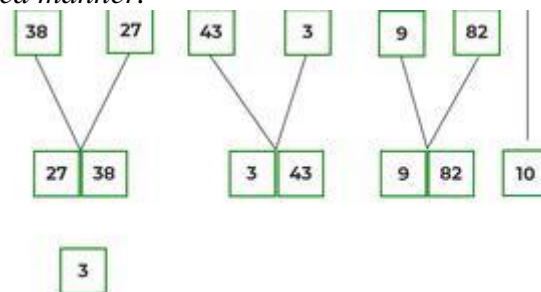


- Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.

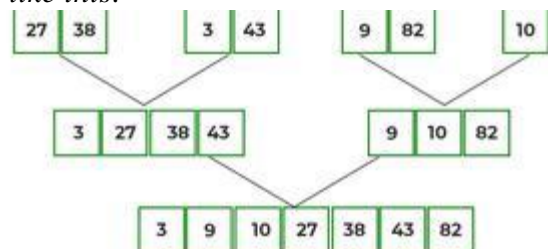


After dividing the array into smallest units merging starts, based on comparison of elements.

- After dividing the array into smallest units, start merging the elements again based on comparison of size of elements
- Firstly, compare the element for each list and then combine them into another list in a sorted manner.

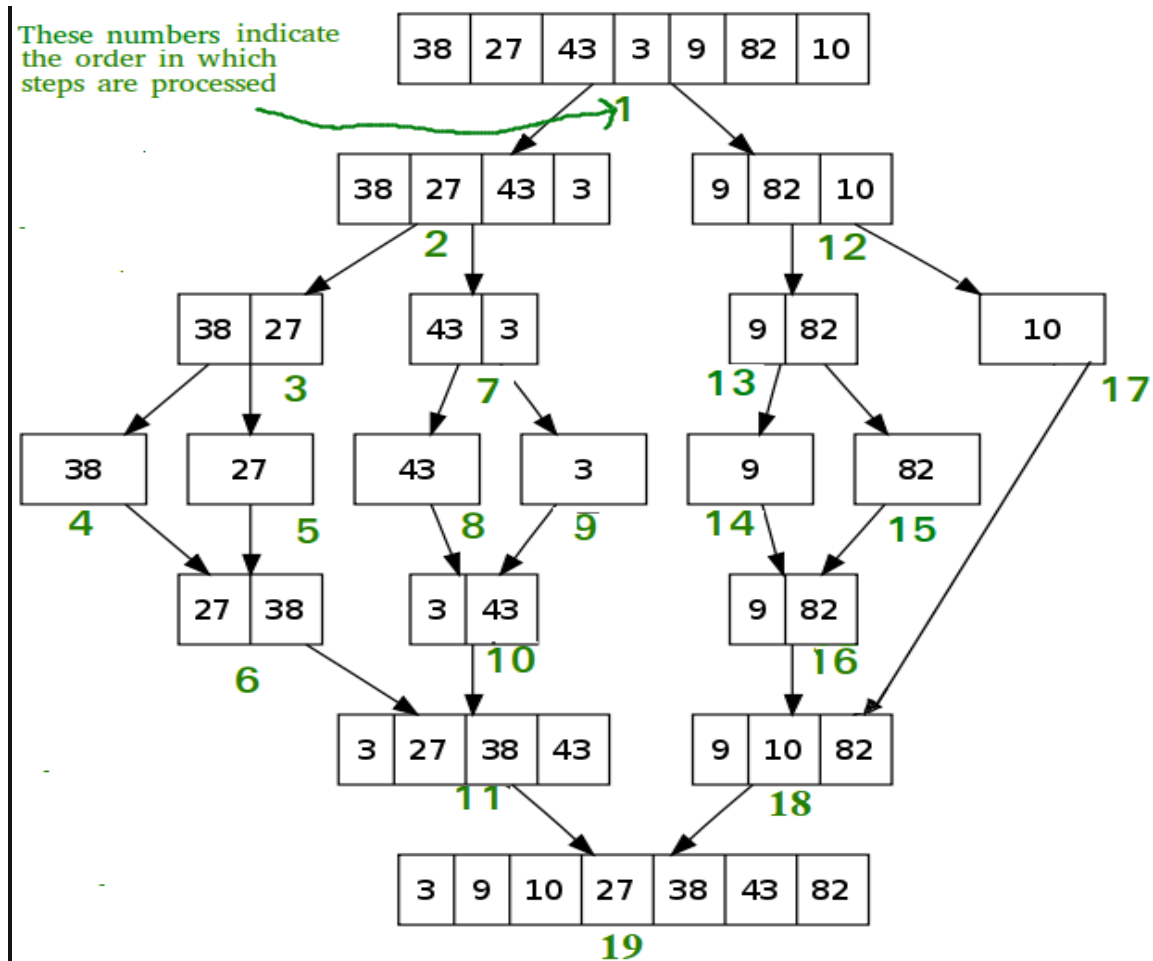


- After the final merging, the list looks like this:



The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Recursive steps of merge sort

Algorithm:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

if left > right

return

mid= (left+right)/2

mergesort(array, left, mid)

mergesort(array, mid+1, right)

merge(array, left, mid, right)

step 4: Stop

Follow the steps below to solve the problem:

MergeSort(arr[], l, r)

If $r > l$

- Find the middle point to divide the array into two halves:
 - middle $m = l + (r - l) / 2$
- Call mergeSort for first half:
 - Call mergeSort(arr, l, m)
- Call mergeSort for second half:
 - Call mergeSort(arr, m + 1, r)
- Merge the two halves sorted in steps 2 and 3:
 - Call merge(arr, l, m, r)

PROGRAM B) MERGE SORT ALGORITHM

```
# Python program for implementation of MergeSort
def mergeSort(arr):
    if len(arr) > 1:

        # Finding the mid of the array
        mid = len(arr)//2

        # Dividing the array elements
        L = arr[:mid]

        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
```

```

        arr[k] = R[j]
        j += 1
        k += 1

# Code to print the list

def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()

# Driver Code
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)

# This code is contributed by Karan Korpe

```

OUTPUT:

Given array is

12 11 13 5 6 7

Sorted array is

5 6 7 11 12 13

Bubble Sort Algorithm

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

How does Bubble Sort Work?

Input: $arr[] = \{6, 3, 0, 5\}$

First Pass:

- Bubble sort starts with very first two elements, comparing them to check which one is greater.
 - $(6\ 3\ 0\ 5) \rightarrow (3\ 6\ 0\ 5)$, Here, algorithm compares the first two elements, and swaps since $6 > 3$.
 - $(3\ 6\ 0\ 5) \rightarrow (3\ 0\ 6\ 5)$, Swap since $6 > 0$
 - $(3\ 0\ 6\ 5) \rightarrow (3\ 0\ 5\ 6)$, Swap since $6 > 5$

Second Pass:

- Now, during second iteration it should look like this:
 - $(3\ 0\ 5\ 6) \rightarrow (0\ 3\ 5\ 6)$, Swap since $3 > 0$
 - $(0\ 3\ 5\ 6) \rightarrow (0\ 3\ 5\ 6)$, No change as $5 > 3$

Third Pass:

- Now, the array is already sorted, but our algorithm does not know if it is completed.

- The algorithm needs one **whole** pass without **any** swap to know it is sorted.
 - (0 3 5 6) \rightarrow (0 3 5 6), No change as $3 > 0$

Array is now sorted and no more pass will happen.

Follow the below steps to solve the problem:

- Run a nested for loop to traverse the input array using two variables **i** and **j**, such that $0 \leq i < n-1$ and $0 \leq j < n-i-1$
- If **arr[j]** is greater than **arr[j+1]** then swap these adjacent elements, else move on
- Print the sorted array

Below is the implementation of the above approach:

PROGRAM C) BUBBLE SORT ALGORITHM

```
# Python program for implementation of Bubble Sort

def bubbleSort(arr):

    n = len(arr)

    # optimize code, so if the array is already sorted, it doesn't need
    # to go through the entire process

    swapped = False

    # Traverse through all array elements

    for i in range(n-1):

        # range(n) also work but outer loop will
        # repeat one time more than needed.

        # Last i elements are already in place

        for j in range(0, n-i-1):

            # traverse the array from 0 to n-i-1
```

```
        # Swap if the element found is greater
        # than the next element

        if arr[j] > arr[j + 1]:

            swapped = True

            arr[j], arr[j + 1] = arr[j + 1], arr[j]

    if not swapped:

        # if we haven't needed to make a single swap, we
        # can just exit the main loop.

        return

# Driver code to test above

arr = [64, 34, 25, 12, 22, 11, 90]

bubbleSort(arr)

print("Sorted array is:")

for i in range(len(arr)):

    print("% d" % arr[i], end=" ")

# This code is modified by Karan Korpe.
```

OUTPUT:

Sorted array:

11 12 22 25 34 64 90

Conclusion: Hence, we have successfully studied about program for implementing quick sort/ merge sort/ bubble sort.

EXPERIMENT NO. - 10

AIM: Write a function which takes a tuple as a parameter and returns a new tuple as the output, where every other element of the input tuple is copied, starting from the first one.

THEORY:

A tuple is a collection of objects which ordered and immutable. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example –

```
tup1 = ('physics', 'chemistry', 1997, 2000);  
tup2 = (1, 2, 3, 4, 5 );  
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing –

```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value –

```
tup1 = (50,);
```

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example –

```
#!/usr/bin/python  
  
tup1 = ('physics', 'chemistry', 1997, 2000);  
tup2 = (1, 2, 3, 4, 5, 6, 7 );  
print "tup1[0]: ", tup1[0];  
print "tup2[1:5]: ", tup2[1:5];
```

When the above code is executed, it produces the following result –

```
tup1[0]: physics  
tup2[1:5]: [2, 3, 4, 5]
```

PROGRAM A)

```
def oddTuples(aTup): #Function with tuple as an argument  
    rTup = () #Initially the output tuple rTup is empty  
    index = 0  
    while index < len(aTup):  
        rTup += (aTup[index],)  
        index += 2 #index increased by 2  
    return rTup  
t=(1, 3, 2, 4, 6, 5)  
print(oddTuples(t))
```


Output:

(1, 2, 6)

Explanation In the above program, initially a tuple 't' is created. This tuple 't' is passed as a parameter to a function. The while loop iterates till the length of the tuple. In each iteration, the number stored at an odd index is accessed and stored into the output tuple 'rTup'.

PROGRAM B)

```
#function definition
def odd_indices(tup):
    #create temporary tuples
    Odd_Tup = ()
    #iterate tuple
    for ind in range(len(tup)):
        #check if the element from the tuples gives the remainder
        if ind % 2 != 0:
            #if the number returns the remainder then that is an odd index
            #store that odd indexed value in a new tuple
            Odd_Tup += (tup[ind],)
    #return tuple
    return Odd_Tup

myTup = (1, 2, 3, 4, 5, 6)
print(odd_indices(myTup))
```

Explanation:

- The function takes one argument 'tup'.
- Create another temporary tuple for keeping track of odd indexed values from the 'tup'.
- Iterate tup using the loop, and the if statement check whether the index is odd or not.
- If the index gives a remainder then that is odd and store that index value in Odd_tup tuples
- Return the Odd_tup to the called function.

EXERCISE:

1. Write a function which takes a tuple as a parameter and returns a new tuple as the output, where every other element of the input tuple is copied, starting from the first one.

T = ('Hello', 'Are', 'You', 'Loving', 'Python?')

Output_Tuple = ('Hello', 'You', 'Python?')

Conclusion: Hence, we have successfully studied about program for writing a function which takes a tuple as a parameter and returns a new tuple as the output, where every other element of the input tuple is copied, starting from the first one.

EXPERIMENT NO. - 11

AIM: Write a function called how many, which returns the sum of the number of values associated with a dictionary.

THEORY:

Python dictionary is an ordered collection (starting from **Python 3.7**) of items. It stores elements in **key/value** pairs. Here, **keys** are unique identifiers that are associated with each **value**.

Let's see an example,

If we want to store information about countries and their capitals, we can create a dictionary with country names as **keys** and capitals as **values**.

Keys	Values
Nepal	Kathmandu
Italy	Rome
England	London

Create a dictionary in Python

Here's how we can create a dictionary in Python.

```
capital_city = {"Nepal": "Kathmandu", "Italy": "Rome", "England": "London"}  
print(capital_city)  
Run Code
```

Output

```
{'Nepal': 'Kathmandu', 'Italy': 'Rome', 'England': 'London'}
```

In the above example, we have created a dictionary named `capital_city`. Here,

1. **Keys** are `"Nepal"`, `"Italy"`, `"England"`

2. **Values** are "Kathmandu", "Rome", "London"

Note: Here, **keys** and **values** both are of string type. We can also have **keys** and **values** of different data types.

Example 1: Python Dictionary

```
# dictionary with keys and values of different data types
numbers = {1: "One", 2: "Two", 3: "Three"}
print(numbers)
Run Code
```

Output

```
[3: "Three", 1: "One", 2: "Two"]
```

In the above example, we have created a dictionary named `numbers`. Here, **keys** are of integer type and **values** are of string type.

Add Elements to a Python Dictionary

We can add elements to a dictionary using the name of the dictionary with `[]`. For example,

```
capital_city = {"Nepal": "Kathmandu", "England": "London"}
print("Initial Dictionary: ",capital_city)

capital_city["Japan"] = "Tokyo"

print("Updated Dictionary: ",capital_city)
Run Code
```

Output

```
Initial Dictionary: {'Nepal': 'Kathmandu', 'England': 'London'}
Updated Dictionary: {'Nepal': 'Kathmandu', 'England': 'London', 'Japan': 'Tokyo'}
```

In the above example, we have created a dictionary named `capital_city`. Notice the line,

```
capital_city["Japan"] = "Tokyo"
```

Here, we have added a new element to `capital_city` with **key:** Japan and **value:** Tokyo.

Change Value of Dictionary

We can also use `[]` to change the value associated with a particular key. For example,

```
student_id = {111: "Eric", 112: "Kyle", 113: "Butters"}
```

```
print("Initial Dictionary: ", student_id)
```

```
student_id[112] = "Stan"
```

```
print("Updated Dictionary: ", student_id)
```

Run Code

Output

```
Initial Dictionary: {111: 'Eric', 112: 'Kyle', 113: 'Butters'}
```

```
Updated Dictionary: {111: 'Eric', 112: 'Stan', 113: 'Butters'}
```

In the above example, we have created a dictionary named `student_id`. Initially, the value associated with the key `112` is `"Kyle"`. Now, notice the line,

```
student_id[112] = "Stan"
```

Here, we have changed the value associated with the key `112` to `"Stan"`.

Accessing Elements from Dictionary

In Python, we use the keys to access their corresponding values. For example,

```
student_id = {111: "Eric", 112: "Kyle", 113: "Butters"}
```

```
print(student_id[111]) # prints Eric
```

```
print(student_id[113]) # prints Butters
```

Run Code

Here, we have used the keys to access their corresponding values.

If we try to access the value of a key that doesn't exist, we'll get an error. For example,

```
student_id = {111: "Eric", 112: "Kyle", 113: "Butters"}
```

```
print(student_id[211])
```

```
# Output: KeyError: 211
```

Run Code

Removing elements from Dictionary

We use the `del` statement to remove an element from the dictionary. For example,

```
student_id = {111: "Eric", 112: "Kyle", 113: "Butters"}
```

```
print("Initial Dictionary: ", student_id)
```

```
del student_id[111]
```

```
print("Updated Dictionary ", student_id)
```

Run Code

Output

```
Initial Dictionary: {111: 'Eric', 112: 'Kyle', 113: 'Butters'}
```

```
Updated Dictionary {112: 'Kyle', 113: 'Butters'}
```

Here, we have created a dictionary named `student_id`. Notice the code,

```
del student_id[111]
```

The `del` statement removes the element associated with the key `111`.

We can also delete the whole dictionary using the `del` statement,

```
student_id = {111: "Eric", 112: "Kyle", 113: "Butters"}
```

```
# delete student_id dictionary
```

```
del student_id
```

```
print(student_id)
```

```
# Output: NameError: name 'student_id' is not defined
```

Run Code

We are getting an error message because we have deleted the `student_id` dictionary

and `student_id` doesn't exist anymore.

Iterating Through a Dictionary

We can iterate through each key in a dictionary using a `for` loop.

```
# Iterating through a Dictionary
```

```
squares = {1: 1, 3: 9, 5: 25, 7: 49, 9: 81}
```

```
for i in squares:
```

```
print(squares[i])
```

Run Code

Output

```
1
9
25
49
81
```

Here, we have iterated through each **key** in the `squares` dictionary using the `for`

EXERCISE:

Write a function called `how_many`, which returns the sum of the number of values associated with a dictionary.

```
T= animals = {'L':['Lion'],'D':['Donkey'],'E':['Elephant']}
```

```
>>>print(how_many(animals))
```

Conclusion: Hence, we have successfully studied about program for writing a function called `how_many`, which returns the sum of the number of values associated with a dictionary.