Unit-II

Definite Iteration: The for Loop

There are two types of loops-
1. count-controlled loop
2. condition-controlled loop

1. count-controlled loop:

A count controlled loop is often called the **definite repetition** loop as the number of repetitions is known before the loop begins executing. When we do not know in advance the number of times we want to execute a statement, we cannot use count-controlled repetition.

A count-controlled loop iterates a specific number of times. In Python, generally for statement is used to write a count-controlled loop(definite repletion loop). The for loop is an easy and convenient control statement for describing a definite iteration.

Python provides a built-in function named range( ) that simplifies the process of writing a count controlled for loop.

range( ) function:

range() is a built-in function of python which returns a range object. The range() function generates the integer numbers between the given start integer to the stop integer, which is generally used to iterate over with for Loop

syntax:

range (start, stop[, step])

The range( ) function takes three arguments. Out of the three, 2 arguments are optional. i.e., Start and Step are the optional arguments.

- A start argument is a starting number of the sequence. i.e., lower limit. By default, it starts with 0 if not specified.
- A stop argument is an upper limit. i.e. generate numbers up to this number(doesn't include this number in the result.)
- The step is a difference between each number in the result. The default value of the step is 1 if not specified.

Ex:

Program	output
for i in range(5): print(i, end=' ')	0 1 2 3 4

Program	output
for i in range(5): print("hello world")	hello world hello world hello world hello world hello world

Program	output
for i in range(2,10): print(i,end=' ')	2 3 4 5 6 7 8 9

Program	output
for i in range(2,10,2): print(i,end=' ')	2 4 6 8

Program	output
for count in range(10, 0, -1): print(count, end = " ")	10 9 8 7 6 5 4 3 2 1

Program	output
import random for roll in range(10): print(random.randint(1, 6), end = " ")	Output1: 6 1 1 3 2 5 2 3 3 1

Ex: write a program that displays the numbers 1 through 10 and their squares

Program:

```
print('Number\tSquare')
print('--------------')

#Print the numbers 1 through 10 and their squares

for number in range(1, 11):
    square = number**2
    print(number, '\t', square)
```

Output:

```
Number   Square
--------------
1          1
2          4
3          9
4          16
5          25
6          36
7          49
8          64
9          81
10         100
```

Count Control with a while Loop:

while loop can also be used as a count-controlled loop.

Program	output
#program to illustrate count controlled while loop # Summation with a while loop theSum = 0 count = 1 while count <= 100000: theSum += count count += 1 print("The sum is:",theSum)	The sum is: 5000050000

Program	output
#Counting down with a while loop count = 10 while count >= 1: print(count, end = " ") count -= 1	10 9 8 7 6 5 4 3 2 1

Write a program to compute the sum of even numbers between 1 and 10 using for loop

Sol:
#program to compute the sum of the even numbers between 1 and 10 Sum = 0 for count in range(2, 11, 2):   Sum += count print("the required sum is",Sum)
Output: the required sum is 30

Write a loop that reads a string and prints each character in the string

Sol:
str1=input("enter the string:") print("the characters of the string are:") for character in str1:   print(character)
Output: enter the string: python the characters of the string are: p y t h o n

Formatting Text for Output:

Many data-processing applications require output that has a tabular format, where numbers and other information are aligned in columns that can be either left-justified or right-justified.

The total number of data characters and additional spaces for a given datum in a **formatted string** is called its field width. When we format **floating-point numbers** for output, we often would like to specify the number of digits of precision to be displayed as well as the field width.

A format string and its operator % allow the programmer to format data using a field width and a precision.

Python includes a general formatting mechanism that allows the programmer to specify field widths for different types of data.

To format a sequence of data values, construct a format string that includes a format specifier for each datum and place the data values in a tuple following the % operator.

The form of this operation is as follows:

<format string> % (<datum–1>, ..., <datum–*n*>)

The format specifiers like %s ,%d etc, can be used to specify the format string

To format the **string data** value either left justified or right justified, the notation

%<field width>s

can be used in the format string. When the field width is positive, the datum is right-justified; when the field width is negative, you get left-justification. If the field width is less than or equal to the datum's print length in characters, no justification is added.

Ex:
The code that right-justify and left-justify the **string** "four" with a **field width of 6** is given below.

```
>>> "%6s" % "four"                    # Right justify
' four'
```

```
>>> "%-6s" % "four"                    # Left justify
'four '
```

To format **integer data**, the notation

%<field width>d

can be used in the format string.

Ex:

Program:

```
for exponent in range(7, 11):
    print("%-3d%12d" % (exponent, 10 ** exponent))
```

Output:

```
7       10000000
8      100000000
9     1000000000
10   10000000000
```

To format **float data**, the notation

%<field width>.<precision>f

where .<precision> is optional.

Ex:	Output
salary = 100.00 print("salary is $%0.2f" % salary)	salary is $100.00

Ex:

Write a code segment that reads the values of the integers x,y,z and display them such that each value is left-justified in a field width of 6 in a single line.

Program:

```
x,y,z = map(int,input("enter three integers separated by comma:").split(","))
print("%-6d %-6d %-6d" %(x,y,z))
```

Output:

```
enter three integers separated by comma:13,24,100
13     24     100
```

Ex:

Write a loop that outputs the numbers in a list named salaries. The outputs should be formatted in a column that is right-justified, with a field width of 12 and a precision of 2.

Program

```
salaries = [1,2,3,4,5,6,7,8]
for item in salaries:
   print("%12.2f" % item)
```

Output:

```
        1.00
        2.00
        3.00
        4.00
        5.00
        6.00
        7.00
        8.00
```

Selection Statements:

1. if statement:

The simplest form of selection statement is the **if statement**. This type of control statement is also called a **one-way selection statement**, because it consists of a condition and just a single sequence of statements. If the condition is True, the sequence of statements is run. Otherwise, control proceeds to the next statement following the entire selection statement.

Program	output
#program to illustrate simple if statement #program to print the absolute value of a number num=eval(input("enter a number")) if(num<0): num=-num print("absolute value of a number is",num)	Output1: enter a number5 absolute value of a number is 5 Output2: enter a number-45 absolute value of a number is 45

2.if-else statement:

The **if-else statement** is also called a **two-way selection statement**, because it directs the computer to make a choice between two alternative courses of action.

```
#program to find maximum and minimum among two integers
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
if first > second:
   maximum = first
   minimum = second
else:
   maximum = second
   minimum = first
print("Maximum:", maximum)
print("Minimum:", minimum)
```

Output:

```
Enter the first number: 34
Enter the second number: 100
Maximum: 100
Minimum: 34
```

3. if…elif…else statement:

The statement if….elif….else statement is also called as **multway selection statement** because it allows for testing of several conditions and responding accordingly. The syntax is

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
statement-x
```

Ex:

program

```
number = int(input("Enter the numeric grade: "))
if number > 89:
    letter = 'A'
elif number > 79:
    letter = 'B'
elif number > 69:
    letter = 'C'
else:
    letter = 'F'
print("The letter grade is", letter)
```

Output:

```
Enter the numeric grade: 97
The letter grade is A
```

Logical Operators:

Python includes all three Boolean or logical operators, and, or, and not.

The logical and operator and the logical or operator allow you to connect multiple Boolean expressions to create a compound Boolean expression. The logical not operator reverses the truth of a Boolean expression. They can be described as follows.

Operator	Meaning
and	The and operator connects two Boolean expressions into one compound expression. Both subexpressions must be true for the compound expression to be true.
or	The or operator connects two Boolean expressions into one compound expression. One or both subexpressions must be true for the compound expression to be true. It is only necessary for one of the subexpressions to be true, and it does not matter which.
not	The not operator is a unary operator, meaning it works with only one operand. The operand must be a Boolean expression. The not operator reverses the truth of its operand. If it is applied to an expression that is true, the operator returns false. If it is applied to an expression that is false, the operator returns true.

The behaviour of each operator can be completely specified in a **truth table** for that operator.

A	B	A and B
True	True	True
True	False	False
False	True	False
False	False	False

A	B	A or B
True	True	True
True	False	True
False	True	True
False	False	False

A	not A
True	False
False	True

Short-circuit Evaluation:

Sometimes the value of a Boolean expression is known before it has evaluated all of its operands.

The approach where the evaluation of Boolean expression is stopped as soon as the truth value of the Boolean expression has been determined without evaluating all of its operands is called short circuit evaluation. The evaluation of expression takes place from left to right.

Both the logical and and logical or operators perform short-circuit evaluation. The short-circuit evaluation working with the logical and operator is given here.

If the expression on the left side of the and operator is false, the expression on the right side will not be checked. Because the compound expression will be false if only one of the subexpressions is false, it would waste CPU time to check the remaining expression. So, when the and operator finds that the expression on its left is false, it shortcircuits and does not evaluate the expression on its right.

For ex, in the expression 'A and B', if A is false, then the expression becomes false, and so B does not get evaluated.

Similarly in the expression 'A or B', if A is true, then so is the expression, and B does not get evaluated.

Conditional Iteration –while loop:

condition-controlled loop:

A condition-controlled loop uses a true/false condition to control the number of times that it repeats. This is also called **indefinite repetition** loop as the number of repetitions is not known before the loop begins executing.

In Python, generally while loop is used to write a condition-controlled loop (indefinite repletion loop).

Program

```
#program to illustrate condition controlled loop
#prompts the user for a series of numbers,computes their sum,and outputs the result.
theSum = 0.0
data = input("Enter a number or just enter to quit: ")
while data != "":
    number = float(data)
    theSum += number
    data = input("Enter a number or just enter to quit: ")
print("The sum is", theSum)
```

Output1:

```
Enter a number or just enter to quit: 2
Enter a number or just enter to quit: 3
Enter a number or just enter to quit: 4
Enter a number or just enter to quit: 5
Enter a number or just enter to quit: 6
Enter a number or just enter to quit: 7
Enter a number or just enter to quit:
The sum is 27.0
```

Output2:

```
Enter a number or just enter to quit: 4
Enter a number or just enter to quit:
The sum is 4.0
```

Program
#program to illustrate condition controlled loop theSum = 0.0 while True: data = input("Enter a number or just enter to quit: ")   if data == "":   break   number = float(data)   theSum += number print("The sum is", theSum)
Output: Enter a number or just enter to quit: 4 Enter a number or just enter to quit: 2 Enter a number or just enter to quit: 3 Enter a number or just enter to quit: The sum is 9.0

Strings:

A string is a sequence of zero or more characters enclosed in single quotation marks('...') or double quotation marks("...").

A string's length is the number of characters it contains. Python's len( ) function returns this value when it is passed a string.

Ex:

```
>>> len("python programming")
18

>>> len("")   #length of empty string
0
```

The string is an **immutable data structure**. This means that its characters, can be accessed, but cannot be replaced, inserted, or removed.

Accessing characters in a string:

The individual characters of a string can be accessed using indexing and a range of characters can be accessed using slicing. The index must be an integer and Index starts from 0. Python also allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

Trying to access a character out of index range will raise an IndexError.

Ex:

```
>>>str = 'python programming'
>>>print('str = ', str)
str =  python programming

>>>str = 'python programming'
>>>print('str[0] = ', str[0])
str[0] =  p

>>>str = 'python programming'
>>>print('str[-1] = ', str[-1])
str[-1] =  g

>>>str = 'python programming'
>>>print('str[1:5] = ', str[1:5])
str[1:5] =  ytho

>>>str = 'python programming'
>>>print('str[5:-2] = ', str[5:-2])
str[5:-2] =  n programmi
```

write a program that reads the string and display the characters and their positions

Sol:

```
#count controlled loop to display the characters and their positions
data=input("enter the string:")
for index in range(len(data)):
    print(index,data[index])
```

Output:

```
enter the string:python
0 p
1 y
2 t
3 h
4 o
5 n
```

Write a program that traverses a list of filenames and prints just the filenames that have a '.**txt** 'extension

Sol:

```
fileList = ["myfile.txt", "myprogram.exe", "yourfile.txt"]
for fileName in fileList:
    if ".txt" in fileName:
        print(fileName)
```

Output:

```
myfile.txt
yourfile.txt
```

Data Encryption:

Data travelling on the Internet is vulnerable to spies and potential thieves. For this reason, most applications now use **data encryption** to protect information transmitted on networks

In cryptography, encryption is the process of encoding information. This process converts the original representation of the information, known as plaintext, into an alternative form known as cipher text.

The sender **encrypts** a message by translating it to a secret code, called a **cipher text**. At the other end, the receiver **decrypts** the cipher text back to its original **plaintext** form. Both sender and receiver use one or more **keys** that allow them to encrypt and decrypt messages

Encryption is important because it allows to securely protecting data so that any unauthorized person cannot have access to it.

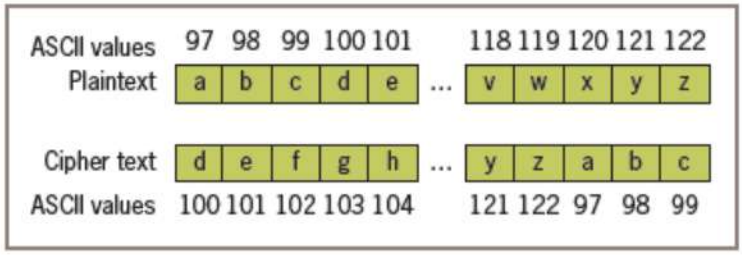
A very simple encryption method is a **Caesar cipher**.

This encryption strategy replaces each character in the plaintext with the character that occurs a given distance away in the sequence. (Assuming character set for text is ordered as a sequence of distinct values.). For positive distances, the method wraps around to the beginning of the sequence to locate the replacement characters for those characters near its end.

For example, if the distance value of a Caesar cipher equals three characters, the string "invaders" would be encrypted as "lqydghuv".

To decrypt this cipher text back to plaintext, you apply a method that uses the same distance value but looks to the left of each character for its replacement. This decryption method wraps around to the end of the sequence to find a replacement character for one near its beginning.

The following Figure shows the first five and the last five plaintext characters of the lowercase alphabet and the corresponding cipher text characters, for a Caesar cipher with a distance of +3.
The numeric ASCII values are listed above and below the characters.



Python programs that implement Caesar cipher encryption and decryption technique for any strings that contain the lowercase letters of the alphabet and for any distance values between 0 and 26 are given below.

Caesar cipher encryption technique

```
"""
Encrypts an input string of lowercase letters and prints
the result. The other input is the distance value.
"""
plainText = input("Enter a one-word, lowercase message: ")
distance = int(input("Enter the distance value: "))
code = ""
for ch in plainText:
    ordvalue = ord(ch)
    cipherValue = ordvalue + distance
    if cipherValue > ord('z'):
      cipherValue = ord('a') + distance - (ord('z') - ordvalue + 1)
    code += chr(cipherValue)
print(code)
```

Output:

```
Enter a one-word, lowercase message: invaders
Enter the distance value: 3
lqydghuv
```

Caesar cipher decryption technique

```
"""
Decrypts an input string of lowercase letters and prints
the result. The other input is the distance value.
"""
code = input("Enter the coded text: ")
distance = int(input("Enter the distance value: "))
plainText = ""
for ch in code:
    ordvalue = ord(ch)
    cipherValue = ordvalue - distance
    if cipherValue < ord('a'):
        cipherValue = ord('z') - distance +(ord('a') - ordvalue - 1)
    plainText += chr(cipherValue)
print(plainText)
```

Output:

Enter the coded text: lqydghuv
Enter the distance value: 3
invaders

Strings and Number Systems :

Number systems are the technique to represent numbers in the computer system architecture, every value that you are saving or getting into/from computer memory has a defined number system. Computer architecture supports following number systems.

- **Binary number system**
- **Octal number system**
- **Decimal number system**
- **Hexadecimal (hex) number system**

BINARY NUMBER SYSTEM

A Binary number system has only two digits that are **0 and 1**. Every number (value) represents with 0 and 1 in this number system. The base of binary number system is 2, because it has only two digits.

OCTAL NUMBER SYSTEM

Octal number system has only eight (8) digits from **0 to 7**. Every number (value) represents with 0,1,2,3,4,5,6 and 7 in this number system. The base of octal number system is 8, because it has only 8 digits.

DECIMAL NUMBER SYSTEM

Decimal number system has only ten (10) digits from **0 to 9**. Every number (value) represents with 0,1,2,3,4,5,6, 7,8 and 9 in this number system. The base of decimal number system is 10, because it has only 10 digits.

HEXADECIMAL NUMBER SYSTEM

A Hexadecimal number system has sixteen (16) alphanumeric values from **0 to 9** and **A to F**. Every number (value) represents with 0,1,2,3,4,5,6, 7,8,9,A,B,C,D,E and F in this number system. The base of hexadecimal number system is 16, because it has 16 alphanumeric values. Here A is 10, B is 11, C is 12, D is 14, E is 15 and F is 16.

The following table summarizes the different number systems

Number system	Base(Radix)	Used digits	Example
Binary	2	0,1	$(11110000)_2$
Octal	8	0,1,2,3,4,5,6,7	$(360)_8$
Decimal	10	0,1,2,3,4,5,6,7,8,9	$(240)_{10}$
Hexadecimal	16	0,1,2,3,4,5,6,7,8,9, A,B,C,D,E,F	$(F0)_{16}$

Conversions:

Converting Binary to Decimal:

PROGRAM:

```
#converting binary number to decimal

binary = input('Enter a Binary number: ') #bit string
decimal = 0

#checking whether input is binary or not
flag=1
for digit in binary:
    if digit!='0' and digit!='1':
        flag=0
if flag==0:
    print("entered number is not a binary number")
else:

    #length of binary input
    exponent = len(binary)

    #loop through each digit of binary
    for digit in binary:
        exponent = exponent-1
        decimal = decimal+( pow(2,exponent) * int(digit))

    print("the decimal value is:",decimal)

#another way
dec=int(binary,2)
print("the decimal value is:",dec)
```

Output:

```
Enter a Binary number: 1111
the decimal value is: 15
the decimal value is: 15
```

Converting Decimal to Binary:

program

```
"""
File: decimaltobinary.py
Converts a decimal integer to a string of bits.
"""
decimal = int(input("Enter a decimal integer: "))
if decimal == 0:
   bitstring=0
else:
   bitstring = ""
   while decimal > 0:
      remainder = decimal % 2
      decimal = decimal // 2
      bitstring = str(remainder) + bitstring

print("The binary representation is", bitstring)
```

Output:

```
Enter a decimal integer: 34
The binary representation is 100010
```

String methods:

capitalize( ):

returns the capitalized version of the string i.e., make the first character have upper case and the rest lower case

title( ):

Return a title cased version of the string, i.e., all words start with uppercase and rest of the characters in words are in lowercase.

lower( ):
Return a copy of the string converted to lowercase.

upper( ):
Return a copy of the string converted to uppercase.

swapcase( ):
Return a copy of the string with uppercase characters converted to lowercase and vice versa.

count(sub[, start[, end]]) :

Return the number of non-overlapping occurrences of substring sub in string in the range [start:end]. Optional arguments start and end are interpreted as in slice notation. String Search is case-sensitive.

find(sub[, start[, end]]) :

Return the lowest index in the string where substring sub is found, such that sub is contained within the string in the range [start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure.

rfind(sub[, start[, end]]) :

Return the highest index in the string where substring sub is found, such that sub is contained within the string in the range [start:end]. Optional arguments start and end are interpreted as in slice notation. Return -1 on failure

index(sub[, start[, end]]) :

Return the lowest index in the string where substring sub is found, such that sub is contained within the string in the range [start:end]. Optional arguments start and end are interpreted as in slice notation. **Raises ValueError when the substring is not found.(same as find but return value on failure different)**

rindex(sub[, start[, end]]) :

Return the highest index in the string where substring sub is found, such that sub is contained within the string in the range [start:end]. Optional arguments start and end are interpreted as in slice notation

replace(old, new[, count]):

Return a copy of S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

Ex:

```
>>> var='This is a good example'
>>> str='was'
>>> var.replace('is',str)
'Thwas was a good example'
>>> var.replace('is',str,1)
'Thwas is a good example'
```

split(sep=None, maxsplit=-1) :

Return a list of the words in the string, 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.

Ex:

```
>>> var='This is a good example'
>>> var.split(sep=' ')
['This', 'is', 'a', 'good', 'example']
>>> var.split(' ')
['This', 'is', 'a', 'good', 'example']
>>> var.split(sep='@')
['This is a good example']
>>> var.split(sep=' ',maxsplit=2)
['This', 'is', 'a good example']

>>> var.split(sep=' ',maxsplit=3)
['This', 'is', 'a', 'good example']

>>> "myfile.txt".split('.')
['myfile', 'txt']
```

Program that reads the date in the form dd/mm/yyyy and split the date
Create a string with a date. date_string=input("enter the date(dd/MM/yyyy):") # Split the date. date_list = date_string.split('/') # Display each piece of the date. print('Day:', date_list[0]) print('Month:', date_list[1]) print('Year:', date_list[2])
Output: enter the date(dd/MM/yyyy):08/12/2020 Day: 08 Month: 12 Year: 2020

join(iterable):

Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.

Ex:

```
>>> list1=['This', 'is', 'a', 'good', 'example']
>>> list1
['This', 'is', 'a', 'good', 'example']
>>> string=" "
>>> string.join(list1)
'This is a good example'
```

istitle( ):

Return True if S is a titlecased string and false otherwise

Ex:

```
>>>string=”this is a good example”
>>>string.istitle( )
False
```

startswith(substring):

The startswith method determines whether a string starts with a specified substring.The substring argument is a string. The method returns true if the string starts with substring.

endswith(substring):

The endswith method determines whether a string ends with a specified substring.The substring argument is a string. The method returns true if the string ends with substring.

Ex:

Program that reads the filename and checks whether it is textfile, pythonfile, word processing document

```
filename = input('Enter the filename with extension: ')
if filename.endswith('.txt'):
   print('That is the name of a text file.')
elif filename.endswith('.py'):
   print('That is the name of a Python source file.')
elif filename.endswith('.doc'):
   print('That is the name of a word processing document.')
else:
   print('Unknown file type.')
```

Output:

```
Enter the filename with extension: ex1.py
That is the name of a Python source file.
```

Ex: write a python program to print the following pattern

```
p
py
pyt
pyth
pytho
python
```

sol:

Program:	Output:
name = input('Enter your name: ') for i in range(len(name)): print(name[:i+1], end='\n')	Enter your name: python p py pyt pyth pytho python

Some more string methods:

Method	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters, and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters, and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`lstrip()`	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string.
`rstrip()`	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string.
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.