Introduction to Loops

In Python, loops are control structures used to repeat a block of code multiple times until a certain condition is met. There are two main types of loops in Python: **for** loops and **while** loops.

Introduction to While Loops

The while loop is somewhat similar to an **if** statement, it executes the code block inside if the expression/condition is **True.** However, as opposed to the **if** statement, the while loop continues to execute the code repeatedly, as long as the expression is **True**. In other words, a while loop iterates over a block of code.

In Python, the body of a **while** loop is determined by the indentation. It starts with indentation and ends at the first unindented line.

The most important part of a **while** loop is the looping variable. This looping variable controls the flow of iterations. An increment or decrement in this looping variable is important for the loop to function. It determines the next iteration level. In case of the absence of such increment/decrement, the loop gets stuck at the current iteration and continues forever until the process is manually terminated.

The general syntax of a **while** loop is similar to an **if** statement with a few differences. It is shown below:

```
while(Expression/Condition/Boolean):

    <Execute this code block till the Expression is True>

    #Increment/Decrement in looping variable
```

Problem Statement: Given an Integer n, print the first “n” natural numbers.

```
n = 10
i=1 #Initialising the looping variable to 1
while i<=n:#The loop will continue till the value of i<number
    print(i)
    i+=1 #Value of i is updated at the end of every iteration
```

We get the output as:

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

Problem Statement: Given an Integer n, find the sum of the first n natural numbers.

```
n = 4
sum = 0
i = 1 #Initialising the looping variable to 1
while (i<=n): #The loop will continue till the value of i<number
        sum = sum + i
        i = i+1 #Value of i is updated at the end of every iteration
print(sum)
```

We get the output as:

```
10
```

Break Statement

The break statement enables a program to skip over a part of the code. A break statement terminates the very loop it lies within. The working of a break statement is as follows:

```
while <Expression/Condition/Statement>:
      #Statement1
      if <condition1>:
            break #If "condition" is true, then code breaks out
      #Statement2
      #Statement3
#Statement4: This statement is executed if it breaks out of the
loop
#Statement5
```

For Example :

```
n=5
i=1
while i<=n:
    if i==4:#If "condition" is true, then code breaks out
        break
    print(i)
    i=i+1
```

We get the output as:

```
1
2
3
```

Continue Statement

The continue statement jumps out of the current iteration and forces the next iteration of the loop to take place.
The working of a continue statement is as follows:

```
while <Expression/Condition/Statement>:
     #Statement1
     if <condition1>:
          continue
     #Statement2
     #Statement3
#Statement4: This statement is executed if it breaks out of
the loop
#Statement5
```

In the given code snippet, if **condition1** is true, the **continue** statement will cause the skipping of Statement2 and Statement3 in the current iteration, and the next iteration will start.

The following code fragment shows you an example of the **continue** statement:

```
n=5
i=1
while i<=n:
    if i==4:
        i=i+1
        continue
    print(i)
    i=i+1
```

We get the output as:

```
1
2
3
5
```

Pass Statement

The pass statement is a null statement.

- It is generally used as a placeholder for future code i.e. in cases where you want to implement some part of your code in the future but you cannot leave the space blank as it will give a compilation error.
- Sometimes, pass is used when the user does not want any code to execute.
- Using the pass statement in loops, functions, class definitions, and if statements, is very useful as empty code blocks are not allowed in these.

The syntax of a pass statement is:

```
pass
```

Given below is a basic implementation of a conditional using a pass statement:

```
n=2
if n==2:
   pass #Pass statement: Nothing will happen
else:
   print ("Executed")
```

In the above code, the if statement condition is satisfied because n==2 is True. Thus there will be no output for the above code because once it enters

the if statement block, there is a pass statement. Also, no compilation error will be produced.

Consider another example:

```
n=1
if n==2:
   print ("Executed")
else:
   pass #Pass statement: Nothing will happen
```

In the above code, the **if** statement condition is not satisfied because `n==2` is **False**. Thus there will be no output for the above code because it enters the **else** statement block and encounters the **pass** statement.

Check Prime: Using While Loop and Nested If Statements

Problem Statement: Given any Integer, check whether it is Prime or Not.

Approach to be followed: A prime number is always positive so we are checking that at the beginning of the program. Next, we are dividing the input number by all the numbers in the range of 2 to (number - 1) to see whether there are any positive divisors other than 1 and the number itself (Condition for Primality). If any divisor is found then we display, “Is Prime”, else we display, “Is Not Prime”.

Note:- We are using the break statement in the loop to come out of the loop as soon as any positive divisor is found as there is no further check required. The purpose of a break statement is to break out of the current iteration of the loop so that the loop stops. This condition is useful, as once we have found a positive divisor, we need not check for more divisors and hence we can break out of the loop. You will study about the break statement in more detail in the latter part of this course.

```
# taking input from the user
number = int(input("Enter any number: "))

isPrime= True #Boolean to store if number is prime or not
if number > 1: # prime number is always greater than 1
    i=2
    while i< number:
        if (number % i) == 0: # Checking for positive
divisors
            isPrime= False
            break
        i=i+1

if(number<=1): # If number is less than or equal to 1
    print("Is Not Prime")
elif(isPrime): # If Boolean is true
    print("Is Prime")
else:
    print("Is Not Prime")
```

range() Function

The range() function in Python generates a List which is a special sequence type. A sequence in Python is a succession of values bound together by a single name.

The syntax of the range() function is given below:

```
range(<Lower Limit>, <Upper Limit>) # Both limits are
integers
```

The function, range(L, U), will produce a list having values L, L+1, L+2....(U-1).

Note: The lower limit is included in the list but the upper limit is not.

For Example:

```
range(0,5)
[0,1,2,3,4] #Output of the range() function
```

Note: The default step value is +1, i.e. the difference in the consecutive values of the sequence generated will be +1.

If you want to create a list with a step value other than 1, you can use the following syntax:

```
range(<Lower Limit>, <Upper Limit>, <Step Value>)
```

The function, range(L, U, S), will produce a list [L, L+S, L+2S...<= (U-1)].

For Example:

```
range(0,5,2)
[0,2,4] #Output of the range() function
```

For Loop

The for loop of Python is designed to process the items of any sequence, such as a list or a string, one by one. The syntax of a for loop is:

```
for <variable> in range(<sequence>):
     Statements_to_be_executed
```

For example, consider the following loop:

```
for i in range(1,6):
     print(i)
```

Output :

```
1
2
3
4
5
```

Factors: Using For Loop

Problem Statement: Given any Integer, print all the factors between 2 to N.

```
n = 6

for i in range(2, n):
    if n % i == 0:
        print(i,end=" ")
```

```
Output : 2 3
```

Note: for loop that iterates through numbers from 2 to n - 1 (inclusive). It starts from 2 because 1 is always a factor of any number, and we're excluding 1 and n from the list of factors. Secondly, check if the current number i is a factor of n by using the modulo operator (%). If n is divisible by i without leaving a remainder, then i is a factor of n.

Fibonacci series: Using Loop

Problem Statement: Write a Python program to generate the Fibonacci series up to N.

Note : Fibonacci series is a sequence of numbers where each number is the sum of the two preceding ones, usually starting with 0 and 1.

```
n = 5

second_last_num = 0
last_num = 1

print(second_last_num)
print(last_num)

count = 2 # this denotes how many values we printed already


while(count < n):
```

```
    next_fib = last_num + second_last_num
    print(next_fib)

    second_last_num = last_num
    last_num = next_fib
    Count +=1
```

Output:

```
0
1
1
2
3
```

Decimal to Binary Conversion

Problem Statement: Write a Python program to convert Decimal number to Binary number.

Note : Initialize an empty string ans to store the binary representation.

While n is greater than 0:

- If n is even (i.e., n % 2 == 0), append "0" to the beginning of ans.
- If n is odd (i.e., n % 2 != 0), append "1" to the beginning of ans.
- Divide n by 2 using floor division n //= 2.

Print the binary representation stored in ans.

```
n = 10
ans = ""
while n > 0:
    if n%2 == 0:
        # even
        ans = "0" + ans
    else:
        # odd
```

```
        ans = "1" + ans
    n //= 2
print(ans)
```

Output :

```
1010
```

Binary to Decimal Conversion

Problem Statement: Write a Python program to convert Binary number to Decimal number.

Note: Initialize ans to store the final decimal value and pow_of_2 as 1, representing the current power of 2.

While n is greater than 0:

- Extract the last digit of n using modulo 10 (n % 10), which gives the rightmost digit.
- Multiply the last digit by the current power of 2 and add it to ans.
- Multiply pow_of_2 by 2 to update the power for the next iteration.
- Update n by removing the last digit using integer division n //= 10.

Print the decimal representation stored in ans.

```
n = 1010

ans = 0 # final decimal will be stored
pow_of_2 = 1

while n > 0:
    last_digit = n % 10 # remainder
    ans += last_digit * pow_of_2
```

```
    pow_of_2 *= 2
    n //= 10 # eliminate the last bit

print(ans)
```

Output :

```
10
```