More On Loops

`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
```

`in` Operator

The `in` operator tests if a given value is contained in a sequence or not and returns `True` or `False` accordingly. For eg.,

```
3 in [1,2,3,4]
```

will return `True` as value 3 is contained in the list.

```
"A" in "BCD"
```

will return `False` as `"A"` is not contained in the String `"BCD"`.

`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 <sequence>:
      Statements_to_be_executed
```

For example, consider the following loop:

```
for a in [1, 4, 7]:
      print(a)
```

The given `for` loop will be processed as follows:

1. Firstly, the looping variable `a` will be assigned the first value from the list i.e. 1, and the statements inside the for loop will be executed with this value of `a`. Hence 1 will be printed.
2. Next, `a` will be assigned 4 and 4 will be printed,
3. Finally, `a` will be assigned 7, and 7 will be printed.
4. All the values in the list are executed, hence the loop ends.

Check Prime: Using For Loop

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".**

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

isPrime= True
if number > 1: # prime number is always greater than 1
    for i in range(2, number):
        if (number % i) == 0: # Checking for positive divisors
            isPrime= False
            break

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

Jump Statements: `break` and `continue`

Python offers two jump statements (within loops) to jump out of the loop iterations. These are `break` and `continue` statements. Let us see how these statements work.

`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
```

In the given code snippet, if `condition1` is true, then the flow of execution will break out of the loop. The next statement to be executed will be `Statement4`.

Note: This loop can terminate in 2 ways:

1. Throughout the iterations of the loop, if `condition1` remains false, the loop will go through its normal course of execution and stops when its `condition/expression` becomes false.
2. If during any iteration, `condition1` becomes true, the flow of execution will break out of the loop and the loop will terminate.

The following code fragment shows you an example of the `break` statement:

```
for val in "string":
    if val == "i":
        break
    print(val)
print("The end")
```

The output will be:

```
s
t
r
```

Here, as soon as the value of `val` becomes equal to `"i"`, the loop terminates.

`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:

```
for val in "string":
    if val == "i":
        continue
    print(val)
print("The end")
```

The output will be:

```
s
t
r
n
g
The end
```

Here, the iteration with `val = "i"`, gets skipped and hence `"i"` is not printed.

`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.