Chapter 4: Loops

Loops are repetition structures, generally used to process data collections, such as lines of a file or records of a database that must be processed by the same code block.

The range() function

We can generate a sequence of numbers using range() function. range(10) will generate numbers from 0 to 9 (10 numbers).

We can also define the start, stop and step size as `range(start,stop,step size)`. step size defaults to 1 if not provided.

This function does not store all the values in memory, it would be inefficient. So it remembers the start, stop, step size and generates the next number on the go.

To force this function to output all the items, we can use the function list().

```
# Output: range(0, 10)
print(range(10))

# Output: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(range(10)))

# Output: [2, 3, 4, 5, 6, 7]
print(list(range(2, 8)))

# Output: [2, 5, 8, 11, 14, 17]
print(list(range(2, 20, 3)))

print(list(range(20, 2, -3)))
```

```
range(0, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[2, 3, 4, 5, 6, 7]
[2, 5, 8, 11, 14, 17]
[20, 17, 14, 11, 8, 5]
```

For

It is the repetition structure most often used in Python. The statement accepts not only static sequences, but also sequences generated by iterators. Iterators are structures that allow iterations, i.e. access to items of a collection of elements, sequentially.

During the execution of a *for* loop, the reference points to an element in the sequence. At each iteration, the reference is updated, in order for the *for* code block to process the corresponding element.

The clause *break* stops the loop and *continue* passes it to the next iteration. The code inside the `else` is executed at the end of the loop, except if the loop has been interrupted by *break*.

Syntax:

```
for <reference> in <sequence>:
    <code block>
    continue
    break
else:
    <code block>
```

Example:

```
# Sum 0 to 99
s = 0
for x in range(100, 1, -5):
    print(x)
    s = s + x
print("sum of 0 to 99 is", s)
```

```
100
95
90
85
80
75
70
65
60
55
50
```

```
45
40
35
30
25
20
15
10
5
sum of 0 to 99 is 1050
```

```
cols = ["Red", "Green", "Yellow", "White"]
for color in cols:
    print(color)
```

```
Red
Green
Yellow
White
```

```
color = {"c1": "Red", "c2": "Green", "c3": "Orange"}
for value in color.values():
    print(value)
```

```
Red
Green
Orange
```

```
color = {"c1": "Red", "c2": "Green", "c3": "Orange"}
for col in color:
    print(col, color[col])
```

```
c1 Red
c2 Green
c3 Orange
```

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

for x in x_test:
    print(x)
    a = x[0]
```

```
    b = x[1]
    print (a, b)

print("-"*20)

for x, y in x_test:
    print(x, y)
```

```
[1, 2]
1 2
[3, 4]
3 4
[5, 6]
5 6
--------------------
1 2
3 4
5 6
```

The function `range(m, n, p)`, is very useful in loops, as it returns a list of integers starting at `m` through smaller than `n` in steps of length `p`, which can be used as the order for the loop.

While

Executes a block of code in response to a condition.

Syntax:

```
while <condition>:
    <code block>
    continue
    break
else:
    <code block>
```

The code block inside the *while* loop is repeated while the loop condition is evaluated as true.

Example:

```
# Sum 0 to 99
s = 0
x = 1

while x < 100:
    s = s + x
    x = x + 1
else:
    print("Sorry")

print ("Sum of 0 to 99", s)

while x < 0:
    print("Hello")
else:
    print("Sorry")
```

```
Sorry
Sum of 0 to 99 4950
Sorry
```

NOTE: The *while* loop is appropriate when there is no way to determine how many iterations will occur and there is a sequence to follow.

```
s = 0
x = 100

while x < 100:
    s = s + x
    x = x + 1
else:
    print("x is already equal or greater than 100")
print(s)
```

```
x is already equal or greater than 100
0
```

```
x = 1;
s = 0
while (x < 10):
    s = s + x
    x = x + 1
    if (x == 5):
```

```
        break
else:
     print('The sum of first 9 integers : ',s)
print('The sum of ',x,' numbers is :',s)
```

```
The sum of  5  numbers is : 10
```

Break

The break statement is used to exit a for or a while loop. The purpose of this statement is to end the execution of the loop (for or while) immediately and the program control goes to the statement after the last statement of the loop. If there is an optional else statement in while or for loop it skips the optional clause also

```
num_sum = 0
count = 0
for x in range(1, 9):
    print(x)
    num_sum = num_sum + x
    count = count + 1
    if count == 5:
        break
print("Sum of first ",count,"integers is : ", num_sum)
```

```
1
2
3
4
5
Sum of first  5 integers is :  15
```

Continue Statement

The continue statement is used in a while or for loop to take the control to the top of the loop without executing the rest statements inside the loop. Here is a simple example.

```
for x in range(8):
    if (x == 3 or x==6):
        print("\tSkipping:", x)
        continue
```

```
        print("This should never print")
    else:
        print(x)
```

```
0
1
2
    Skipping: 3
4
5
    Skipping: 6
7
```

The `else` in for

```
for x in [1,10,4]:
    if x == 10:
        continue
    print("Hello", x)
else:
    print("processing completed without issues.")
```

```
--------------------
Hello 1
Hello 4
processing completed without issues.
```

```
print("-" * 20)
for x in [1,10,4]:
    if x == 10:
        break
    print("Hello", x)
else:
    print("processing completed without issues.")
```

```
--------------------
Hello 1
```

Usecases for `else`

A common use case for the else clause in loops is to implement search loops; say you're performing a search for an item that meets a particular condition, and need to perform additional processing or raise an error if no acceptable value is found:

```
for x in data:
    if meets_condition(x):
        break
else:
    # raise error or do additional processing
```

```
for n in range(2, 10):
    for x in range(2, n):
#         print("N =", n)
        if n % x == 0:
            print(n, 'equals', x, '*', n/x)
            break
    else:
        # loop fell through without finding a factor
        print(n, 'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2.0
5 is a prime number
6 equals 2 * 3.0
7 is a prime number
8 equals 2 * 4.0
9 equals 3 * 3.0
```

NOTE: When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no break occurs. For more on the try statement and exceptions, see Handling Exceptions.

Excersise

- Create a program to count by prime numbers. Ask the user to input a number, then print each prime number up to that number.
- Instruct the user to pick an arbitrary number from 1 to 100 and proceed to guess it correctly within seven tries. After each guess, the user must tell whether their number is higher than, lower than, or equal to your guess.
- Print all the characters in sentance "The continue statement is used in a while or for loop"
- Write a program which will find all such numbers which are divisible by 7 but are not a multiple of 5, between 2000 and 3200 (both included). The numbers obtained should be printed in a comma-separated sequence on a single line.
- When does while loop in the following code exit?

```
x=input('Enter a passkeyid: ')
while x != str(10):
    print(x),
    x=input('Enter a passkeyid: ')
key=raw_input("Press key...")
```

- What would be printed from the following Python code segments?

a.

```
for i in range(10,0,-3):
      print(i)
```

b.

```
* for x in range(1, 6):
     for y in range(1, x+1):
            print(x, ' ', y)
```

c.

```
x=10
while x>5:
     print(x),
     x-=1
```

```
x=int(input('Enter a passkeyid: '))
print(type(x))
while x != 10:
    print(x),
    x=int(input('Enter a passkeyid: '))
key=raw_input("Press key...")
```