Chapter 4 - Loops

4 Loops

In the previous chapter on Conditional Statements, we have seen that the statements are executed sequentially that means, the statements are executed in the order they are listed, one after the other and/or after an expression results in either a `True` or `False` value. What happens if we want to execute a block of statements for a certain number of times or even till an expression resolves to a False value? What we need is called a looping statement and they allow blocks of statement/s to either be executed a finite number of times or till an expression resolves to a False value.

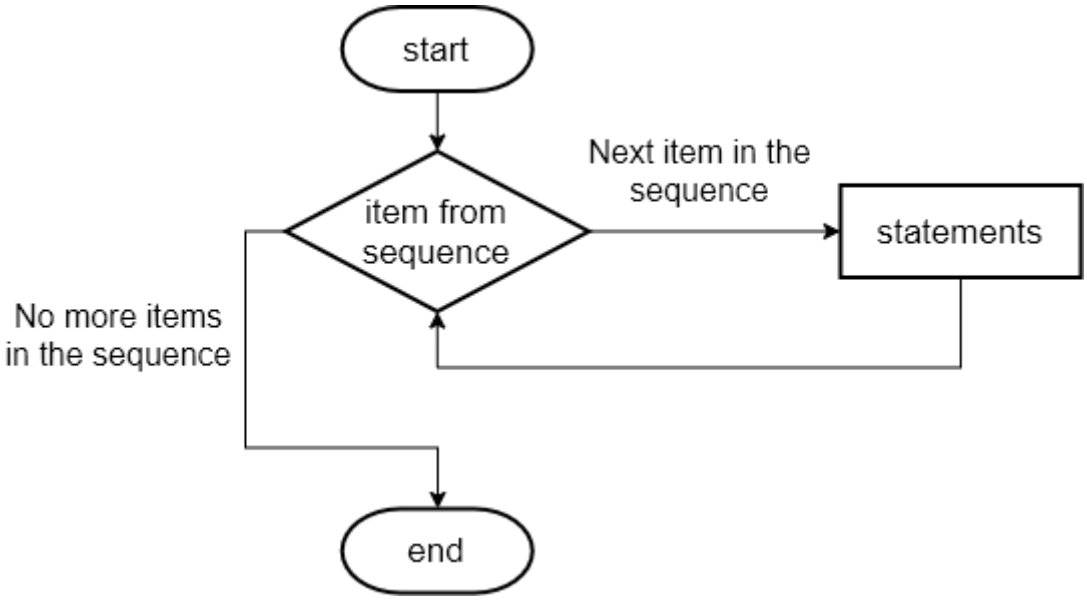
Python has 3 types of looping statements:

- `for` loop - finite number of execution
- `while` loop - executes till a expression resolves to a False
- Nested loops - any combination of one of more loops inside either a `for` or `while` loop

4.1 `for` loops

`for` loops are loops that executes a block of statement/s with a finite number of times. It is generally used for looping over any iterable object, such as strings, lists, tuple, dictionary, etc. The syntax for the `for` loop and its flow chart is as follows:

```
for var in iterable:
    statement(s)
```



Flowchart 1 - `for` loop

The `sequence` is can be an iterable object variable or a function evaluates to an iterable object. The first item of the `sequence` is place in the `var` before the statement/s are executed. The statements within the `for` loop are executed till the entire iterable object in the `sequence` is exhausted. An example is on the next page.

```
# in this example we will be using the built-in
# function range() to help with generating a
# number sequence
for i in range(0, 10, 2):
    print(i)
```

The built-in function `range(start, stop, step)` has 3 inputs:

- **start** - The number from which the sequence should start generating from. This parameter is optional and the default is 0.
- **stop** - The number from which the sequence will stop at.
- **step** - The number that determines the step size of the number sequence generated. The step parameter can take both positive and negative values thus the start and stop parameters have to be adjusted accordingly. This parameter is optional and the default is 1. Note that the step size is always `step < stop_value`.

The output is:

```
0
2
4
6
8
```

Strings in Python are also regarded as iterable objects and thus we can use `for` loops to loop through them.

Example:

```
for i in 'Python':
    print(i)
```

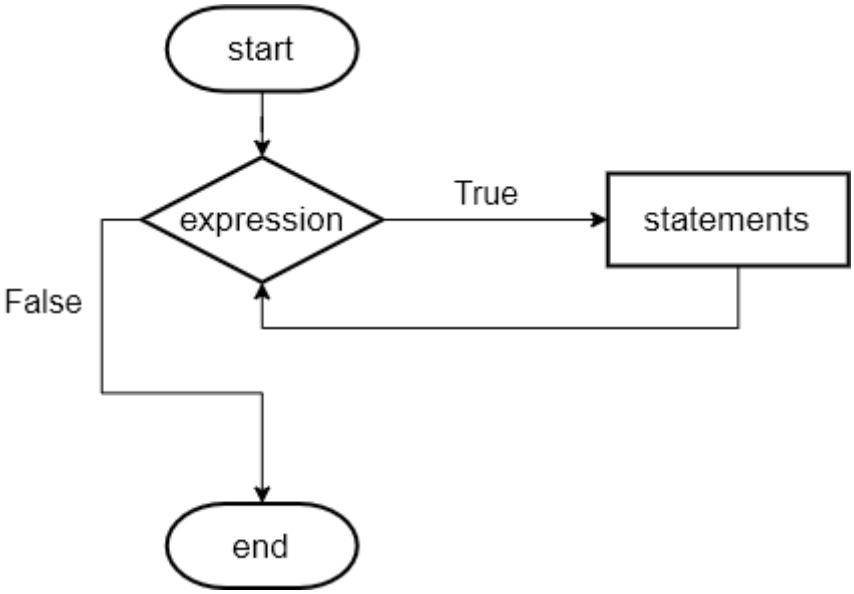
The output is

```
P
y
t
h
o
n
```

4.2 `while` loops

While the `for` loop execution is finite, the `while` loop will execute repeatedly until the expression resolves to a False value. The syntax and flow chart is as follows:

```
while expression:
    statement(s)
```



Flowchart 2 - `while` loop

To help the expression resolve to a False value, an external variable will be required as a loop tracker. Example:

```
# variable to make sure that the expression resolves
# to a False value after 5 counts
count = 0
while count < 5:
    print("Count is: ", count)
    count += 1      # keeps track of the number of repeats
print("Finished!")
```

the output is

```
Count is:  0
Count is:  1
Count is:  2
Count is:  3
Count is:  4
Finished!
```

`while` loops have a nasty habit of resulting in an infinite loop should the expression **never** resolves to a False value. These loops are call Infinite loops. To stop an infinite loop, use the `ctrl+c` keys while in the command line window to exit the program. If you are using Jupyter Notebook, use the *Interrupt* option in the Kernel menu. If that does not work, Restart the Python Kernel by terminating it in the command window and restarting it again from the command window.

Example:

```
cnt = 1
while cnt == 1:     # this results in an infinite loop
    num = int(input("Enter a number  :"))
    print ("You entered: ", num)
```

The 2 built-in functions used here are `input(prompt)` and `int(string)`.

- `input(prompt)` - Accepts a user input from the terminal and returns it as a string. The prompt is a message used to show to the user that user input is required at this point.
- `int(string)` - Converts a string value to its integer equivalent.

When the program is executed from command line via a Python script, the output is:

```
Enter a number  :20
You entered:  20
Enter a number  :3
You entered:  3
Enter a number  :Traceback (most recent call last):
   File "examples\test.py", line 5, in
      num = int(input("Enter a number  :"))
KeyboardInterrupt
```

4.3 Nested loops

Nested loops are used when there is a need to iterate nested objects such as a 2-dimensional iterable object. Both `for` and `while` loops can be nested with each other or in combination. There can be any number of loops within a nested loop. The syntax is as follows:

```
# Nested for loop
for var in sequence1:
    for var in sequence2:
        statement(s)
    statement(s)

# Nested while loop
while expression1:
    while expression2:
        statement(s)
    statement(s)

# Nested for and while loops
for var in sequence:
    while expression:
        statement(s)
    statement(s)

while expression:
    for var in sequence:
        statement(s)
    statement(s)
```

Example:

```
# prints a 5 by 5 matrix of numbers
for row in range(1,6):      # outer for loop
   for col in range(1,6):       # inner for loop
      k = row * col
      print(f'{k:02}', end=' ')
   print()  # belongs to the outer for loop
```

The first `print()` statement on line 5 used here has included some new Python3 string formatting to output numbers with a predefined width. More detailed explanation can be found in the Strings chapter but a short description is as follows for `f'{k:02}'`:

- the `f` stands for the new type Python string formatting available since Python 3.6
- the `{:}` is part of the string formatting syntax
- the `k` is the variable to be printed
- the `02` pads the number with leading zeros and sets the width of the column to 2 digits

The `end` parameter is part of the `print()` and is used to append a whitespace character instead of the default newline character after the statement is executed.

The output is:

```
01 02 03 04 05
02 04 06 08 10
03 06 09 12 15
04 08 12 16 20
05 10 15 20 25
```

The algorithm to this nested for loop is to complete the computation for the inner for loop before moving on the outer for loop.

4.4 Loop Control Statements

There are also special control statements that changes the execution flow of the loops from its normal sequence. These statements are:

- **break** - stops and exits the loop and transfers the execution to the statement immediately following the loop.
- **continue** - causes the loop to skip the remaining of its statements and immediately retest the expression before continuing with the loop again.
- **pass** - used when the statement is required syntactically but no execution of codes are needed.

Examples:

```
# break statement example
print("Break Statement"+("-"*13))
for letter in "GoodDay":
    if letter == "d":
        break
    print ("Current Letter :", letter)
print()
# continue statement example
print("Continue Statement"+("-"*10))
for letter in "GoodDay":
```

```
    if letter == "d":
        continue
    print ("Current Letter :", letter)
print()
# pass statement example
print("Pass Statement"+("-"*14))
for letter in "GoodDay":
    if letter == "d":
        pass
        print("passed block")
    print ("Current Letter :", letter)
```

the output is

```
Break Statement-------------
Current Letter : G
Current Letter : o
Current Letter : o

Continue Statement----------
Current Letter : G
Current Letter : o
Current Letter : o
Current Letter : D
Current Letter : a
Current Letter : y

Pass Statement--------------
Current Letter : G
Current Letter : o
Current Letter : o
passed block
Current Letter : d
Current Letter : D
Current Letter : a
Current Letter : y
```

4.5 References

1. Python 3 - Loops, https://www.tutorialspoint.com/python3/python_loops.htm
2. Lubanovic, 2019, 'Chapter 6. Loop with while and for' in Introducing Python, 2nd Edition, O`Reilly Media, Inc.