Loop:

- When do I use for loops?

for loops are traditionally used when you have a block of code which you want to repeat a fixed number of times. The Python *for* statement iterates over the members of a sequence in order, executing the block each time. Contrast the *for* statement with the "while" loop, used when a condition needs to be checked each iteration, or to repeat a block of code forever. For example:

For loop from 0 to 2, therefore running 3 times.

```
for x in range(0, 3):
    print("We're on time %d" % (x))
```

While loop from 1 to infinity, therefore running forever.

```
x = 1
while True:
    print("To infinity and beyond! We're getting close, on %d now!" % (x))
    x += 1
```

As you can see, these loop constructs serve different purposes. The for loop runs for a fixed amount - in this case, 3, while the while loop runs until the loop condition changes; in this example, the condition is the boolean True which will never change, so it could theoretically run forever. You could use a for loop with a huge number in order to gain the same effect as a while loop, but what's the point of doing that when you have a construct that already exists? As the old saying goes, "why try to reinvent the wheel?".

- *How do they work?*

If you've done any programming before, you have undoubtedly come across a for loop or an equivalent to it. Many languages have conditions in the syntax of their for loop, such as a relational expression to determine if the loop is done, and an increment expression to determine the next loop value. In Python this is controlled instead by generating the appropriate sequence. Basically, any object with an iterable method can be used in a for loop. Even strings, despite not having an iterable method - but we'll not get on to that here. Having an iterable method basically means that the data can be presented in list form, where there are multiple values in an orderly fashion. You can define your own iterables by creating an object with next() and iter() methods. This means that you'll rarely be dealing with raw numbers when it comes to for loops in Python - great for just about anyone!

- *Nested loops*

*When you have a block of code you want to run **x** number of times, then a block of code within that code which you want to run **y** number of times, you use what is known as a "nested loop". In Python, these are heavily used whenever someone has a list of lists - an iterable object within an iterable object.*

```
for x in range(1, 11):
    for y in range(1, 11):
        print('%d * %d = %d' % (x, y, x*y))
```

- *Early exits*

Like the while loop, the for loop can be made to exit before the given object is finished. This is done using the break statement, which will immediately drop out of the loop and contine execution at the first statement after the block. You can also have an optional else clause, which will run should the for loop exit cleanly - that is, without breaking.

```
for x in range(3):
    if x == 1:
        break
```

Examples-1:

```
for x in range(3):
    print(x)
else:
    print('Final x = %d' % (x))
```

Strings as an iterable

```
string = "Hello World"
for x in string:
    print(x)
```

Lists as an iterable

```
collection = ['hey', 5, 'd']
for x in collection:
    print(x)
```

Loop over Lists of lists

```
list_of_lists = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list in list_of_lists:
    for x in list:
        print(x)
```

Examples-2:

Python

```python
# Python program to illustrate
# while loop
count = 0
while (count < 3):
    count = count + 1
    print("Hello Geek")
```

1. Output:

```
Hello Geek
Hello Geek
Hello Geek
```

Using else statement with while loops: As discussed above, while loop executes the block until a condition is satisfied. When the condition becomes false, the statement immediately after the loop is executed.
The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.
If else like this:

Examples-3:

```python
#Python program to illustrate
# combining else with while
count = 0
while (count < 3):
    count = count + 1
    print("Hello Geek")
else:
    print("In Else Block")
```

- Output:

```
Hello Geek
Hello Geek
Hello Geek
In Else Block
```

Single statement while block: Just like the if block, if the while block consists of a single statement the we can declare the entire loop in a single line as shown below:

```python
# Python program to illustrate
# Single statement while block
count = 0
while (count == 0): print("Hello Geek")
```

Note: It is suggested **not to use** this type of loops as it is a never ending infinite loop where the condition is always true and you have to forcefully terminate the compiler.

for in Loop: For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is “for in” loop which is similar to for each loop in other languages. Let us learn how to use for in loop for sequential traversals.
Syntax:

Examples-4:

Python3

```
# Python program to illustrate
# Iterating over range 0 to n-1

n = 4
for i in range(0, n):
    print(i)
```

1. Output :

```
0
1
2
3
```

Python

```python
# Python program to illustrate
# Iterating by index

list = ["geeks", "for", "geeks"]
for index in range(len(list)):
    print list[index]
```

1. Output:

```
geeks
for
geeks
```

Using else statement with for loops: We can also combine else statements with for loop like in while loop. But there is no condition for the loop based on which the execution will terminate so the else block will be executed immediately after the block finishes execution.

Examples-5:

Python

```python
for iterator_var in sequence:
    for iterator_var in sequence:
        statements(s)
        statements(s)
```

Nested Loops: Python programming language allows the use of one loop inside another loop. Following section shows a few examples to illustrate the concept.