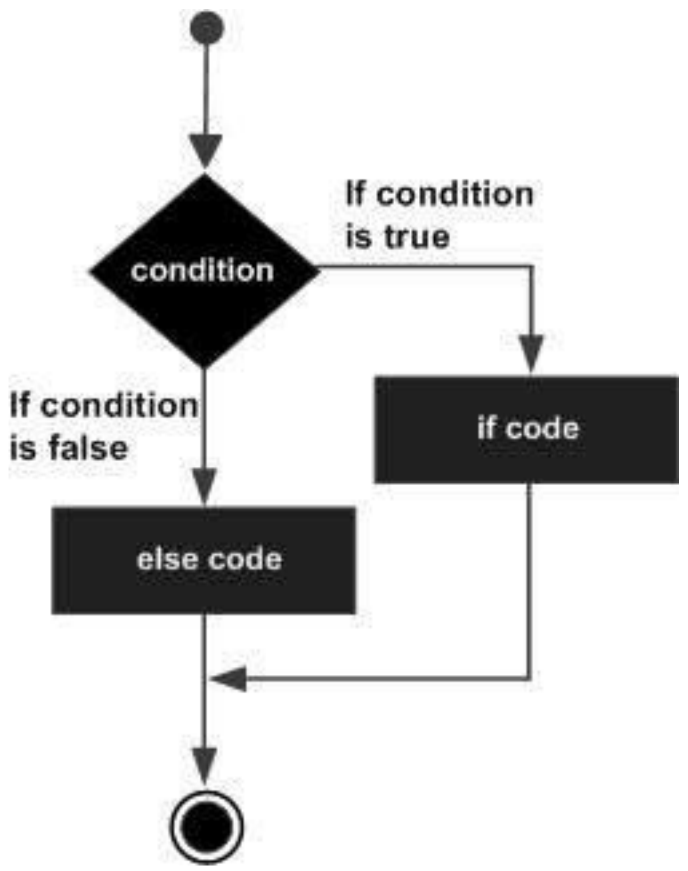
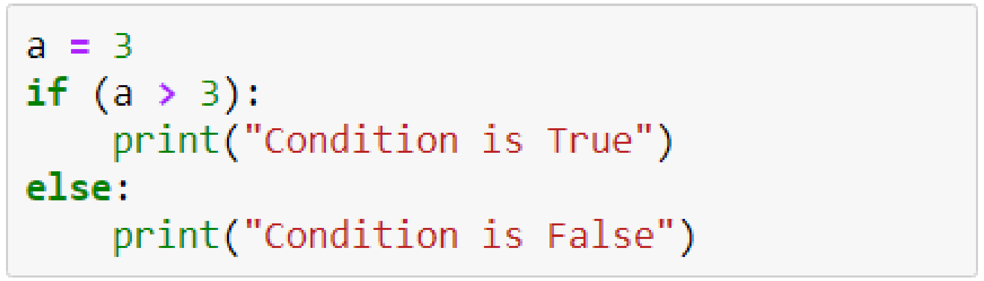
Conditional Statements ⇨ if-else

- An **if statement** can be followed by an optional **else statement**, which executes when the Boolean expression is **False**.
- An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the **if** statement resolves to **0** or a **False** value.
- The **else** statement is an optional statement and there could be at most only one **else** statement following **if**.

```
if expression:
   statement(s)
else:
   statement(s)
```

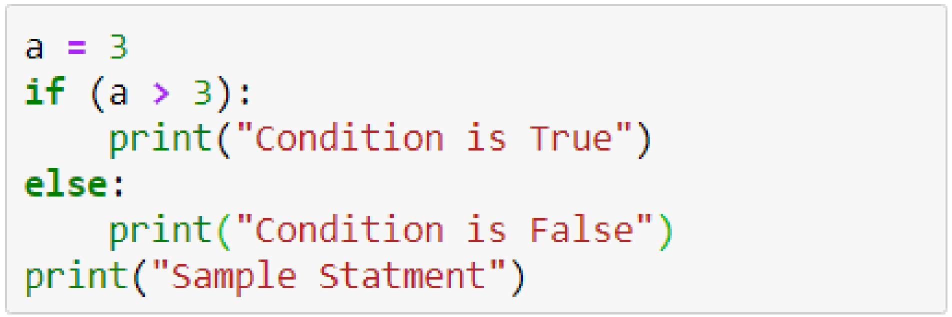


Conditional Statements ➪ if - else



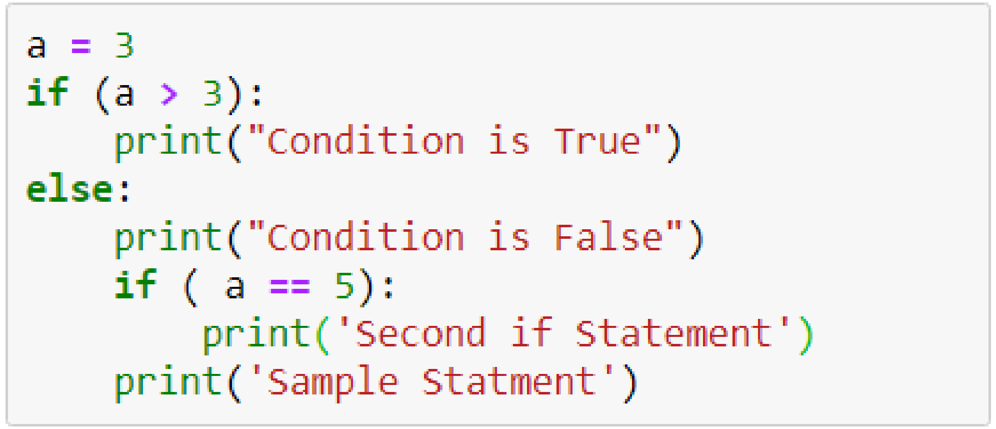
```
a = 3
if (a > 3):
    print("Condition is True")
else:
    print("Condition is False")
```

```
Condition is False
```



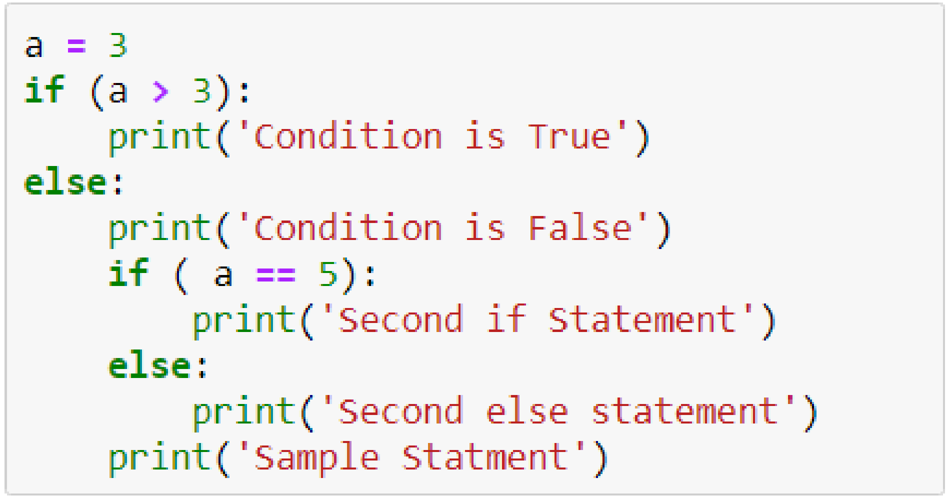
```
a = 3
if (a > 3):
    print("Condition is True")
else:
    print("Condition is False")
print("Sample Statment")
```

```
Condition is False
Sample Statment
```



```
a = 3
if (a > 3):
    print("Condition is True")
else:
    print("Condition is False")
    if ( a == 5):
        print('Second if Statement')
    print('Sample Statment')
```

```
Condition is False
Sample Statment
```

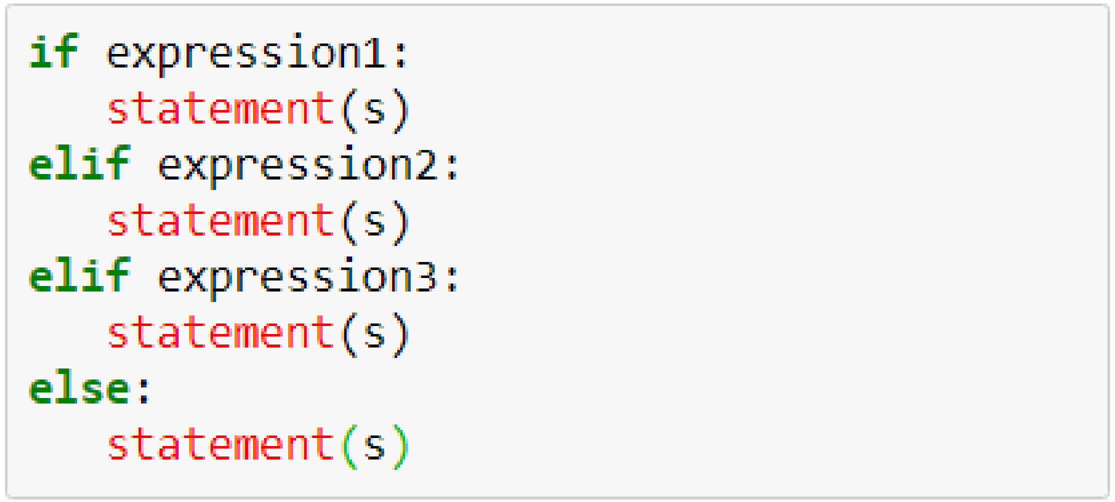


```
a = 3
if (a > 3):
    print('Condition is True')
else:
    print('Condition is False')
    if ( a == 5):
        print('Second if Statement')
    else:
        print('Second else statement')
    print('Sample Statment')
```

```
Condition is False
Second else statement
Sample Statment
```

Conditional Statements ⇨ if - elif

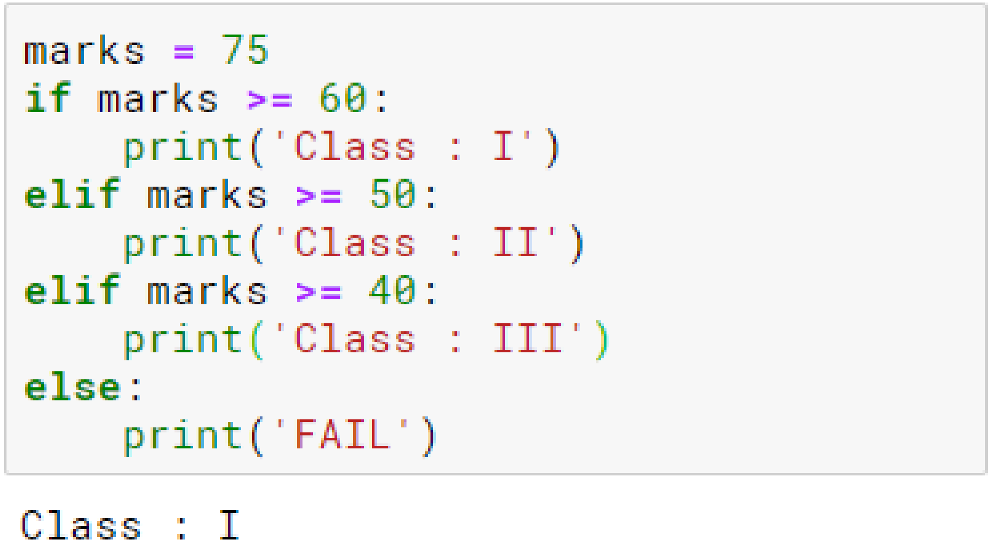
- The **elif** statement allows one to check multiple expressions for **True** and execute a block of code as soon as one of the conditions evaluates to **True**.
- Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if**.



```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

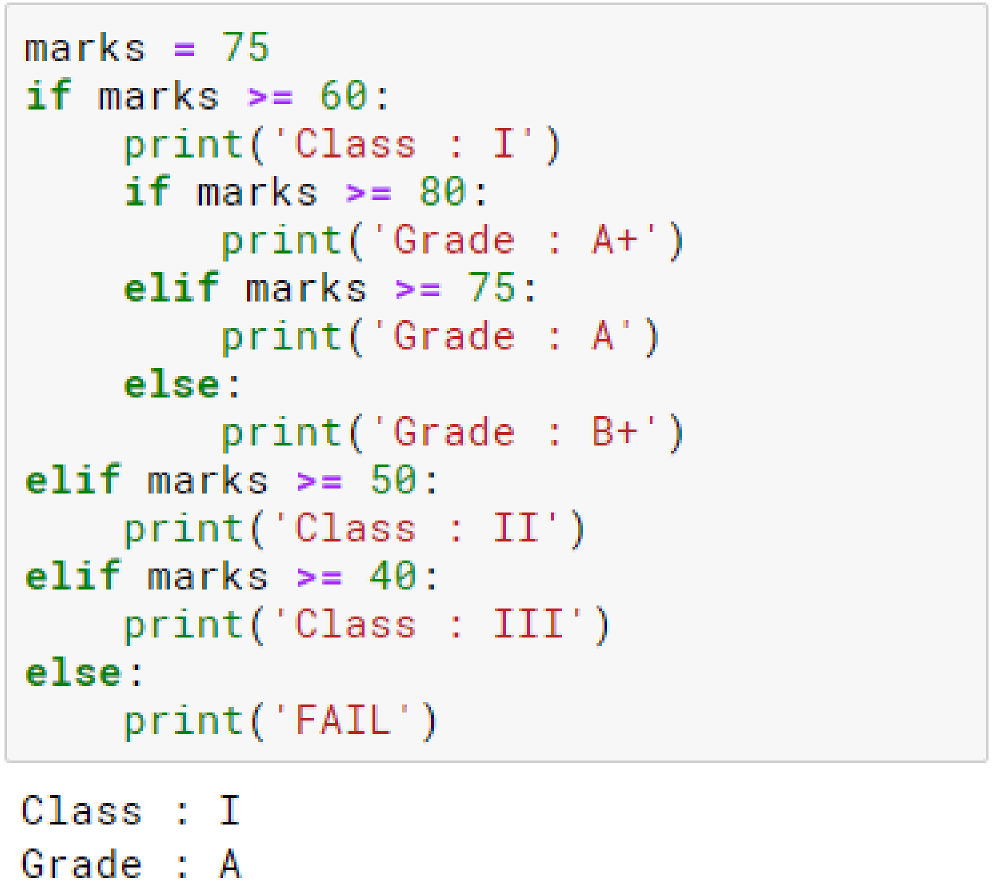
- Core Python does not provide switch or case statements as in other languages, but we can use **if...elif** statements to simulate switch case.

Conditional Statements ⇨ if - elif



```python
marks = 75
if marks >= 60:
    print('Class : I')
elif marks >= 50:
    print('Class : II')
elif marks >= 40:
    print('Class : III')
else:
    print('FAIL')
```

```
Class : I
```

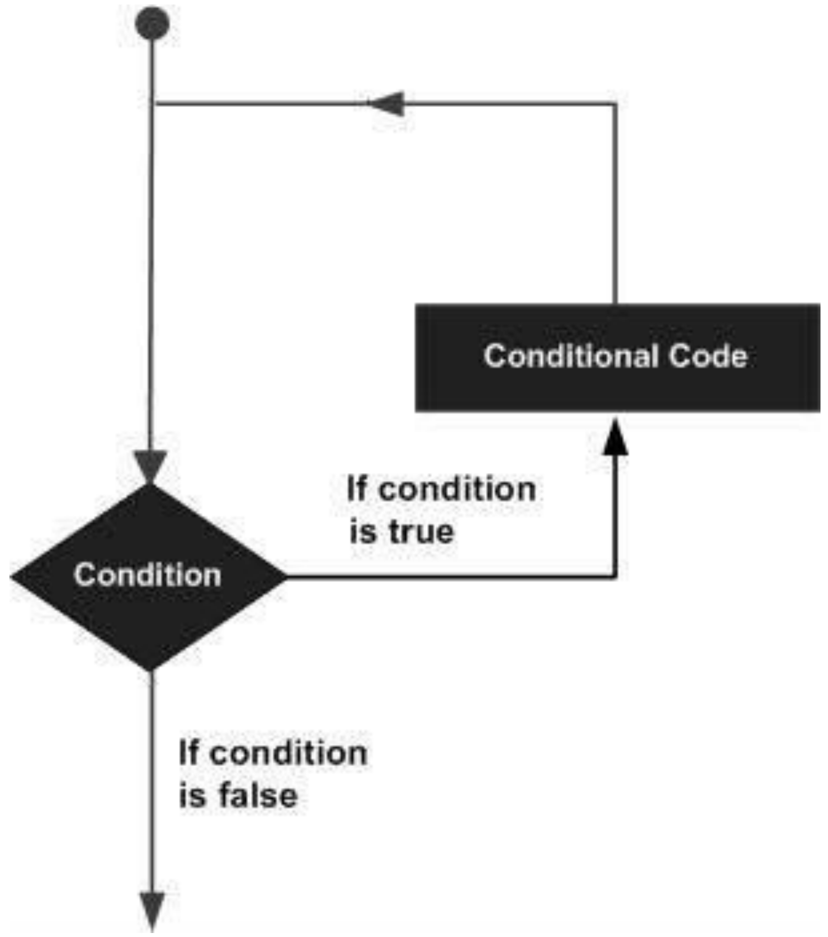


```python
marks = 75
if marks >= 60:
    print('Class : I')
    if marks >= 80:
        print('Grade : A+')
    elif marks >= 75:
        print('Grade : A')
    else:
        print('Grade : B+')
elif marks >= 50:
    print('Class : II')
elif marks >= 40:
    print('Class : III')
else:
    print('FAIL')
```

```
Class : I
Grade : A
```

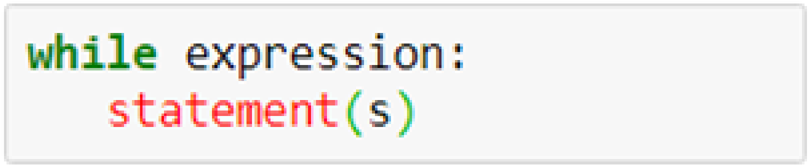
Looping Statements

- In general, statements are executed sequentially. The first statement in a program is executed first, followed by the second, and so on. There may be a situation when one need to execute a block of code several number of times.
- Python Programming language provide various control structures that allow for more complicated execution paths.
- A loop statement allows us to execute a statement or group of statements multiple times. The diagram illustrates a loop statement.



Looping Statements ⇨ While

- A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is **True**.



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

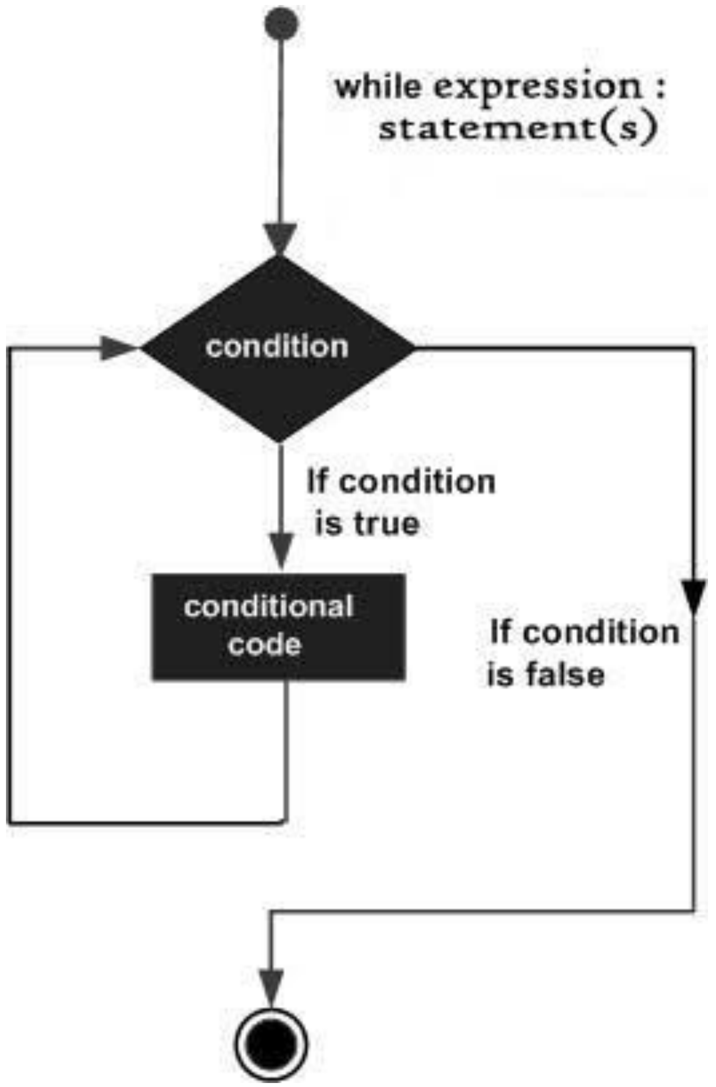
- Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and **True** is any non-zero value.

- The loop iterates while the condition is **True**. When the condition becomes **False**, program control passes to the line immediately following the loop.
- In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.
- Key point of the while loop is that the loop might not ever run. When the condition is tested and the result is **False**, the loop body will be skipped and the first statement after the while loop will be executed.

Looping Statements ➡ While

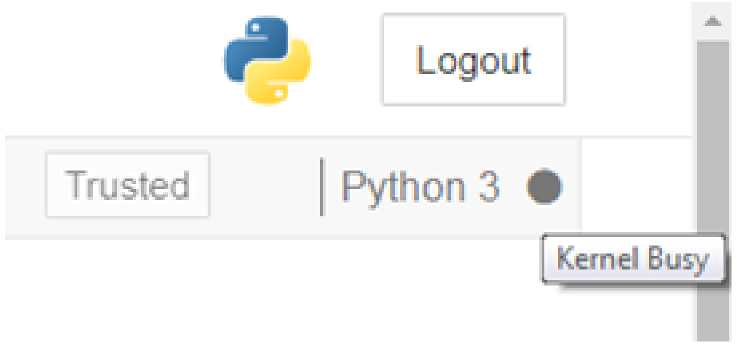
```
count = 0
while (count < 9):
    print("The count is: ", count)
    count = count + 1

print("Other Statement")
```

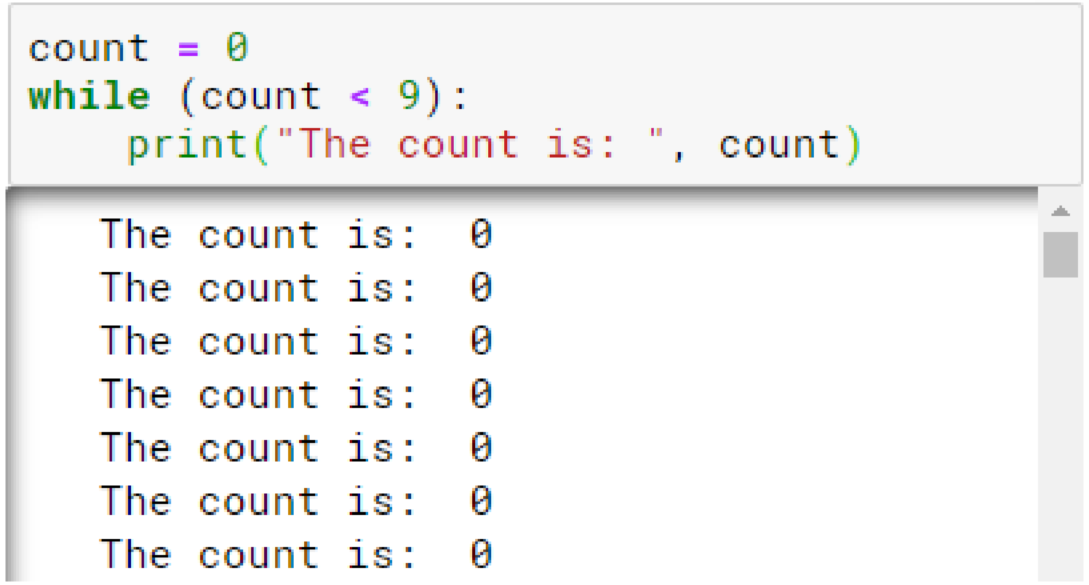
```
The count is:  0
The count is:  1
The count is:  2
The count is:  3
The count is:  4
The count is:  5
The count is:  6
The count is:  7
The count is:  8
Other Statement
```



Looping Statements ➡ Infinite Loop

- A loop becomes infinite loop if a condition never becomes **False**. One must use caution when using while loops because of the possibility that given condition never resolves to a False value. This results in a loop that never ends. Such a loop is called an **infinite** loop.
- An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.
- When the given code is executed, it goes in an infinite loop. The Kernel of our Jupyter Noteboo becomes Busy.





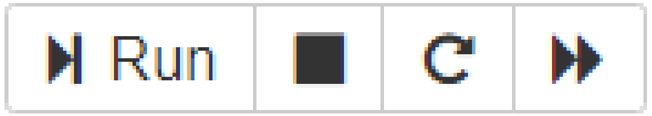
```
count = 0
while (count < 9):
    print("The count is: ", count)
```

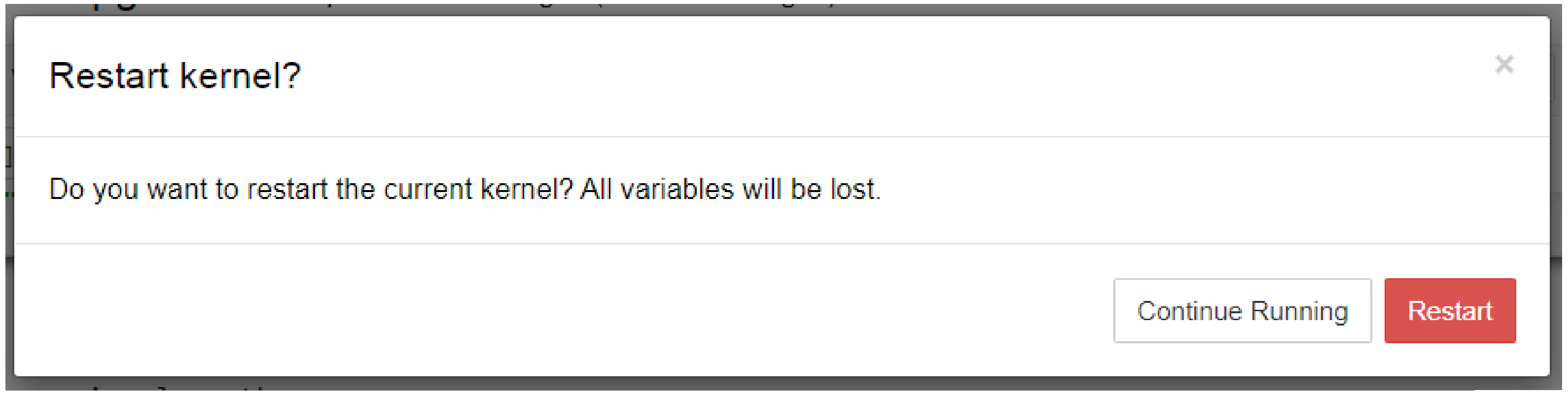
```
The count is:  0
The count is:  0
The count is:  0
The count is:  0
The count is:  0
The count is:  0
The count is:  0
```

Looping Statements ➪ Infinite Loop

Steps to follow to interrupt the program running in infinite loop.

Locate the **Restart** button and click on it.





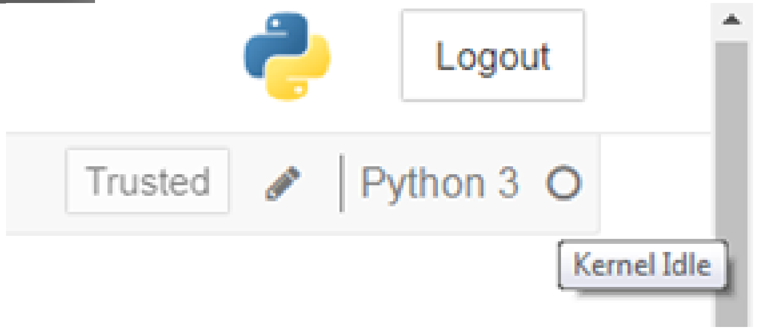
Restart kernel?

Do you want to restart the current kernel? All variables will be lost.

Continue Running | Restart

On the pop up dialog box ,click on Restart button.

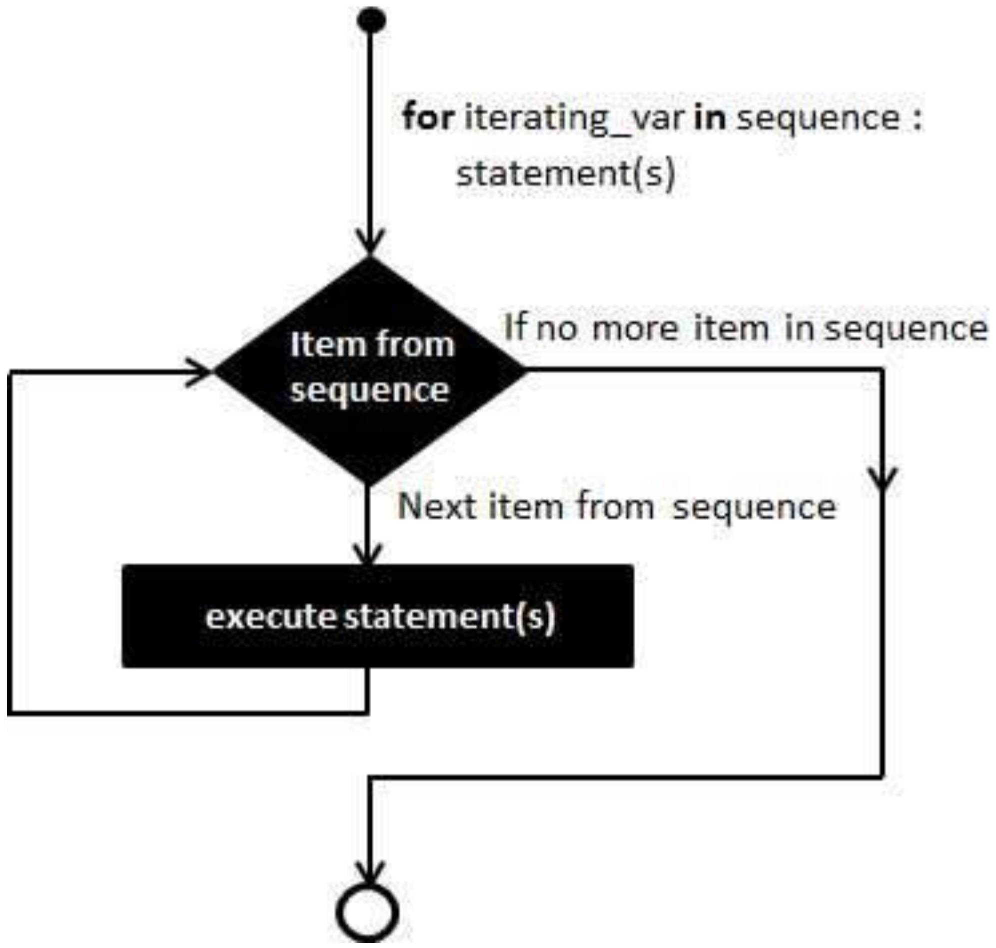
Wait for Kernel to become Ideal again.



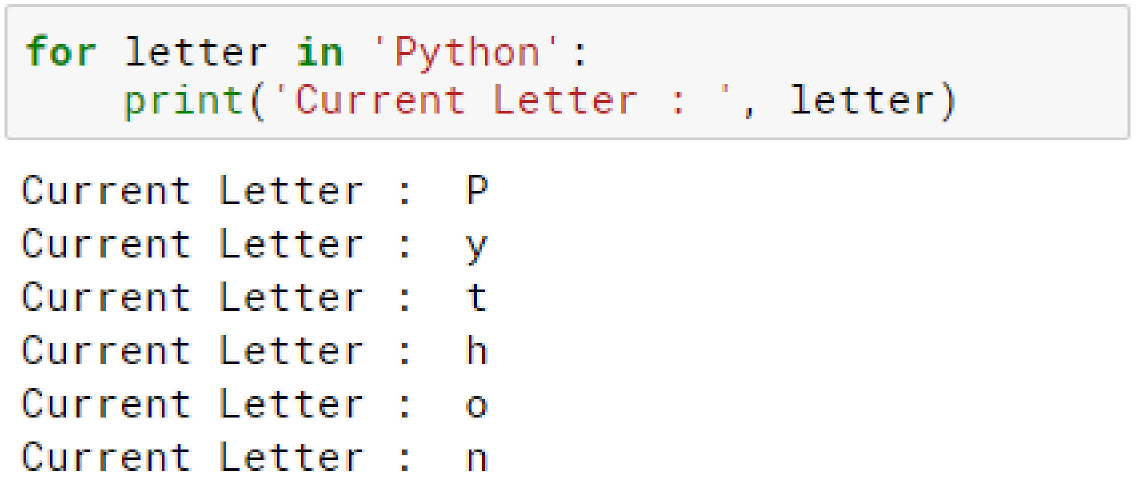
Looping Statements ➡ for

- Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
- It has the ability to iterate over the items of any sequence, such as a list or a string.
- If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable ***iterating_var***. Next, the statements block is executed.
- Each item in the list is assigned to ***iterating_var***, and the statement(s) block is executed until the entire sequence is exhausted.

```
for iterating_var in sequence:
   statements(s)
```



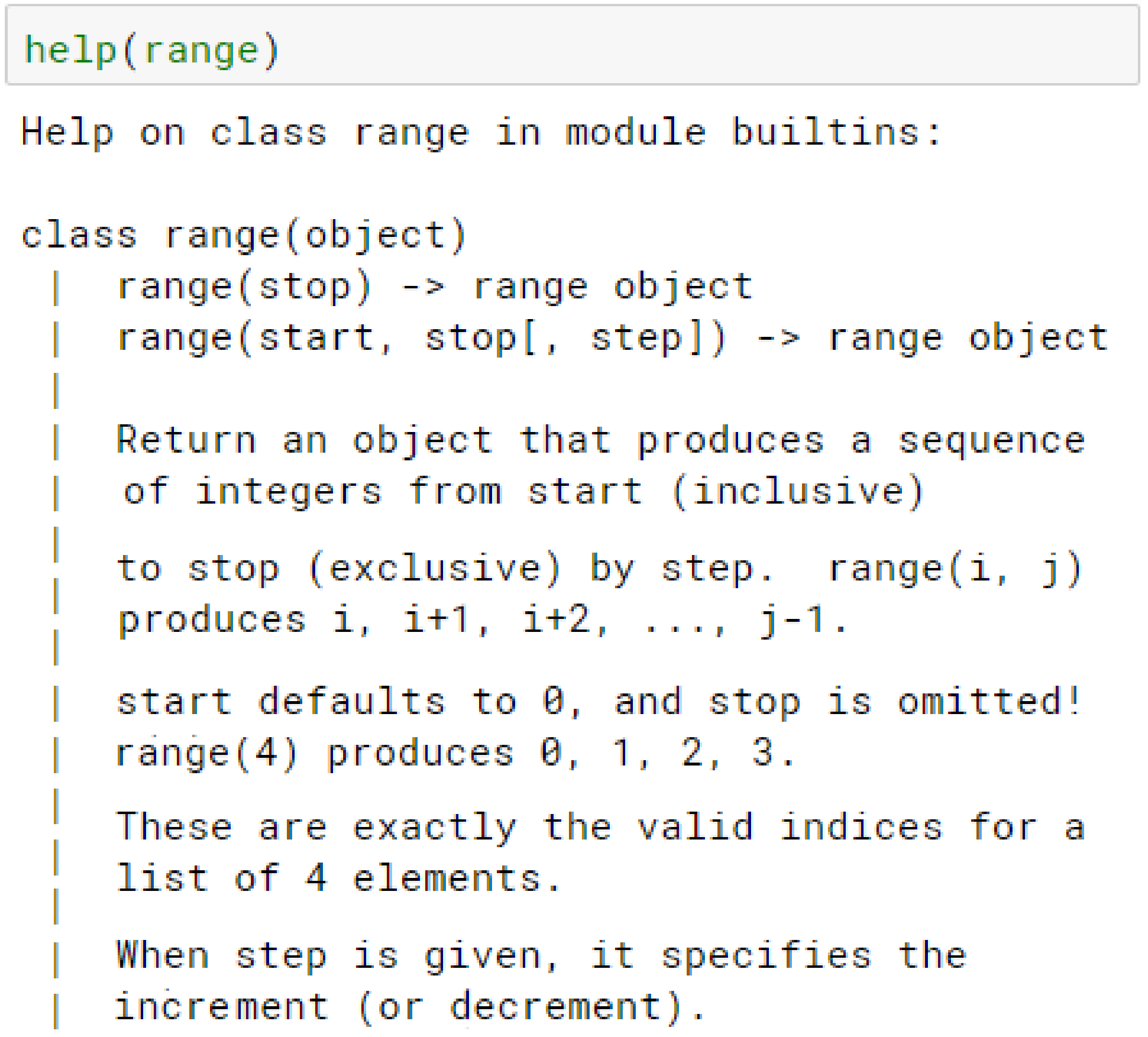
Looping Statements ➡ for



```python
for letter in 'Python':
    print('Current Letter : ', letter)
```

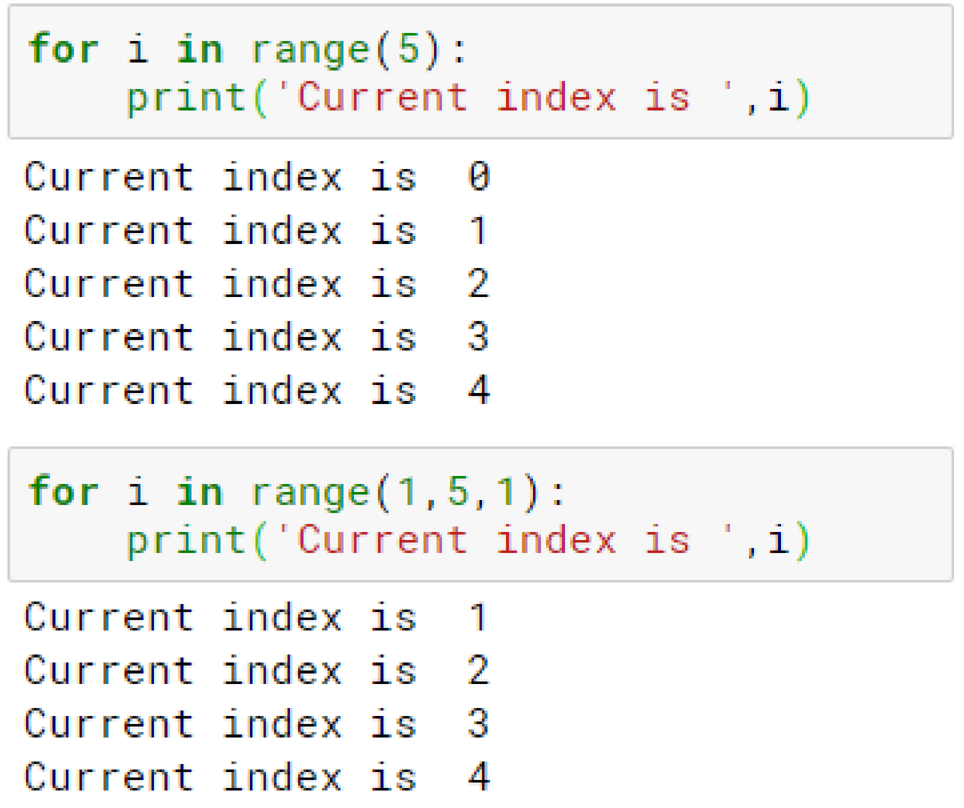
```
Current Letter :  P
Current Letter :  y
Current Letter :  t
Current Letter :  h
Current Letter :  o
Current Letter :  n
```

Looping Statements ⇨ for



```
help(range)
```

```
Help on class range in module builtins:

class range(object)
 |  range(stop) -> range object
 |  range(start, stop[, step]) -> range object
 |
 |  Return an object that produces a sequence
 |   of integers from start (inclusive)
 |
 |  to stop (exclusive) by step.  range(i, j)
 |  produces i, i+1, i+2, ..., j-1.
 |
 |  start defaults to 0, and stop is omitted!
 |  range(4) produces 0, 1, 2, 3.
 |
 |  These are exactly the valid indices for a
 |  list of 4 elements.
 |
 |  When step is given, it specifies the
 |  increment (or decrement).
```



```
for i in range(5):
    print('Current index is ',i)
```

```
Current index is  0
Current index is  1
Current index is  2
Current index is  3
Current index is  4
```

```
for i in range(1,5,1):
    print('Current index is ',i)
```

```
Current index is  1
Current index is  2
Current index is  3
Current index is  4
```

`range(3,10,2)` ⇨ 3, 5, 7, 9

`range(3,10)` ⇨ 3, 4, 5, 6, 7, 8, 9

`range(10,1,-1)` ⇨ 10, 9, 8, 7, 6, 5, 4, 3, 2