

Basic Programming with Python

Prepared By:

Professor Dr. Md. Mijanur Rahman

Department of Computer Science & Engineering

Jatiya Kabi Kazi Nazrul Islam University, Bangladesh.

www.mijanrahman.com

3

Control Structures in Python

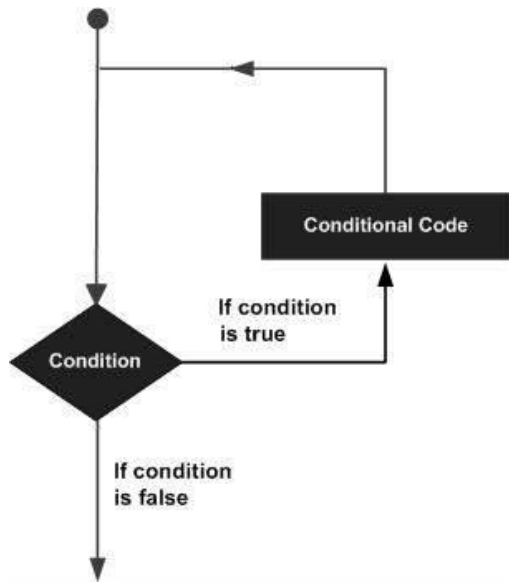
CONTENTS

3.2. Iteration (Loop) Statements	1
3.2.1. for Loop.....	2
3.2.2. for...else Loop.....	5
3.2.3. while Loop.....	7
3.2.4. while...else Loop.....	11
3.2.5. Using else Statement with Loops	13
3.2.6. Nested Loops	15

3.2. ITERATION (LOOP) STATEMENTS

In Python, iteration or loop statements are used to execute a block of code repeatedly. They allow us to automate repetitive tasks, process large amounts of data efficiently, and perform operations on sequences like lists, tuples, and dictionaries.

Python loops allow us to execute a statement or group of statements multiple times. The following diagram illustrates how a loop statement works:



In Python, a loop condition refers to the expression or condition that determines whether the loop should continue iterating or terminate. Python provides several types of iteration statements, including for loops, while loops, and loop control statements like break, continue, and pass.

It's essential to ensure that the loop condition is properly defined to prevent infinite loops, which can lead to excessive CPU usage and program crashes. Also, loop conditions can be updated within the loop body to control the loop's behavior based on changing circumstances.

Types of Loops in Python:

Python programming language provides various types of loops to handle looping requirements. There are two primary types of loops in Python: for loops and while loops. These loops allow us to execute a block of code repeatedly, but they have different use cases and syntax.

1. **for Loops:** for loops are used to iterate over a sequence (such as a list, tuple, string, or range) or any iterable object. They execute a block of code for each item in the sequence.
1. **while Loops:** while loops are used to execute a block of code repeatedly as long as a specified condition evaluates to True. They continue iterating until the condition becomes False.

In both types of loops, the loop condition determines the termination criteria for the loop. If the condition evaluates to False (in the case of a while loop) or if there are no more items in the iterable (in the case of a for loop), the loop terminates, and the program execution continues after the loop.

3.2.1. for Loop

In Python, a **for** loop is used to iterate over a sequence (such as a list, tuple, string, or range) or any iterable object. It allows us to execute a block of code repeatedly, once for each item in the sequence. The **for** loop is widely used in Python programming for various tasks, such as processing collections of data, performing calculations, and executing repetitive tasks.

The general syntax of a **for** loop in Python is given below:

```
for item in sequence:  
    code-block
```

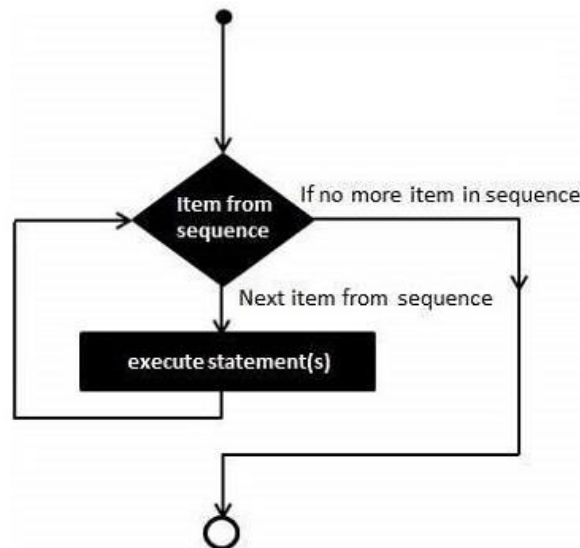
3. Control Structures in Python

In this syntax:

- Item: This is a variable that represents each item in the iterable object (sequence) during each iteration of the loop. We can choose any valid variable name for this.
- sequence: This is the iterable object over which the loop iterates. It can be a list, tuple, string, range object, or any other iterable.
- Code block: This is the indented block of code that is executed for each item in the iterable. It is executed once for each item in the iterable.

If a sequence contains an expression list, it is evaluated first. Then, the first item (at 0th index) in the sequence is assigned to the iterating variable item. Next, the code-block is executed. Each item in the list is assigned to item, and the code-block is executed until the entire sequence is exhausted.

The following flowchart illustrates the working of for loop:



Python's built-in range() function returns an iterator object that streams a sequence of numbers. We can run a for loop with range as well. For example:

```
for i in range(5): # Loop condition: range(5)
    print(i)
```

In this example, the loop condition range(5) generates an iterable sequence of numbers from 0 to 4, and the loop iterates over each number.

The following is another example of using a for loop to iterate over a list of fruits:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

In this example, fruits is a list containing three strings. The for loop iterates over each item in the fruits list. During each iteration, the variable fruit takes on the value of each item in the list ("apple", "banana", "cherry" in sequence). The print(fruit) statement inside the loop prints each fruit name. The output of this script will be:

3. Control Structures in Python

```
apple
banana
cherry
```

The **for** loop continues iterating until all items in the iterable have been processed. Once the loop has finished iterating over all items, the program execution continues with the code following the loop. **for** loops are commonly used in Python for various tasks, such as iterating over the elements of a list, processing characters in a string, generating sequences of numbers using `range()`, and iterating over the key-value pairs of a dictionary.

Example 3.9: Computing sum and average of numbers using for loop in Python.

```
# List of numbers
numbers = [10, 20, 30, 40, 50]

# Initialization
total_sum = 0
count = 0

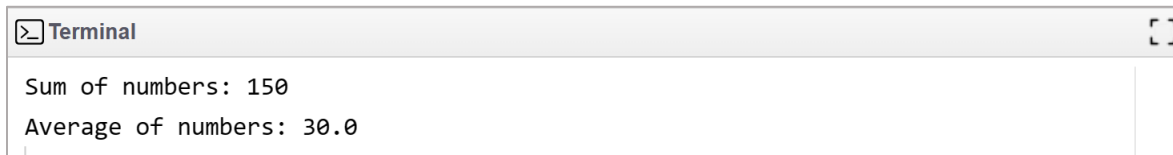
# Compute sum of numbers using a for loop
for num in numbers:
    total_sum += num
    count += 1

# Compute average
average = total_sum / count

# Print sum and average
print("Sum of numbers:", total_sum)
print("Average of numbers:", average)
```

In this script, we have a list of numbers `numbers`. We initialize two variables `total_sum` and `count` to store the sum of numbers and the count of numbers, respectively. We use a `for` loop to iterate over each number in the `numbers` list. Inside the loop, we add each number to the `total_sum` and increment the `count`. After the loop, we compute the average by dividing the total sum by the count. Finally, we print the sum and average of the numbers.

Output:



```
Terminal
Sum of numbers: 150
Average of numbers: 30.0
```

Example 3.10: Computing factorial of a given number using for loop in Python.

```
# Function to compute factorial
def factorial(n):
    fact = 1
    for i in range(1, n + 1):
```

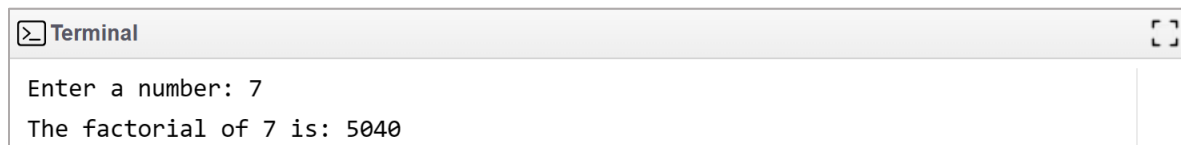
```
    fact = fact * i

    return fact

# Example usage
number = int(input("Enter a number: "))
result = factorial(number)
print(f"The factorial of {number} is: {result}")
```

In this script, we define a function `factorial(n)` that takes a number `n` as input and returns its factorial. We initialize the variable `fact` to 1, as the factorial of 0 and 1 is 1. We use a `for` loop to iterate over integers from 1 to `n` (inclusive). Inside the loop, we multiply the current value of `fact` by the loop variable `i`. After the loop, we return the final value of `fact`, which represents the factorial of the input number. We prompt the user to enter a number, compute its factorial using the `factorial()` function, and then print the result.

Output:



```
Terminal
Enter a number: 7
The factorial of 7 is: 5040
```

3.2.2. `for...else` Loop

In Python, the `for...else` loop is a variation of the regular `for` loop that includes an optional `else` block. This `else` block is executed when the loop completes normally, i.e., when it does not encounter a `break` statement that prematurely terminates the loop. The `else` block is not executed if the loop is terminated by a `break` statement.

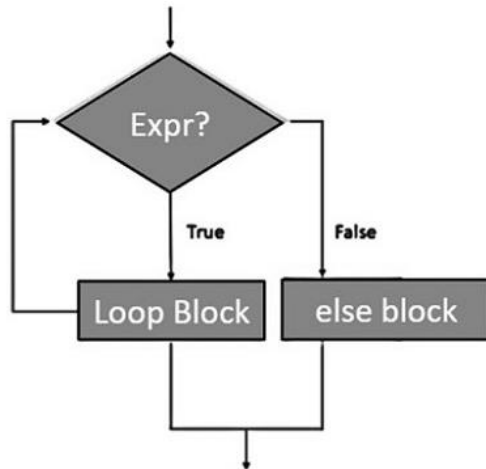
The general syntax of the `for...else` loop in Python is given below:

```
for item in iterable:
    # Code block in the loop
else:
    # Code block in the else clause
```

The `for` loop iterates over each item in the iterable, executing the specified code block for each iteration. After the loop completes all iterations without encountering a `break` statement, the optional `else` block is executed. If the loop is terminated prematurely by a `break` statement, the `else` block is skipped.

If the `else` statement is used with a `for` loop, the `else` statement is executed when the sequence is exhausted before the control shifts to the main line of execution. The following flowchart illustrates how to use `else` statement with `for` loop:

3. Control Structures in Python



The **for...else** loop is commonly used when we want to execute some additional code after the loop has iterated over all items in the iterable, but only if the loop completes successfully without any early termination. It provides a convenient way to handle scenarios where we need to perform actions based on the outcome of the loop iteration. It's important to note that the else block of a for...else loop will also be executed if the iterable is empty because the loop will complete normally without any iterations.

The following is an example to illustrate the usage of the **for...else** loop:

```
for i in range(5):
    print (f"Iteration no. {i}")
else:
    print ("for loop over. It's else block")
print ("End loop.")
```

This example illustrates the combination of an else statement with a for statement in Python. Till the `i` is less than 5, the iteration count is printed. As it becomes 5, the print statement in else block is executed, before the control is passed to the next statement in the main program.

On executing, this code will produce the following output:

```
Iteration no. 0
Iteration no. 1
Iteration no. 2
Iteration no. 3
Iteration no. 4
for loop over. It's else block
End loop.
```

Example 3.11: Searching a specific number in a list using the **for...else** loop in Python.

```
numbers = [10, 20, 30, 40, 50]
target = int(input("Enter the number to search for: "))

for index, num in enumerate(numbers):
    if num == target:
        print(f"Number {target} found at index {index}.")
```

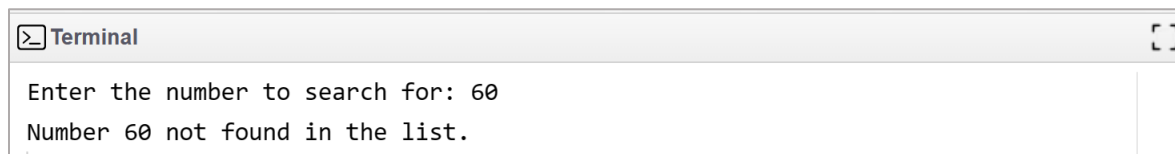
3. Control Structures in Python

```
        break
    else:
        print(f"Number {target} not found in the list.")
```

This Python script uses a for...else loop to search for a specific number in a list. If the number is found, it prints its index; otherwise, it prints a message indicating that the number was not found.

We use a **for...else** loop to iterate over each number in the numbers list. Inside the loop, we use the enumerate() function to get both the index and the value of each number. We check if the current number (num) is equal to the target number (target). If the target number is found, we print a message indicating its index and break out of the loop using the break statement. If the loop completes all iterations without finding the target number, the else block is executed, and a message indicating that the number was not found is printed.

Output:



```
Terminal
Enter the number to search for: 60
Number 60 not found in the list.
```

3.2.3. while Loop

In Python, a **while** loop is used to repeatedly execute a block of code as long as a specified condition evaluates to True. The loop continues to execute as long as the condition remains True, and it terminates when the condition becomes False. while loops are useful when the number of iterations is not known beforehand, or when you want to repeat a block of code until a specific condition is met.

Hence, a **while** loop statement in Python repeatedly executes a target statement as long as a given boolean expression is true. The general syntax of a **while** loop in Python is given below:

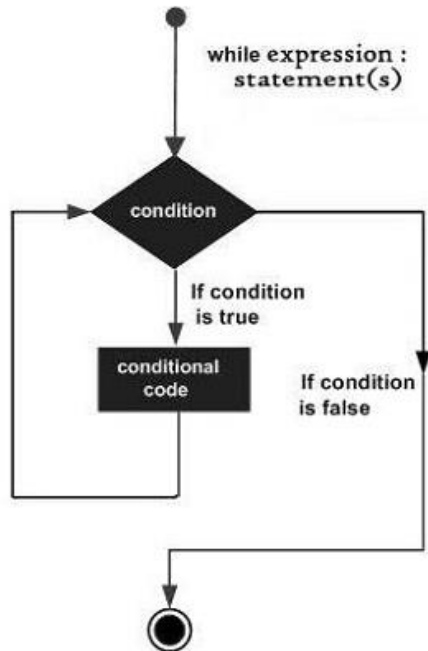
```
while condition:
    Code-block          # Execute while condition is True
```

In this **while** loop syntax:

- condition: This is a boolean expression that is evaluated before each iteration of the loop. If the condition evaluates to True, the loop continues executing; if it evaluates to False, the loop terminates.
- Code-block: This is the indented block of code that is executed repeatedly as long as the condition remains True. It contains the statements or operations that you want to repeat.

As soon as the expression becomes false, the program control passes to the line immediately following the loop. The following flowchart illustrates how the while loop works in Python:

3. Control Structures in Python



In a while loop, the loop condition is defined by a boolean expression that is evaluated before each iteration. The loop continues iterating as long as the condition evaluates to True.

The following is an example of using a while loop to count from 1 to 5:

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

In this example, we initialize the variable `i` to 1. The while loop checks if `i` is less than or equal to 5. If the condition `i <= 5` is True, the loop executes the indented block of code, which prints the current value of `i` and then increments `i` by 1 using the `i += 1` statement. The loop continues iterating until `i` becomes 6 (when `i` is no longer less than or equal to 5), at which point the condition becomes False, and the loop terminates.

It's essential to ensure that the condition of a while loop eventually becomes False to prevent infinite loops, which can lead to excessive CPU usage and program crashes. Moreover, we can use loop control statements like `break` and `continue` to control the flow of the while loop and handle special cases.

Example 3.12: Adding a list of numbers using the while loop in Python.

```
# List of numbers
numbers = [10, 20, 30, 40, 50]

total = 0
i = 0

# Compute sum of numbers using a while loop
while i < len(numbers):
```

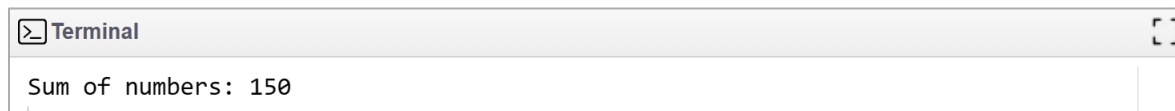

3. Control Structures in Python

```
total += numbers[i]
i += 1

# Print sum
print("Sum of numbers:", total)
```

In this script, we have a list of numbers `numbers`. We initialize two variables `total` and `i` to store the sum of numbers and the current index, respectively. We use a `while` loop to iterate over the list of numbers. Inside the loop, we add the number at the current index to the `total` and increment the index. The loop continues iterating until the index reaches the length of the `numbers` list. After the loop, we print the sum of the numbers.

Output:



```
Terminal
Sum of numbers: 150
```

Example 3.13: Reading a list of numbers and calculating sum and average of numbers using the `while` loop in Python.

```
numbers = []
# Read numbers from the user
while True:
    x = input("Enter a number (or press Enter to finish): ")
    if x == "":
        break
    try:
        number = float(x)
        numbers.append(number)
    except ValueError:
        print("Invalid input. Please enter a valid number.")

# Calculate sum and average of numbers
total = 0
i = 0
while i < len(numbers):
    total += numbers[i]
    i += 1

average = total / len(numbers)

# Print sum and average
print("Sum of numbers:", total)
print("Average of numbers:", average)
```

In this script, we initialize an empty list `numbers` to store the numbers entered by the user. We use a `while` loop with a condition `True` to repeatedly prompt the user for input until an empty input is provided (i.e., when the user presses Enter without typing anything).

3. Control Structures in Python

Inside the loop, we read the user input using the `input()` function and convert it to a floating-point number using `float()`. If the input is not a valid number, we handle the `ValueError` exception and prompt the user to enter a valid number. We append the valid number to the numbers list.

Once the user enters an empty input, indicating the end of input, we exit the loop using the `break` statement. After reading all the numbers, we initialize variables `total` and `index` to store the sum of numbers and the current index, respectively.

We use another `while` loop to iterate over the numbers list and compute their sum. Then we compute the average of the numbers. Finally, we print the sum of the numbers.

Output:

```
Terminal
Enter a number (or press Enter to finish): 12
Enter a number (or press Enter to finish): 24
Enter a number (or press Enter to finish):
Sum of numbers: 36.0
Average of numbers: 18.0
```

Example 3.14: Finding a series of Fibonacci numbers using the while loop in Python.

```
# Function to generate Fibonacci series
def fibonacci(n):
    # Initialize first two Fibonacci numbers
    a, b = 0, 1

    fib = []

    # Generate Fibonacci numbers using a while loop
    while a < n:
        fib.append(a)
        a, b = b, a + b

    return fib

# User input
n = int(input("Enter the limit for the Fibonacci series: "))

series = fibonacci(n)
print("Fibonacci series up to", n, ":", series)
```

In this script, we define a function `fibonacci(n)` that generates Fibonacci numbers up to a given number. We initialize variables `a` and `b` to represent the first two Fibonacci numbers (0 and 1, respectively). We initialize an empty list `fib` to store the Fibonacci numbers. We use a `while` loop to generate Fibonacci numbers until `a` exceeds the specified limit. Inside the loop, we append the current Fibonacci number `a` to the `fib` list, and then update the values of `a` and `b` to generate the next Fibonacci number. Once the loop exits, we return the list of Fibonacci numbers.

3. Control Structures in Python

We prompt the user to enter a number for the Fibonacci series. We call the `fibonacci()` function with the specified limit to generate the Fibonacci series. Finally, we print the generated Fibonacci series.

Output:

```
Terminal
Enter the limit for the Fibonacci series: 10
Fibonacci series up to 10 : [0, 1, 1, 2, 3, 5, 8]
```

3.2.4. while...else Loop

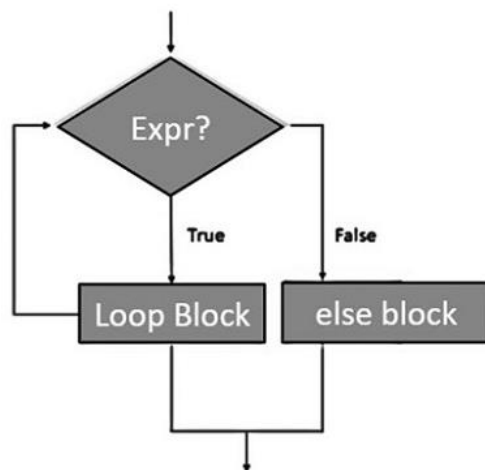
Python supports having an `else` statement associated with a `while` loop statement. If the `else` statement is used with a `while` loop, the `else` statement is executed when the condition becomes false before the control shifts to the main line of execution. Hence, a **while...else** statement is a variation of the regular `while` loop that includes an optional `else` block. The `else` block is executed when the `while` loop completes normally, i.e., when the loop condition becomes `False`. The `else` block is not executed if the loop is terminated prematurely using a `break` statement.

The general syntax of a **while...else** statement in Python is given below:

```
while condition:
    Code-block          # Execute while condition is True
else:
    Code-block          # Execute if the loop completes without a break
```

The **while** loop repeatedly executes the code block as long as the condition evaluates to `True`. After the loop completes all iterations without encountering a `break` statement, the optional **else** block is executed. If the loop is terminated prematurely by a `break` statement, the `else` block is skipped.

The following flowchart shows how to use `else` with `while` statement:



The **while...else** statement is commonly used when we want to execute some additional code after the `while` loop has finished iterating, but only if the loop completes successfully without any early termination. It provides a convenient way to handle scenarios where we need to perform actions based on the outcome of the loop iteration.

3. Control Structures in Python

The following is an example to illustrate the usage of the **while...else** statement in Python:

```
i = 0
while i < 3:
    i += 1
    print("Inside while loop: ", i)
else:
    print("Inside else block.")
```

In this example, the **while** loop iterates as long as the *i* is less than 3. Inside the loop, the message "Inside while loop" is printed three times. After the loop completes all iterations, the message "Now in else block" is printed, indicating that the else block is executed because the loop completed normally without encountering a **break** statement. If a **break** statement were encountered within the while loop, causing the loop to terminate early, the else block would not be executed.

On executing, this code will produce the following output:

```
Inside while loop: 1
Inside while loop: 2
Inside while loop: 3
Now in else block
```

Example 3.15: Computing a factorial of a number using the while...else loop in Python.

```
# Function to calculate the factorial of a number
def factorial(n):
    fact = 1
    i = 1
    # Calculate the factorial using a while loop
    while i <= n:
        fact *= i
        i += 1
    else:
        print("Factorial calculated successfully.")

    return fact

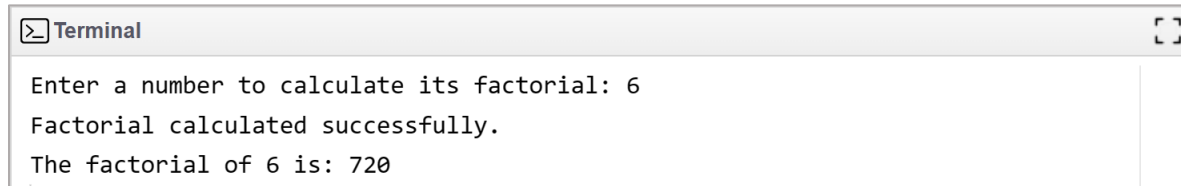
# User input
num = int(input("Enter a number to calculate its factorial: "))
# Calculate the factorial of the number
fact = factorial(num)
# Print the factorial
print(f"The factorial of {num} is: {fact}")
```

In this script, we define a function `factorial(n)` that calculates the factorial of a given number *n*. Inside the function, we initialize variables `fact` and `i`. `fact` is set to 1 initially, and `i` is used as a counter. We use a **while** loop to calculate the factorial. The loop runs as long as `i` is less than or equal to `n`. Inside the loop, we multiply `fact` by `i` and then increment `i` by 1. After the loop completes successfully (i.e., `i` becomes greater than `n`), the else block is executed, and a message is printed indicating that the factorial was calculated successfully.

3. Control Structures in Python

We prompt the user to enter a number and call the factorial() function with the user-provided number to calculate its factorial. Finally, we print the factorial of the given number.

Output:



```
Terminal
Enter a number to calculate its factorial: 6
Factorial calculated successfully.
The factorial of 6 is: 720
```

3.2.5. Using else Statement with Loops

As discussed earlier, Python allows us to implement the else functionality with for/while loops as well. The else functionality is available for use only when the loop terminates normally. In case of forceful termination of the loop else statement is overlooked by the interpreter and hence its execution is skipped. When the loop is not terminated by a break statement, the else block immediately after for/while is executed.

Method 1: For-Else Construct with normal termination (without break statement)

The following program shows how to use the else statement with for loop:

```
for i in ['A','B','C']:
    print(i)
else:
    # Loop else statement
    print("For-else statement successfully executed.")
```

On executing, this script will generate the following output:

```
A
B
C
For-else statement successfully executed.
```

Method 2: For-Else Construct with forceful termination (with break statement)

The following program shows how else conditions work in case of a break statement:

```
for i in ['A','B','C']:
    print(i)
    break
else:
    # Loop else statement
    print("For-else statement successfully executed.")
```

On executing, this script will generate the following output:

```
A
```

This type of else is only useful if there is an if condition inside the loop that is dependent on the loop variable in some way. In **Method 1**, the loop else statement is executed since the for loop

3. Control Structures in Python

terminates normally after iterating over the list ['A','B','C']. However, in **Method 2**, the loop-else statement is not executed since the loop is forcedly stopped using jump statements, such as break. These methods clearly show that when the loop is forcedly terminated, the loop-else expression is not executed.

Method 3: For-Else Construct with break statement and if conditions

Now we will consider an example in which the loop-else statement is performed in some conditions but not in others. The following example shows how else conditions works in case of break statement and conditional statements:

```
# Function to check even or odd
def number():
    for i in [1, 2, 3]:
        if i%2 == 0:
            print ("EVEN")
        else:
            print ("ODD")
            # break
    # Else statement of the for loop
    else:
        print ("Loop-else Executed")

# Function Call
number()
```

On executing, this script will generate the following output:

```
ODD
EVEN
ODD
Loop-else Executed
```

Method 4: Else-While without break statement

The following program demonstrates the use of the else statement in the while loop:

```
i = 0
while i<5:
    # incrementing by 1
    i += 1
    # printing i value
    print("i = ", i)
else:
    print("This is a while-else block")
```

On executing, this script will generate the following output:

```
i = 1
i = 2
i = 3
```

```
i = 4
i = 5
This is an else block
```

Method 5: Else-While with break statement

The following program demonstrates the use of the else statement in the while loop with a break:

```
def Numbers(l):
    n = len(l)
    i = 0
    while i < n:
        if l[i] % 2 == 0:
            print("The input list contains an even number")
            break
        i += 1
    else:
        print("The input list doesn't contain an even number")

# Example
print("Input list 1:")
Numbers([3, 9, 4, 5])
print("Input list 2:")
Numbers([7, 3, 5, 1])
```

On executing, this script will generate the following output:

```
Input list 1:
The input list contains an even number
Input list 2:
The input list doesn't contain an even number
```

3.2.6. Nested Loops

Nested loops in Python refer to the situation where one loop is placed inside another loop. This allows us to iterate over multiple levels of data structures, such as lists of lists or nested dictionaries. Nested loops are used when we need to perform operations on each element of a collection of collections or when dealing with multi-dimensional data.

The syntax for nested loops in Python is straightforward. We can simply place one loop inside another loop, typically with each inner loop indented within the outer loop.

The following is an example of nested for loop syntax:

```
for i in outer_collection:           #Outer loop
    for j in inner_collection:       #Inner loop
        # Perform operations using i and j item
```

The outer loop iterates over the elements of the outer collection (e.g., a list or range). For each iteration of the outer loop, the inner loop iterates over the elements of the inner collection (e.g., a sublist or another list). Inside the nested loops, we can perform operations using both the *i* and *j* item.

3. Control Structures in Python

Nested loops can be used for various tasks, such as traversing multi-dimensional arrays, sorting elements in an array, and performing matrix operations.

The following is an example of nested loops used to traverse a 2D list or array:

```
matrix = [[1, 2, 3],
          [4, 5, 6],
          [7, 8, 9]]

for row in matrix:
    for element in row:
        print(element, end=' ')
    print()
```

In this example, the outer loop iterates over each row of the matrix, and the inner loop iterates over each element in the current row. This allows us to print each element of the matrix row by row.

Example 3.16: Sorting elements of an array using nested loops in Python.

```
# Define an array of elements
array = [30, 10, 40, 50, 90, 20, 60, 70, 30, 99]

# Get the length of the array
n = len(array)

# Perform bubble sort using nested loops
for i in range(n):
    for j in range(0, n-i-1):
        if array[j] > array[j+1]:
            # Swap elements if they are in the wrong order
            array[j], array[j+1] = array[j+1], array[j]

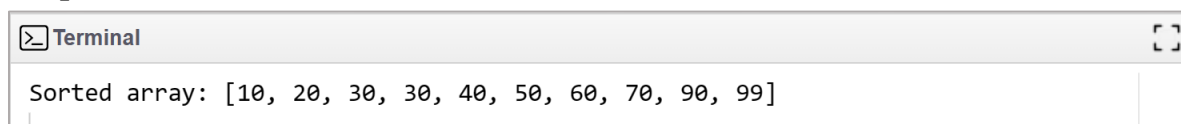
# Print the sorted array
print("Sorted array:", array)
```

In this script, we define an array array with some elements. We use nested loops to perform bubble sort:

- The outer loop (for i in range(n)) iterates over each element of the array.
- The inner loop (for j in range(0, n-i-1)) iterates over each pair of adjacent elements in the array.
- Inside the inner loop, we compare adjacent elements and swap them if they are in the wrong order.

After the loops complete, the array will be sorted in ascending order. Finally, we print the sorted array.

Output:



```
Terminal
Sorted array: [10, 20, 30, 30, 40, 50, 60, 70, 90, 99]
```


Example 3.17: Adding two matrices using nested loops in Python.

```
# Define two matrices as lists of lists
matrix1 = [[1, 2, 3],
           [4, 5, 6],
           [7, 8, 9]]

matrix2 = [[9, 8, 7],
           [6, 5, 4],
           [3, 2, 1]]

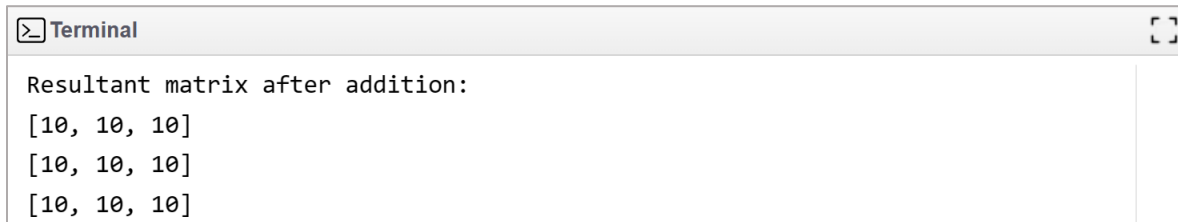
# Initialize an empty matrix to store the result
matrix3 = []

for i in range(len(matrix1)):
    sum_row = []
    for j in range(len(matrix1[i])):
        sum_row.append(matrix1[i][j] + matrix2[i][j])
    # Append sum_row to the result matrix
    matrix3.append(sum_row)

# Print the result matrix
print("Resultant matrix after addition:")
for row in matrix3:
    print(row)
```

In this example, we define two matrices `matrix1` and `matrix2` as lists of lists. We initialize an empty matrix `matrix3` to store the result of the addition. We iterate over the rows of both matrices using a nested loop. The outer loop iterates over the rows of `matrix1` (or `matrix2`, as they have the same number of rows), and the inner loop iterates over the elements of each row. Inside the nested loop, we add corresponding elements from `matrix1` and `matrix2`, and append the sum to a new row `sum_row`. After the inner loop completes for each row, we append `sum_row` to the `matrix3`. Finally, we print the `matrix3`, which contains the sum of the two input matrices.

Output:



```
Terminal
Resultant matrix after addition:
[10, 10, 10]
[10, 10, 10]
[10, 10, 10]
```

