MASTER OF COMPUTER APPLICATIONS

SEMESTER 1

PYTHON PROGRAMMING

Unit 3

Control Structures

Table of Contents

1. INTRODUCTION

Unit 2 was all about laying the groundwork with Python syntax and variables, wasn't it? You dove into how Python's flexibility allows for dynamic typing and variable assignment, and how different data types and their operations shape the way we structure our code. It was like getting the pieces of a puzzle, each variable and data type a building block for more complex structures.

Now, as we step into Unit 3, it's time to bring those pieces together with Control Structures. Think of it as moving from learning how to write sentences to crafting full-fledged stories. Control structures will enable your code to make decisions, loop through data, and handle tasks more efficiently. Why is this important, you ask? Well, without control structures, programs would run from the first line to the last without any sense of interactivity or decision-making. Pretty dull, right?

But here's where it gets exciting. You'll learn to use if, else, and elif statements to make your programs make decisions on the fly. Then, there are loops - for and while - that will repeat tasks without you having to write the same code over and over. And let's not forget the pass statement, a handy tool for when you're sketching out your program's structure but aren't quite ready to fill in all the details.

So, how do you tackle this new chapter? First, start by understanding the theory behind each control structure—knowing when and why to use each one is key. Then, get your hands dirty with code. Experiment with different scenarios, tweak conditions, and see how your program's behavior changes. And remember, making mistakes is part of the process. Each error is a step towards mastering Python's control structures.

By the end of this unit, your goal is to not just understand control structures but to think algorithmically. You'll be analyzing problems and deciding which control structure best suits each situation. Ready to make your code smarter and more efficient? Let's dive in!

1.1 Learning Objectives

At the end of this unit, you will be able to:

- *Analyze different scenarios to determine the most appropriate control structure for efficient problem-solving.*
- *Apply conditional statements effectively to direct the flow of a Python program based on dynamic conditions.*
- *Construct looping structures to automate repetitive tasks and manage sequences of data efficiently.*
- *Evaluate the use of nested control structures for complex decision-making processes and optimize their implementation.*
- *Design robust Python programs that incorporate if, else, elif, for, while, and pass statements to handle a variety of tasks and conditions.*
- *Demonstrate the ability to use the pass statement as a placeholder in developing program structures, ensuring code readability and maintainability.*

INSPIRED BY LIFE

2. CONTROL STRUCTURES

Control structures are the backbone of programming in Python, enabling developers to dictate the flow of their program's execution. These structures guide the decision-making processes and repetitive execution of code blocks, making dynamic and responsive programming possible.

At the core of any Python program lie control structures, which are mechanisms that allow for conditional execution and repetition of code segments. These structures are indispensable for creating programs that can react to different inputs and conditions, making decisions, and performing tasks multiple times without manual intervention.

2.1. Types of Control Structures in Python

Conditional Statements: These statements evaluate conditions and execute code blocks based on the outcome of the evaluation. They are the building blocks for decision-making in programming.

Looping Structures: Loops enable the execution of a code block multiple times, either for a set number of iterations or until a certain condition is met, making them ideal for tasks that require repetition.

INSPIRED BY LIFE

3. THE IF-ELSE STATEMENT

'If' is the simplest conditional statement in Python. When the specified condition of the 'if' statement is met: the block of code or compound statement under it is executed. The 'if' statement has the following structure in Python:

if < conditional expression > :

< block >

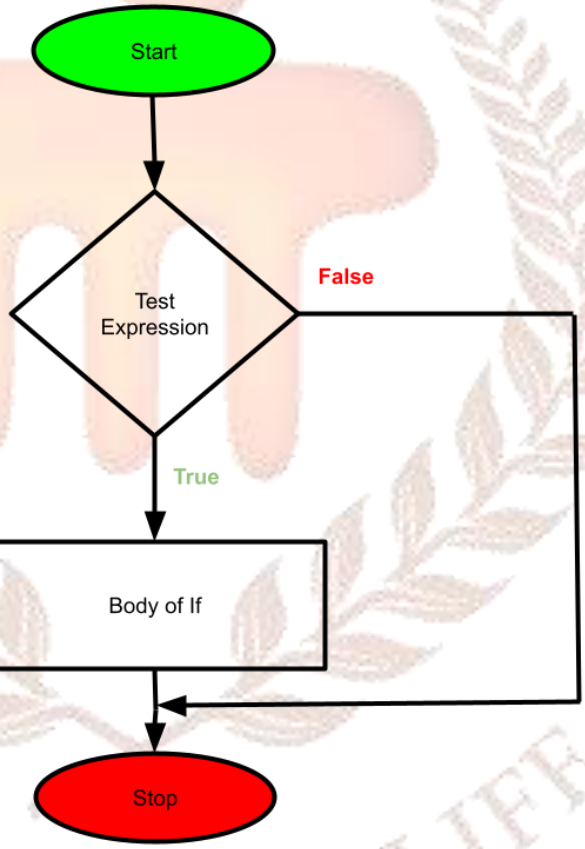

Fig 1: Flow Chart of if Statement

Example:

```
T = float(input('What is the water temperature? '))
if T > 24:
print('Water is safe!')
print('Jump in!')        # First line after if part
```

In the above example, Python will ask the user to input the water temperature. If the temperature is more than 24, the condition is 'True', and the print commands are executed. If the temperature is less than 24, the condition is 'False'. The program moves to the code after the if block without executing anything. In other words, the program gives no response when the temperature is below 24.

[Note that we need to convert the string input to float before we assign it to 'T']

However, as a programmer, you will want your application to perform a set of steps when a condition is met (True statement) and another set of steps when it is not (False statement). We use the if...Else statement to execute different blocks of code based on the True or False criteria.

The if...Else structure is as follows:

```
if < conditional expression > :
    < block >
else :
    < block >
```

INSPIRED BY LIFE

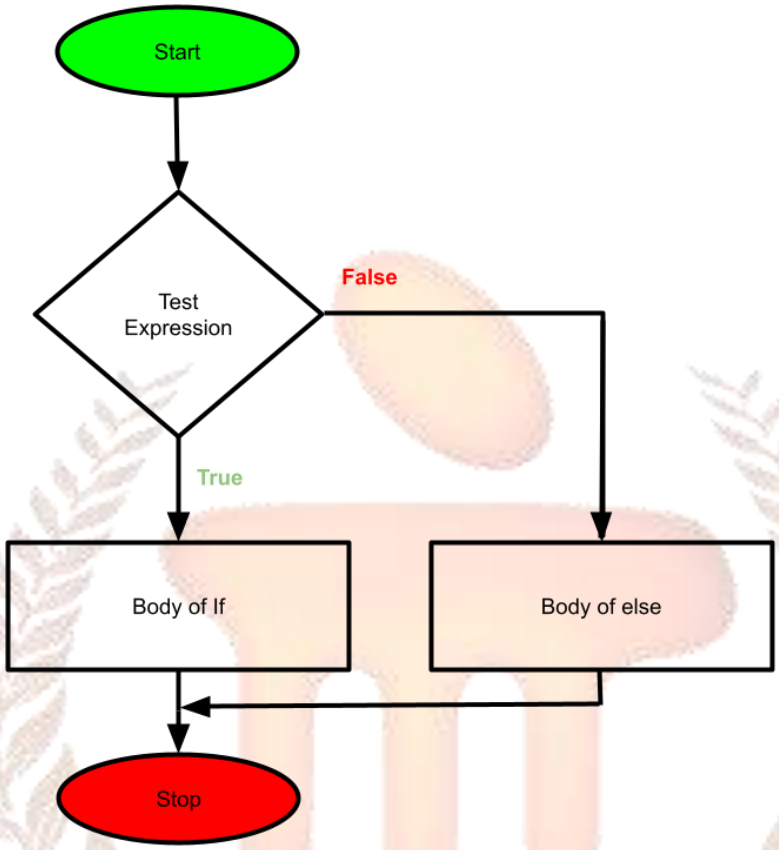

Fig 2: Flow Chart of if-else Statement

Example:

We can use the same example of temperature mentioned above:

```
T = float(input('What is the water temperature? '))
if T > 24:
    print('Water is safe!')
    print('Jump in!')
else:
    print('Better not go in!')
```

When the if condition evaluates to 'False', the print command in the else part is executed automatically. It is only logical because if the first condition is false, then the second one is true.

3.1. The el-if statement

The branching execution of the el-if (short for else if) statement is based on several alternatives. It is also known as the if...el-if...else statement and involves using many el-if statements. Python evaluates all the el-if statements in succession and only executes the block under the first el-if expression. In other words, it does not matter if there are more than one 'True' condition; python stops at the first 'True' value, executes it, and then exits the if...el-if...else structure.

In case no elif statement returns the "True" value, the else block is executed. In most cases, however, the else expression is not required and can be omitted. It is up to the programmer to determine whether an otherwise expression is required in the code.

You can use the elif expression in two ways:

3.2. Chained Conditionals:

Using el-if in the chained conditional expressions is quite straight forward as mentioned above and has the following syntax:

```
if < conditional expression > :
    < block >
elif < conditional expression > :
    < block >
elif < conditional expression > :
    < block >
elif < conditional expression >:
    < block >
elif < conditional expression > :
    < block >
.
.
.
else :
    < block >
```

Example:

```
if choice == 'a':
    print("You chose 'a'.")
elif choice == 'b':
    print("You chose 'b'.")
elif choice == 'c':
    print("You chose 'c'.")
else:
    print("Invalid choice.")
```

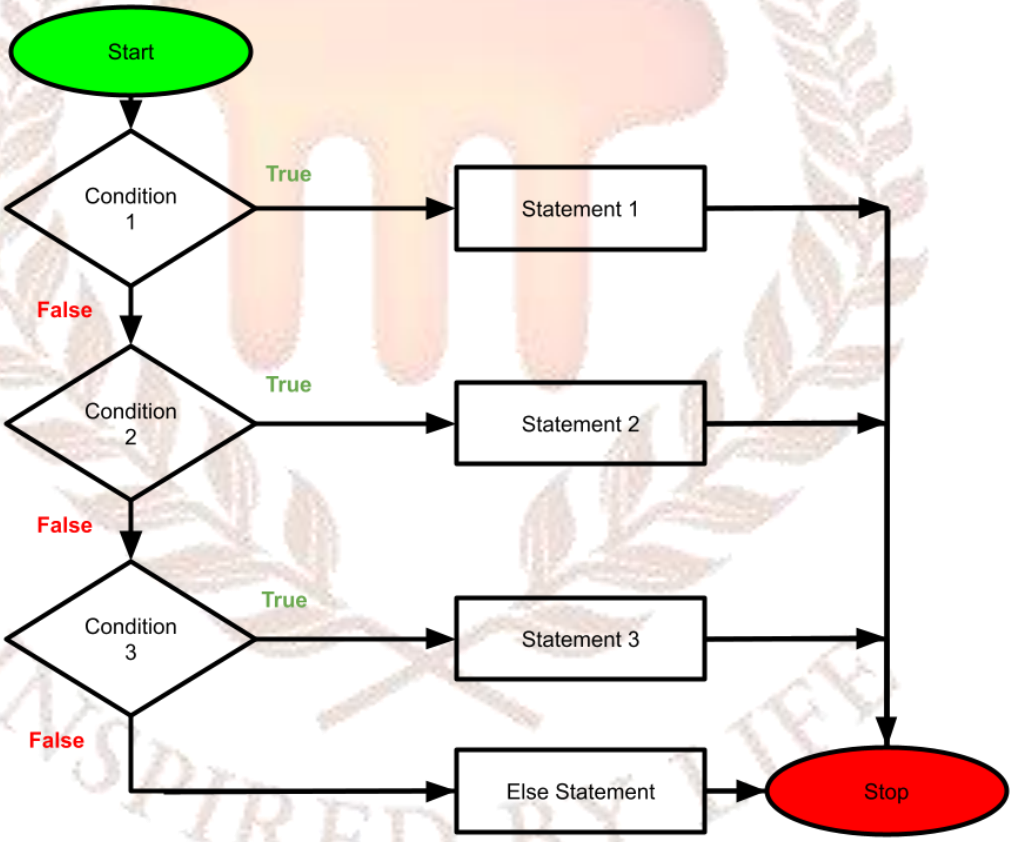

Fig 3: Flow Chart of if_el-if_else [IF-Else Ladder]

Each condition is checked in the given order. If the second elif statement returns 'True', "You chose 'b'." is printed, and the cursor exits the chained structure to move to the next code line.

Nested Conditionals:

In Nested conditionals, we can nest a set of conditional expression within another. It has the following syntax:

if < conditional expression > :

    < block >

else:

if < conditional expression > :

    < block >

else :

    < block >

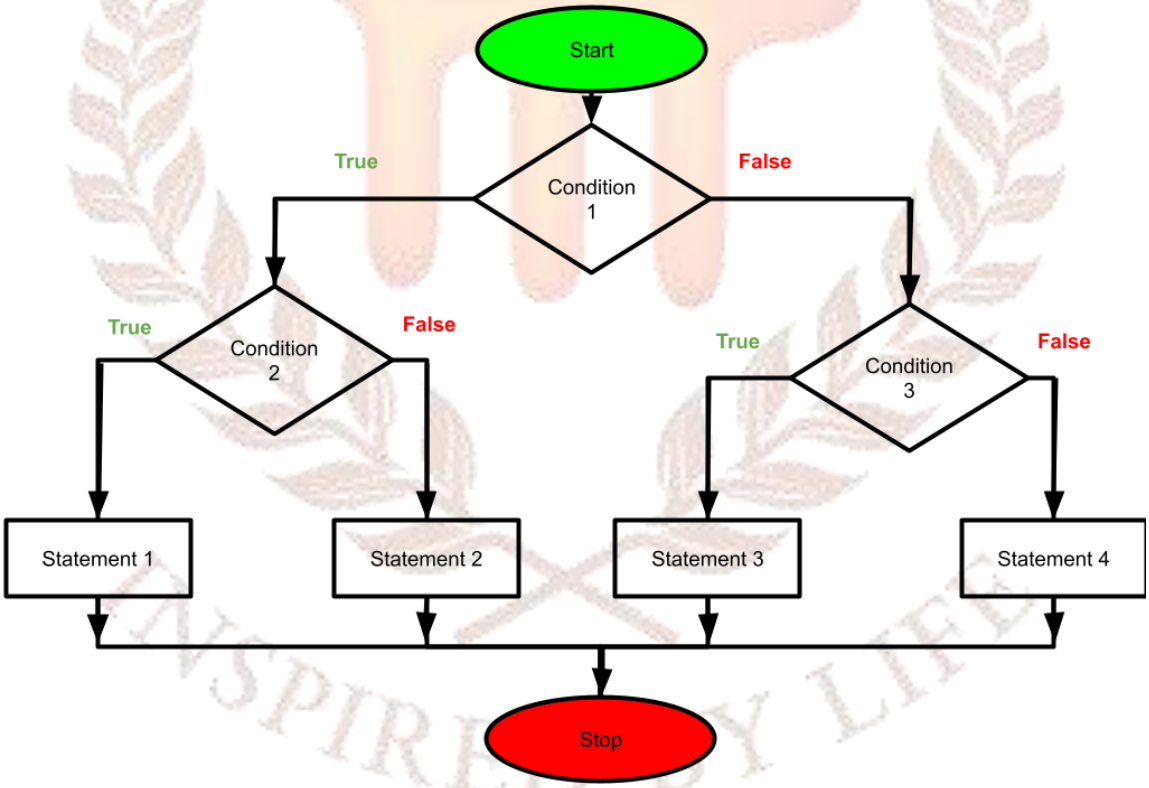

Fig 4: Flow Chart of Nested If-else

The nested format has an outer conditional with two branches. The second branch has an if expression with two branches of its own, and so on. Although this format has good structure

with indentation, it becomes very difficult to read nested expressions in the long run. Hence, it is better to avoid this when you can.

3.2. Composite Expressions:

To combine two conditional expressions, we use the logical operators: "and", "or", or both to make a composite conditional expression.

Example:

if < conditional expression > and < conditional expression > :

    < block >

OR

if < conditional expression > or < conditional expression > :

    < block >

3.3. Python if-else conditions

Conditional expressions in Python are logical expressions where logical operators compare, evaluate, or check if the input (operand) meets the conditions and returns 'True'. The block of code gets executed based on this. If the expression returns a 'False', the block does not execute, and the cursor moves on to the next line of code.

Python follows the common logical conditions/operators used in mathematics. These include:

- Equals to (==) → x==y: 'True' if value of x equals value of y. Otherwise, it is 'False'.
  if < conditional expression > == < conditional expression > :
  < block >
- Not equal to (!=) → x!=y: 'True' if the value of x does not equal the value of y. Otherwise, it is 'False'.
  if < conditional expression > != < conditional expression > :
  < block >

- Less than (<) →x<y: 'True' if the value of x is less than the value of y. Otherwise, it is 'False'.

  if < conditional expression > < < conditional expression > :

  < block >

- Less than or equal to (<=) → x<=y: 'True' if the value of x is less than or equal to the value of y. Otherwise, it is 'False'.

  if < conditional expression > <= < conditional expression > :

  < block >

- Greater than (>) → x>y: 'True' if the value of x is more than the value of y. Otherwise, it is 'False'.

  if < conditional expression > > < conditional expression > :

  < block >

- Greater than or equal to (>=) x>=y: 'True' if the value of x is more than or equal to the value of y. Otherwise, it is 'False'.

  if < conditional expression > >= < conditional expression > :

  < block >

INSPIRED BY LIFE

4. PYTHON LOOPS

The process of looping or iteration is fundamental to all programming languages and involves the repetition of tasks. A loop is a set of instructions (block) repeatedly executed if the defined conditions are met or satisfied. In Python, we use two statements for looping: the "for" statement and the "while" statement.

4.1. For Loop

The for loop executes each statement in a sequence and is hence used with Python's iterables that include: a string, list, tuple, dictionary, and set. The structure of for loop in Python is:

for < var > in < iterable > :

< statement >

< statement >

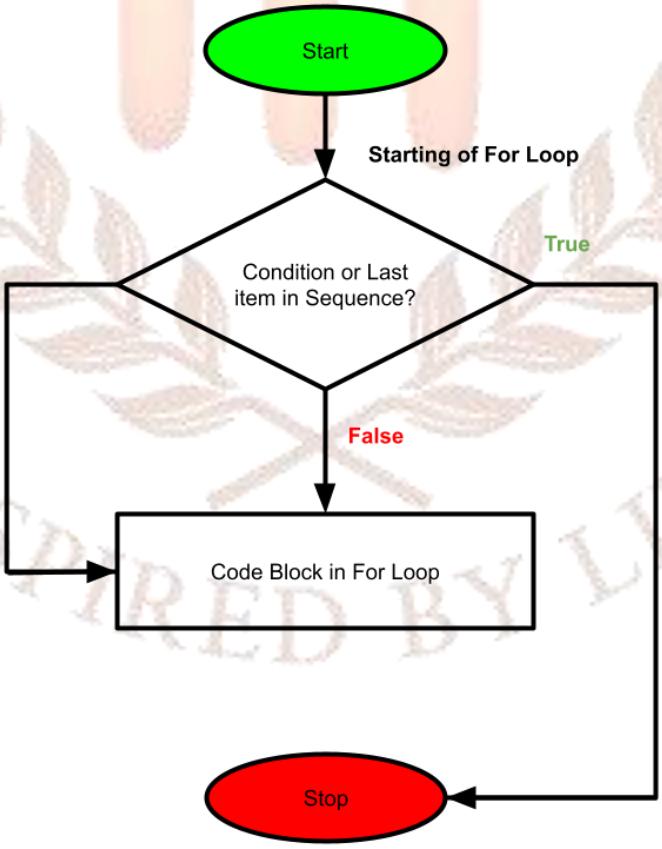

Fig 5: Flow Chart For Loop

In the above format, <var> represents the looping variable that updates to the next value of the iterable < iterable > each time the loop is executed.

Unlike the for loop in other programming languages, the Python for loop can incorporate the break statement, continue statement and the else clause.

As looping is controlled by the number of items in the iterable, infinite loops cannot likely be formed. However, the variable can accidentally be manipulated in the body of the loop, resulting in an infinite loop. Hence, programmers should check, vet, and recheck the code before execution.

Using Sequence

As mentioned above, loops use a set of items called iterables to execute statements. A Sequence is a generic term used to refer to iterables. Python has many types of sequences; however, the most common types are:

- Lists - These are the most versatile sequences. The elements of a list can be any object that can be added, reassigned or removed.
  fruits = ["apple", "banana", "cherry"]
  for x in fruits:
  print(x)

- Tuples - These sequences are similar to lists. However, they cannot be changed.
  thistuple = ("apple", "banana", "cherry")
  for x in thistuple:
  print(x)

- Strings - These are special sequences that can only store characters and have a unique notation. Apart from characters, sequence operators can also be stored as a string.
  for x in "banana":
  print(x)

Sequence Operators

- '+' operation combines two sequences through concatenation.

- '*' operation repeats a sequence a given number of times. Example: x*n; sequence x will add with itself 'n' number of times.
- x will return 'True' in Seq if it is an element of Seq. Otherwise, it will return 'False'.
- seq[i] will return the ith item in the sequence. You should note that Python starts with the '0' index; that is, the first has index 0, the second has index 1, and so on.
- seq[-i] will return the ith element from the end of the sequence.

All in one Example:

```
# Define a list, a string, and a tuple
list1 = [1, 2, 3]
string1 = "Hello"
tuple1 = (4, 5, 6)

# Use '+' to concatenate two sequences
list2 = list1 + [7, 8, 9] # list2 = [1, 2, 3, 7, 8, 9]
string2 = string1 + " World" # string2 = "Hello World"
tuple2 = tuple1 + (10, 11, 12) # tuple2 = (4, 5, 6, 10, 11, 12)

# Use '*' to repeat a sequence a given number of times
list3 = list1 * 2 # list3 = [1, 2, 3, 1, 2, 3]
string3 = string1 * 3 # string3 = "HelloHelloHello"
tuple3 = tuple1 * 2 # tuple3 = (4, 5, 6, 4, 5, 6)

# Use 'in' to check if an element is in a sequence
print(2 in list1) # True
print("H" in string1) # True
print(7 in tuple1) # False

# Use '[]' to access an element by index
print(list1[0]) # 1
print(string1[1]) # e
print(tuple1[2]) # 6
```

```
# Use '[-]' to access an element from the end of the sequence
print(list1[-1]) # 3
print(string1[-1]) # o
print(tuple1[-1]) # 6
```

Using Range

The range function is used to repeat or loop a block of code a given number of times. We know that Python is a zero-indexed programming language. Hence, by default, it returns a sequence of elements starting from zero indexes. The index sees an increment by 1 till the set range is reached.

The range() function is used as:

```
for < variable > in range( < number > ) :
    < block >
```

The loop will run from a 0 value variable to a value equal to the number specified minus one.

Example:

```
for x in range( 5 ) :
    print(x)
```

The loop will run from x = 0 through to x = 4 with increments of 1.

The starting value of a range() be change from the default 0:

```
for x in range( 2, 5 ) :
    print(x)
```

The loop will run from x = 2 through to x = 4 with increments of 1.

The increment values can also be changed as follows:

```
for x in range( 5, 16, 2 ) :
    print(x)
```

The loop will run from x = 5 through to x = 15 with increments of 2.

4.2. Nested For Loops in Python

Similar to the nested if conditional statements, programmers can also nest for loops within each other for better functionality. Python gives programmers the flexibility to nest loops in conditional statements, composite conditional statements, and vice versa without any restrictions. The nesting expression can be written as follows:

```
for [first iterating variable] in [outer loop]: # Outer loop
        < block > # Optional
    for [second iterating variable] in [nested loop]: # Nested loop
        < block >
```

The program evaluates the outer loop first, executing the first iteration. The iteration triggers the next (nested) loop to completion. The program returns to the top of the outer loop, completes the second iteration, and triggers the nested loop again. The loop continues till the sequence is complete or there is a disruption in the process.

Let's take a look at an example to understand how we can nest for loops in Python:

```
num_list = [1, 2, 3]
alpha_list = ['a', 'b', 'c']
for number in num_list:
    print(number)
    for letter in alpha_list:
        print(letter)
```

For this program, we get the output as:

Output

```
1
a
b
c
2
a
b
```

c

3

a

b

c

From the above output, we can see that the program executed the first iteration to print '1', goes to the nested loop to print 'a, b, c' consecutively. Once the inner loop is complete, it goes back to print '2' in the outer loop, and again prints 'a, b, c' by triggering the inner loop, so on.

Using Else Statement in Loop

As mentioned earlier, Python gives a programmer the freedom to nest and also use various conditional statements in loops. The else statement is one such statement that can be used in loops even without the if conditional statement.

However, else is only functional when the loop is terminated normally. If the loop is terminated forcefully, the else statement is overlooked and not executed.

The else statement can be executed in the for loop as follows:

```
for < var > in < iterable > :
    < statement >
else :
    < statement >
```

Let's look at an example:

```
for currentLetter in 'Hi everyone!':
    print ('The current letter is', currentLetter)
else:
    print ('All letters were printed.')
```

The output will be:

The current letter is H

The current letter is i

The current letter is

```
The current letter is e
The current letter is v
The current letter is e
The current letter is r
The current letter is y
The current letter is o
The current letter is n
The current letter is e
The current letter is !
All letters were printed.
```

We get the above output because the loop iteration ended normally and the else statement was executed.

Let's see what happens when the loop does not end normally:

```
for currentLetter in 'Hi everyone!':
    print ('The current letter is', currentLetter)
    if currentLetter == 'r':
        break
else:
    print ('All letters were printed.')
```

Output:

```
The current letter is H
The current letter is i
The current letter is
The current letter is e
The current letter is v
The current letter is e
The current letter is r
```

The for loop terminated as soon as the letter 'r' was encountered due to the if condition provided. The break statement disrupts the code abruptly. Hence, the else statement did not execute as the loop did not complete normally.

5. WHILE LOOP

The while loop is used to repeatedly execute a block of code as long as the given condition is satisfied. The while loop has the following syntax:

while < expression > :

< block >

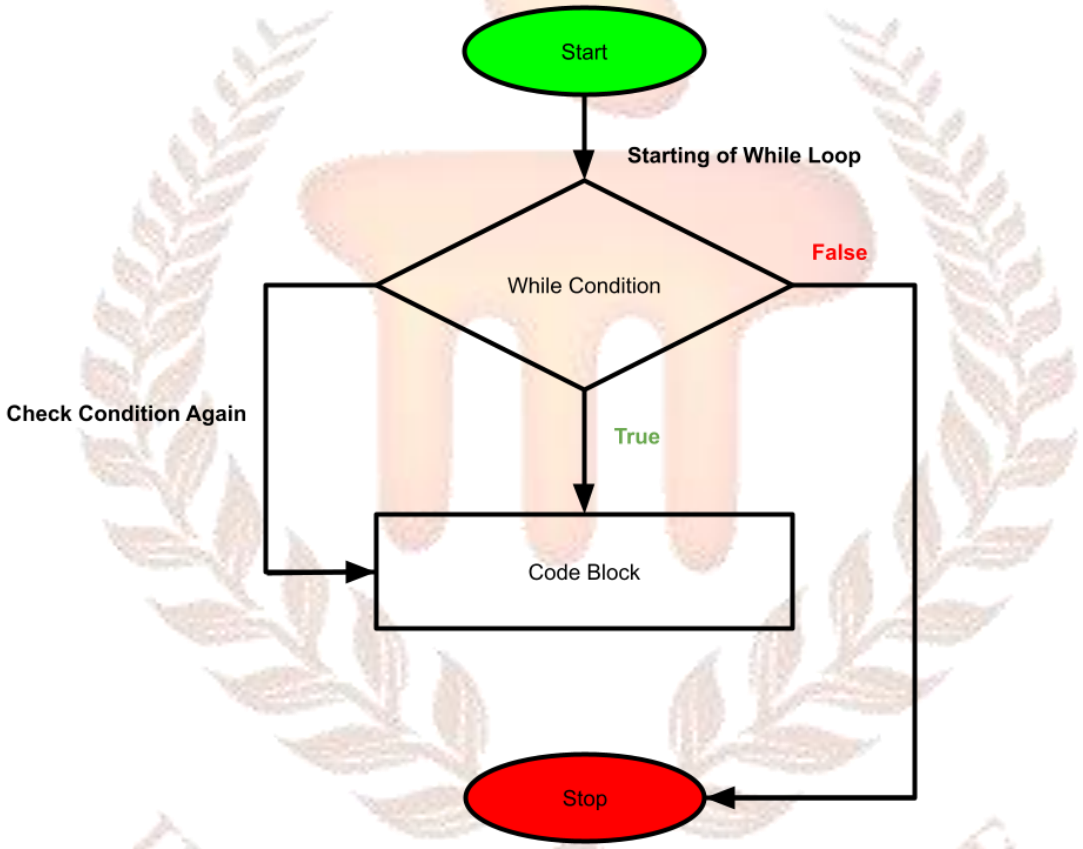

Fig 6: Flow Chart of While Loop

The expression controls the loop execution and involves looping variable(s) with logical operations. The looping expression sets the condition based on which the loop will continue to run or terminate.

While executing an assignment statement in the while loop, the right-hand side of the expression is evaluated first. The evaluated result is then assigned to the variable. In loop expressions, the variable is evaluated and repeatedly updated while the given condition is 'True'; otherwise, it is terminated.

An Example of a while loop:

```
x = 5
while x < 10:
    print (x)
    x+=1
```

Output:

```
5
6
7
8
9
```

The statement x += 1 adds 1 to x in each loop iteration. This ensures that the loop does not become an endless loop.

INSPIRED BY LIFE

6. BREAK, CONTINUE AND PASS

6.1. The Break Statement

In Python, the break statement is used to abruptly terminate the loop execution. It is mostly used under the if statement that sets the condition to break out of the loop. Once the program breaks out of the loop, it resumes the execution at the first line after the loop.

The break statement is used as:

```
while < expression > :
      < block >
if < statement >:
        break
< block >
```

Example:

```
word = input ('Type a word that is less than 5 characters long: ')
letterNumber = 1
for i in word:
print ('Letter', letterNumber, 'is', i)
letterNumber += 1
if letterNumber > 5:
break
print('The word you have entered is more than 5 characters long!')
```

The above code will print each letter of the word you have entered. However, if the character count exceeds 5, the break statement in the code will terminate the code after 5 characters.

Type a word that is less than 5 characters long: Avalanche

Letter 1 is A

Letter 2 is v

Letter 3 is a

Letter 4 is l

Letter 5 is a

The word you have entered is more than 5 characters long!

3.2. The continue Statement

Similar to the break statement, the continue suddenly stops the running iteration. However, instead of terminating the loop, it goes back to the loop expression for re-evaluation before continuing. If the re-evaluated results do not comply with the expression, the loop terminates, and the program moves to the next line after the body of the loop. If the expression is still valid, the program continues with the next iteration.

The continue statement is used as:

```
while < expression > :
        < block >
if < statement >:
        continue
< block >
```

Example:

```
for i in range(6,15):
        if i==9:
          continue
      print (i)
```

Output:

6
7
8
10
11
12
13

14

The program prints all numbers between 6 to 15 except 9. When the variable i is equal to 9, the if statement is executed, and the continue statement under it will force the program to skip the print statement before it moves to the next iterable.

6.3. The pass statement

The pass statement is unique in Python, acting as a syntactic placeholder where the language requires a statement, but no action needs to be taken. It's the equivalent of an empty statement or a no-op (no operation), but it plays a crucial role in maintaining the structural integrity and readability of Python code.

Purpose and Usage

The primary purpose of the pass statement is to fill syntactical requirements without altering the behaviour of the program. It is commonly used in blocks where a statement is syntactically necessary, but the programmer does not want to execute any code.

Scenarios for Using pass:

Under Development: When writing new functions or control structures, pass can be used as a placeholder, allowing the programmer to focus on the higher-level structure of the code before implementing the details.

```
def future_function():
    pass  # Future implementation goes here
```

Conditional Statements: In complex conditional branches, sometimes no action is required in one or more branches. The pass statement can be used to denote these intentionally empty branches.

```
if condition_met():
    pass  # No action required
else:
    perform_alternative_action()
```

Loops: In loops, particularly with while, you might encounter scenarios where the loop's condition is managed within the loop itself, making the loop body unnecessary under certain conditions.

```
while condition_met():
    pass  # Condition managed elsewhere
```

Class Definitions: When defining a class structure without implementing any methods, pass can be used to create a minimal class body.

```
class EmptyClass:
    pass  # No attributes or methods yet
```

Examples:

Create a function named placeholder_function using the pass statement. Then, replace pass with a print statement that says "Function Implemented".

```
def placeholder_function():
    pass  # Replace this with a print statement


# After modification
def placeholder_function():
    print("Function Implemented")
```

Define a loop that iterates over a list of numbers. Use the pass statement to skip even numbers and print odd numbers.

```
numbers = [1, 2, 3, 4, 5]
for number in numbers:
    if number % 2 == 0:
        pass  # Modify this to skip even numbers
    else:
        print(number)
```

Common Pitfalls

- Overuse: While pass is useful as a placeholder, overusing it can lead to code that's hard to read and maintain. It's best used sparingly and replaced with actual code as soon as practical.
- Silent Failure: Using pass in error handling, such as within an except block, can lead to silent failures, making debugging difficult. Always log or handle exceptions explicitly.

```
try:
    risky_operation()
except Exception:
    pass  # This can hide errors. Consider logging the exception or handling it explicitly.
```

Best Practices

- Comments: When using pass, it's helpful to include a comment explaining why no action is taken or indicating that the code block is a placeholder for future implementation.
- Refactoring: Regularly revisit sections of code that contain pass to assess if they can be refactored or implemented. This keeps the codebase active and prevents stagnation.

INSPIRED BY LIFE

7. SUMMARY

In this unit, we embarked on a journey through the essential control structures in Python, uncovering the mechanisms that empower our programs to make decisions, repeat tasks, and navigate complex data. We began with an exploration of conditional statements, where we learned to use if, else, and elif to direct program flow based on specific conditions. This not only introduced us to the basics of decision-making in programming but also highlighted the importance of evaluating conditions accurately to achieve desired outcomes.

We then ventured into the realm of looping structures, where for and while loops opened up new possibilities for automating repetitive tasks and efficiently managing sequences of data. Through practical examples and exercises, we experienced first-hand how loops can transform cumbersome tasks into streamlined processes, making our code more efficient and our programming tasks less daunting.

The unit also introduced the pass statement, a unique feature in Python that serves as a syntactical placeholder. We discovered how pass can be strategically used in developing program structures, providing flexibility during the initial stages of coding and ensuring clarity and maintainability in the long run.

Throughout this unit, interactive exercises and real-world examples reinforced our understanding, allowing us to apply these control structures creatively in various scenarios. We tackled common pitfalls, such as the nuances of Python's indentation and the logical intricacies of conditional expressions, emerging with a deeper comprehension and an enhanced ability to troubleshoot and optimize our code.

As we conclude this unit, we're equipped not just with theoretical knowledge but with practical skills that will serve as a foundation for more advanced programming concepts. The control structures we've mastered are not mere syntax; they are powerful tools that enable us to bring logic, efficiency, and sophistication to our Python programs.

8. GLOSSARY

- Control Structures: Mechanisms in programming that dictate the flow of program execution, enabling conditional execution and repetition of code blocks.
- Conditional Statements: Code structures that allow execution of certain code blocks based on the evaluation of a condition.
  - if Statement: A basic control structure that executes a block of code if a specified condition is true.
  - else Statement: Used in conjunction with if, it specifies a block of code to be executed if the if condition is false.
  - elif Statement: Short for "else if", it allows for multiple conditional expressions to be evaluated in sequence, providing alternative execution paths.
- Looping Structures: Control structures that repeat a block of code multiple times until a certain condition is met or a sequence is exhausted.
  - for Loop: Iterates over a sequence (such as a list, tuple, or string) and executes a block of code for each item.
  - while Loop: Continuously executes a block of code as long as a specified condition remains true.
- pass Statement: A null operation in Python used as a placeholder in situations where a statement is syntactically required but no action is needed.
- Iteration: The process of looping through items in a sequence or performing a block of code repeatedly.
- Nested Loops: A loop inside another loop, allowing for the iteration over complex, multi-dimensional structures.
- Break Statement: Terminates the nearest enclosing loop, skipping any remaining iterations.
- Continue Statement: Skips the rest of the code inside the current loop iteration and moves to the next iteration of the loop.
- Infinite Loop: A loop that has no terminating condition or one that never becomes false, causing the loop to run indefinitely.
- Sequence: An ordered collection of items, such as lists, tuples, and strings, over which iteration can occur.

- Boolean Expression: An expression that evaluates to either True or False, commonly used as the condition in control structures.
- Composite Condition: A condition formed by combining multiple boolean expressions using logical operators like and, or, and not.
- Truthy and Falsy Values: In Python, values that evaluate to True in a boolean context are considered truthy, while those that evaluate to False are considered falsy.

9. SELF-ASSESSMENT QUESTIONS

1. What does the if statement do in Python?
   A) Repeats a block of code multiple times
   B) Executes a block of code if a certain condition is true
   C) Defines a new function
   D) None of the above
2. How do you write an if statement in Python to check if a is equal to 10?
   A) if a = 10:
   B) if a == 10:
   C) if a equals 10:
   D) if a in 10:
3. What is the purpose of the else statement in Python?
   A) To specify a block of code to execute if no conditions are true
   B) To create a new variable
   C) To exit the program
   D) To loop through a sequence
4. Which loop is used to iterate over a sequence (like a list, tuple, or string)?
   A) while loop
   B) do-while loop
   C) for loop
   D) repeat-until loop
5. What does the while loop in Python do?
   A) Executes a set of statements as long as a condition is false
   B) Executes a set of statements as long as a condition is true
   C) Iterates over a sequence of numbers
   D) Iterates over a fixed sequence
6. How can you break out of a loop in Python?
   A) exit()
   B) break
   C) stop
   D) return

7. What is the use of the pass statement in Python?
   A) To exit a loop
   B) To pass control to another part of the program
   C) To act as a placeholder for future code
   D) To pass arguments to functions
8. Where can the pass statement be used?
   A) Inside a function definition only
   B) Inside loops only
   C) Anywhere a statement is syntactically required but no action is desired
   D) Inside conditional statements only
9. Which of the following is a correct use of the pass statement?
   A) if x > 5: pass
   B) pass if x > 5
   C) if x > 5: execute pass
   D) if pass: x > 5
10. What would be the output of the following code snippet?

```
x = 10

if x > 5:

print("Greater")

else:

print("Smaller")
```

   A) Greater
   B) Smaller
   C) Error
   D) No output
11. Which statement is used to skip the current iteration of a loop in Python?
   A) skip
   B) break
   C) continue

D) pass

12. How do you check multiple conditions in a sequence in Python?
    A) Using multiple if statements
    B) Using the elif statement
    C) Using the else statement
    D) Using the for loop
13. What does the following code print?

```
for i in "hello":
if i == "l":
continue
print(i)
```

    A) heo
    B) hello
    C) helo
    D) he
14. Which of the following is true about the while loop in Python?
    A) The while loop can only execute a single statement per iteration.
    B) The condition for a while loop is checked after executing the loop's code block.
    C) The while loop is used to iterate over a sequence of numbers.
    D) A while loop continues to execute as long as the given condition is true.
15. In Python, how can you execute a block of code at least once and then repeat it as long as a condition is true?
    A) Using a do-while loop
    B) Using a for loop with an initial execution before the loop
    C) Using a while loop with an initial execution before the loop
    D) Python does not support this behavior

10. TERMINAL QUESTIONS

1. How would you write an if statement in Python to check if a variable a is equal to 100?
2. Describe how to use an if-else statement to print "Odd" if a number n is odd and "Even" if the number is even.
3. Explain the purpose and usage of the elif statement in Python with an example.
4. Write a for loop in Python that prints numbers from 1 to 10.
5. Describe how a while loop works in Python with an example that prints numbers from 1 until a variable stop becomes True.
6. Explain how to use a for loop to iterate over a list named items and print each item.
7. What is the pass statement, and when would you use it in Python programming?
8. Provide an example of how the pass statement can be used in a function that is yet to be implemented.
9. Describe a scenario where the pass statement is used within a loop in Python.
10. How can you nest an if statement within a for loop to check for even numbers in a list and print them?
11. Describe how to combine if and elif statements within a while loop to create a simple menu-driven program.
12. Explain the concept of nested loops with an example that prints a 3x3 matrix of asterisks (*).
13. How would you write a Python script that uses both break and continue statements within a loop to process a list of numbers and stop when it finds a number greater than 10?
14. Describe a practical use case for using nested if statements within a for loop in data processing.
15. Explain how to use a while loop with an else clause in Python, with an example.
16. Write a Python program using control structures to read a list of temperatures and classify them as 'High', 'Medium', or 'Low'.
17. How can you use control structures to validate user input in a Python program that requires an integer input between 1 and 10?
18. Describe a scenario where you would use a combination of if-elif-else statements and loops to process user commands in a text-based adventure game.

19. How do you handle potential errors when using control structures to access elements in a list based on user input?
20. Discuss the best practices for nesting control structures in Python to maintain code readability and efficiency.

11. ANSWERS

Self-Assessment Questions

1. Answer: B
2. Answer: B
3. Answer: A
4. Answer: C
5. Answer: B
6. Answer: B
7. Answer: C
8. Answer: C
9. Answer: A
10. Answer: A
11. Answer: C
12. Answer: B
13. Answer: A
14. Answer: D
15. Answer: C

Terminal Questions

1. Answer Location: Section on Conditional Statements
2. Answer Location: Section on Conditional Statements - The if-else statement
3. Answer Location: Section on Conditional Statements - The elif statement
4. Answer Location: Looping Structures - for Loop
5. Answer Location: Looping Structures - while Loop
6. Answer Location: Looping Structures - for Loop
7. Answer Location: Section on The pass Statement

8. Answer Location: Section on The pass Statement
9. Answer Location: Section on The pass Statement
10. Answer Location: Composite and Nested Control Structures
11. Answer Location: Composite and Nested Control Structures
12. Answer Location: Composite and Nested Control Structures
13. Answer Location: Advanced Control Flow
14. Answer Location: Advanced Control Flow
15. Answer Location: Advanced Control Flow
16. Answer Location: Practical Applications of Control Structures
17. Answer Location: Practical Applications of Control Structures
18. Answer Location: Practical Applications of Control Structures
19. Answer Location: Error Handling and Best Practices in Control Structures
20. Answer Location: Error Handling and Best Practices in Control Structures

INSPIRED BY LIFE