Control Statements

Unit 2

Introduction

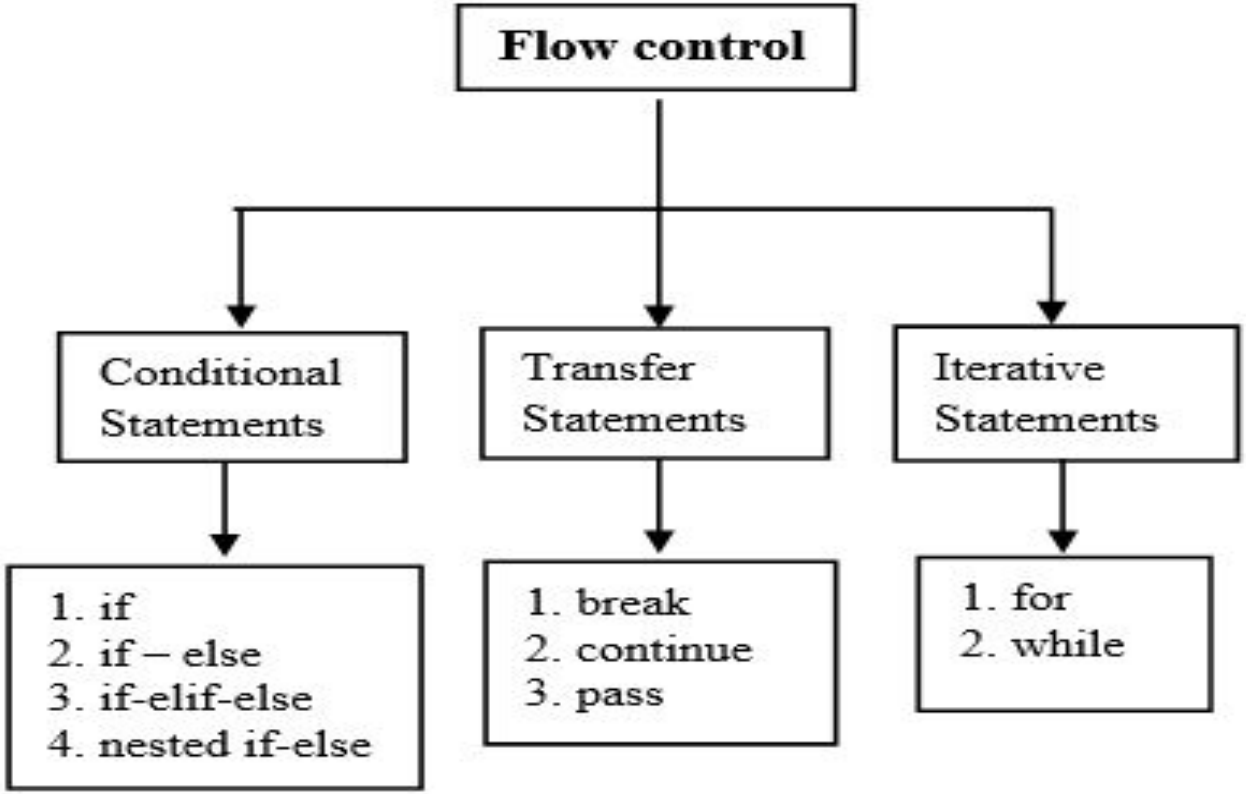
Control statements in Python are fundamental constructs that enable you to control the flow of execution within a program.

They allow you to make decisions, repeat actions, and handle exceptions based on certain conditions.

Control Flow Statements

The flow control statements are divided into three categories

1. Conditional statements
2. Iterative statements.
3. Transfer statements


Flow control
Conditional Statements
Transfer Statements
Iterative Statements
1. if
2. if – else
3. if-elif-else
4. nested if-else
1. break
2. continue
3. pass
1. for
2. while

Conditional statements

In Python, condition statements act depending on whether a given condition is true or false. You can execute different blocks of codes depending on the outcome of a condition. Condition statements always evaluate to either True or False.

There are three types of conditional statements.

- if statement
- if-else
- if-elif-else
- nested if-else

Iterative statements

In Python, iterative statements allow us to execute a block of code repeatedly as long as the condition is True. We also call it a loop statements.

Python provides us the following two loop statement to perform some actions repeatedly

- for loop
- while loop

Transfer statements

In Python, transfer statements are used to alter the program’s way of execution in a certain manner.

For this purpose, we use three types of transfer statements.

- break statement
- continue statement
- pass statements

Conditional statements

If statement

In control statements, The if statement is the simplest form. It takes a condition and evaluates to either True or False.

If the condition is True, then the True block of code will be executed, and if the condition is False, then the block of code is skipped, and The controller moves to the next line

Syntax of the if statement

```
if condition:
    statement 1
    statement 2
    statement n
```

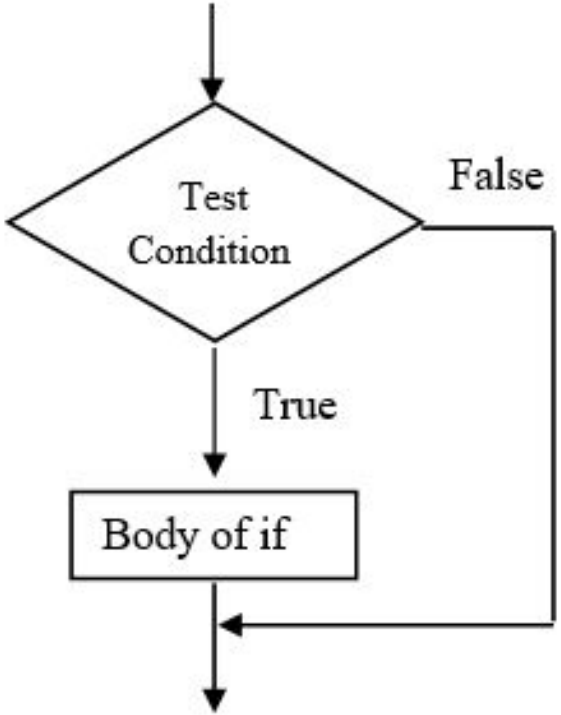


Fig. Flowchart of if statement

Example

```
number = 6

if number > 5:

    print(number * number)

print('Next lines of code')

// here indentation is important
```

If – else statement

The if-else statement checks the condition and executes the if block of code when the condition is True, and if the condition is False, it will execute the else block of code.

Syntax of the if-else statement

```
if condition:
    statement 1
else:
    statement 2
```

If the condition is True, then statement 1 will be executed If the condition is False, statement 2 will be executed. See the following flowchart for more detail.

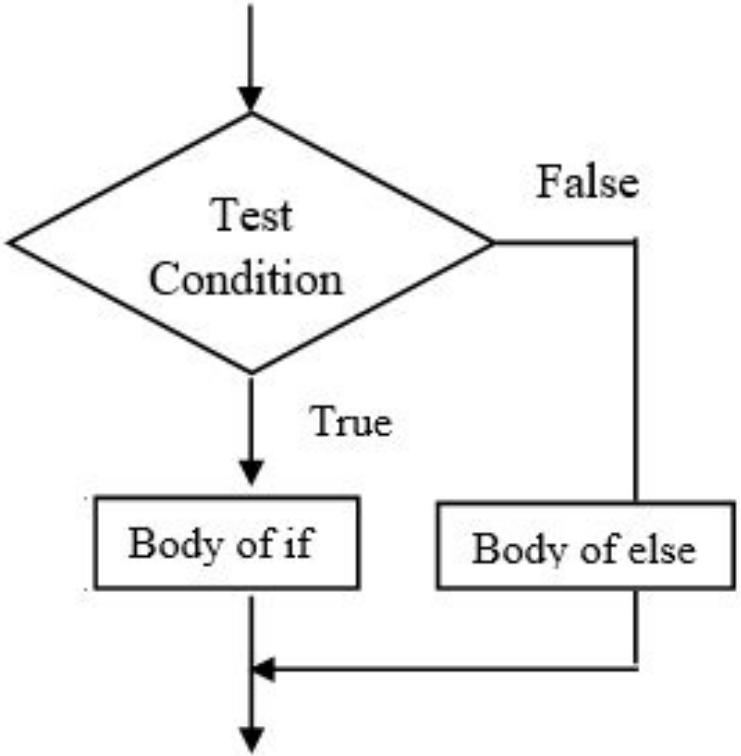


Fig. Flowchart of if-else

example

```
a =20
if a%2==0:
    print("a is even")
else:
    print("a is odd")
```

Chain multiple if statement

In Python, the if-elif-else condition statement has an elif blocks to chain multiple conditions one after another. This is useful when you need to check multiple conditions.

With the help of if-elif-else we can make a tricky decision. The elif statement checks multiple conditions one by one and if the condition fulfills, then executes that code.

Syntax of the if-elif-else statement

```
if condition-1:
    statement 1
elif condition-2:
    stetement 2
elif condition-3:
    stetement 3
    ...
else:
    statement
```

Example

```
a =20
if a>0:
    print("a is positive")
elif a<0:
    print("a is negative")
else:
    print("a is equal to zero")
```

Nested if-else statement

In Python, the nested if-else statement is an if statement inside another if-else statement. It is allowed in Python to put any number of if statements in another if statement.

Indentation is the only way to differentiate the level of nesting. The nested if-else is useful when we want to make a series of decisions.

Syntax of the nested-if-else:

```
if conditon_outer:
    if condition_inner:
        statement of inner if
    else:
        statement of inner else:
    statement ot outer if
else:
    Outer else
statement outside if block
```

Example: Find a greater number between two numbers

```
num1 = int(input('Enter first number '))
num2 = int(input('Enter second number '))


if num1 >= num2:
    if num1 == num2:
        print(num1, 'and', num2, 'are equal')
    else:
        print(num1, 'is greater than', num2)
else:
    print(num1, 'is smaller than', num2)
```

If there is single statement how to write it?

Single statement suites

Whenever we write a block of code with multiple if statements, indentation plays an important role. But sometimes, there is a situation where the block contains only a single line statement.

Instead of writing a block after the colon, we can write a statement immediately after the colon.

Example

```
number = 20

if number > 0: print("positive")

else: print("negative")
```

what are not allowed in python conditional statement

1: No Parentheses Around Conditions

In Python, unlike some other programming languages, parentheses around conditions are not required in conditional statements. While using them doesn't cause syntax errors, it's considered unnecessary

```
if (x > 0):(not recommended)
    # Do something
if x > 0:(recommended)
    # Do something
```

2: No Assignment Inside Conditions:

Assignment statements inside conditions are not allowed in Python. This is because assignment (=) is a statement, not an expression, and therefore cannot be used as part of a condition.

```
if (x = 10):  # Assignment inside condition

    # Do something
```

3: No Expression Evaluation Inside Conditions:

Python's if statement expects a boolean expression to determine the flow of execution. Passing non-boolean types like strings or numbers directly as conditions without comparison or boolean operators is not allowed.

```
if "hello":  # Non-boolean expression
    # Do something
```

4: No Use of Assignment Operator (=) for Comparison:

The single equals sign (=) is used for assignment in Python. When comparing values for equality in a conditional statement, use the double equals sign (==). Using the assignment operator instead of the equality operator will not produce the intended behavior and might lead to logical errors.

```
if x = 10:  # Incorrect usage of assignment operator for comparison
    # Do something


if x == 10:  # Correct usage of equality operator for comparison
    # Do something
```

5: indentation is important

Indentation plays a crucial role in Python because it is used to denote blocks of code. So, statement written after if block should be maintained properly.

If a<2:

print(“Positive”); (error)

match-case statement

Python is under constant development and it didn’t have a match statement till python < 3.10. Python match statements were introduced in python 3.10 and it is providing a great user experience, good readability, and cleanliness in the code which was not the case with clumsy Python if elif else ladder statements.

What is a ‘match-case’ statement?

A match case statement is a conditional statement that matches the value of an expression against a set of patterns and executes the code block associated with the first matching pattern.

It is similar to the switch-case statement in other programming languages.

The match case statement was introduced in Python 3.10 as a new feature to simplify conditional branching and improve code readability.

Syntax and Usage

The syntax of the match case statement in Python is as follows:

```
match expression:
    case pattern1:
        # code block for pattern1
    case pattern2:
        # code block for pattern2
    ...
    case patternN:
        # code block for patternN
    case _:
        # default code block
```

The match keyword is followed by the expression that we want to match.

A pattern and a colon follow each case keyword.

The code block associated with each pattern is indented below the case statement.

We can have multiple case statements and a default case denoted by an underscore (_).

```
user = "Teksan"
match user:
        case "Om":
            print("Om do not have access  to the database only for the api code.")
        case "Vishal":
            print(
                "Vishal do not have access to the database , only for the frontend code.")
        case "Rishabh":
            print("Rishabh have the access to the database")
        case _:
            print("You do not have any access to the code")
```

what are not allowed in python match-case statement

Non-Constant Patterns

Patterns used in a match statement must be constants or literals. They cannot be variables, expressions, or function calls.

The patterns must be known at compile-time, allowing for efficient compilation and optimization.

```
value = 5


match value:
    case 5:  # Allowed, 5 is a constant pattern
        # Case logic
    case x:  # Not allowed, x is a variable
        # Case logic
```

Limited Types for Patterns

Not all types can be used as patterns in a match statement. Only types that support structural pattern matching can be used.

Commonly supported types include literals, constants, data classes, tuples, lists, dicts, enums, and instances of classes with __match__ methods.

Complex Patterns

Complex patterns involving nested structures, such as nested lists or nested tuples, may not be directly supported or may require additional handling.

Pattern matching in Python is primarily designed for simple and structured patterns.

```
nested_list = [[1, 2], [3, 4]]

match nested_list:
    case [[1, 2], [3, 4]]:  # Allowed, matching against a nested list
        # Case logic
```

Side Effects in Patterns

Patterns should not have side effects such as modifying variables or performing I/O operations.

Patterns are used for matching values, and side effects can lead to unexpected behavior.

value = 5

match value:

    case print("Hello"): # Not allowed, print() has a side effect

        # Case logic

Using if- else as Ternary Operator

Ternary operators are more commonly known as conditional expressions in Python. These operators evaluate something based on a condition being true or not.

The Python ternary operator determines if a condition is true or false and then returns the appropriate value in accordance with the result.

The ternary operator in Python is simply a shorter way of writing an if and if...else statements.

Here's what an if...else statement looks like in Python

```
user_score = 90


if user_score > 50:
    print("Next level")
else:
    print("Repeat level")
```

Python Ternary Operator Example

You can shorten it using the ternary operator. Here's what the syntax looks like:

Syntax:

[option1] if [condition] else [option2]

In the syntax above, option1 will be executed if the condition is true. If the condition is false then option2 will be executed.

Here's a more practical example:

```
user_score = 90


print("Next level") if user_score > 50 else print("Repeat level")
```

```
x = 10
y = 5

z = x if x > y else y

print("z = " + str(z))
```

Looping Statements

A loop in programming is a control structure that repeatedly executes a block of code as long as a specified condition remains true.

Python programming language provides two types of Python loop checking time.

– For loop

– While loop to handle looping requirements.

Loops in Python provides three ways for executing the loops.

- for loop
- while loop
- nested loop

For Loop

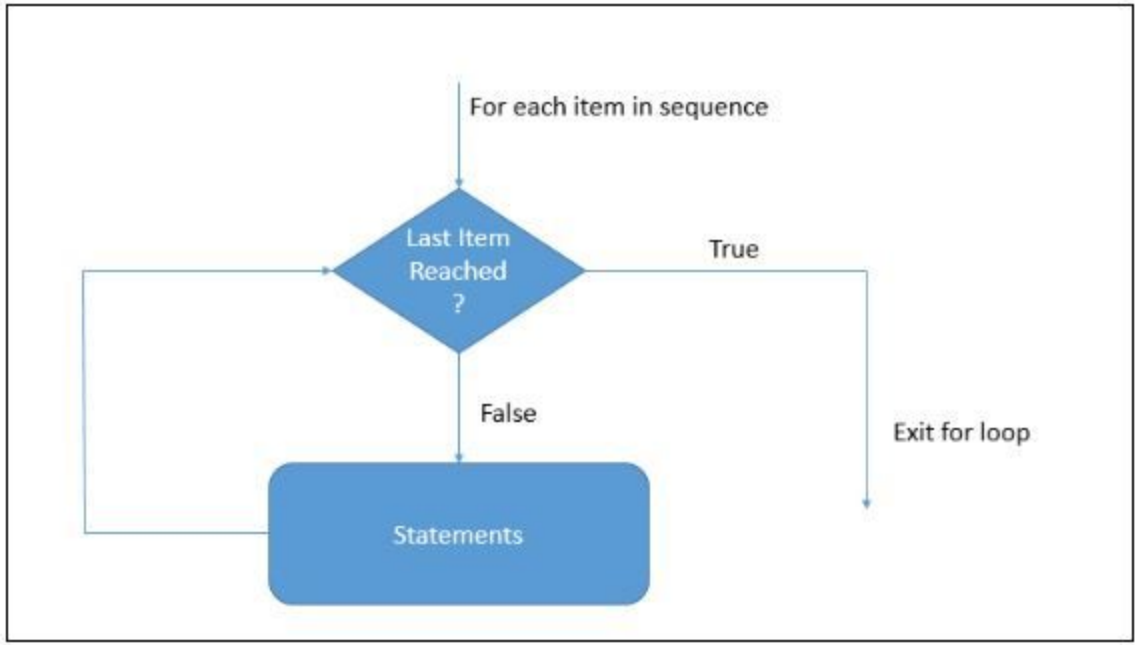
A for loop in Python is used to iterate over a sequence (list, tuple, set, dictionary, and string).

The for loop is used in the case we needs to execute a part of the code until the given condition is satisfied.

The for loop is also called a pre-tested loop.

It is best to use for loop if the number of iterations is known in advance.

Flowchart of Python for Loop



for loop Syntax

```
for val in sequence:

    # statement(s)
```

Here, val accesses each item of the sequence on each iteration. The loop continues until we reach the last item in the sequence.

example

```
languages = ['Swift', 'Python', 'Go']


# access elements of the list one by one

for i in languages:
    print(i)
```

Python For Loop with String

```
language = 'Python'


# iterate over each character in language
for x in language:
    print(x)
```

Python for loop with Range

In Python, the range() function returns a sequence of numbers. For example,

values = range(4)

Here, range(4) returns a sequence of 0, 1, 2 ,and 3.

Since the range() function returns a sequence of numbers, we can iterate over it using a for loop.

```
# iterate from i = 0 to i = 3
for i in range(4):
    print(i)
```

Nested For Loops in Python

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

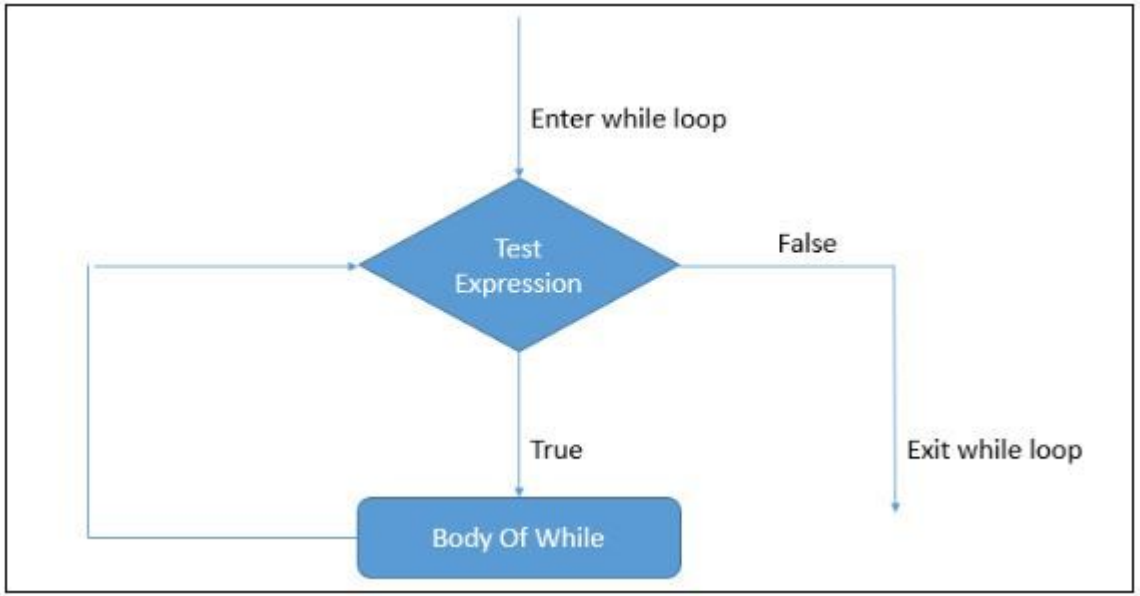
while Loops

the while loop is used to repeatedly execute a block of code as long as a condition is true. The loop continues iterating as long as the condition evaluates to True. Once the condition becomes False, the loop terminates, and the program continues with the next statement after the loop.

Here's the basic syntax of a while loop in Python:

```
while condition:
    # Code block to execute while condition is True
    # This block can contain one or more statements
```

Flowchart


Enter while loop
Test Expression
False
True
Exit while loop
Body Of While

```
number = 1


while number <= 3:
    print(number)
    number = number + 1
```

Infinite while Loop

```
age = 32


# the test condition is always True
while age > 18:
    print('You can vote')
```

Python while Loop example

```
# Calculate the sum of numbers until user enters 0
number = int(input('Enter a number: '))
total = 0
# iterate until the user enters 0
while number != 0:
    total += number
    number = int(input('Enter a number: '))

print('The sum is', total)
```

The else Clause after for or while Loops

Else in For Loop

The 'else' block in a 'for loop' in Python, often referred to as 'for else python', is executed after the for loop completes its iteration over the entire sequence, but only if a break statement didn't terminate the loop.

Syntax

```
for i in range(n) :
    #code
else:
    #code
```

Else block executed

```
numbers = [1, 2, 3, 4, 5]
target = 10


for num in numbers:
    if num == target:
        print("Target number found.")
        break
else:
    print("Target number not found.")
```

Else block NOT executed

```
numbers = [1, 2, 3, 4, 5]
target = 3


for num in numbers:
    if num == target:
        print("Target number found.")
        break
else:
    print("Target number not found.")
```

Else in While Loop

In Python, the while else statement is a combination of a while loop with an else block. The while loop executes a block of code as long as its given condition remains true. Once the condition becomes false, the loop stops, and the else block is executed. This structure provides a robust way to control the flow and termination of a loop, allowing for better readability and organization in your code.

The Python while else loop is an extension of the while loop with an else block that executes when the while's condition is no longer true.

Syntax of the Python while else loop

```
while condition:
    # Code to be executed when the condition is True
else:
    # Code to be executed when the condition becomes False
```

Python while True Else: An Explanation

Let us consider a simple example demonstrating the Python while else loop in action:

```
count = 1
while count <= 5:
    print("Count: ", count)
    count += 1
else:
    print("The loop has ended.")
```

Python While Else with a Break Statement

```
age = 30


# the test condition is always True
while age > 19:
        print('Infinite Loop')
age-=1
else:
        print("Not a infinite loop")
```

Note:

The break statement is used to terminate the loop prematurely and exit the loop without executing the else block.

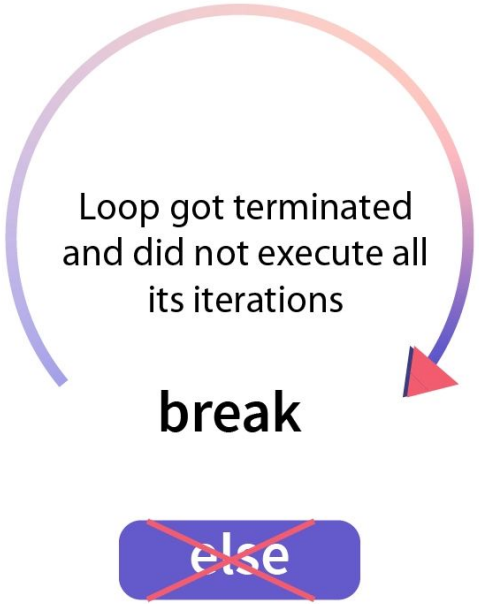
On the other hand, the continue statement skips the remaining code within the loop block for the current iteration and jumps back to the start of the loop to check the condition again.

You can use these statements within a while else loop or for else loop to control the flow more efficiently.

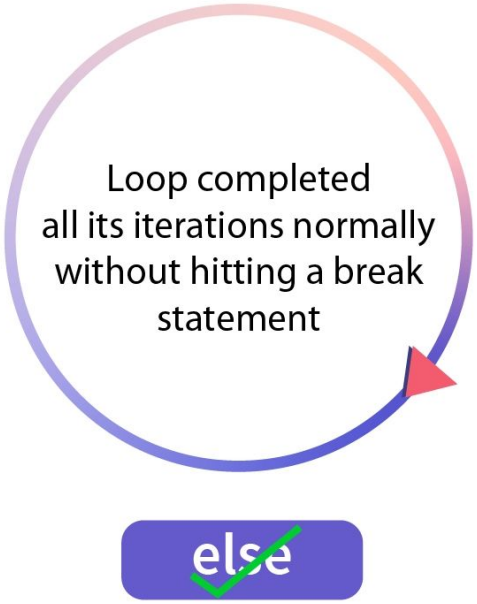
What is For Else and While Else in Python

For-else and while-else are helpful features provided by Python. You can use the else block just after the for and while loop. Otherwise, the block will be executed only if a break statement doesn't terminate the loop.

Simply put, if a loop is executed successfully without termination, then the else block will be executed.



The else clause is not executed



The else clause is executed Once!

Remember:

Else block in for-else or while-else will only execute once when the loop completes all its iterations without termination by a break statement

Transfer Statement

Break Statement

The break statement in Python is used to terminate the loop or statement in which it is present.

After that, the control will pass to the statements that are present after the break statement, if available.

If the break statement is present in the nested loop, then it terminates only those loops which contain the break statement.

Example of Break Statement

```
number = 0


for number in range(10):
    if number == 5:
        break    # break here
    print('Number is ' + str(number))
print('Out of loop')
```

Output

Number is 0

Number is 1

Number is 2

Number is 3

Number is 4

Out of loop

What are the place where we can use break statements?

- Inside Loops only

Continue Statement

The continue statement allows you to skip over the part of a loop where an external condition is triggered, but to go on to complete the rest of the loop.

The current iteration of the loop will be disrupted, but the program will return to the top of the loop.

The continue statement will be within the code block under the loop statement, usually after a conditional if statement.

Using the same for loop program as in the Break Statement section above, we'll use a continue statement rather than a break statement:

Example

```
number = 0


for number in range(10):
    if number == 5:
        continue    # continue here
    print('Number is ' + str(number))
print('Out of loop')
```

Output

Number is 0

Number is 1

Number is 2

Number is 3

Number is 4

Number is 6

Number is 7

Number is 8

Number is 9

Out of loop

What are the place where we can use continue statements?

- Inside Loops only

The pass Statement

Python pass is a null statement provided by the Python programming language in order to ignore a set of code that we do not want to execute.

If a function, class, or a loop needs to be coded and executed in the future, then the python pass statement is used to indicate the Python Interpreter to ignore that particular function, class, or loop while the program is being executed.

What does pass do?

The pass statement does nothing in Python, which is helpful for using as a placeholder in if statement branches, functions, and classes. In layman's terms, pass tells Python to skip this line and do nothing.

Example of how pass works

```
for number in range(1, 71):
    if number % 7 != 0:
        # the number is not a multiple of 7
        pass
    else:
        print({number} is a multiple of 7')
```

Why use pass?

As mentioned previously, pass is usually used as a placeholder for branches, functions, classes. Whenever Python arrives at a pass statement, it passes straight over it (hence the name).

This functionality may seem pointless, but let's try and run our example from the introduction again, without the pass statement:

Example of how pass works

```
for number in range(1, 71):
    if number % 7 != 0:
        # the number is not a multiple of 7
    else:
        print({number} is a multiple of 7')
```

Output

```
File "<ipython-input-2-793ee23f6307>", line 5
    else:
       ^
IndentationError: expected an indented block
```

This code gives us IndentationError: expected an indented block which we're getting because we haven't added anything to our if statement.

A codeless statement like this is known as an empty suite. We can avoid this error by using pass as a placeholder for our if statement as seen in the introduction.

The addition of pass means that Python does nothing in situations where the number variable isn't a multiple of 27. Despite this seeming redundant, it prevents Python from throwing an error.

Another example

```
def func1():
    # implement function later


def func2():
    # implement function later


def func3(a):
    print(a)


func3('Python')
```

Output

```
File "", line 3
    def func2():
    ^
IndentationError: expected an indented block
```

Fix this

```
def func1():
    pass


def func2():
    pass


def func3(a):
    print(a)


func3('Python')
```