UNIT 2

Python Program Flow Control Conditional blocks

UNIT-2: Python Program Flow Control Conditional blocks: if, else and else if, Simple for loops in python, for loop using ranges, string, list and dictionaries. Use of while loops in python, Loop manipulation using pass, continue, break and else. Programming using Python conditional and loop blocks

2.1 INTRODUCTION TO PYTHON PROGRAM FLOW CONTROL & LOOPS:
Python program flow control helps us **decide which code should run** and **repeat code when necessary.**

The Main Components are:

- **Flow Control** → Decides the execution order of code.
- **Conditional Blocks (if, elif, else)** → Used for decision-making**.**
- **Loops (for, while)** → Used for repetition.
- **Loop Manipulation (pass, continue, break)** → Used to control loop behavior.

2.1.1 What is Flow Control in Python?

- Flow Control means deciding which part of the code should run based on certain conditions.
- In Python, flow control is achieved using:
  - ***if***
  - ***elif (else if)***
  - ***else***

These statements control the direction of program execution.

2.1.2 What are Conditional Blocks in Python?

- Conditional blocks like if, elif, and else help the program make decisions.
- They check whether something is **True or False** (Boolean expression).
- Depending on the result:
  - ✓ The program can run different parts of code.
  - ✓ It helps the program decide what to do next.

2.1.3 What are Loops in Python?

- Loops help run the same code multiple times.
- Python has two main loops:
  - ✓ **For loop** → used to iterate through items in a list, string, or range.
  - ✓ **While loop** → runs as long as a condition is True.

- Loops save time and make programs shorter and easier to read.
- Loops in Python can also have an optional else block, executed only if the loop completes normally (without break).

2.1.4 What are Loop Manipulation Statements in Python?

The special statements used inside loops to control their execution are:

- **pass** → Does nothing, used as a placeholder.
- **continue** → Skips the current iteration and moves to the next.
- **break** → Exits the loop immediately.

These statements provide more control over how loops run.

2.2 Conditional Blocks

In Python, conditional blocks are used to control the flow of a program based on logical decisions. They execute certain sections of code only when specific conditions are satisfied.

2.2.1 Conditional Blocks: if

The if statement is a fundamental building block for decision-making in Python.

- It executes a block of code **only if** a specific condition is **True**.
- If the condition is **False**, the code block is skipped.

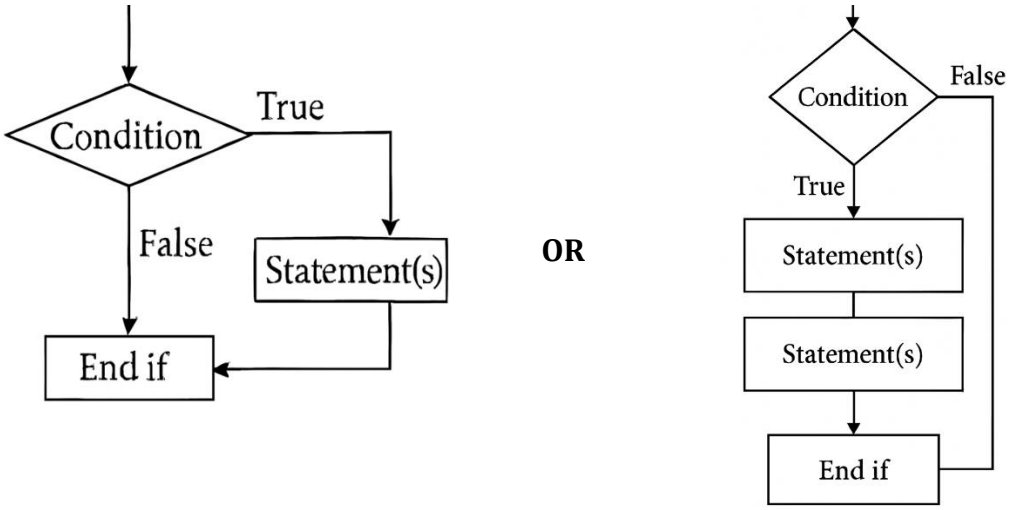

Figure 1: Conditional block if

Syntax:

```
if condition:
    # block of code executed when condition is True
```

Explanation:

- Condition must return **True or False.**
- Indentation is required for the code inside the block.

Example Program: Basic if Statement

```
a = 6
if a > 5:
    print("a is bigger than 5")
```

Output:

```
a is bigger than 5
```

Explanation:

- Since a > 5 is True, the print statement runs.

Example Program: Using if with Input	
number = int(input("Enter a number: ")) *if number % 2 == 0:* *print("Even number")*	*number = int(input("Enter a number: "))* *if number % 2 == 0:* *print("Even number")* *if number % 2 ! = 0:* *print("Odd number")*
Output: *Enter a number: 6* *Even number*	**Output:** *Enter a number: 5* *Odd number*

Explanation:

- If remainder is zero, the number is even.

Example Program: if with Boolean Expressions

is_sunny = True *if is_sunny:* *print("Go for a walk!")*
Output: *Go for a walk!*

Explanation: Since is_sunny is True, the message is printed.

Example Program: if - Code skipped when Condition is False

profit = 50 *if profit > 100:* *print("You can withdraw money.")*
Output: *(no output)*

Explanation:

- Since the condition is False, nothing is printed.

Example Program: Check if a character is a vowel

char = "e" *if char in "aeiou":* *print(f"{char} is a vowel")*
Output: *e is a vowel*

Explanation:

- The condition char in "aeiou" checks if char is one of the vowel letters.
- Since char is "e", it is True, and the message is displayed.
- The part inside {} → {char} → will be replaced with the **value of the variable char**.
- Since char = "e", the expression becomes: print("**e** is a vowel")

What is f in print(f"...")?

- The **f** before the string starts (called an **f-string**) means **formatted string literal**.
- It was introduced in **Python 3.6**.

- It allows you to **directly embed variables or expressions** inside a string using {} braces.

Example Program: Check if a character is a vowel

```
name = "Arun"
age = 25
print(f"My name is {name}, and I will be {age + 1} next year.")
```

Output:

*My name is **Arun**, and I will be 26 next year.*

Explanation:

- **f"..."** → This tells Python it's a formatted string literal.
- Inside the string:
  - ✓ {name} → replaced by the value of the variable name, which is "**Arun**".
  - ✓ {age + 1} → Python evaluates the expression 25 + 1 = 26.

Example Program: Temperature Alert

```
temperature = 40
if temperature > 35:
    print("It's too hot, stay hydrated!")
```

Output:

It's too hot, stay hydrated!

Explanation:

- The condition temperature > 35 is checked.
- Since 40 > 35 is **True**, the print statement runs.
- If the temperature had been 30, nothing would be printed because the condition would be **False**.

2.2.2 Conditional Blocks: else

- The **else statement** runs only if all the earlier conditions (if and elif) are **False**.
- If nothing else matches, the else block executes.
- In simple words: it tells Python what to do **"otherwise."**

Syntax:

```
if condition:
    # code when condition is True
else:
    # code when condition is False
```

Explanation:

- The if statement checks a condition (True/False).
- If the condition is **True**, the block under if runs.
- If the condition is **False**, the block under else runs.

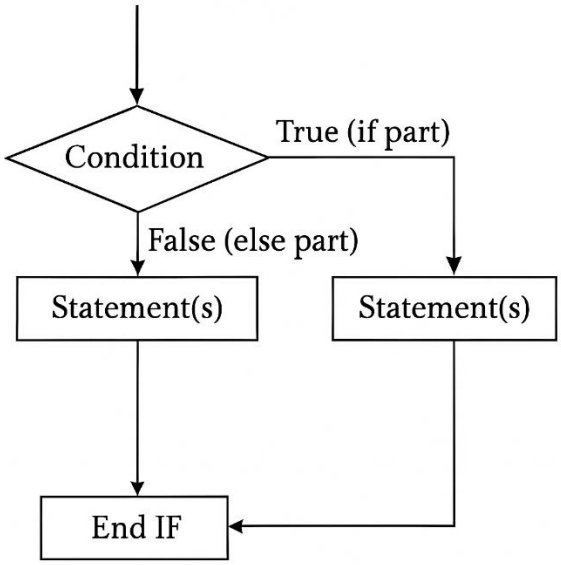

Figure 2: Conditional block if-else

Extended Syntax with elif

```
if condition1:
   # code when condition1 is True
elif condition2:
   # code when condition2 is True
else:
   # code when none of the above conditions are True
```

Explanation:

- **if** → checks the first condition.
- **elif** ("else if") → checks the next condition, but only if the previous one was False.
- **else** → runs when none of the above conditions are True.

Example Program: Grading System

```
marks = 55

if marks >= 90:
   print("Grade: A")
elif marks >= 60:
   print("Grade: B")
else:
   print("Grade: C")
```

Output:

```
Grade: C
```

Explanation:

- marks = 55
- Checks in order:
  - ✓ marks >= 90 → False
  - ✓ marks >= 60 → False
- Since none are True, the else block runs → **Grade: C**

Example Program: ATM Withdrawal (if-else)

```
balance = 200
withdraw = 250
if withdraw <= balance:
   print("Transaction Successful")
else:
   print("Insufficient Funds")
```

Output:

Insufficient Funds

Explanation:

- Condition checked → withdraw <= balance → 250 <= 200 → False.
- Hence, the else block runs → **"Insufficient Funds"**.
- If withdraw = 150, output would be: **"Transaction Successful"**.

2.2.3 Conditional Blocks: elif

- The **elif statement** stands for **"else if"**.
- It allows checking **multiple conditions** one after another.
- **Only the first condition that is True will run**, and the rest are skipped.

Syntax:

```
if condition1:
   # code if condition1 is True
elif condition2:
   # code if condition1 is False and condition2 is True
elif condition3:
   # code if previous conditions are False and condition3 is True
else:
   # code if none of the above conditions are True
```

Explanation:

- The program checks conditions in order, top to bottom.
  - As soon as one condition is **True**, that block executes and others are ignored.
- If none are True, the else block (if present) will run.

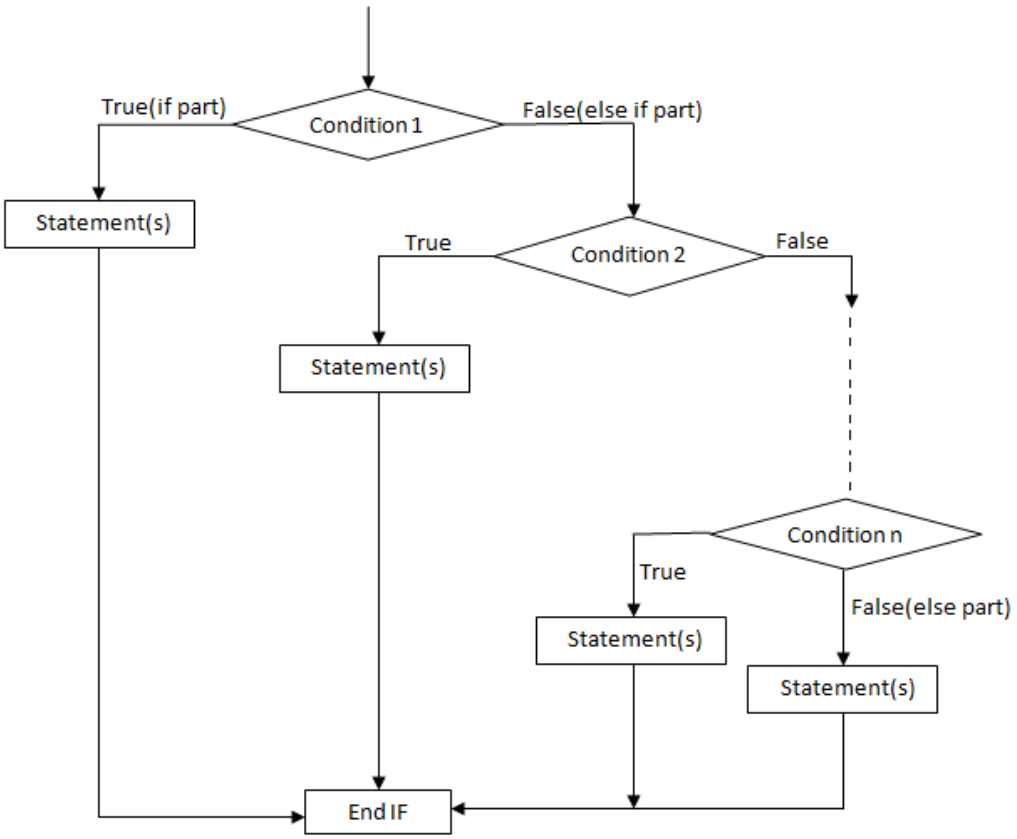

Figure 3: Conditional block elif

Example Program: Grading with Multiple Levels

```
marks = 88

if marks >= 95:
    print("Grade: A")
elif marks >= 80:
    print("Grade: B")
elif marks >= 70:
    print("Grade: C")
else:
    print("Grade: D")
```

Output:

```
Grade: B
```

Explanation:

- marks >= 95 → False
- marks >= 80 → True → prints **Grade: B**
- Remaining checks are skipped.

Example Program: Weather Checker

```
temperature = 25

if temperature > 35:
    print("It's very hot.")
elif temperature > 30:
    print("It's warm.")
elif temperature > 10:
```

```
    print("It's cool.")
else:
    print("It's cold.")
```

Output:

It's cool.

Explanation:

- 25 is not > 35 → False
- 25 is not > 30 → False
- 25 is > 10 → True.

Example Program: Number Range (Multiple Conditions)

```
a = 18

if a < 10:
    print("a is less than 10")
elif a < 20:
    print("a is between 10 and 19")
else:
    print("a is 20 or greater")
```

Output:

a is between 10 and 19

Explanation:

- First condition (a < 10) is False.
- Second condition (a < 20) is True (because 18 < 20), so its block is executed.
- The rest (including else) is skipped.

Example Program: Age Classifier (Multiple Conditions)

```
age = 25

if age < 13:
    print("Child")
elif age < 20:
    print("Teenager")
elif age < 60:
    print("Adult")
else:
    print("Senior")
```

Output:

Adult

Explanation:

- Checks if age < 13 → False (25 is not less than 13)
- Then age < 20 → False (25 not less than 20)
- Then age < 60 → True (25 is less than 60)
- "Adult" is printed, and the rest skipped.

Example Program: Checking Grades

```
score = 75

if score >= 90:
   print("Grade: A")
elif score >= 80:
   print("Grade: B")
elif score >= 70:
   print("Grade: C")
else:
   print("Grade: D or below")
```

Output:

```
Grade: C
```

Explanation:

- The first (score >= 90) and second (score >= 80) are False.
- The third (score >= 70) is True (75 ≥ 70), so "Grade: C" prints.

Example Program: Days of the Week

```
day = "Wednesday"

if day == "Monday":
   print("Start of the work week")
elif day == "Friday":
   print("Almost the weekend!")
elif day == "Sunday":
   print("Weekend!")
else:
   print("Midweek day")
```

Output:

```
Midweek day
```

Explanation:

- None of the specific day checks match, so the else block is executed.

Example Program: Check day of the week

```
day = "Sunday"

if day == "Saturday" or day == "Sunday":
   print("Weekend")
elif day == "Friday":
   print("Almost weekend")
else:
   print("Weekday")
```

Output:

```
Weekend
```

Explanation:

- The condition checks if day is Saturday or Sunday; day is Sunday so True.
- Prints "Weekend".
- Skips remaining checks.

Example Program: User Input with elif

```
num = int(input("Enter a number: "))
if num > 0:
   print("Positive number")
elif num == 0:
   print("Zero")
else:
   print("Negative number")
```

Output:

```
Zero
```

Explanation:

- The program checks if num is greater than 0 (False for 0).
- Then checks if num equals 0 (True).
- Since the elif condition is true, "Zero" is printed.
- The else block is skipped.

Example Program: Simple if-elif-else to check a number sign

```
num = 0

if num > 0:
   print("Positive number")
elif num == 0:
   print("Zero")
else:
   print("Negative number")
```

Output:

```
Enter a number: -3
Negative number
```

Explanation:

- For input -3, neither the first nor second conditions are met, so the else runs.

Example Program: if-elif-else (Multiple Choices) → Traffic Light System

```
signal = "Yellow"
if signal == "Green":
   print("Go")
elif signal == "Yellow":
   print("Slow Down")
else:
   print("Stop")
```

Output:

```
Slow Down
```

Explanation:

- First, the program checks if signal == "Green" → False.
- Next, it checks signal == "Yellow" → True.
- Since this is True, it executes print("Slow Down").
- The else block is skipped because one condition has already matched.

2.2.4 COMPARISON TABLE: if, elif & else:

Feature	if	elif	else
Purpose	Starts the conditional check. Executes if its condition is **True**.	Checks another condition if previous if/elif are **False**.	Executes only if **all previous conditions are False**.
Condition	Must have a condition (boolean expression).	Must have a condition (boolean expression).	No condition (default block).
Requirement	**Mandatory** to start a conditional chain.	**Optional**, can appear multiple times.	**Optional**, can appear only once, at the end.
Execution Flow	If **True**, executes its block and skips others.	Checked only if previous conditions are **False**.	Runs if none of the above conditions are satisfied.
Indentation	Required.	Required.	Required.
Typical Use Case	To test the first/main condition.	To test multiple alternate conditions.	To handle all remaining/unmatched cases.
Syntax	***if condition:*** *# block of code*	***if condition1:*** *# block if condition1 is true* ***elif condition2:*** *# block if condition2 is true*	***if condition1:*** *# block if condition1 is true* ***elif condition2:*** *# block if condition2 is true* ***else:*** *# block if none of the above are True*
Example Program	***Using if, elif, else together:*** *x = 10* *if x > 10:* *print("x is greater than 10")* *elif x == 10:* *print("x is equal to 10")* *else:* *print("x is less than 10")* ***Explanation:*** 1. *if x > 10: → False (10 is not greater than 10).* 2. *elif x == 10: → True, so "x is equal to 10" is printed.* 3. *else is skipped because one condition was satisfied.*		

2.3 SIMPLE for LOOPS

- A for loop in Python is used to iterate (repeat) over a sequence (like a list, string, tuple, or range of numbers) and execute a block of code for each item in the sequence.
- In each iteration, the loop variable automatically takes the next value from the sequence.

Syntax:

```
for variable in sequence:
    # code to execute for each item
```

Explanation:

- **variable**: A temporary name that holds each item from the sequence.
- **sequence**: A list, string, tuple, range, or any other iterable object.
- **Indentation** is mandatory for the "body" (the block of code inside the loop).

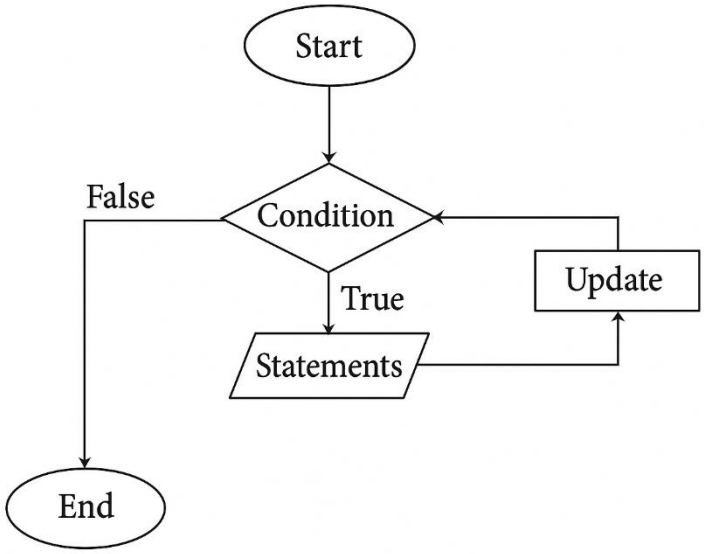

Figure 4: for loop

Example Program: Basic loop using range()

```
for i in range(4):
    print("Number:", i)
```

Output:

```
Number: 0
Number: 1
Number: 2
Number: 3
```

Explanation:

- range(4) creates a sequence of numbers: 0, 1, 2, 3
- The loop runs 4 times, and each time i takes the next number.
- The print( ) function shows the current value of i.

Example Program: Loop through a list

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

Output:

```
I like apple
I like banana
I like cherry
```

Explanation:

1. The list fruits contain 3 elements → "apple", "banana", "cherry".
2. The for loop iterates over the list one element at a time.
   - ✓ ***First iteration: fruit = "apple" → prints "I like apple".***
   - ✓ ***Second iteration: fruit = "banana" → prints "I like banana".***
   - ✓ ***Third iteration: fruit = "cherry" → prints "I like cherry".***
3. The loop ends after all items in the list are processed.

Example Program: Loop through a list

fruits = ["apple", "banana", "cherry"] *for i in range(3):*	*fruits = ["apple", "banana", "cherry"]* *for i in range(3):*

print(fruits[0]) # prints "apple" 3 times *print(fruits[1]) # prints "banana" 3 times* *print(fruits[2]) # prints "cherry" 3 times*	*print(fruits[0]) # prints "apple" 3 times* *print(fruits[1]) # prints "banana" once* *print(fruits[2]) # prints "cherry" once*
Output: *apple* *banana* *cherry* *apple* *banana* *cherry* *apple* *banana* *cherry*	**Output:** *apple* *apple* *apple* *banana* *cherry*
Explanation: • The loop runs 3 times (range(3)). • Inside each iteration, it prints: ✓ ***fruits[0] → "apple"*** ✓ ***fruits[1] → "banana"*** ✓ ***fruits[2] → "cherry"*** • So each fruit is printed 3 times, once per iteration.	**Explanation:** • *fruits[0] refers to "apple" and is printed inside the loop, so it prints 3 times.* • *fruits[1] and fruits[2] refer to "banana" and "cherry" respectively, and are printed once each, after the loop ends.*

Explanation:

- The loop takes each element from the list fruits in turn ("apple", then "banana", then "cherry"), assigning it to fruit and runs the print statement for each.

Example Program: Iterate Over a String

```
text = "hi"
for letter in text:
    print(letter)
```

Output:

h
i

Explanation:

- The loop prints each character of the string separately.

Example Program: Using range() for Repeating Tasks

```
for x in range(1, 5):
    print(x)
```

Output:

1
2
3
4

Explanation:

- range(1, 5) generates numbers from 1 up to (but not including) 5.
- In each iteration, i is assigned the next number in the range.

Example Program: Iterating Over a Tuple

```
colors = ("red", "green", "blue")
for color in colors:
    print(color)
```

Output:

red
green
blue

Explanation:

- Just like with lists, you can iterate through tuples.

KEY POINTS:

- The loop variable automatically takes each value from the sequence, no index is needed.
- For loops stop after the last item in the sequence.
- You can loop through any **iterable**: lists, tuples, strings, dictionaries, ranges, etc.

Write a python program to count the vowels present in given input string. Explain the output of program through example.

Solution:

Method-1

```
string = input("Enter a string: ")
count = 0

for ch in string:                    # loop through each character
    if ch in "aeiouAEIOU":           # check if it's a vowel
        count += 1                   # increase count

print("Number of vowels:", count)
```

Output:

Enter a string: Hello JSS Academy of Technical Education, Noida
Number of vowels: 15

Explanation:

1. string = input("Enter a string: ") → Takes input from the user and stores it in the variable string.
2. count = 0 → Creates a counter variable count and sets it to 0.
3. for ch in string: → Loops through each character (ch) in the input string.
4. if ch in "aeiouAEIOU": → Checks if the character is a vowel (either small or capital).
5. count += 1 → If it is a vowel, increase the counter by 1.
6. print("Number of vowels:", count) → Prints the total number of vowels found in the input string.

Counting vowels

- *Hello → e, o → 2*
- *JSS → none → 0*
- *Academy → A, a, e, a → 4*
- *of → o → 1*
- *Technical → e, i, a → 3*
- *Education → E, u, a, i, o → 5*
- *Noida → o, i, a → 3*

Total vowels = 2 + 0 + 4 + 1 + 3 + 5 + 3 = 18

Method-2

```
string = input("Enter a string: ")
vowels = "aeiouAEIOU"
count = 0

for ch in string:
    if ch in vowels:
        count += 1

print("Number of vowels:", count)
```

Output:

```
Enter a string: Hello JSS Academy of Technical Education, Noida
Number of vowels: 15
```

2.3.1 SIMPLE for LOOP USING range()

What is range( )?

- range( ) is a built-in function in Python that returns a sequence of numbers.
- It's commonly used in for loops to repeat actions a specific number of times.

Syntax of range( )

```
range(stop)
range(start, stop)
range(start, stop, step)
```

Explanation:

- **start** – starting value (default is 0).
- **stop** – ending value (exclusive, not included).
- **step** – how much to increment by (default is 1).

Syntax of for loop using range( ):

```
for variable in range(start, stop, step):
    # code to execute in each iteration
```

Explanation:

- **start** (optional): The starting number (inclusive). Default is 0.
- **stop**: The ending number (exclusive).
- **step** (optional): The increment between numbers. Default is 1.
- **variable**: The loop variable that takes the current number from the sequence in each iteration.

Example Program: Basic for loop using range(stop)

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

Output:

```
0
1
2
3
4
```

Explanation:

- range(5) generates numbers from 0 to 4 (not including 5).

- i takes each value one by one.
- print() displays the value in each loop iteration.

Example Program: for loop using range(start, stop)

for i in range(1, 6): *print(i)*	*for num in range(2, 6):* *print("Number:", num)*
Output: *1* *2* *3* *4* *5*	**Output:** *Number: 2* *Number: 3* *Number: 4* *Number: 5*
Explanation: • The sequence generated is from 1 (inclusive) to 6 (exclusive). • So the numbers 1 through 5 are printed.	**Explanation:** • range(2, 6) generates: 2, 3, 4, 5 (6 is not included). • The loop prints numbers starting from 2 to 5.

Example Program: range(start, stop, step)

for n in range(1, 10, 2): *print(n)*	*for i in range(0, 10, 2):* *print(i)*
Output: *1* *3* *5* *7* *9*	**Output:** *0* *2* *4* *6* *8*
Explanation: • range(1, 10, 2) starts at 1, goes up to 9, incrementing by 2. • Skips every second number.	**Explanation:** • Starts at 0, stops before 10, stepping by 2 each time. • Prints even numbers from 0 to 8.

Example Program: Using negative step to count downwards

```
for i in range(5, 0, -1):
    print(i)
```

Output:

5
4
3
2
1

Explanation:

- Starts at 5, stops before 0, stepping backwards by 1.
- Prints numbers counting down from 5 to 1.

Example Program: Multiplication Table with for Loop

```
number = int(input('Enter a number: '))

for count in range(1, 11):
   product = number * count
```

```
    print(number, '*', count, '=', product)
```

Output:

```
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
```

Explanation

1. User enters a number (e.g., 7).
2. range(1,11) generates values from 1 to 10.
3. Each iteration multiplies the number with count.
4. Prints the multiplication table.

Example Program: Nested for Loop

```
# outer for loop
for number in range(1, 3):
    # inner for loop
    for value in range(1, 6):
        product = number * value
        print(number, '*', value, '=', product)
```

Output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
```

Explanation

1. Outer loop → runs for numbers 1 and 2.
2. Inner loop → runs 5 times for each outer loop iteration.
3. Together, 2 × 5 = 10 total iterations.

Write a Python program to construct the following pattern, using a nested for loop.

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

Solution:

Using a nested for loop: **Method-1**

```
# Define number of stars in each row
rows = [1, 2, 3, 4, 5, 4, 3, 2, 1]

for count in rows:
    for i in range(count):
        print("*", end=" ")
    print()
```

Output:

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

Explanation

1. We created a list rows = [1, 2, 3, 4, 5, 4, 3, 2, 1]
   - ✓ Each number represents how many stars should be printed in that row.
   - ✓ It increases from 1 → 5 and then decreases back to 1 (diamond-like pattern without spaces).
2. Outer loop → for count in rows:
   - ✓ Iterates over each number in the list (number of stars in that row).
3. Inner loop → for i in range(count):
   - ✓ Prints the stars in a row, based on the current number count.
4. print("*", end=" ")
   - ✓ Prints stars side by side in the same line.
5. print() (blank print)
   - ✓ Moves to the next line after finishing each row.

Using a nested for loop: **Method-2**

```
# First half of the pattern (increasing stars)

for i in range(1, 6):              # i goes from 1 to 5
    for j in range(i):             # j goes from 0 to i-1
        print("*", end=" ")
    print()                        # Move to the next line

# Second half of the pattern (decreasing stars)
for i in range(4, 0, -1):          # i goes from 4 down to 1
    for j in range(i):             # j goes from 0 to i-1
        print("*", end=" ")
    print()                        # Move to the next line
```

Explanation:

First Half (Increasing Pattern)

- *for i in range(1, 6) → This loop runs 5 times (i = 1 to 5).*

- *For each i, the inner loop for j in range(i) prints i stars.*
- *end=" " keeps the stars on the same line with spaces.*
- *print() moves to the next line after each row.*

Second Half (Decreasing Pattern)

- *for i in range(4, 0, -1) → This loop runs in reverse (i = 4 to 1).*
- *For each i, the inner loop prints i stars.*
- *This creates the descending part of the triangle.*

Note:

Method-1: Without Nested for loop	***Method-2: Without Nested for loop***
# Simple star pattern *for i in range(1, 6):* *print("* " * i)* *for i in range(4, 0, -1):* *print("* " * i)*	***# Simple star pattern*** *n = 5* *for i in range(1, n + 1):* *print("* " * i)* *for i in range(n - 1, 0, -1):* *print("* " * i)*
Explanation: • *The **first loop** (range(1, 6)) prints increasing stars from 1 to 5.* • *The **second loop** (range(4, 0, -1)) prints decreasing stars from 4 to 1.* • *print("* " * i) uses string multiplication to repeat "* " i times.*	**Explanation:** • *n = 5 → The pattern will go up to 5 stars in the middle.* • ***First loop (for i in range(1, n+1)):*** • *Prints stars from 1 star up to 5 stars (increasing order).* • ***Second loop (for i in range(n-1, 0, -1)):*** • *Prints stars from 4 stars down to 1 star (decreasing order).* • *print("* " * i) → Prints "* " repeated i times.*

Write a program to print a right-angled triangle of stars.

Method-1*: Right-Angled Triangle of Stars *Simplest Method
 # Define number of stars in each row *rows = [1, 2, 3, 4, 5]* *for count in rows:* *for i in range(count):* *print("*", end=" ")* *print()*
Output: * * * * * * * * * * * * * * *

Explanation

1. **rows = [1, 2, 3, 4, 5]**
   → This list tells how many stars to print in each row.
   - ✓ 1st row → 1 star
   - ✓ 2nd row → 2 stars
   - ✓ 3rd row → 3 stars
   - ✓ 4th row → 4 stars
   - ✓ 5th row → 5 stars
2. **Outer loop for count in rows:**
   - ✓ Goes through each number in the list (1, 2, 3, 4, 5).
   - ✓ That number (count) tells how many stars to print in that row.
3. **Inner loop for i in range(count):**
   - ✓ Runs count times.
   - ✓ Prints "* " each time on the same line because of end=" ".
4. **print() after inner loop**
   - ✓ Moves the cursor to the next line after finishing one row.

Method-2	***Method-3***
rows = 5 ***# number of rows*** *for i in range(1, rows + 1):* ***# Outer loop for rows*** *for j in range(i):* ***# Inner loop for stars*** *print("*", end=" ")* *print()* ***# Move to next line after each row***	***# Outer loop for rows (from 1 to 5)*** *for i in range(1, 6):* ***# Inner loop for printing stars in each row*** *for j in range(i):* *print("*", end=" ")* ***# Print star and stay on the same line*** *print()* ***# Move to the next line after each row***
Output: * * * * * * * * * * * * * * *	***Output:*** * * * * * * * * * * * * * * *
Explanation: 1. ***Outer loop (i)*** → *controls how many rows (5 rows).* 2. ***Inner loop (j)*** → *prints stars in each row.* ✓ *Row 1 → 1 star* ✓ *Row 2 → 2 stars* ✓ *Row 3 → 3 stars … and so on.* 3. *print() → moves to the next line after finishing each row.*	***Line-by-Line Explanation*** *for i in range(1, 6):* • *This is the* ***outer loop****.* • *It runs 5 times: i = 1 to i = 5.* • *Each value of i represents a* ***row number****.* *for j in range(i):* • *This is the* ***inner loop****, nested inside the outer loop.* • *It runs i times for each row.* ✓ *Row 1 → 1 star* ✓ *Row 2 → 2 stars* ✓ *Row 3 → 3 stars* ✓ *and so on…* *print("*", end=" ")* • *Prints a star followed by a space.* • *end=" " keeps the output on the same line.* *print()*

	• *Moves to the **next line** after printing all stars in a row.* • *Without this, all stars would appear on a single line.*

5. Write a program to create a hollow pyramid pattern given below:

```
*
* *
* * * *
* * * * * * * * *
```

Solution:

Nested Loops in Python

- A **nested loop** means one loop **inside another loop**.
- The **outer loop** controls the number of rows.
- The **inner loop** controls the number of columns (what to print in each row).
- Commonly used for patterns, matrices, tables, etc.

Example Program: hollow pyramid pattern

```
# Define number of stars in each row
rows = [1, 2, 4, 9]

for count in rows:
   for i in range(count):
      print("*", end=" ")
   print()
```

Output:

```
*
* *
* * * *
* * * * * * * * *
```

Explanation

1. **rows = [1, 2, 4, 9]**
   - ✓ This list defines how many stars will be printed in each row.
   - ✓ Row 1 → 1 star
   - ✓ Row 2 → 2 stars
   - ✓ Row 3 → 4 stars
   - ✓ Row 4 → 9 stars
2. **Outer loop for count in rows:**
   - ✓ Picks each number from the list (1, 2, 4, 9).
   - ✓ That number tells how many stars to print in the current row.
3. **Inner loop for i in range(count):**
   - ✓ Runs count times.
   - ✓ Prints "* " for each iteration, staying on the same line (end=" ").
4. **print() after inner loop**
   - ✓ Moves the cursor to the next line after finishing one row.

6. Write a program to create a hollow pyramid pattern given below:

```
******
*
*
******
*
*
******

******
*
*
*
******

******
*
*
******
*
*
******
```

Solution:

```
# E letter
rows = [6, 1, 1, 6, 1, 1, 6]
for count in rows:
    for i in range(count):
        print("*", end=" ")
    print()


# C letter
rows = [6, 1, 1, 1, 6]
for count in rows:
    for i in range(count):
        print("*", end=" ")
    print()

# E letter
rows = [6, 1, 1, 6, 1, 1, 6]
for count in rows:
    for i in range(count):
        print("*", end=" ")
    print()
```

Output:

```
* * * * * *
*
*
* * * * * *
*
*
* * * * * *

* * * * * *
*
*
*
* * * * * *

* * * * * *
*
*
* * * * * *
*
*
* * * * * *
```

Explanation

1. **Letter E** (rows = [6,1,1,6,1,1,6])
   - ✓ Row 1 → 6 stars
   - ✓ Row 2 → 1 star
   - ✓ Row 3 → 1 star
   - ✓ Row 4 → 6 stars
   - ✓ Row 5 → 1 star
   - ✓ Row 6 → 1 star
   - ✓ Row 7 → 6 stars

   This creates the letter **E** shape.
2. **Letter C** (rows = [6,1,1,1,6])
   - ✓ Row 1 → 6 stars
   - ✓ Row 2 → 1 star
   - ✓ Row 3 → 1 star
   - ✓ Row 4 → 1 star
   - ✓ Row 5 → 6 stars

   This creates the letter **C** shape.
3. **Letter E** (repeated same as the first).

   Thus, the final result is the word **E – C – E** drawn in stars.

Example Program: for Loop with else

```
for number in range(1, 3):
  for value in range(1, 6):
    product = number * value
    print(number, '*', value, '=', product)
  else:
    print('Multiplication Table for', number, 'is completed.')
```

Output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
1 * 4 = 4
1 * 5 = 5
Multiplication Table for 1 is completed.
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
Multiplication Table for 2 is completed.
```

Explanation

1. For each number in range (1,2), inner loop prints the table.
2. Once inner loop finishes, the else block executes.
3. It prints a completion message for each multiplication table.

2.3.2 *for* loop using string:

- A **string** is a sequence of characters.
- In Python, you can use a **for loop** to process each character in a string, one by one.
- This is useful for tasks like **counting letters, searching characters, skipping spaces, or reversing strings**.

Syntax of loop using string

```
for variable in string:
    # code to execute for each character
```

Explanation:

- **variable:** Will hold each character from the string, one by one.
- **string:** The string to be processed.

Example Program: Print each character of a string

```
text = "hello"
for ch in text:
    print(ch)
```

Output:

```
h
e
l
l
o
```

Explanation:

- The string "hello" has 5 characters.
- The loop runs 5 times.
- Each time, ch takes one character and prints it on a new line.

Example Program: Count vowels in a string

```
sentence = "Python is easy"
vowels = "aeiouAEIOU"
count = 0

for letter in sentence:
   if letter in vowels:
      count += 1

    print("Number of vowels:", count)
```

Output:

```
Number of vowels: 4
```

Explanation:

- The loop checks each character of "Python is easy".
- If the character is a vowel, the counter increases.
- Finally, the total vowel count is printed.

Example Program: Skip Spaces While Printing

```
sentence = "Hello World"
for c in sentence:
   if c == " ":
      continue  # skip spaces
       print(c, end="")
```

Output:

```
HelloWorld
```

Explanation:

- The continue statement skips the space " ".
- All non-space characters are printed continuously.
- end="" avoids line breaks, printing characters side by side.

Example Program: Reversing a String Using a for Loop

```
text = "hello"
reversed_text = ""
for char in text:
   reversed_text = char + reversed_text
    print(reversed_text)
```

Output:

```
olleh
```

Explanation:

- Each character is added in front of the previous string.
- This reverses "hello" into "olleh".

2.3.3 *for* LOOP USING LIST

- A **list** in Python is a collection of ordered items that can store different data types (e.g., numbers, strings).
- Lists are written using **square brackets [ ]**.
- A for loop can be used to iterate through all elements of a list.
- During each iteration, a variable takes the value of the next item in the list.
- The loop automatically stops after visiting every element.

Syntax of loop using list

```
for variable in list_name:
    # code to execute for each item
```

Explanation:

- **variable:** Holds the value of each element as the loop iterates.
- **list_name:** The list you want to loop through.

Example Program: Print all items of a list

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

Output:

```
apple
banana
cherry
```

Explanation:

- Each time through the loop, fruit takes the value of the next item in the fruits list.
- print(fruit) outputs each fruit name, one per line.

Example Program: Count Items in a List

```
numbers = [10, 20, 30, 40, 50]
count = 0

for num in numbers:
    count += 1
    print("Number of items:", count)
```

Output:

```
Number of items: 5
```

Explanation:

- The loop iterates through each number in the list.
- For each item, the counter increases by 1.
- After the loop, the total count is displayed.

Example Program: Print Only Even Numbers

```
numbers = [1, 2, 3, 4, 5, 6]
for n in numbers:
    if n % 2 == 0:
        print(n)
```

Output:

```
2
4
6
```

Explanation:

- The loop iterates through all numbers.
- The condition n % 2 == 0 checks if a number is even.
- Only even numbers are printed.

Example Program: Add 5 to Each Value

```
marks = [60, 70, 80]
for m in marks:
    print(m + 5)
```

Output:

```
65
75
85
```

Explanation:

- For each mark in the list, 5 is added.
- The updated value is printed.

2.3.4 *for* loop using dictionaries:

What is a Dictionary?

- A dictionary is a collection of key-value pairs.
- It is defined using curly braces { }.
  Example:
  student = {"name": "Ravi", "age": 20, "grade": "A"}

Simple for Loop Using Dictionaries:

- A dictionary in Python is an **unordered collection** of key-value pairs.
- With a for loop, you can iterate over:
  - ✓ **Keys** (default)
  - ✓ **Values**
  - ✓ **Key-Value Pairs**
- Useful for reading, processing, or displaying dictionary contents.

Syntax of loop using dictionaries: Iterating Over Keys (default)

```
for key in dictionary:
    # code using key
```

Syntax of loop using dictionaries: Iterating Over Keys (equivalently)

```
for key in dictionary.keys():
    # code
```

Explanation:

- By default, looping over a dictionary goes through its **keys**.
- dictionary.keys() explicitly returns all keys.
- You can access each value as dictionary[key].

Example Program: Print keys and values using keys

```
student = {"name": "Rahul", "age": 38}
for key in student:
    print(key, "->", student[key])
```

Output:

```
name -> Rahul
age -> 38
```

Explanation:

- The loop iterates over the keys of the dictionary student (by default, looping over a dictionary iterates over its keys).
- For each key, it accesses the corresponding value using student[key] and prints the key along with its value, formatted as key -> value.

Syntax of loop using dictionaries: Iterating Over Values

```
for value in dictionary.values():
    # code using value
```

Explanation:

- .values() returns all the **values** in the dictionary.
- Use this when only values matter.

Example Program: Print each character of a string

```
student = {"name": "Rahul", "age": 35}
for value in student.values():
    print(value)
```

Output:

```
Rahul
35
```

Explanation:

- student.values() returns all the **values** in the dictionary: "Rahul" and 35.
- The for loop prints each value one by one.

Syntax of loop using dictionaries: Iterating Over Key-Value Pairs

```
for key, value in dictionary.items( ):
    # code using key and value
```

Explanation:

- .items() returns **(key, value) pairs** as tuples.
- Both key and value can be unpacked in the loop.

Example Program: Iterating Over Key-Value Pairs

```
student = {"name": "Rahul", "age": 35}
for key, value in student.items():
    print(f"{key}: {value}")
```

Output:

```
name: Rahul
age: 35
```

Explanation:

- student.items() returns each **key-value pair** in the dictionary.
- The for loop assigns:
  - ✓ key → the name of the field (like "name" or "age")
  - ✓ value → the actual data (like "Rahul" or 35)
- print(f"{key}: {value}") displays each pair in a readable format.

Example Program: Print keys only

```
student = {"name": "Rahul", "age": 35, "grade": "A"}

for key in student:
    print(key)
```

Output:

name
age
grade

Explanation:

- student is a **dictionary** containing key-value pairs.
- The for key in student: loop goes through each **key** in the dictionary.
- print(key) displays each key one by one.

Example Program: Print keys and values

```
student = {"name": "Rahul", "age": 35, "grade": "A"}

for key, value in student.items():
    print(key, ":", value)
```

Output:

name : Rahul
age : 35
grade : A

Explanation:

- student.items() returns each **key-value pair** from the dictionary.
- The for loop assigns:
  - ✓ key → the name of the field (like "name", "age", "grade")
  - ✓ value → the actual data (like "Rahul", 35, "A")
- print(key, ":", value) displays each pair in the format key : value.

Example Program: Count total marks

```
marks = {"Math": 85, "Science": 90, "English": 80}
total = 0
for subject in marks:
   total += marks[subject]

    print("Total Marks:", total)
```

Output:

Total Marks: 255

Explanation:

- marks is a **dictionary** where each subject is a key and the marks are the values.
- total = 0 initializes the total score.
- The for loop goes through each **subject** in the dictionary.
- marks[subject] fetches the mark for that subject.
- total += marks[subject] adds the mark to the running total.
- After the loop, print("Total Marks:", total) displays the final sum.

Example Program: Print all Keys

```
person = {"name": "Rahul", "age": 35, "city": "New York"}

for key in person:
    print(key)
```

Output:

name
age
city

Explanation:

- person is a **dictionary** with three key-value pairs.
- The for key in person: loop goes through each **key** in the dictionary.
- print(key) displays each key one by one.

Example Program: Print all Values

```
person = {"name": "Rahul", "age": 35, "city": "Noida"}

for value in person.values():
    print(value)
```

Output:

```
Rahul
35
Noida
```

Explanation:

- person is a **dictionary** with three key-value pairs.
- person.values() returns a list-like view of all the **values**: "Rahul", 35, and "Noida".
- The for loop goes through each value and prints it one by one.

Example Program: Print Keys and Values

```
person = {"name": "Rahul", "age": 35, "city": "Noida"}

for key, value in person.items():
    print(key, ":", value)
```

Output:

```
name : Rahul
age : 35
city : Noida
```

Explanation:

- person.items() returns each **key-value pair** from the dictionary.
- The for loop assigns:
  - ✓ key → the name of the field (like "name", "age", "city")
  - ✓ value → the actual data (like "Rahul", 35, "Noida")
- print(key, ":", value) displays each pair in the format key : value.

Example Program: Count Items in Dictionary

```
fruits = {"apple": 3, "banana": 2, "cherry": 5}
count = 0
for fruit in fruits:
   count += 1
    print("Number of fruit types:", count)
```

Output:

```
Number of fruit types: 3
```

Explanation:

- fruits is a **dictionary** where each key is a fruit name and each value is the quantity.
- The for fruit in fruits: loop goes through each **key** in the dictionary.
- count += 1 adds 1 for every fruit type found.
- After the loop, count holds the total number of fruit types (keys).
- print() displays the result.

Example Program: Only Print Items with Value Above a Threshold

```
scores = {"Ram": 85, "Sham": 72, "Rahul": 91}

for name, score in scores.items():
    if score > 80:
        print(name, "scored above 80.")
```

Output:

```
Ram scored above 80.
Rahul scored above 80.
```

Explanation:

- scores.items() returns each name and score as a pair.
- The if score > 80: condition checks whether the score is greater than 80.
- If true, it prints the message with the student's name.

Note:

- By default, looping over a dictionary iterates through its keys.
- Use .values() to loop over just values, .items() for key-value pairs.
- Very useful for processing and analyzing named data or lookup tables in Python.

SUMMARY:

Syntax	Iterates Over	Typical use
for key in dictionary:	keys	When you need keys (and possibly values)
for value in dictionary.values():	values	When you only need values
for key, value in dictionary.items():	key-value	When you need both key and its corresponding value

Note:

- for key in dict (or dict.keys()) gives you keys
- for value in dict.values() gives you values
- for key, value in dict.items() gives you both, as a pair, in each iteration

2.4 USE OF WHILE LOOPS IN PYTHON:

What is a while loop?

- A **while loop** in Python repeatedly executes a block of code **as long as a given condition is True**.
- It is useful when you **do not know in advance** how many times you need to repeat an action.
- The loop ends when the condition becomes **False**.

Key Points:

- The condition is checked **before** each iteration.
- If the condition is **initially False**, the loop body may not run at all.
- If the condition never becomes False, the loop becomes **infinite** (unless broken with break).

Syntax

```
while condition:
    # code block to execute repeatedly
```

Explanation:

- **condition:** A Boolean expression that controls the loop continuation.
- The indented block under while runs as long as the condition is true.

- Always ensure that the loop changes something (inside or outside) that eventually makes the condition False, otherwise it will run forever.

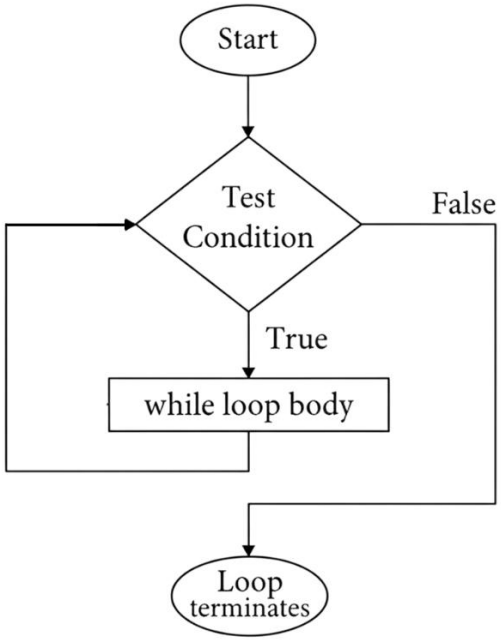

Figure 5: while loop

Example Program: Simple Counting

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

Output:

```
1
2
3
4
5
```

Explanation:

- Starts with count = 1.
- Prints numbers up to 5.
- Stops when count becomes 6 (condition count <= 5 is False).

Example Program: Taking User Input Until a Condition

```
text = ""
while text.lower() != "exit":
    text = input("Type 'exit' to quit: ")
        print("You typed:", text)
```

Output:

```
Type 'exit' to quit: hello
You typed: hello
Type 'exit' to quit: python
You typed: python
Type 'exit' to quit: exit
You typed: exit
```

Explanation:

- The loop continues as long as text.lower() != "exit".
- Each time, it asks for input and prints back the typed text.
- When the user types "exit" (in any case like EXIT, Exit, etc. because of .lower()), the condition becomes False and the loop stops.

Example Program: Factorial Calculation

```
n = 5
fact = 1
i = 1

while i <= n:
   fact *= i
   i += 1

    print("Factorial of", n, "is", fact)
```

Output:

```
Factorial of 5 is 120Type 'exit' to quit: exit
You typed: exit
```

Explanation:

- Multiplies numbers 1×2×3×4×5.
- Gives factorial 120.

Example Program: Sum of numbers until user enters 0

```
total = 0
num = int(input("Enter a number (0 to stop): "))

while num != 0:
   total += num
   num = int(input("Enter a number (0 to stop): "))

    print("Total sum:", total)
```

Output:

```
Enter a number (0 to stop): 5
Enter a number (0 to stop): 10
Enter a number (0 to stop): 0
Total sum: 15
```

Explanation:

- Continuously takes input numbers.
- Adds each number to total.
- Stops when the user enters 0.

Example Program: Reverse Countdown

```
count = 5
while count > 0:
   print(count)
       count -= 1
```

Output:

```
5
4
3
```

```
2
1
```

Explanation:

- Starts from 5 and decreases by 1.
- Stops when count reaches 0.

Example Program: Multiplication Table of 3

```
i = 1
while i <= 10:
   print("3 x", i, "=", 3*i)
       i += 1
```

Output:

```
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
...
3 x 10 = 30
```

Explanation:

- Multiplies 3 with numbers 1 to 10.

Example Program: Infinite loop (use with caution)

```
while True:
       print("This runs forever unless stopped!")
```

Explanation:

- This loop **never ends** on its own.
- Must be stopped using break or manually by the user.

Example Program: Reverse Countdown

```
count = 5
while count > 0:
   print(count)
       count -= 1
```

Output:

```
5
4
3
2
1
```

Explanation:

- Starts from 5 and decreases by 1.
- Stops when count reaches 0.

Example Program: Print Characters of a String

```
text = "Python"
i = 0

while i < len(text):
   print(text[i])
       i += 1
```

Output:

P
y
t
h
o
n

Explanation:

- Goes through each index of the string.
- Prints letters one by one.

Example Program: Infinite loop (use with caution)

```
while True:
    print("This runs forever unless stopped!")
```

Explanation:

- This loop **never ends** on its own.
- Must be stopped using break or manually by the user.

Note:

- while loops are useful when the number of iterations is not predetermined.
- Always ensure the condition eventually becomes False or use break to avoid infinite loops.
- Can be combined with else (executed if loop terminates normally without break).

Example Program: Multiply Two Numbers Without Using * Operator

```
first_num = int(input("Enter the first number: "))
second_num = int(input("Enter the second number: "))

result = 0
counter = first_num

while counter > 0:
   result += second_num
   counter -= 1

    print("The product of", first_num, "and", second_num, "is", result)
```

Output:

```
Enter the first number: 4
Enter the second number: 5
The product of 4 and 5 is 20
```

Explanation:

- Initialize result = 0 to store the final product.
- The loop runs first_num times.
- In each iteration, we add second_num to result.
- This simulates multiplication as repeated addition.
- For example: 5 + 5 + 5 + 5 = 20 → which is 4 * 5

2.5 Difference Between *for* and *while* Loops

Feature	for loop	while loop
Syntax	***while condition:*** *# code block to execute repeatedly*	***for variable in sequence:*** *# code to execute for each item*
Usage	Used when we know **how many times** we want to repeat.	Used when we **don't know how many times** to repeat, depends on a condition.
Control	Iterates over a **sequence** (range, list, string, etc.) or fixed number of times.	Repeats until the **condition becomes False**.
Stopping	Stops automatically after the last element / fixed range.	Must update variables inside the loop, otherwise it can run forever.
Best for	When number of iterations is **known in advance**.	When iterations depend on **user input or condition**.
Example program	***# Using for loop (fixed range)*** *for i in range(1, 6):* *print(i)* **Output:** 1 2 3 4 5 **Note:** We **know** we want numbers from 1 to 5 → perfect for **for loop.**	***# Using while loop (condition-based)*** *count = 1* *while count <= 5:* *print(count)* *count += 1* **Output:** 1 2 3 4 5 **Note:** Here, the loop **runs until condition count <= 5 is false**.

SUMMARY:

Feature	Description
Condition	Checked before each loop iteration
Use case	When number of repetitions is unknown
Exit loop	Condition becomes False, or use break to stop
Special case	while True: creates an infinite loop (exit using break)
Optional else	else block can be added; runs if the loop ends normally (not by break)

2.3.1 Loop Manipulation using pass in Python:

What is pass?

- pass is a **null statement** in Python.
- It does nothing when executed.
- It is used as a **placeholder** where syntactically a statement is required but you don't want any action to occur.
- This helps avoid **syntax errors** when writing code structures that are not yet fully implemented.

Why use pass in loops?

- To **maintain valid code structure** when the loop body is not written yet.
- Useful during **code development, debugging, or prototyping**.
- Clearly indicates that the loop (or a condition inside the loop) is **intentionally left blank.**

Syntax:

```
for item in sequence:
    pass                    # Loop body intentionally left empty

while condition:
    pass                    # Placeholder for future logic
```

Explanation:

- for variable in sequence: pass → The loop runs, but does nothing inside.
- while condition: pass → The loop condition is checked, but the body executes no action.
- It prevents **syntax errors** from having an empty loop body.

Example Program: Using pass in a for loop

```
# Example 1: Using pass in a for loop
for i in range(5):
    pass   # Does nothing, just a placeholder

print("Loop executed successfully.")
```

Output:

```
Loop executed successfully.
```

Explanation:

- The for i in range(5) runs 5 times (i = 0, 1, 2, 3, 4).
- Inside the loop, pass is used, which means **"do nothing"**.
- After the loop completes, the print statement executes.

Example Program: Using pass as a placeholder inside a condition

```
for letter in "Python":
    if letter == "h":
        pass   # Placeholder for future logic
    else:
            print("Letter:", letter)
```

Output:

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

Explanation:

- When the loop encounters 'h', it executes pass (does nothing).
- For other letters, it prints them.
- Later, logic can be added where pass is written.

Example Program: pass in a loop over a list

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

Output:

```
Loop finished.
```

Explanation:

- The loop iterates over all fruits but does nothing inside.

- The program structure remains valid.

Example Program: Conditional pass in a loop

```
for i in range(5):
    if i == 2:
        pass  # Do nothing for i == 2
    else:
        print(f"Processing {i}")
```

Output:

```
Processing 0
Processing 1
Processing 3
Processing 4
```

Explanation:

- The loop runs from 0 to 4.
- At i = 2, the pass statement is executed (no action).
- Other numbers are processed normally.

Note:

- pass **does nothing**; it is only a placeholder.
- It is **not the same as continue or break**:
  - ✓ ***continue*** → *skips to the next iteration.*
  - ✓ ***break*** → *exits the loop entirely.*
  - ✓ ***pass*** → *simply "do nothing" and move ahead.*
- It is commonly used in **developing code outlines**, **empty loop bodies**, **empty functions**, and **class definitions**.

2.3.2 Loop Manipulation Using continue:

What is continue?

- The continue statement is used inside loops to **skip the current iteration** and move directly to the **next iteration** of the loop.
- The loop does **not terminate**; it simply bypasses the remaining code in that iteration.

When to use continue

- When you want to **skip certain values** during iteration.
- Useful for **filtering specific conditions** while still continuing the loop.

Explanation:

The **continue statement** in Python is used within loops (for or while) to **skip the rest of the code inside the loop for the current iteration** and immediately proceed to the next iteration. When the continue statement is executed:

- It **skips the rest of the code** inside the loop for that iteration.
- Control immediately goes to the **next iteration**.
- Works with both **for** and **while** loops.

This is helpful when you want to ignore specific cases without stopping the entire loop.

Syntax: for Loop with continue

```
for item in sequence:
    if condition:
        continue
    # remaining code
```

Explanation:

- for item in sequence: This starts a loop that goes through each item in a sequence (like a list, tuple, or string).
- if condition: This checks a specific condition for the current item.
- continue If the condition is true, continue tells Python to **skip the rest of the code** inside the loop for that item and **move to the next item**.
- # remaining code This part runs **only if the condition is false** i.e., if continue is not triggered.

Example Program: Skip Even Numbers in a for Loop

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

for num in numbers:
    if num % 2 == 0:
        continue
        print(num)
```

Output:

```
1
3
5
```

Explanation:

- Even numbers are skipped; only odd numbers are printed.

Example Program: Skip numbers less than 5 in a for Loop

```
numbers = [2, 5, 7, 3, 8]

for num in numbers:
    if num < 5:
        continue
        print(num)
```

Output:

```
5
7
8
```

Explanation:

- Numbers less than 5 (2 and 3) are skipped.
- Only numbers 5, 7, and 8 are printed.

Example Program: Skipping a value in a for Loop

```
# Program to print all numbers from 1 to 5 except 3

for i in range(1, 6):
    if i == 3:
        continue
        print(i)
```

Output:

1
2
4
5

Explanation:

- The loop runs with i from 1 to 5.
- When i is 3, continue skips the print(i) statement for that iteration.
- So, 3 is not printed.

Example Program: Printing Only Odd Numbers

```
for num in range(1, 10):
    if num % 2 == 0:
        continue  # Skip even numbers
        print(num)
```

Output:

1
3
5
7
9

Explanation:

- The loop checks if num is even. If true, it executes continue and skips printing.
- This results in only odd numbers being printed.

Example Program: Skip vowels in a string

```
text = "welcome"
for char in text:
    if char in "aeiou":
        continue
        print(char, end=" ")
```

Output:

w l c m

Explanation:

- Whenever a vowel is found, continue skips printing it.
- Only consonants are displayed.

Syntax: while Loop with continue

```
for item in sequence:
    if condition:
        continue
    # remaining code
```

Explanation:

- while condition: Starts a loop that runs **as long as the condition is true**.
- if condition_to_skip: Checks if a certain situation should be skipped.
- continue If the condition is met, continue **skips the rest of the code** inside the loop for that iteration and **goes back to check the condition again**.
- # remaining code This part runs **only if the skip condition is false**.

Example Program: Skip Multiples of 3 in a while Loop

```
num = 1
while num <= 6:
   if num % 3 == 0:
      num += 1
      continue
   print(num)
       num += 1
```

Output:

```
1
2
4
5
```

Explanation:

- Numbers 3 and 6 are skipped because they are divisible by 3.
- The loop continues without printing them.

Example Program: continue in a While Loop

```
n = 0
while n < 5:
   n += 1
   if n == 3:
      continue
       print("Current value:", n)
```

Output:

```
Current value: 1
Current value: 2
Current value: 4
Current value: 5
```

Explanation:

- n is incremented each time.
- When n == 3, continue skips the print() statement for that iteration.

Note:

- continue skips only the **current iteration**, not the entire loop.
- Any code after continue inside the loop body is not executed for that iteration.
- Works with both for and while loops.
- Useful for **filtering, skipping unwanted cases, or selective processing**.

2.3.3 Loop manipulation using break:

What is break in Python?

- The break statement is used to **exit a loop immediately**, regardless of whether the loop condition is still true or the sequence has more items.
- Once break is executed:
  - ✓ The **current loop** (either for or while) terminates instantly.
  - ✓ The program continues with the **next statement after the loop**.

When to Use break

- To stop a loop early when a certain condition is met.
- To optimize performance by avoiding unnecessary iterations.
- ***Commonly used in:***

- ✓ Search operations (stop after finding the target).
- ✓ Menu-driven programs.
- ✓ Error handling within loops.

Syntax of **break** in a **for loop**

```
for variable in sequence:
    if condition:
        break                    # Exit the loop completely
    # Code here runs only if break is not executed
```

Explanation:

- The for loop goes through each item in the sequence.
- If the **condition is true**, break stops the loop right away.
- Otherwise, the code inside the loop continues.
- After the loop ends, the program continues with the next statement after the loop.

Example Program: Break in a for Loop

```
# Stop the loop when number 5 is found
for i in range(1, 11):
    if i == 5:
        break
        print(i)
```

Output:

```
1
2
3
4
```

Explanation:

- The loop starts from 1 and goes up to 10.
- When i == 5, break executes, ending the loop immediately.

Example Program: Using break in a search operation

```
# Searching for a name in a list

names = ["Rahul", "Bharat", "Arun", "Manoj"]
search_name = "Rahul"

for name in names:
    if name == search_name:
        print(f"{search_name} found!")
        break
else:
        print(f"{search_name} not found.")
```

Output:

```
Rahul found!
```

Explanation:

- A list of names is created: ["Rahul", "Bharat", "Arun", "Manoj"].
- We want to search for "Rahul" in that list.
- The for loop checks each name one by one.
- If a match is found (name == search_name), it prints "Rahul found!" and **exits the loop** using break.

- The else after the loop runs **only if the loop completes without finding the name** (i.e., no break occurred).
  - ✓ If "Rahul" is not in the list, it prints "Rahul not found."

Example Program: Finding the First Multiple in a List

```
numbers = [7, 13, 21, 25, 40]
for num in numbers:
    if num % 5 == 0:
        print("First multiple of 5 is:", num)
            break
```

Output:

First multiple of 5 is: 25

Explanation:

- The loop checks each number for divisibility by 5.
- Stops at the first match (25).

Example Program: Searching for an Element

```
fruits = ["apple", "banana", "cherry", "mango"]
target = "cherry"
for f in fruits:
    if f == target:
        print(f"{target} found!")
        break
else:
        print(f"{target} not found!")
```

Output:

cherry found!

Explanation:

- The loop searches for "cherry".
- Once found, it prints and breaks out of the loop.
- The else block after the loop is run only if break never executes (i.e., if the target is not found).

Example Program: Break Statement in *for Loop*	**Example Program: Continue Statement in *for Loop***
# create a list of strings *names = ['Rahul', 'Bharat', 'Manoj', 'Kazi']* ***# access each list value*** *for name in names:* *if name == 'Manoj':* *break* *print(name)*	***# create a list of strings*** *names = ['Rahul', 'Bharat', 'Manoj', 'Kazi']* ***# access each list value*** *for name in names:* *if name == 'Bharat':* *continue* *print(name)*
Output: ***Rahul*** ***Bharat***	***Output:*** ***Rahul*** ***Manoj*** ***Kazi***
Explanation:	***Explanation:*** *1. First iteration: "Rahul" → condition name == 'Bharat' is False, so it prints "Rahul".*

1. *Loop starts with "Rahul" → condition name == 'Manoj' is False, so it prints "Rahul".* 2. *Next iteration: "Bharat" → condition still False, so it prints "Bharat".* 3. *Next iteration: "Manoj" → condition becomes True, so the break statement executes and stops the loop immediately.* 4. *"Kazi" is never executed because the loop ends when "Manoj" is found.*	2. *Second iteration: "Bharat" → condition is True, so the continue statement is executed. The loop skips printing "Bharat" and moves to the next iteration.* 3. *Third iteration: "Manoj" → condition is False, so it prints "Manoj".* 4. *Fourth iteration: "Kazi" → condition is False, so it prints "Kazi".* ***Final output excludes "Bharat" because continue skipped it.***

Syntax of break in a while loop

```
while condition:
    if other_condition:
        break  # Exit the loop immediately
    # More code inside the loop
# Code here runs after the loop ends
```

Explanation:

- The while loop keeps running **as long as the condition is true**.
- If other_condition becomes true, break will stop the loop immediately.
- Otherwise, the loop body continues to run.
- When the loop ends (either naturally or by break), the program moves on.

Example Program: break in a while loop

```
# Program to find and stop at the first even number

i = 1
while i <= 10:
    if i % 2 == 0:
        print("First even number found:", i)
        break
        i += 1
```

Output:

```
First even number found: 2
```

Explanation:

- The loop increments i from 1.
- When i == 2, the condition is true, and break ends the loop.

Example Program: Using break in a While Loop

```
count = 1
while count <= 10:
    if count == 6:
        break
    print("Value:", count)
        count += 1
```

Output:

```
Value: 1
Value: 2
Value: 3
```

```
Value: 4
Value: 5
```

Explanation:

- The loop runs while count <= 10.
- At count == 6, break exits the loop.

Note:

- break immediately exits the **current loop** (not outer loops).
- Code after break inside the loop is never executed.
- Useful for:
  - ✓ Early termination of loops.
  - ✓ Search operations.
  - ✓ Avoiding unnecessary iterations.
- In **nested loops**, only the **innermost loop** is terminated by break.

SYNTAX: break in **while** loop and **for** loop:

while Loop Syntax with break	***for Loop Syntax with break***
while <test-condition>: *statement_1* *if <condition>:* ***break*** *statement_2* *statement_3* ***statement_4***	***for <var> in <sequence>:*** *statement_1* *if <condition>:* ***break*** *statement_2* *statement_3* ***statement_4***
Explanation: • *The loop continues as long as <test-condition> is true.* • *If <condition> becomes true inside the loop, break exits the loop immediately.* • *statement_4 executes after the loop terminates.*	**Explanation:** • *The loop iterates over each item in <sequence>.* • *If <condition> is met, break exits the loop early.* • *statement_4 runs after the loop ends.*

2.3.4 Loop Manipulation Using else:

What is else in Loops?

- In Python, a for loop or a while loop can have an **else block**.
- The else block runs **only if the loop completes normally** (i.e., without being stopped by a break statement).

Detailed Explanation:

- If the loop **finishes all iterations** (for **for loops**) or the condition becomes False (for while loops), then the else block runs.
- If the loop **exits early due to a break**, then the else block is skipped.
- This feature is especially useful in:
  - ✓ **Search operations** → to check if an item was found or not.
  - ✓ **Post-loop tasks** → something to do only if the loop was not interrupted.

Syntax: for loop with else *(for + else)*

```
for variable in sequence:
    # loop code
else:
        # code that runs if loop completes without break
```

Explanation:

- If the loop iterates through the entire sequence without hitting break, the else block executes.
- If break is used inside the loop, the else block is skipped.

Example Program: for-else: Searching for a Number

```
numbers = [2, 4, 6, 8, 10]
target = 5
for n in numbers:
    if n == target:
        print("Found:", target)
        break
else:
        print(target, "not found.")
```

Output:

```
5 not found.
```

Explanation:

- The loop checks each number.
- Since 5 is not in the list, the loop completes without break.
- The else block runs.

Example Program: for-else (with break)

```
numbers = [1, 3, 5, 7]
for n in numbers:
    if n % 2 == 0:
        print("Even number found:", n)
        break
else:
        print("No even number found.")
```

Output:

```
No even number found.
```

Explanation:

- The list has only odd numbers, so break is never executed.
- The else block confirms that no even number was found.

Example Program: for loop with else and break

```
for i in range(5):
    if i == 3:
        break
    print("Number:", i)
else:
        print("Loop finished.")
```

Output:

```
Number: 0
Number: 1
Number: 2
```

Explanation:

- When i is 3, the break statement stops the loop.
- Therefore, the else block is not executed.

Syntax: while loop with else

```
while condition:
   # loop code
else:
      # code that runs if loop condition becomes false naturally
```

Explanation:

- The else block executes only if the loop ends when the condition becomes False.
- If the loop is stopped using break, the else block does not run.

Example Program: for loop with else (No break)

```
for i in range(3):
   print("Number:", i)
else:
      print("Loop finished without break.")
```

Output:

```
Number: 0
Number: 1
Number: 2
Loop finished without break.
```

Explanation:

- The loop runs while x < 4.
- When x becomes 4, the condition is False, so the else block runs.

Example Program: while-else (with break)

```
x = 1
while x < 4:
   print("x is", x)
   x += 1
else:
      print("x is no longer less than 4.")
```

Output:

```
x is 1
x is 2
x is 3
x is no longer less than 4.
```

Explanation:

- When x reaches 3, break interrupts the loop.
- Because of break, the else block is skipped.

Example Program: while loop with else and break

```
x = 0
while x < 5:
   if x == 3:
      break
   print("x =", x)
```

```
    x += 1
else:
        print("Completed loop.")
```

Output:

```
x = 0
x = 1
x = 2
```

Explanation:

- The loop stops early at x = 3 due to break.
- So, the else block is skipped.

Example Program: while-else

```
x = 0
while x < 3:
    print(x)
    x += 1
else:
        print("Loop finished without interruption.")
```

Output:

```
0
1
2
Loop finished without interruption.
```

Explanation:

- The while loop reaches its natural end (x < 3 becomes False), so the else block executes.

Example Program: while-else Skipped Due to break

```
x = 0
while x < 5:
    if x == 2:
        break
    print(x)
    x += 1
else:
        print("Loop finished without interruption.")
```

Output:

```
0
1
```

Explanation:

- When x becomes 2, the break statement interrupts the loop.
- The else block is not executed.

Example Program: Using while with break

```
n = 1
while True:  # infinite loop
    print(n)
    if n == 3:
        break  # exit the loop
        n += 1
```

Output:

1
2
3

Explanation:

- while True creates an infinite loop.
- Prints numbers starting from 1.
- When n reaches 3, break exits the loop.

Note:

- The else clause in loops is **not** an error handler.
- It is a clean way to run code **only when the loop finishes without interruption**.
- Best use case: **search tasks** (e.g., check if an item exists in a list).

SUMMARY:

Loop Type	Loop ends with break	Loop ends normally (without break)	Does *else* Run?
for	Yes (stops early)	No	Only without break
while	Yes	No	Only without break

2.3.5 Loop manipulation using pass, continue, break and else:

Example Program: Program that demonstrates the use of pass, continue, break, and else within loop manipulation

```
numbers = [1, 2, 0, 4, 5, -1, 6]

for num in numbers:
    if num == 0:
        pass  # Do nothing for 0, just a placeholder
    elif num < 0:
        print("Negative number encountered! Stopping loop.")
        break  # Exit loop if number is negative
    elif num % 2 == 0:
        continue  # Skip even numbers, proceed to next iteration
    print("Processing:", num)
else:
        print("Loop finished without break.")
```

Output:

Processing: 1
Processing: 5
Negative number encountered! Stopping loop.

Explanation:

- **pass** is used as a placeholder when num == 0. No operation is performed for this case.
- **break** is used to exit the loop immediately if a negative number is found.
- **continue** is used to skip the rest of the loop body for even numbers (except 0, which is handled by pass).
- **else** (tied to the for-loop) runs only if the loop completes without a break.

Step-by-Step Explanation:

1. **Iteration 1 (num = 1):**
   - *Not 0 (so not pass), not negative, not even.*

- *print("Processing:", num) runs → Output: Processing: 1*

2. **Iteration 2 (num = 2):**
   - *Not 0, not negative, is even (continue): skip the print, move to the next iteration.*
3. **Iteration 3 (num = 0):**
   - *Is 0 (pass): nothing happens in the body; move to next iteration.*
4. **Iteration 4 (num = 4):**
   - *Not 0, not negative, is even (continue): skip printing.*
5. **Iteration 5 (num = 5):**
   - *Not 0, not negative, not even.*
   - *print("Processing:", num) runs → Output: Processing: 5*
6. **Iteration 6 (num = -1):**
   - *Not 0, is negative (break): prints message and exits loop → Output: Negative number encountered! Stopping loop.*
   - *The loop is terminated; else is skipped.*
7. **Iteration 7 (num = 6):**
   - *Never reached, because loop was broken at previous step.*

What if There Was No Negative Number?
Let's say `numbers = [1, 2, 0, 4, 5, 7, would be:

OUTPUT:

Processing: 1
Processing: 5
Processing: 7
Loop finished without break.

- All numbers processed unless skipped by continue or pass.
- Since no break occurred, the else block runs after the loop.

Example Program: Process Odd Numbers, Skip Evens, Stop on Negative, Track Loop Progress

```
numbers = [3, 2, 0, 7, -1, 9]
for num in numbers:
    if num == 0:
        pass  # Placeholder; do nothing for zero
    elif num < 0:
        print("Negative found. Exiting loop.")
        break  # Exit loop when negative encountered
    elif num % 2 == 0:
        continue  # Skip even numbers
    print("Odd number:", num)
else:
        print("No negative numbers found; loop finished normally.")
```

Output:

Odd number: 3
Odd number: 7
Negative found. Exiting loop.

Explanation:

- pass: Does nothing if number is 0.
- continue: Skips the print for even numbers (2).
- break: Exits when a negative number is found (-1).
- else: Skipped because loop is terminated with break.

Example Program: Process Odd Numbers, Skip Evens, Stop on Negative, Track Loop Progress

```
words = ["hello", "", "ok", "example", "stop", "continue"]
target = "example"

for word in words:
   if word == "":
      pass  # No operation for empty strings
   elif word == "stop":
      print("Stop word encountered! Breaking loop.")
      break
   elif len(word) < 3:
      continue  # Skip words shorter than 3 letters
   if word == target:
      print("Found target:", word)
else:
      print("Target not found or 'stop' not encountered.")
```

Output:

```
Found target: example
Stop word encountered! Breaking loop.
```

Explanation:

- Skips empty string with pass, skips "ok" (less than 3 letters) with continue.
- Prints the target when matched.
- Breaks on "stop", so else does not execute.

Example Program: Validating Inputs - Pass on Blank, Continue for Non-integers, Break on Zero, Else if No Zero

```
inputs = ["5", "abc", "", "7", "0", "3"]
for item in inputs:
   if item == "":
      pass  # Placeholder for blank entries
   elif not item.isdigit():
      continue  # Skip non-numeric input
   elif int(item) == 0:
      print("Zero detected! Ending input check.")
      break
   print("Valid number:", item)
else:
      print("All items checked; no zero found.")
```

Output:

```
Valid number: 5
Valid number: 7
Zero detected! Ending input check.
```

Explanation:

- Ignores blanks with pass, skips "abc" with continue.
- Ends loop when "0" is found via break; else is skipped.

Example Program: Loop Over Scores, Use pass for Negatives, continue for Failing, break on 100, else if No 100

```
scores = [42, -1, 85, 67, 100, 73]
for score in scores:
```

```
    if score < 0:
        pass  # Ignore negative scores
    elif score < 50:
        continue  # Skip failing scores below 50
    elif score == 100:
        print("Perfect score found! Stopping search.")
        break
    print("Passing score:", score)
else:
    print("No perfect score found in the list.")
```

Output:

```
Passing score: 85
Passing score: 67
Perfect score found! Stopping search.
```

Explanation:

- Negative value triggers pass, below-50 triggers continue, 100 triggers break. else block is NOT executed since loop breaks.

Example Program: Checking Names - Pass for Placeholder, continue for Short, break for 'END', else if Normal Exit

```
names = ["Rahul", "___", "Bharat", "Manoj", "Kazi", "END"]

for name in names:
    if name == "___":
        pass  # Placeholder - skip
    elif len(name) <= 3:
        continue  # Skip names of 3 or fewer characters
    elif name == "END":
        print("END marker found, stopping.")
        break
    print("Accepted name:", name)
else:
    print("All names processed without END marker.")
```

Output:

```
Accepted name: Rahul
Accepted name: Bharat
Accepted name: Manoj
Accepted name: Kazi
END marker found, stopping.
```

Explanation

1. "Rahul" → not "___", length > 3, not "END" → printed.
2. "___" → matches placeholder → pass does nothing, so loop continues silently.
3. "Bharat" → passes all checks → printed.
4. "Manoj" → passes all checks → printed.
5. "Kazi" → length = 4 (>3), so printed.
6. "END" → matches break condition → "END marker found, stopping." printed, loop ends.
7. Because of the break, the else block is skipped.

Example Program: Python Loop Control: Combined Use of pass, continue, break, and else

```
for i in range(7):
    if i == 0:
        pass  # Placeholder, does nothing
    elif i == 3:
        continue  # Skip printing 3
    elif i == 5:
        break  # Exit the loop
    print("Current value:", i)
else:
        print("Completed loop.")
```

Output:

```
Current value: 0
Current value: 1
Current value: 2
Current value: 4
```

Explanation:

- i == 0 → pass is used.
- i == 3 → continue skips printing 3.
- i == 5 → break exits the loop.
- The else block is skipped because of the break.

Example Program: Skipping Even Numbers and Breaking at 9

```
for i in range(1, 11):
    if i % 2 == 0:
        continue  # Skip even numbers
    elif i == 9:
        break     # Stop loop at 9
    elif i == 7:
        pass      # Do nothing special
    print("Odd number:", i)
else:
        print("Loop finished successfully.")
```

Output:

```
Odd number: 1
Odd number: 3
Odd number: 5
Odd number: 7
```

Explanation:

- Even numbers (i % 2 == 0) are skipped using continue.
- pass does nothing when i == 7, just a placeholder.
- Loop ends at 9 due to break.
- else block doesn't run because the loop was broken.

Example Program: Searching for a Target Number

```
target = 6
for num in range(1, 10):
    if num < target:
        pass  # Ignore numbers less than target
    elif num == target:
```

```
        print("Found target:", num)
        break  # Exit loop when found
    print("Checking:", num)
else:
        print("Target not found.")
```

Output:

```
Checking: 1
Checking: 2
Checking: 3
Checking: 4
Checking: 5
Found target: 6
```

Explanation:

- Numbers less than target are passed over.
- Break exits the loop once target is found.
- else is skipped due to break.

2.3.6 Programming using Python conditional and loop blocks:

Syntax: if, elif and else

```
for variable in sequence:
    if condition1:
        # Code block if condition1 is True
    elif condition2:
        # Code block if condition2 is True
    else:
        # Code block if no conditions are True
```

Syntax: while, if and else

```
while condition:
    if condition1:
        # Code block
    else:
        # Another code block
```

Combined Syntax: for, while with if, elif, else

```
# For loop with conditionals
for item in iterable:
    if condition1:
        # Block 1
    elif condition2:
        # Block 2
    else:
        # Block 3

# While loop with conditionals
while condition:
    if conditionA:
        # Block A
    else:
        # Block B
```

Example Program: List of numbers to check even/odd using for loop

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

print("Checking numbers using for loop:")
for num in numbers:
   if num % 2 == 0:
      print(num, "is even")
   elif num % 2 != 0:
      print(num, "is odd")
   else:
      print(num, "Unknown type")

print("\nCounting with while loop until we find a multiple of 7:")

count = 1
while count <= 10:
   if count % 7 == 0:
      print(count, "is divisible by 7 - stopping loop.")
      break
   else:
      print(count, "is not divisible by 7")
       count += 1
```

Output:

```
Checking numbers using for loop:
1 is odd
2 is even
3 is odd
4 is even
5 is odd

Counting with while loop until we find a multiple of 7:
1 is not divisible by 7
2 is not divisible by 7
3 is not divisible by 7
4 is not divisible by 7
5 is not divisible by 7
6 is not divisible by 7
7 is divisible by 7 - stopping loop.2 is positive.
-8 is negative.
0 is zero.
5 is positive.
Summary:
Positives: 4
Negatives: 2
Zeros    : 2
```

Explanation:

for loop section:

- Iterates over a list of numbers [1, 2, 3, 4, 5].
- Uses if, elif, else to check each number and print whether it is even or odd.

while loop section:

- Starts from count = 1 and increments until it finds a number divisible by 7.
- Uses if to check for a multiple of 7, and break to exit the loop when found.
- Otherwise, it keeps printing and increasing the count.

Example Program: Categorize Numbers in a List

```
numbers = [10, -3, 0, 7, 2, -8, 0, 5]

positive_count = 0
negative_count = 0
zero_count = 0

for n in numbers:
    if n > 0:
        print(f"{n} is positive.")
        positive_count += 1
    elif n < 0:
        print(f"{n} is negative.")
        negative_count += 1
    else:
        print(f"{n} is zero.")
        zero_count += 1

print("Summary:")
print("Positives:", positive_count)
print("Negatives:", negative_count)
    print("Zeros    :", zero_count)
```

Output:

```
10 is positive.
-3 is negative.
0 is zero.
7 is positive.
2 is positive.
-8 is negative.
0 is zero.
5 is positive.
Summary:
Positives: 4
Negatives: 2
Zeros    : 2
```

Explanation:

- The program iterates through each number in the numbers list using a for loop.
- Inside the loop, it uses if-elif-else conditional blocks:
    - If the number is **greater than zero**, it is counted as positive.
    - If the number is **less than zero**, it is counted as negative.
    - If the number is **zero**, it is counted in the zero category.
- Each category's count is incremented accordingly.
- After processing all numbers, a summary of the number of positives, negatives, and zeros is printed.

Example Program: Finding Prime Numbers within a Range

```
start = 0
end = 20

print(f"Prime numbers between {start} and {end}:")

for number in range(start, end + 1):
    if number > 1:
        for i in range(2, int(number ** 0.5) + 1):
            if number % i == 0:
                break
        else:
                print(number)
```

Output:

```
Prime numbers between 0 and 20:
2
3
5
7
11
13
17
19
```

Explanation:

- The program loops through numbers from 0 to 20.
- For each number greater than 1, it checks if there are any divisors other than 1 and itself.
- If not, it prints the number as a prime.

Example Program: Student Grade Categorizer

```
students = {
    "Rahul": 78,
    "Arun": 66,
    "Bharat": 92,
    "Joshi": 54,
    "Krishna": 48
}

for name, score in students.items():
    if score >= 90:
        grade = "A"
    elif score >= 75:
        grade = "B"
    elif score >= 60:
        grade = "C"
    elif score >= 50:
        grade = "D"
    else:
        grade = "F"
         print(f"{name} scored {score} and got grade {grade}.")
```

Output:

Rahul scored 78 and got grade B.
Arun scored 66 and got grade C.
Bharat scored 92 and got grade A.
Joshi scored 54 and got grade D.
Krishna scored 48 and got grade F.

Explanation:

- The program uses a for loop to iterate over student names and their scores from a dictionary.
- Inside the loop, conditional blocks (if-elif-else) determine the corresponding grade based on the score.
- Each student's name, score, and grade are printed accordingly.

This example clearly demonstrates how conditional and loop blocks work together to process and categorize data efficiently.

COMPARISON OF LOOP CONTROL STATEMENTS IN PYTHON

Statement	What it does	When it is used	Effect on loop	Example use case
break	Exits the loop immediately	When you want to stop the loop early once a condition is met	Loop ends completely, execution continues after the loop	Stop searching when the target element is found
continue	Skips the rest of the code in the current iteration and moves to the next iteration	When you want to skip certain values/cases but continue looping	Current iteration is skipped, loop goes to the next cycle	Skip even numbers and print only odd numbers
pass	Does nothing (a placeholder statement)	When a statement is required syntactically but no action is needed yet	No effect, loop continues normally	Writing loop structure during development but leaving body empty
else (with loop)	Runs only if the loop finishes **without break**	When you want to confirm that the loop completed normally	Executes after the loop ends if **no break** occurred	Search loop → print "Not found" only if no break

Example: break

```
for i in range(5):
    if i == 3:
        break
    print(i)
# Output: 0 1 2
```

Example: continue

```
for i in range(5):
    if i == 3:
        continue
    print(i)
# Output: 0 1 2 4
```

Example: pass

```
for i in range(5):
    pass   # placeholder
print("Loop executed")
# Output: Loop executed
```

Example: else with loop

```
for i in range(3):
    print(i)
else:
    print("Loop finished without break")
# Output: 0 1 2 Loop finished without break
```

Example Python Programs

Q1. Basic if

```
a = 6
if a > 5:
   print("a is bigger than 5")
```

Output:
a is bigger than 5

Explanation: Condition is True, so message prints.

Q2. if...else

```
balance = 200
withdraw = 250
if withdraw <= balance:
   print("Transaction Successful")
else:
   print("Insufficient Funds")
```

Output:
Insufficient Funds

Explanation: 250 ≤ 200 is False → else part runs.

Q3. if...elif...else (Grading)

```
marks = 88

if marks >= 95:
   print("Grade: A")
elif marks >= 80:
   print("Grade: B")
elif marks >= 70:
   print("Grade: C")
else:
   print("Grade: D")
```

Output:
Grade: B

Explanation: 88 ≥ 80 → prints **B**; lower checks are skipped.

Q4. for with range(stop)

```
for i in range(4):
   print("Number:", i)
```

Output:
Number: 0
Number: 1
Number: 2
Number: 3

Explanation: range(4) gives 0,1,2,3.

Q5. for over a list

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

Output:

```
I like apple
I like banana
I like cherry
```

Explanation: Loops through each list item.

Q6. for over a string (count vowels)

```
text = "Python is easy"
vowels = "aeiouAEIOU"
count = 0

for ch in text:
   if ch in vowels:
      count += 1

print("Number of vowels:", count)
```

Output:

```
Number of vowels: 4
```

Explanation: Adds 1 for each vowel found.

Q7. for with range(start, stop, step)

```
for n in range(1, 10, 2):
   print(n)
```

Output:

```
1
3
5
7
9
```

Explanation: Starts at 1, steps by 2, stops before 10.

Q8. Countdown with negative step

```
for i in range(5, 0, -1):
   print(i)
```

Output:

```
5
4
3
```

```
2
1
```

Explanation: Counts down from 5 to 1.

Q9. while loop (count 1→5)

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

Output:

```
1
2
3
4
5
```

Explanation: Repeats while condition is True.

Q10. while loop (reverse countdown)

```
count = 5
while count > 0:
   print(count)
   count -= 1
```

Output:

```
5
4
3
2
1
```

Explanation: Decreases until condition becomes False.

Q11. continue (skip evens)

```
for num in range(1, 10):
   if num % 2 == 0:
      continue
   print(num)
```

Output:

```
1
3
5
7
9
```

Explanation: Skips printing when number is even.

Q12. break (stop at first multiple of 5)

```
numbers = [7, 13, 21, 25, 40]
for n in numbers:
   if n % 5 == 0:
      print("First multiple of 5 is:", n)
      break
```

Output:

First multiple of 5 is: 25

Explanation: Exits loop as soon as a match is found.

Q13. pass (placeholder)

```
for i in range(3):
   pass
print("Loop executed successfully.")
```

Output:

Loop executed successfully.

Explanation: pass does nothing; keeps code valid.

Q14. for ... else (search with "not found")

```
numbers = [2, 4, 6, 8, 10]
target = 5

for n in numbers:
   if n == target:
      print("Found:", target)
      break
else:
   print(target, "not found.")
```

Output:

5 not found.

Explanation: Loop ends without break → else runs.

Q15. while ... else (finishes normally)

```
x = 0
while x < 3:
   print("x =", x)
   x += 1
else:
   print("Loop finished without break.")
```

Output:

```
x = 0
x = 1
x = 2
Loop finished without break.
```

Explanation: Condition becomes False → else executes.

Q16. Multiplication table (for loop)

```
number = 7
for count in range(1, 11):
   print(number, '*', count, '=', number * count)
```

Output:

```
7 * 1 = 7
7 * 2 = 14
7 * 3 = 21
7 * 4 = 28
7 * 5 = 35
7 * 6 = 42
7 * 7 = 49
7 * 8 = 56
7 * 9 = 63
7 * 10 = 70
```

Explanation: Multiplies 7 by 1..10.

Q17. Right-angled triangle (nested loops)

```
rows = 5
for i in range(1, rows + 1):
   for j in range(i):
      print("*", end=" ")
   print()
```

Output:

```
*
* *
* * *
* * * *
* * * * *
```

Explanation: Row i prints i stars.

Q18. Dictionary loop (sum of marks)

```
marks = {"Math": 85, "Science": 90, "English": 80}
total = 0
for subject in marks:
   total += marks[subject]
print("Total Marks:", total)
```

Output:

```
Total Marks: 255
```

Explanation: Adds each value from the dictionary.

Q19. Skip spaces while printing string

```
sentence = "Hello World"
for c in sentence:
```

```
  if c == " ":
     continue
  print(c, end="")
```

Output:
HelloWorld

Explanation: continue skips spaces.

Q20. Nested loops (tiny table 1–2 × 1–3)

```
for n in range(1, 3):
  for v in range(1, 4):
     print(n, "*", v, "=", n * v)
```

Output:
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6

Explanation: Outer loop chooses number; inner loop multiplies.