Seed Programming

Python Fundamentals

Lecture 02

Introduction

In Lecture 1, we learned how to display output, take input, and use variables. In this lecture, we will learn how to make programs repeat tasks automatically using loops. We will also learn how to print different patterns using nested loops.

1 Using the `sep` Parameter

The `sep` parameter of the `print()` function changes the separator placed between multiple values. By default, Python separates values with a single space.

1.1 Default and Custom Separators

```
print("Ali", 20, "Lahore")
print("Ali", 20, "Lahore", sep="-")
```

Output

```
Ali 20 Lahore
Ali-20-Lahore
```

More than three values can be formatted in the same way:

```
print(1, 2, 3, 4, 5, sep=",")
print("A", "B", "C", sep="*")
```

Output

```
1,2,3,4,5
A*B*C
```

The `sep` parameter is useful for neat formatting, CSV-like data, and custom output styles.

2 The `while` Loop

A `while` loop repeats a block of code for as long as its condition remains `True`. It avoids writing the same statement many times.

2.1 Syntax and Example

The general syntax is:

```
while condition:
    statements
```

The indented statements form the body of the loop.

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

Output

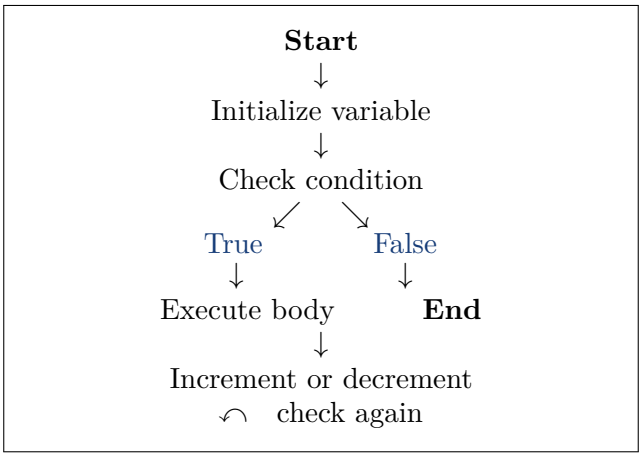
```
1
2
3
4
5
```

Without a loop, printing `"Hello"` five times requires five separate statements. The same result can be produced more clearly as follows:

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

3 Flow of a `while` Loop

A `while` loop initializes its control variable, checks a condition, executes the body when the condition is true, updates the variable, and then checks the condition again. Execution leaves the loop when the condition becomes false.



Consider this loop:

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

It prints `1`, `2`, and `3`. After the third iteration, `count` becomes `4`; the condition `4 <= 3` is false, so the loop stops.

4 Dry Run

A *dry run* means tracing a program manually to understand how its values and conditions change.

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

Iteration	Count before	Condition	Output	Count after
1	1	True	1	2
2	2	True	2	3
3	3	True	3	4
4	4	False	Loop ends	—

5 Using Multiple Variables

A program can store and use several values at the same time.

```
name = "Ali"
age = 20
city = "Lahore"

print(name)
print(age)
print(city)
```

Output

```
Ali
20
Lahore
```

6 Conditions

Conditions allow a program to make decisions. They are also used to decide whether a `while` loop should continue.

Operator	Meaning
`==`	Equal to
`!=`	Not equal to
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to

```
age = 18
if age >= 18:
    print("You can vote.")
```

7 Counters

A *counter* is a variable that keeps track of how many times a loop runs. In the following example, `count` is the counter:

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

Its initial value is `1`, and it increases after every iteration until the loop's condition becomes false.

8 Increment

Increment means increasing a variable's value. The statement `count += 1` is a shorter form of `count = count + 1`.

```
1 number = 10
2 number += 1
3 print(number)
```

Output

```
11
```

9 Decrement

Decrement means decreasing a variable's value. The statement `count -= 1` is a shorter form of `count = count - 1`.

```
1 number = 10
2 number -= 1
3 print(number)
```

Output

```
9
```

10 Infinite Loops

An *infinite loop* never ends because its condition never becomes false.

```
1 while True:
2     print("Hello")
```

Output

```
Hello
Hello
Hello
...
```

An infinite loop may occur because the condition is always true, the counter is not updated, or the program has no valid stopping condition.

Remark. *An intentional infinite loop can be useful in interactive or continuously running programs, but it must normally contain a controlled way to stop.*

11 Nested `while` Loops

A *nested while loop* is a `while` loop inside another `while` loop. Nested loops are useful for patterns, tables, rows and columns, and grids. The outer loop usually controls the rows, while the inner loop controls the columns.

11.1 Rectangle Example

```
row = 1
while row <= 3:
    column = 1
    while column <= 5:
        print("*", end=" ")
        column += 1
    print()
    row += 1
```

Output

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

The outer loop runs three times, once for each row. During every outer-loop iteration, the inner loop prints five stars. The empty `print()` call then moves the cursor to the next line.

12 Pattern-Printing Logic

Pattern printing is an effective way to understand nested loops. First count the rows, then use the outer loop for rows and the inner loop for columns or symbols. Print a new line after completing each row.

$$\text{Outer loop} \longrightarrow \text{rows}, \qquad \text{inner loop} \longrightarrow \text{columns or symbols}.$$

12.1 Pattern 1: Rectangle

```
row = 1
while row <= 3:
    col = 1
    while col <= 5:
        print("*", end=" ")
        col += 1
    print()
    row += 1
```

Output

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

12.2 Pattern 2: Square

```
row = 1
while row <= 4:
    col = 1
    while col <= 4:
        print("#", end=" ")
        col += 1
    print()
    row += 1
```

Output

```
# # # #
# # # #
# # # #
# # # #
```

12.3 Pattern 3: Increasing Triangle

```
row = 1
while row <= 5:
    col = 1
    while col <= row:
        print("*", end=" ")
        col += 1
    print()
    row += 1
```

Output

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

Row 1 prints one star, row 2 prints two stars, and the number of stars continues to increase until row 5.

12.4 Pattern 4: Inverted Triangle

```
row = 5
while row >= 1:
    col = 1
    while col <= row:
        print("*", end=" ")
        col += 1
    print()
    row -= 1
```

Output

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