Review of Lecture 3

- **The if Statement**

```
if condition:
    statement
```

```
if number > 0:
    print("Positive number")
```

- **The if-else Statement**

```
if condition:
    statement_if_true
else:
    statement_if_false
```

```
if temperature > 35:
    print("High temperature")
else:
    print("Temperature is acceptable")
```

- **The if-elif-else Statement**

```
if condition1:
    statement1
elif condition2:
    statement2
else:
    statement3
```

```
score = 13

if score >= 15:
    print("Excellent")
elif score >= 10:
    print("Good")
else:
    print("Weak")
```

- **Logical operators:** `and, or, not`
- **Nested conditional statements**

```
if temperature <= 30:
    if pressure <= 100:
        print("Conditions are acceptable")
```

Info 3

Introduction to Python

M. Bouzenita
2nd year Engineer - University of Jijel

Lecture 4
Loops and Iteration

1. Introduction

Consider the following program:

Display five multiples of 5

```
print(5)
print(10)
print(15)
print(20)
print(25)
```

Display 100 multiples of 5

```
print(5)
print(10)
print(15)
print(20)
print(25)
print(30)
print(35)
print(40)
print(45)
print(50)
print(55)
print(60)
print(65)
print(70)
```

Python provides a more efficient solution: **Loop**

```
for number in range(1, 101):
    print(number * 5)
```

2. Loops

A loop is a programming structure
that **repeats** a block of instructions.

Loops are useful when:

1. An operation must be **repeated a fixed number of times**.
2. An operation must **continue until a condition is satisfied**.
3. A program must **process a sequence of values**.
4. A calculation must be **performed repeatedly using different inputs**.

3. The `while` loop

A **`while`** loop **repeats** a block of instructions as long as a specified **condition is true**.

```
while condition:
    statements
```

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

```
1
2
3
4
5
```

4. Counter-controlled loops

A **counter-controlled** loop repeats a block of instructions **a known number** of times.

```
cycle = 1

while cycle <= 5:
    print("Test cycle:", cycle)
    cycle = cycle + 1
```

```
Test cycle: 1
Test cycle: 2
Test cycle: 3
Test cycle: 4
Test cycle: 5
```

However, Python also provides a convenient alternative for this situation: the **`for`** loop.

5. Infinite loops

Consider this example:

```
number = 1

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

```
1
1
1
1
1
1
1
1
1
1
1
```

An **infinite loop** is a loop that **does not terminate** because its continuation condition remains true indefinitely

This is a common beginner mistake and the corrected program is:

```
number = 1

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

6. The `for` loop

In general, we use the **for** loop to **repeat a set of statements with a known number** of iterations.

```
for variable in sequence:
    statements
```

The loop `variable` receives an item from the `sequence` during **each iteration**.

The sequence may be **generated** using `range()` or another iterable object.

7. The `range()` function

The **`range()`** function **generates a sequence of integers** and it is frequently used with **for** loops.

`range(stop)`	`x in range(3)` ➡ `x ∈{0, 1, 2}`
`range(start, stop)`	`x in range(2,5)` ➡ `x ∈{2, 3, 4}`
`range(start, stop, step)`	`x in range(1,6,2)` ➡ `x ∈{1, 3, 5}`

7. The `range()` function

The **`range()`** function **generates a sequence of integers** and it is frequently used with **for** loops.

```
range(stop)

range(start, stop)

range(start, stop, step)
```

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

```
0
1
2
3
4
```

7. The `range()` function

The **`range()`** function **generates a sequence of integers** and it is frequently used with for loops.

```
range(stop)

range(start, stop)

range(start, stop, step)
```

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

```
2
3
4
```

7. The `range()` function

The **`range()`** function **generates a sequence of integers** and it is frequently used with for loops.

```
range(stop)

range(start, stop)

range(start, stop, step)
```

```
for number in range(0, 11, 2):
    print(number)
```

```
0
2
4
6
8
10
```

7. The `range()` function

The **`range()`** function **generates a sequence of integers** and it is frequently used with for loops.

```
range(stop)

range(start, stop)

range(start, stop, step)
```

A **negative step** can be used to generate decreasing values:

```
for number in range(10, 0, -3):
    print(number)
```

```
10
7
4
1
```

8. Nested loops

A nested loop is a loop **placed inside another loop**.

Nested loops are **useful** when a problem **involves two levels of repetition**.

```
for row in range(1, 3):
    for column in range(1, 4):
        print("Row:",row, "Column:",column)
```

```
Row: 1 Column: 1
Row: 1 Column: 2
Row: 1 Column: 3
Row: 2 Column: 1
Row: 2 Column: 2
Row: 2 Column: 3
```

9. The `break` statement

In general, we use the **`break`** statement **to jump out of** the loop.

```
for number in range(1, 10):
    print(number)
    if number == 5:
        break
```

```
1
2
3
4
5
```

When `number` becomes **5**, the **`break`** statement terminates the loop.

The number **5** is printed because the condition is checked after the `print()` statement.

10. The `continue` statement

The **`continue`** statement **skips the remaining instructions** in the current iteration and proceeds to the next iteration.

For example, suppose we want to display the numbers from 1 to 5 except 3.

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

```
1
2
4
5
```

10. Comparing `while` and `for`

`while`

```
number = 1

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



`for`

```
for number in range(1, 6):
    print(number)
```

Rule of thumb:

- Use `for` when iterating over **a known range or sequence**.
- Use `while` when repetition must continue **until a condition changes**.

11. Common beginner mistakes

1. Forgetting to update a while loop counter

Incorrect:

```
number = 1

while number <= 5:
    print(number)
```

Correct:

```
number = 1

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

The value **never changes**, so the condition remains true.

11. Common beginner mistakes

2. Using the wrong stop value in `range()`

Consider:

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

Output:

```
1
2
3
4
```

The number 5 is not included. Always remember that the stop value is excluded.

To display the integers from 1 to 5, we write:

```
for number in range(1, 6):
    print(number)
```

11. Common beginner mistakes

3. Incorrect Indentation

Incorrect:

```
for number in range(1, 4):
print(number)
```

Correct:

```
for number in range(1, 4):
    print(number)
```

11. Common beginner mistakes

4. Initializing an accumulator incorrectly

Incorrect:

```
total = 10

for number in range(1, 4):
    total = total + number

print(total)
```

The program displays 16 because the initial value is 10.

If the purpose is to calculate the sum of 1, 2, and 3, the accumulator should begin at zero.

Correct:

```
total = 0

for number in range(1, 4):
    total = total + number

print(total)
```

Output:

```
6
```

11. Common beginner mistakes

5. Confusing break and continue

The **`break`** statement terminates the nearest enclosing loop.
The **`continue`** statement skips the remaining statements in the current iteration and proceeds to the next iteration.

For example:

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

    print(number)
```

Output:

```
1
2
4
5
```

Using break

```
for number in range(1, 6):
    if number == 3:
        break

    print(number)
```

Output:

```
1
2
```

11. Common beginner mistakes

6. Incorrect indentation in nested loops

Consider the following program:

```
for row in range(1, 4):
    for column in range(1, 3):
        print(row, column)
```

The `print()` statement belongs to the inner loop. Therefore, it executes for every column in every row. If the indentation changes, the program's behavior may change as well. When using nested loops, carefully identify which loop contains each statement.

Practice