**Assignment2:-**

Q1.

The if-elif-else construct is used to control the flow of your program by evaluating conditions in sequence:

if condition1:

# Code block runs if condition1 is True

elif condition2:

# Runs if condition1 is False and condition2 is True

else:

# Runs if none of the above conditions are True

Flow of Control

* Python evaluates conditions top to bottom.
* As soon as one condition is True, that block runs.
* Remaining elif or else blocks are skipped.

Example: Check Number Sign

def check\_number\_sign(num):

if num > 0:

return "Positive"

elif num < 0:

return "Negative"

else:

return "Zero"

Sample Output

print(check\_number\_sign(10)) # Positive

print(check\_number\_sign(-3)) # Negative

print(check\_number\_sign(0)) # Zero

Q2.

**Key Differences**

| **Feature** | **for loop** | **while loop** |
| --- | --- | --- |
| **Purpose** | Loop over **known range or sequence** | Loop while a **condition is True** |
| **Use Case** | Iteration through collections, ranges | Indefinite loops, wait for a condition |
| **Control** | Count-controlled | Condition-controlled |
| **Typical Syntax** | for item in iterable: | while condition: |

def is\_prime(n):

if n <= 1:

return False

for i in range(2, int(n\*\*0.5) + 1):

if n % i == 0:

return False

return True

def print\_primes\_upto\_100():

for num in range(1, 101):

if is\_prime(num):

print(num, end=" ")

print\_primes\_upto\_100()

**Output:**

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Q3.

A **nested if** statement is an if statement **inside another if or else block**. It lets you **check multiple levels of conditions** when decisions depend on earlier checks.

**🧠 Syntax Example:**

if condition1:

if condition2:

# Run this if both condition1 and condition2 are True

else:

# Runs if condition1 is True but condition2 is False

else:

# Runs if condition1 is False

**Problem: Deep Nesting**

Nested if statements can become **hard to read**, like this:

if a > 0:

if b > 0:

if c > 0:

print("All positive")

**Coding Challenge: Find the Largest of Three Numbers Using Nested if Statements**

def find\_largest(a, b, c):

if a >= b:

if a >= c:

return a

else:

return c

else:

if b >= c:

return b

else:

return c

**Example:**

print(find\_largest(5, 8, 3)) # 8

print(find\_largest(12, 12, 10)) # 12

print(find\_largest(-1, -5, -3)) # -1

Q4.

**Effects of break**

* Immediately exits the **nearest enclosing loop** (for or while).
* Code **after the loop continues**, skipping the rest of the current iteration or any future ones.

**Alternatives to break**

| **Alternative** | **When to Use** |
| --- | --- |
| while with condition | When loop termination condition is known early |
| Use flag variable | To signal and exit cleanly |
| return (in functions) | To exit entirely if within a function |

**Coding Challenge**

**Problem:**

Read numbers until a **negative number** is entered.  
Print the **sum of all positive numbers** entered.

**Using break:**

def sum\_positive\_numbers():

total = 0

while True:

num = int(input("Enter a number (negative to stop): "))

if num < 0:

break

total += num

print("Sum of positive numbers:", total)

# Run the function

# sum\_positive\_numbers()

**Alternative (No break):**

def sum\_positive\_numbers\_no\_break():

total = 0

num = 0

while num >= 0:

num = int(input("Enter a number (negative to stop): "))

if num >= 0:

total += num

print("Sum of positive numbers:", total)

**Example Input/Output**

Enter a number (negative to stop): 5

Enter a number (negative to stop): 10

Enter a number (negative to stop): -2

Sum of positive numbers: 15

Q5.

**Syntax:**

for item in iterable:

if condition:

break

else:

# This runs only if loop wasn’t broken

**Key Differences from if...else:**

| **if...else** | **for/while...else** |
| --- | --- |
| Executes else if condition is False | Executes else if loop **completes fully** (no break) |
| Used for **decision making** | Used for **search failure handling, retries, etc.** |

**Use Case: Searching for an element in a list**

If item is found → break.  
If not found after looping → else runs.

**Coding Challenge: Find Index Using for Loop and else**

def find\_element\_index(lst, target):

for index, value in enumerate(lst):

if value == target:

return index

else:

return -1

Q6.

Floating-point numbers in Python (and most programming languages) are represented **imprecisely** due to binary rounding errors.  
This makes **direct comparisons (==) unreliable** in many cases.

**Example of the Problem**

python

CopyEdit

x = 0.1 + 0.2

print(x == 0.3) # ❌ False

print(x) # 0.30000000000000004

This fails because floating-point math can’t represent some decimal values **exactly**, leading to small precision errors.

**How to Avoid This Pitfall**

Use **tolerance-based comparison** — check if two values are **"close enough"** instead of exactly equal.

**Coding Challenge: Approximate Float Comparison Function**

def approx\_equal(a, b, tol=1e-9):

return abs(a - b) < tol

**Example Usage**

print(approx\_equal(0.1 + 0.2, 0.3)) # ✅ True

print(approx\_equal(0.123456, 0.1234567)) # ✅ True if within tolerance

print(approx\_equal(1.000001, 1.000002)) # May return True if tol is large enough

Q7.

**General Pattern:**

for item in collection:

if condition(item):

# Do something with item

Given a list of integers, create a new list containing only the **even numbers** using a for loop and if statement.

**Solution:**

def filter\_even\_numbers(numbers):

even\_numbers = []

for num in numbers:

if num % 2 == 0:

even\_numbers.append(num)

return even\_numbers

**Example Usage:**

nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

print(filter\_even\_numbers(nums))

# Output: [2, 4, 6, 8, 10]

Q8.

**Short-circuit evaluation** is a feature in Python where logical expressions are **evaluated from left to right**, and the evaluation **stops as soon as the result is determined**.

**How It Works:**

| **Operator** | **Behavior** |
| --- | --- |
| and | Stops at the **first False** (because the result is already False) |
| or | Stops at the **first True** (because the result is already True) |

**Examples:**

# 'and' stops early if it sees False

False and print("This won’t run") # ➝ Doesn't print

# 'or' stops early if it sees True

True or print("This won’t run either") # ➝ Doesn't print

**Performance Benefits:**

* **Avoids unnecessary computation** (especially useful with function calls or expensive operations).
* Can be used to **safely check attributes** or **prevent errors** (e.g., null checks).

if obj is not None and obj.value > 10:

print("Safe access")

If obj is None, the second part isn't evaluated, preventing an error.

**Coding Challenge:**

Write a function that takes three boolean values and returns True if **at least two of them are True**, using short-circuit logic.

**Solution Using Short-Circuit Logic:**

def at\_least\_two\_true(a, b, c):

return (a and b) or (a and c) or (b and c)

**Example Usage:**

print(at\_least\_two\_true(True, False, True)) # ➝ True

print(at\_least\_two\_true(False, False, True)) # ➝ False

print(at\_least\_two\_true(True, True, True)) # ➝ True

Q9.

The continue statement is used **inside loops** to **skip the rest of the current iteration** and move on to the **next one** immediately.

**How It Works:**

for num in range(5):

if num == 2:

continue

print(num)

Output:

0

1

3

4

def print\_divisible\_by\_3(numbers):

for num in numbers:

if num % 3 != 0:

continue

print(num)

Example Usage:

nums = [1, 3, 4, 6, 7, 9, 10, 12]

print\_divisible\_by\_3(nums)

Output:

3

6

9

12

Q10.

**List comprehensions** are a compact way to create lists in Python.  
You can include **if conditions** inside them to **filter** or **modify** data on the fly.

**Syntax:**

[expression for item in iterable if condition]

This is equivalent to:

result = []

for item in iterable:

if condition:

result.append(expression)

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

even\_numbers = [x for x in numbers if x % 2 == 0]

# ➞ [2, 4]

squares\_of\_evens = [x\*\*2 for x in range(1, 21) if x % 2 == 0]