SOFT40161_Lab02_Solution

October 7, 2024

Nottingham Trent University

Department of Computer Science

1 SOFT40161 - Lab 02: Solution

1.1 Control Structure with Python in Jupyter Notebook

- Welcome to Lab 2! In this session, we will explore one of the core concepts of programming: control structures. These structures allow us to control the flow of execution in a program, making it more flexible and dynamic. By mastering these tools, you'll gain the ability to make decisions within your code, execute tasks repeatedly, and handle different scenarios efficiently.
- In this lab, we will focus on four essential control structures:
  - Sequence: The default mode where code is executed line by line.
  - Branching: Using if-else statements to choose different paths based on conditions.
  - Conditions: Checking whether certain criteria are met, allowing for complex decision-making.
  - Looping: Repeating tasks using for and while loops to handle repetitive tasks.

Throughout the lab, we will work through examples with increasing complexity, including nested if statements, multiple for loops, and understanding infinite loops. You'll also have the opportunity to apply your knowledge through hands-on exercises designed to solidify your understanding.

By the end of this lab, you should feel confident in using control structures to manage how your programs run. These skills will be essential as we move forward to more advanced topics. Let's get started!

1.2 Lab Learning Outcomes

By the end of this lab, you will be able to: - **Understand and apply control structures:** Implement sequence, branching, and looping to control program flow. - **Utilize conditional statements:** Use if, elif, and else statements to handle decision-making in your code. - **Work with looping structures:** Implement for and while loops to automate repetitive tasks. - **Implement nested control structures:** Use nested if statements and multiple for loops for more complex

scenarios. - **Identify and avoid infinite loops:** Understand infinite loops and how to prevent or break them in both for and while loops. - **Write readable and efficient code:** Practice proper code structuring and commenting to enhance readability and maintainability.

These outcomes will help you gain a strong foundation in programming, particularly in controlling the logic and flow of Python programs.

2 Exercise

Exercise 1: Branching with Nested if - Create a program that simulates a simple traffic light system. - Input a color (red, yellow, or green) and output whether cars should "stop", "slow down", or "go". - Extend this by adding another if to check if there are pedestrians present. If yes, even for green light, cars should "wait".

[ ]:
```
# Input for traffic light color
light_color = input("Enter the traffic light color (red, yellow, green): ").
 ↪lower()

# Input to check if pedestrians are present
pedestrians = input("Are there pedestrians present? (yes/no): ").lower()

# Logic for traffic light system with nested if
if light_color == "red":
    print("Cars must stop.")
elif light_color == "yellow":
    print("Cars should slow down.")
elif light_color == "green":
    if pedestrians == "yes":
        print("Cars must wait for pedestrians.")
    else:
        print("Cars can go.")
else:
    print("Invalid color entered. Please enter red, yellow, or green.")
```

Explanation The program asks the user to input the traffic light color and whether pedestrians are present. The program first checks the traffic light color using a series of if-elif statements: - If the light is red, cars must stop. - If the light is yellow, cars should slow down. - If the light is green, it checks whether pedestrians are present. If pedestrians are present, cars must wait, otherwise, they can go.

If the user inputs an invalid traffic light color, the program outputs an error message.

[ ]:
```

```

Exercise 2: Multiple for Loops - Write a program that outputs the multiplication tables from 1 to 5. - Use nested for loops to iterate over both the multiplier and the multiplicand.

[ ]:
```python
# Outer loop for the multiplicand (1 to 5)
for i in range(1, 6):
    print(f"Multiplication Table for {i}:")

    # Inner loop for the multiplier (1 to 10)
    for j in range(1, 11):
        # Print the result of the multiplication
        print(f"{i} x {j} = {i * j}")

        # Alternatively, for the above command, you can write the following-
        #print(i ,"x", j, "=", i * j)

    # Print a blank line to separate the tables
    print()
```

Explanation: - The outer loop (for i in range(1, 6)) iterates through the numbers 1 to 5, each representing the base number for the multiplication table. - The inner loop (for j in range(1, 11)) iterates from 1 to 10, representing the multiplier. - For each iteration of the inner loop, the product of i (multiplicand) and j (multiplier) is printed. - After each table, an empty line is printed to separate the tables for readability.

[ ]:
```python

```

Exercise 3: Grade Classification - Task: Prompt the user to enter a grade (out of 100). Use if-elif-else statements to classify the grade into A, B, C, or Fail. - Hint: Use branching to evaluate the grade range and print the corresponding result.

[ ]:
```python
# Get the grade input from the user
grade = int(input("Enter your grade (out of 100): "))

# Classify the grade using if-elif-else branching
if grade >= 90:
    print("Grade: A")
elif grade >= 75:
    print("Grade: B")
elif grade >= 50:
    print("Grade: C")
else:
    print("Grade: Fail")
```

Explanation:

Input: The program prompts the user to input their grade, which is converted to an integer using int(). Branching:

- If the grade is 90 or above, it classifies as Grade A.
- If the grade is between 75 and 89, it classifies as Grade B.
- If the grade is between 50 and 74, it classifies as Grade C.
- If the grade is below 50, it classifies as a Fail.

Output: Based on the grade input, the corresponding grade classification is printed.

[ ]:

Exercise 4: Sum of Even Numbers - Task: Write a program that calculates the sum of all even numbers between 1 and 100 using a for loop. - Hint: Use the range() function with a step value of 2 to iterate through even numbers.

[ ]:
```
# Initialize a variable to store the sum
sum_even = 0

# Use a for loop to iterate through even numbers from 2 to 100
for number in range(2, 101, 2):
    sum_even += number

# Print the result
print("The sum of even numbers between 1 and 100 is:", sum_even)
```

Explanation:

Initialization: We initialize a variable sum_even to 0, which will hold the running total of the sum of even numbers.

For Loop: - The loop iterates through the even numbers from 2 to 100 using range(2, 101, 2), where 2 is the starting value, 101 is the ending value (exclusive), and 2 is the step value, meaning it skips odd numbers. - In each iteration, the current number is added to the sum_even variable.

Output: Once the loop is done, the program prints the final sum.

[ ]:

Exercise 5: Bank Withdrawal Simulation - Task: Simulate a simple ATM withdrawal system using a while loop. The user starts with a balance of £1000. The loop should continue to prompt the user for an amount to withdraw, and after each withdrawal, display the remaining balance. Stop the loop when the balance becomes zero or less. - Hint: Make sure the loop exits when the balance is depleted, and handle cases where the withdrawal amount is greater than the available balance.

[ ]:
```
# Initial balance
balance = 1000

# Start the ATM withdrawal loop
while balance > 0:
    # Prompt the user for the withdrawal amount
    withdrawal = float(input(f"Your current balance is ${balance:.2f}. Enter␣
 ↪the amount to withdraw: $"))

    # Check if the withdrawal amount is greater than the available balance
    if withdrawal > balance:
```

```
        print(f"Insufficient balance! You can only withdraw up to ${balance:.
↪2f}.")
    else:
        # Deduct the withdrawal amount from the balance
        balance -= withdrawal
        print(f"Withdrawal successful! Your remaining balance is ${balance:.2f}.
↪")

# Exit message when balance is zero or less
print("Your balance is now zero. No further withdrawals can be made.")
```

Explanation:

Initial Balance: The user starts with a balance of $1000.

While Loop:

The loop continues as long as the balance is greater than 0. - Inside the loop, the program prompts the user to input a withdrawal amount. - Checking the Withdrawal Amount: - If the entered withdrawal amount is greater than the current balance, the program prints a message indicating insufficient funds. - Otherwise, the program deducts the withdrawal amount from the balance and displays the remaining balance.

Loop Termination: The loop will stop when the balance becomes zero or less, and a final message will be printed to indicate that no more withdrawals can be made.

[ ]: