Python Class 11 Series

Flow of Control Chapter: 6

Saroj Kumar Jha

Saroj Codes

Please subscribe

INTRODUCTION

Flow of control in Python refers to the order in which statements and expressions are executed in a Python program. Python follows a specific flow of control, which is determined by various control structures and conditional statements. Understanding the flow of control is essential for writing efficient and logical programs.

The primary flow of control structures in Python includes:

Sequence: The default flow of control in Python is sequential. Statements are executed one after another in the order they appear in the code.

Conditional Statements: Conditional statements allow the program to make decisions based on certain conditions.

Loops: Loops allow you to execute a block of code repeatedly until a certain condition is met.

SELECTION

In computer programming, "selection" typically refers to the process of making decisions based on certain conditions. In Python, selection is primarily achieved using conditional statements like the "if," "elif," and "else" statements. These conditional statements allow the program to execute different blocks of code based on whether certain conditions are true or false.

if statement: The "if" statement is used to execute a block of code if a given condition is true. If the condition is false, the block is skipped.

Example:

```
x = 10
if x > 0:
    print("x is positive")
```

SELECTION

if-else statement: The "if-else" statement allows the program to execute one block of code if the condition is true, and another block if the condition is false.

Example:

```
x = -5
if x > 0:
    print("x is positive")
else:
    print("x is non-positive")
```

if-elif-else statement: The "if-elif-else" statement is used when there are multiple conditions to check. It allows the program to test multiple conditions one by one and execute the block of code associated with the first true condition. If no condition is true, the "else" block (optional) will be executed.

SELECTION

Example:

```
score = 85
if score >= 90:
    print("A")
elif score >= 80:
    print("B")
elif score >= 70:
    print("C")
else:
    print("Below C")
```

Nested if statements: You can also nest one or more "if" statements inside another to create complex decision-making structures.

SELECTION

Example:

```
x = 10
if x > 0:
    if x % 2 == 0:
        print("x is a positive even number.")
    else:
        print("x is a positive odd number.")
else:
    print("x is non-positive.")
```

Using selection constructs like "if" statements allows you to create programs that can dynamically adapt to different scenarios and make informed decisions based on input or calculations. This is essential for writing flexible and robust code.

INDENTATION

In Python, "indentation" refers to the use of whitespace at the beginning of lines to define the structure and hierarchy of code blocks. Unlike many other programming languages that use braces or other symbols to indicate code blocks, Python uses indentation as part of its syntax.

The most common use of indentation is in defining the blocks of code within control structures such as loops, conditional statements, and functions. Proper indentation is crucial in Python because it determines which statements belong to which block and how the flow of control should be followed.

Let's look at some examples to understand how indentation is used:

Indentation in if statement:

```
x = 10
if x > 0:
    print("x is positive")
    print("This statement is inside the if block")
print("This statement is outside the if block")
```

INDENTATION

In this example, the indented lines after the "if" statement form the block of code that will be executed only if the condition (x > 0) is true. The last print statement is outside the "if" block and will be executed regardless of the condition's result.

Indentation in loops:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    print("Number:", num)
    print("This statement is inside the loop")
print("This statement is outside the loop")
```

In this case, the indented lines after the "for" statement constitute the block of code that will be executed for each item in the list numbers. The last print statement is outside the loop and will be executed after the loop is completed.

INDENTATION

Indentation in functions:

```
def greet(name):
    print("Hello, " + name + "!")
    print("Welcome to the function.")
print("This statement is outside the function")

greet("John")
```

In this example, the indented lines under the "def" statement form the body of the greet function. When the function is called with greet("John"), the indented code inside the function is executed, and then the control returns to the outer code.

INDENTATION

It's important to note that Python relies on consistent and correct indentation to determine the structure of the code. Mixing different indentation styles or forgetting to indent properly can lead to syntax errors or incorrect program behavior.

The recommended indentation style in Python is to use four spaces per indentation level, although you can also use tabs or a mix of spaces and tabs as long as you're consistent. Using a good code editor or IDE can help you automatically manage indentation and avoid indentation-related errors.

REPETITION

In programming, "repetition" refers to the process of executing a block of code multiple times. It allows you to automate tasks that need to be performed repeatedly without writing the same code over and over again. Python provides two primary loop structures for implementing repetition:

while loop: The while loop executes a block of code as long as a given condition is true. It continuously checks the condition before each iteration.

Syntax:

```
while condition:
    # Code block to be repeated
```

Example:

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

REPETITION

Output:

Count: 1

Count: 2

Count: 3

Count: 4

Count: 5

for loop: The for loop iterates over a sequence (e.g., list, tuple, string) and executes a block of code for each item in the sequence.

Syntax:

```
for item in sequence:
    # Code block to be repeated for each item
```

Example:

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

REPETITION

Output:

apple

banana

orange

The range() function is often used in conjunction with for loops to generate a sequence of numbers within a specified range.

Example:

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

Output:

1

2

3

4

5

REPETITION

In addition to loops, Python also provides the "**break**" and "**continue**" statements that allow you to modify the flow of repetition:

break: It terminates the loop prematurely when a certain condition is met.
continue: It skips the current iteration and moves to the next one.
Example using break:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 3:
        break
    print(num)
```

Output:

1
2

REPETITION

Example using continue:

```
numbers = [1, 2, 3, 4, 5]
for num in numbers:
    if num == 2:
        continue
    print(num)
```

Output:

```
1
3
4
5
```

Repetition is a fundamental concept in programming, and using loops efficiently can significantly improve code readability and maintainability while reducing redundancy in your programs.

REPETITION

the range

In Python, the range() function is a built-in function that generates a sequence of numbers within a specified range. It is commonly used in conjunction with loops, particularly with the for loop, to iterate over a sequence of numbers.

The range() function has three possible arguments:

start (optional): The starting value of the sequence. If not provided, it defaults to 0.

stop: The stopping value of the sequence. The range() function generates numbers up to, but not including, this value.

step (optional): The step or increment between each number in the sequence. If not provided, it defaults to 1.

Syntax:

range(stop)

range(start, stop)

range(start, stop, step)

Let's see some examples of how the range() function is used:

REPETITION

Example 1: Using range() with one argument (stop):

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

Output:

```
0
1
2
3
4
```

Example 2: Using range() with two arguments (start and stop):

```
for i in range(2, 6):
    print(i)
```

Output:

```
2
3
4
5
```

REPETITION

Example 3: Using range() with three arguments (start, stop, and step):

```
for i in range(1, 6, 2):
    print(i)
```

Output:

1

3

5

Example 4: Creating a list using range() and list():

```
my_list = list(range(5, 15, 3))
print(my_list)
```

Output:

[5, 8, 11, 14]

The range() function is especially useful when you need to perform a specific action a predetermined number of times or when you want to create sequences of numbers for various purposes. Remember that the stop value is not included in the sequence generated by range(), so if you want to include the stop value, you can adjust your loop's logic accordingly.

REPETITION

NESTED LOOPS

In Python, "nested loops" refer to the concept of having one loop inside another loop. This allows you to perform repeated actions in a hierarchical manner, creating more complex iterations and patterns. Nested loops are commonly used to work with 2D data structures (e.g., matrices or grids) or to handle combinations of elements from multiple lists.

The general syntax of a nested loop in Python is as follows:

```
for outer_loop_var in outer_sequence:
    # Outer loop code block

    for inner_loop_var in inner_sequence:
        # Inner loop code block
```

Each time the outer loop executes, it triggers the inner loop to run from start to finish. After the inner loop completes, the control returns to the outer loop, which then proceeds to its next iteration and triggers the inner loop again. This process continues until the outer loop finishes all its iterations.

REPETITION

Let's look at some examples of nested loops:

Example 1: Nested loops to print a simple pattern.

```
for i in range(1, 4):
    for j in range(1, 4):
        print(i, j)
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

REPETITION

Example 2: Nested loops to create a multiplication table.

```
for i in range(1, 6):
    for j in range(1, 11):
        print(i, "*", j, "=", i * j)
```

Output:

```
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
...
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

REPETITION

Example 3: Nested loops to iterate through a 2D list (matrix).

```
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
for row in matrix:
    for item in row:
        print(item, end=" ")
    print()  # To move to the next line after each row
```

Output:

```
1 2 3
4 5 6
7 8 9
```

Nested loops are powerful constructs that allow you to handle complex data structures and perform repetitive tasks in various ways. However, be cautious with deeply nested loops, as they can lead to performance issues for large data sets. Always try to optimize your code when working with nested loops to improve efficiency.

EXERCISE

Q. Write a program that takes the name and age of the user as input and displays a message whether the user is eligible to apply for a driving license or not. (the eligible age is 18 years).

Ans

```
def main():
    try:
        name = input("Enter your name: ")
        age = int(input("Enter your age: "))

        if age >= 18:
            print(f"Hello, {name}! You are eligible to apply for a driving license.")
        else:
            print(f"Sorry, {name}. You are not eligible to apply for a driving license yet.")
    except ValueError:
        print("Invalid input. Please enter a valid age (a whole number).")


if __name__ == "__main__":
    main()
```

EXERCISE

Q. Write a function to print the table of a given number. The number has to be entered by the user

Ans

```
def print_multiplication_table():
    try:
        number = int(input("Enter a number to generate its multiplication table: "))

        print(f"Multiplication table for {number}:")
        for i in range(1, 11):
            result = number * i
            print(f"{number} * {i} = {result}")
    except ValueError:
        print("Invalid input. Please enter a valid number.")


print_multiplication_table()
```

EXERCISE

Q. Write a program that prints minimum and maximum of five numbers entered by the user

Ans

```
def get_min_max():
    try:
        numbers = []
        for i in range(1, 6):
            num = float(input(f"Enter number {i}: "))
            numbers.append(num)

        minimum = min(numbers)
        maximum = max(numbers)

        print(f"Minimum: {minimum}")
        print(f"Maximum: {maximum}")
    except ValueError:
        print("Invalid input. Please enter valid numbers.")


get_min_max()
```

EXERCISE

Q. Write a program to check if the year entered by the user is a leap year or not

Ans

```
def is_leap_year(year):
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        return True
    else:
        return False
def main():
    try:
        year = int(input("Enter a year: "))
        if is_leap_year(year):
            print(f"{year} is a leap year.")
        else:
            print(f"{year} is not a leap year.")
    except ValueError:
        print("Invalid input. Please enter a valid year (a whole number).")

if __name__ == "__main__":
    main()
```

EXERCISE

Q. Write a program to generate the sequence: –5, 10, –15, 20, –25..... upto n, where n is an integer input by the user

Ans

```
def generate_sequence(n):
    sequence = []
    for i in range(1, n + 1):
        if i % 2 == 0:
            sequence.append(i * 5)
        else:
            sequence.append(-i * 5)
    return sequence
def main():
    try:
        n = int(input("Enter the value of n: "))
        if n < 1:
            print("Invalid input. Please enter a positive integer.")
        else:
            result = generate_sequence(n)
            print("Generated Sequence:", result)
    except ValueError:
        print("Invalid input. Please enter a valid integer.")
if __name__ == "__main__":
    main()
```

EXERCISE

Q. Write a program to find the sum of 1+ 1/8 + 1/27.....1/n3, where n is the number input by the user

Ans

```
def calculate_series_sum(n):
    if n < 1:
        return None
    sum_result = 0
    for i in range(1, n + 1):
        sum_result += 1 / (i ** 3)
    return sum_result

def main():
    try:
        n = int(input("Enter the value of n: "))
        series_sum = calculate_series_sum(n)

        if series_sum is not None:
            print(f"The sum of the series is: {series_sum:.4f}")
        else:
            print("Invalid input. Please enter a positive integer.")
    except ValueError:
        print("Invalid input. Please enter a valid integer.")
if __name__ == "__main__":
    main()
```

EXERCISE

Q. Write a program to find the sum of digits of an integer number, input by the user

Ans

```
def calculate_sum_of_digits(number):
    # Convert the number to a string to iterate through its digits
    number_str = str(number)
    sum_of_digits = 0
    for digit in number_str:
        sum_of_digits += int(digit)
    return sum_of_digits
def main():
    try:
        number = int(input("Enter an integer number: "))
        sum_of_digits = calculate_sum_of_digits(number)
        print(f"The sum of the digits of {number} is: {sum_of_digits}")
    except ValueError:
        print("Invalid input. Please enter a valid integer number.")

if __name__ == "__main__":
    main()
```

EXERCISE

Q. Write a function that checks whether an input number is a palindrome or not. [Note: A number or a string is called palindrome if it appears same when written in reverse order also. For example, 12321 is a palindrome while 123421 is not a palindrome]

Ans

```
def is_palindrome(number):
    # Convert the number to a string for easy comparison
    number_str = str(number)
    # Compare the original number with its reverse
    return number_str == number_str[::-1]
def main():
    try:
        number = int(input("Enter a number: "))
        if is_palindrome(number):
            print(f"{number} is a palindrome.")
        else:
            print(f"{number} is not a palindrome.")
    except ValueError:
        print("Invalid input. Please enter a valid integer number.")
if __name__ == "__main__":
    main()
```

EXERCISE

Q. 9. Write a program to print the following patterns:

i)

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

ii)

```
        1
      2 1 2
    3 2 1 2 3
  4 3 2 1 2 3 4
5 4 3 2 1 2 3 4 5
```

iii)

```
1 2 3 4 5
  1 2 3 4
    1 2 3
      1 2
        1
```

EXERCISE

iv)

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

Ans

```
i) def pattern_i(n):
    for i in range(1, n + 1):
        print(" " * (n - i) + "* " * i)

    for i in range(n - 1, 0, -1):
        print(" " * (n - i) + "* " * i)

pattern_i(3)
```

EXERCISE

ii)
```
def print_pattern(n):
    for i in range(1, n + 1):
        # Print spaces before the numbers
        print(" " * (2 * (n - i)), end="")

        # Print numbers in ascending order
        for j in range(i, 0, -1):
            print(j, end=" ")

        # Print numbers in descending order (excluding the last one)
        for j in range(2, i + 1):
            print(j, end=" ")

        # Move to the next line after each row
        print()

print_pattern(5)
```

EXERCISE

iii)
```
def print_pattern(n):
    for i in range(n, 0, -1):
        # Print spaces before the numbers
        print(" " * (2 * (n - i)), end="")

        # Print numbers in ascending order
        for j in range(1, i + 1):
            print(j, end=" ")

        # Move to the next line after each row
        print()

print_pattern(5)
```

EXERCISE

iii)
```
def pattern_iv(n):
    for i in range(n):
        print(" " * (n - i - 1) + "*" + " " * (2 * i) + "*" * (i != 0))

    for i in range(n - 2, -1, -1):
        print(" " * (n - i - 1) + "*" + " " * (2 * i) + "*" * (i != 0))

pattern_iv(3)
```

EXERCISE

Q. Write a program to find the grade of a student when grades are allocated as given in the table below. Percentage of Marks Grade

Above 90% A

80% to 90% B

70% to 80% C

60% to 70% D

Below 60% E

Percentage of the marks obtained by the student is input to the program.

Ans

```
def find_grade(percentage):
    if percentage > 90:
        return "A"
    elif 80 <= percentage <= 90:
        return "B"
    elif 70 <= percentage < 80:
        return "C"
    elif 60 <= percentage < 70:
        return "D"
    else:
        return "E"
```

EXERCISE

```
def main():
    try:
        percentage = float(input("Enter the percentage of marks obtained: "))
        if 0 <= percentage <= 100:
            grade = find_grade(percentage)
            print(f"The grade for {percentage:.2f}% is: {grade}")
        else:
            print("Invalid input. Please enter a valid percentage (between 0 and 100).")
    except ValueError:
        print("Invalid input. Please enter a valid percentage as a number.")

if __name__ == "__main__":
    main()
```

CASE STUDY

Q. CASE STUDY-BASED QUESTIONS

Write a menu driven program that has options to

- accept the marks of the student in five major subjects in Class X and display the same.
- calculate the sum of the marks of all subjects
- Divide the total marks by number of subjects (i.e. 5), calculate percentage = total marks/5 and display the percentage.
- Find the grade of the student as per the following criteria:

Criteria Grade

percentage > 85 A

percentage < 85 && percentage >= 75 B

percentage < 75 && percentage >= 50 C

percentage > 30 && percentage <= 50 D

percentage <30 Reappear .

Ans

```
def accept_marks():
    marks = []
    for i in range(1, 6):
        subject_marks = float(input(f"Enter marks in Subject {i}: "))
        marks.append(subject_marks)
    return marks
```

CASE STUDY

```
def display_marks(marks):
    print("Marks in each subject:")
    for i, subject_marks in enumerate(marks, start=1):
        print(f"Subject {i}: {subject_marks}")

def calculate_percentage(marks):
    total_marks = sum(marks)
    percentage = total_marks / 5
    return percentage
def find_grade(percentage):
    if percentage > 85:
        return "A"
    elif 75 <= percentage < 85:
        return "B"
    elif 50 <= percentage < 75:
        return "C"
    elif 30 <= percentage < 50:
        return "D"
    else:
        return "Reappear"
```

CASE STUDY

```
def main():
    marks = []
    while True:
        print("\nMENU:")
        print("1. Accept marks in five major subjects")
        print("2. Display marks in all subjects")
        print("3. Calculate and display the percentage")
        print("4. Find the grade")
        print("5. Exit")
        choice = int(input("Enter your choice: "))
        if choice == 1:
            marks = accept_marks()
        elif choice == 2:
            if marks:
                display_marks(marks)
            else:
                print("Please enter marks in subjects first.")
```

CASE STUDY

```
        elif choice == 3:
            if marks:
                percentage = calculate_percentage(marks)
                print(f"Percentage: {percentage:.2f}%")
            else:
                print("Please enter marks in subjects first.")
        elif choice == 4:
            if marks:
                percentage = calculate_percentage(marks)
                grade = find_grade(percentage)
                print(f"Grade: {grade}")
            else:
                print("Please enter marks in subjects first.")
        elif choice == 5:
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please select a valid option.")
if __name__ == "__main__":
    main()
```

CASE STUDY

In this program, the user is presented with a menu-driven interface with five options. The user can choose the respective options to enter marks, display marks, calculate percentage, find the grade, or exit the program.

The program stores the marks entered by the user in a list, and then it calculates and displays the percentage and grade based on the given criteria.

Thank you

Saroj Kumar Jha

srojkrjha@gmail.com