Introduction to Programming and Problem Solving

Week 6 : Loop Control Statements

Lecturer: Lemi Agrey Oliver

Department of Information Technology

Kumi University

Email: codingissweet@gmail.com

Recap of week 5 lecture

- Welcome to week 6 of our lecture! In our last lecture, we discussed control statements, which are programming language constructs that allow us to control the flow of execution of a program. We covered all the major types of control statements, including:
- Conditional statements: These statements allow us to execute different code blocks based on a condition. The most common conditional statements are if, else if, and else.
- We touched a bit about Iteration statements: These statements allow us to execute a block of code repeatedly until a certain condition is met. The most common iteration statements are while and for.

Recap of week 5 lecture

- We also discussed Boolean types, which are data types that can have two values: true or false. Boolean expressions are used to evaluate conditions in control statements.
- To help us understand these concepts, we provided many code examples. We also discussed how control statements and Boolean types can be used to implement common programming patterns, such as input validation, error handling, and data processing.

Content

1. Introduction to loop
2. The while Loop
3. The range() Function
4. The for Loop
5. Nested Loops
6. The break Statement
7. The continue Statement

Introduction to loops

- In computer programming, a loop is a sequence of instructions that is continually repeated until a certain condition is reached[1]
- A loop can also be defined as a control structure that permits a repetition execution of a block of code.
- It then follows that with a loop a programmer is able to execute a group of code multiple times without necessarily having to repeat the code.

Types of Loops

- Python provides two main types of loops and these are for loop and while loop.
- These loops are used to automate tasks, iterate through data structures, and perform operations until specific conditions are met.
- Now let us discuss these two types of loops in details
- **For loop**, this type of loop is used for iterating over a sequence and this sequence can be a list, tuple, string or range
- for item in iterable:

For loop

Here:

- for is the keyword that initiates the loop.
- item represents the iterator or variable that takes on each value in the sequence one by one.
- in is a membership operator used to determine whether a value exists in a particular sequence.
- iterable is the sequence, such as a list, tuple, string, or range, over which the loop iterates

For loop+

```
1. fruits = ["apple", "banana",
   "cherry", "date", "Ovacado",
   "passion", "elderberry"]
2. for f in fruits:
3.     if f=='passion':
4.         break
5.     print(f)
6. print("End of the loop”)
```

For loop++

Explanation:

- We used a for loop to iterate over a list of fruits to print each fruit individually however we employed an if statement to look out for the existence of a passion fruit in the list, and if found, the execution should be exited.
- We then printed out a string to mark the end of our for look
- The break statement can be used to exit the loop prematurely if a specific condition is satisfied, such as finding a passion fruit in this case

For loop Example2

```
1. numbers = [3, 5, 23, 6, 5, 1, 2, 9, 8]
2. # initializing a variable that will store the s
   um
3. sum_ = 0
4. # using for loop to iterate over the lis
5. for num in numbers:
6.     sum_ = sum_ + num ** 2
7. print("The sum of squares is: ", sum_)
```

Range Function

- In Python, the range() function is used to generate a sequence of numbers within a specified range. It's commonly used in for loops to iterate over a sequence of numbers. The range() function can take up to three arguments, and its syntax is as follows:
- range([start], stop[, step])
- start (optional): The starting value of the range (inclusive). If not provided, it defaults to 0.

Range Function+

stop (required): The ending value of the range (exclusive). The generated sequence will stop before reaching this value.

step (optional): The step size or increment between values in the range. If not provided, it defaults to 1.

Here are some examples of how you can use the range() function:

Generate a sequence of numbers from 0 to 4 (exclusive of 5):

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

Range Function++

- Generate a sequence of numbers from 2 to 8 (exclusive of 9) with a step size of 2:

```
for i in range(2, 9, 2):
    print(i)
```

- Generate a sequence of numbers from 10 down to 1 (exclusive of 0) with a step size of -1:

```
for i in range(10, 0, -1):
    print(i)
```

Range Function

- You can also convert a range object to a list if you want to create a list containing the numbers in the specified range:

```
my_list = list(range(5))  # Creates a list [0, 1, 2, 3, 4]
```

- Using the range() function efficiently saves memory because it generates numbers on-the-fly and doesn't create a list of all the numbers in memory, which is especially useful when dealing with large ranges.

The while Loop

- In Python, the while loop is a control flow statement that allows you to repeatedly execute a block of
- while condition:
- # Code to execute while the condition is True
- Here's how a while loop works:
- The condition is evaluated. If it's True, the code block inside the while loop is executed. If the condition is initially False, the code block is skipped entirely, and the program moves on to the next statement after the while loop.

The while Loop+

- After the code block is executed, the condition is checked again. If the condition is still True, the code block is executed again.
- This process repeats until the condition becomes False.
- Once the condition is False, the while loop terminates, and the program continues with the next statement after the loop.
- Here's a simple example of a while loop that counts from 1 to 5:

The while Loop++

```
1.count = 1
2.while count <= 5:
3.    print(count)
4.    count += 1
```

- In this example, count starts at 1, and the while loop continues as long as count is less than or equal to Inside the loop, we print the value of count and then increment it by 1 using count += 1. The loop will terminate when count becomes 6 because the condition count <= 5 is no longer True.

The while Loop+++

We can also used the while loop to count backward as can be seen the example below

```
1.count = 10
2.while count >=1:
3.     print(count, end=“”)
4.     count -= 1
```

```
Output: 10 9 8 7 6 5 4 3 2 1
```

Nested Loops

- In Python, you can use nested loops to create a loop inside another loop. This allows you to perform repetitive tasks that involve multiple levels of iteration. Nested loops are a powerful programming construct and are commonly used for tasks such as iterating through two-dimensional arrays or performing combinations and permutations.

The basic structure of nested loops is as follows:

```
for outer_loop_variable in outer_iterable:
    # Code for the outer loop
    for inner_loop_variable in inner_iterable:
        # Code for the inner loop
```

Here's an example of a nested loop:

Nested Loops+

```
for i in range(3):
    for j in range(2):
        print(i, j)
```

In this example, there are two nested loops. The outer loop iterates over values 0, 1, and 2, and for each value of i, the inner loop iterates over values 0 and 1.

The while Loop+

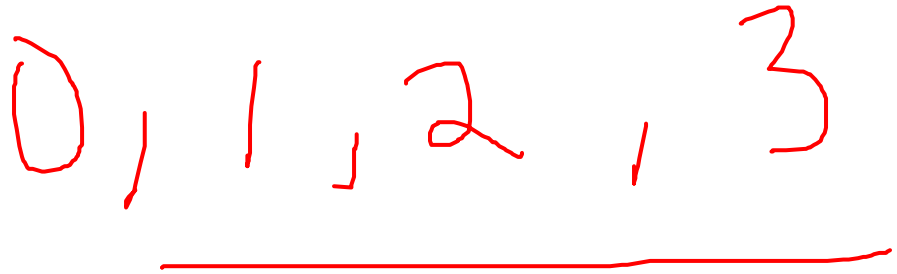
```
matrix = [[1, 2, 3],[4, 5, 6],[7, 8, 9]]
for row in matrix:
    for element in row:
        print(element, end=“”)
Output:1  2  3  4  5  6  7  8  9
```

Break Statements

In Python, the break statement is used to exit or terminate a loop prematurely. It is typically used within loops (such as for or while loops) to immediately stop the loop's execution when a certain condition is met, even if the loop's regular termination condition hasn't been reached.

- The basic syntax of the break statement is as follows
- while condition:
- # Code inside the loop
- if some_condition:
- break # Exit the loop if some_condition is True
- Here's an example using a while loop:

Break Statements




```
1. count = 0
2. while count < 5:
3.     print(count)
4.     if count == 3:
5.         break  # Exit the loop when count reaches 3
6.     count += 1
7. In this example, the loop prints the values of count from 0
 to 3.
```

A program to print a multiplication table

```
num = 21 counter = 1 # we will use a while loop for iterating 10 times for the multiplication table
print("The Multiplication Table of: ", num)
while counter <=7: # specifying the condition
        ans = num * counter
print (num, 'x', counter, '=', ans)
        counter += 1 # expression to increment the counter
```

The Multiplication Table of: 21

21 x 1 = 21

21 x 2 = 42

21 x 3 = 63

21 x 4 = 84

21 x 5 = 105

21 x 6 = 126

21 x 7 = 147

Continue statement

The basic syntax of the continue statement is as follows

```
while condition:
    # Code inside the loop
    if some_condition:
        continue  # Skip the current iteration and move to the next one
```

Here's an example using a while loop:

python

Continue statement+

```
count = 0
while count < 5:
    count += 1
    if count == 3:
        continue      #   Skip   the   current
iteration when count is 3
    print(count)
```

Continue statement+

- In this example, the loop counts from 1 to 5 but skips printing the number 3 due to the continue statement. When count becomes 3, the continue statement is encountered, and the loop proceeds to the next iteration without executing the print statement for that specific iteration.
- You can also use the continue statement with for loops. Here's an example:

```
for i in range(10):
    if i % 2 == 0:
        continue
    print(i)
```

Continue statement++

- In this case, the loop prints only the odd numbers from 0 to 9 and skips the even numbers using the continue statement.
- The continue statement is useful when you want to skip certain iterations of a loop based on a condition without terminating the entire loop. It allows you to customize the behavior of the loop according to your specific needs

A program to check if a number within a defined range is a prime or not +

Let us start by writing an algorithm

- Define a function named is_a_prime that takes an integer num as its parameter.
- Initialize a variable condition to 0. This variable will be used to determine whether the number is prime or not.
- Initialize a variable iteration to 2. This variable is used to iterate through possible divisors of the num.
- Start a while loop with the condition while iteration <= num / 2:.

A program to check if a number within a defined range is a prime or not ++

- This loop continues as long as iteration is less than or equal to half of the input num. This is because any factor of a number will not be greater than its half.
- Inside the while loop:
- Check if num is divisible by iteration without any remainder: if num % iteration == 0:
- If this condition is true, it means that num is not a prime number.
- Set condition to 1 to indicate that the number is not prime.
- Use the break statement to exit the loop since we have determined that the number is not prime.

A program to check if a number within a defined range is a prime or not +++

- After the while loop, check the value of condition:
- If condition is still 0, it means that the number is prime, so print that num is a PRIME number.
- If condition is 1, it means that the number is not prime, so print that num is not a PRIME number.
- Outside the is_a_prime function:
- Iterate through a range of numbers from 1 to 49 (inclusive) with a step of 3 using a for loop: for i in range(1, 50, 3):

A program to check if a number within a defined range is a prime or not ++++

- For each value of i in the range, call the is_a_prime function with i as its argument to determine whether it's prime or not.
- Print the result for each i.
- The algorithm essentially defines a function is_a_prime to check if a number is prime, and then it iterates through a range of numbers from 1 to 49 (inclusive) with a step of 3, calling the function for each number and printing the result.

Code implementation of a program to check if a number within a defined range is a prime or not +

```
def is_a_prime(num):
    condition = 0
    iteration = 2
    while iteration <= num / 2:
        if num % iteration == 0:
            condition = 1
            break
        iteration = iteration + 1
    if condition == 0:
        print(f"{num} is a PRIME number")
    else:
        print(f"{num} is not a PRIME number")
for i in range(1, 50, 3):
 is_a_prime(i)
```

Code Exercise 1: Write a program to display the pattern of stars given as follows[2]

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

Code Exercise 1: Solution

```
print(" Star Pattern Display")
num=1
x=num
for i in range(1,6,1):
    num=num+1;
    for j in range(1,num,1):
        print(" * ",end="")
x=num+1

    print()
print("End of Program")
```

Summary

- In this chapter, we delved into the realm of loops in Python, a pivotal concept for automating repetitive tasks and efficiently processing data.
- Types of Loops Covered:
- for Loops:
- for loops were introduced as a means to iterate over sequences such as lists and strings.
- They empower the execution of specific actions for each item within the sequence.

Summary+

- while Loops:
- while loops were explored as a tool for code repetition based on a specified condition.
- Strategies to prevent infinite loops were discussed.
- Loop Control Statements:
- We introduced loop control statements like break and continue to modify loop behavior and enhance control over program flow.

Summary++

- Advanced Techniques:
- Nested Loops:
- The concept of nested loops, where one loop is enclosed within another, was elucidated.
- Nested loops were showcased for handling intricate repetitive tasks and multi-dimensional data structures.
- Looping with Range:

Summary+++

- We examined the utility of the range() function, which simplifies the management of loop iterations by generating sequences of numbers.
- In Conclusion:
- Mastery of loops is indispensable for any Python programmer, as loops constitute the foundation of diverse programming tasks, spanning from data processing to algorithm implementation.

Summary++++

- Loops are pivotal for automation, efficient data processing, and effective problem-solving, establishing them as a cornerstone of Python programming.
- Equipped with this knowledge, students are well-prepared to tackle a wide range of programming challenges and develop sophisticated Python applications.

Reference:

[1] (2023, September 23). *What is loop?: Definition from TechTarget*. WhatIs.com. Retrieved December 15, 2021, from https://www.techtarget.com/whatis/definition/loop

[2] Introduction to Programming and problem solving with python, Ashok N. Kamthane, Amit A. Kamthane, McGraw Hill Education (India) Private Limited,2018, page 124