AIMERZ.ai
Aim.Act.Achieve

NOTES

Loops and Control Flow

Loops and Control Flow

Loops in Python: While and For

Loops are fundamental in programming, allowing you to execute a block of code multiple times without manually repeating it. This section will introduce both **while** and **for** loops in Python, keeping the explanations simple and clear for both technical and non-technical learners. By the end, learners will understand how to use loops to automate repetitive tasks.

1. While Loop

A **while** loop repeatedly executes a block of code as long as a specified condition is `True`. It's ideal when the number of iterations is unknown beforehand and depends on the condition.

Syntax:

```python
while condition:
    # Code block to execute
```

Example:

```python
count = 1

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

In this example, the loop will print numbers from 1 to 5. The condition `count <= 5` keeps the loop running until `count` exceeds 5, and `count += 1` increments the value by 1 after each iteration.

2. For Loop

The **for** loop is used to iterate over a sequence (such as a list, tuple, dictionary, or string) or a range of numbers. It is typically used when the number of iterations is known or determined by the length of the sequence.

Syntax:

```python
for variable in sequence:
    # Code block to execute
```

Example:

```python
for num in range(1, 6):
    print(num)
```

In this example, the loop iterates over the range of numbers from 1 to 5 (inclusive) and prints each number.

- **range(1, 6)** generates numbers from 1 to 5. The first number is inclusive, and the second number is exclusive, so the loop stops at 5.

Key Differences Between While and For Loops

- **While Loop**: Runs until the condition is False. Typically used when the number of iterations is not known in advance.
- **For Loop**: Runs a fixed number of times, usually based on the length of a sequence or range.

3. Breaking Out of Loops

You can use the **break** statement to exit a loop prematurely when a certain condition is met, even if the loop's condition hasn't yet become `False`.

Example:

```python
for num in range(1, 11):
    if num == 6:
        break
    print(num)
```

In this case, the loop stops when `num` equals 6, so the output will be the numbers 1 through 5.

4. Nested Loops

A **nested loop** means placing one loop inside another. This is useful for working with multi-dimensional data structures such as lists of lists.

Example:

```python
for i in range(1, 4):
    for j in range(1, 4):
        print(f"i: {i}, j: {j}")
```

This will print every combination of `i` and `j` from 1 to 3.

5. Skipping an Iteration

The **continue** statement allows you to skip the current iteration and move to the next one, without terminating the entire loop.

Example:

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

Here, the number 3 is skipped, so the output will be 1, 2, 4, and 5.

6. Looping Through Sequences

The **for** loop is particularly useful for iterating over sequences like lists, strings, and dictionaries. This is key in data science for processing data sets or collections of data.

Example with Lists:

```python
fruits = ["apple", "banana", "cherry"]

for fruit in fruits:
    print(fruit)
```

Output:

```
apple
banana
cherry
```

7. Infinite Loops

An **infinite loop** occurs when the condition in a **while** loop never becomes `False`. While it's generally something to avoid, it can be useful in cases where the program waits for user input or constantly monitors a condition. Ensure learners understand the importance of proper condition handling to avoid these unintentional infinite loops.

Example of an Infinite Loop:

```python
while True:
    print("This will run forever unless stopped.")
```

8. Else in Loops

Python allows an optional **else** block to be added to a loop. The **else** block will execute if the loop finishes normally (i.e., without hitting a **break**).

Example:

```python
for num in range(1, 6):
    print(num)
else:
    print("Loop completed successfully.")
```

Here, after the loop prints numbers 1 to 5, the **else** block runs, printing "Loop completed successfully."

Output:

```
1
2
3
4
5
Loop completed successfully.
```

Problem to solve:

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

# For loop example
for num in numbers:
    print(num)

# While loop example
count = 1
while count <= 5:
    print(count)
    count += 1
```

Find the solution??

Control Flow in Python: break Statement, continue Statement, and range() Function

In this submodule, learners will explore how to control the flow of loops and iterations in Python. We'll cover three important concepts: the **break** statement, the **continue** statement, and the **range()** function. These topics will allow your students to better understand how to manage loops, control when they stop or skip iterations, and work with ranges of numbers.

1. break Statement

The **break** statement is used to **exit** a loop prematurely, stopping the loop when a specific condition is met. It is often used when we want to terminate the loop based on some logic before the loop naturally finishes its iterations.

Syntax:

```python
if condition:
    break
```

Example:

```python
for num in range(1, 11):
    if num == 6:
        break
    print(num)
```

In this example, the loop prints numbers from 1 to 5 and stops when num reaches 6 because of the **break** statement.The loop ends prematurely when the break condition is satisfied.

Output:

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

2. continue Statement

The **continue** statement allows you to **skip** the current iteration of a loop and move directly to the next iteration. This is useful when you want to avoid certain iterations but still continue looping.

Syntax:

```
Python
if condition:
    continue
```

Example:

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

In this example, the number 3 is skipped, and the loop continues with 1, 2, 4, and 5 being printed. **continue** moves to the next iteration of the loop, skipping the rest of the current iteration.

Output:

```
Python
1
2
4
5
```

3. range() Function

The **range()** function is commonly used in **for** loops to generate a sequence of numbers. This function allows you to specify the start, stop, and step size for the sequence.

Syntax:

```
Python
range(start, stop, step)
```

- **start**: The starting number of the sequence (inclusive).
- **stop**: The ending number of the sequence (exclusive).
- **step**: The difference between each number in the sequence (optional).

Common Uses:

1. **range(stop)**: This will generate numbers from 0 up to, but not including, the stop value.
2. **range(start, stop)**: This generates numbers starting from start and stops before stop.
3. **range(start, stop, step)**: This generates numbers starting from start, stopping before stop, and incrementing by step.

Example:

```
Python
# Using start, stop, and step
for num in range(2, 10, 2):
    print(num)
```

This example starts at 2, ends at 10 (exclusive), and increments by 2. The output will be 2, 4, 6, 8. The **range()** function is versatile for generating sequences of numbers, with full control over where the sequence starts and ends.

Output:

```
Python
2
4
6
8
```

Find output and try to write the explanation

```
Python
for num in range(1, 10):
    if num == 5:
        continue  # Skip the number 5
    if num == 8:
        break     # Stop the loop when num is 8
    print(num)
```

Output and explanation???

THANK YOU

AIMERZ.ai
Aim.Act.Achieve