Loop statements in shell scripting allow you to execute a block of code repeatedly, which is essential for tasks like iterating over a list of files, processing arrays, or performing repetitive operations. Here's a detailed guide to understanding loop statements in shell scripting:

1. `for` Loop

The `for` loop is used to iterate over a list of items, such as strings, numbers, or filenames.

Syntax:

```
for variable in list; do
    # Commands to execute
done
```

Example 1: Iterating Over a List of Strings

```
#!/bin/bash

# Define a list of fruits
fruits=("Apple" "Banana" "Cherry")

# Iterate over each fruit
for fruit in "${fruits[@]}"; do
    echo "I like $fruit"
done
```

- This loop iterates over each element in the `fruits` array and prints it.

Example 2: Iterating Over a Sequence of Numbers

```
#!/bin/bash

# Iterate over a sequence of numbers
for i in {1..5}; do
    echo "Number: $i"
done
```

- This loop prints numbers from 1 to 5.

2. `while` Loop

The `while` loop repeatedly executes a block of code as long as the condition remains true.

Syntax:

```
while [ condition ]; do
    # Commands to execute
done
```

Example: Countdown Script

```
#!/bin/bash

# Countdown from 5 to 1
counter=5

while [ $counter -gt 0 ]; do
    echo "Countdown: $counter"
    counter=$((counter - 1))
done

echo "Blast off!"
```

- This script counts down from 5 to 1, then prints "Blast off!".

3. `until` Loop

The `until` loop is similar to the `while` loop, but it executes the code block until the condition becomes true.

Syntax:

```
until [ condition ]; do
```

```
    # Commands to execute
done
```

Example: Incrementing a Counter

```
#!/bin/bash

# Increment counter until it reaches 5
counter=1

until [ $counter -gt 5 ]; do
    echo "Counter: $counter"
    counter=$((counter + 1))
done
```

- This script increments the counter until it exceeds 5.

4. C-style `for` Loop

A C-style `for` loop allows you to define the initialization, condition, and increment in a single line, similar to loops in C or Java.

Syntax:

```
for (( initialization; condition; increment )); do
    # Commands to execute
done
```

Example: Sum of First 5 Numbers

```
#!/bin/bash
```

```
# Sum the first 5 numbers
sum=0

for (( i=1; i<=5; i++ )); do
    sum=$((sum + i))
done

echo "Sum: $sum"
```

- This loop calculates the sum of the first 5 numbers.

5. `break` and `continue` Statements

- **`break`**: Exits the loop prematurely.
- **`continue`**: Skips the current iteration and proceeds with the next one.

Example: Breaking Out of a Loop

```
#!/bin/bash

# Loop through numbers and break when i equals 3
for i in {1..5}; do
    if [ $i -eq 3 ]; then
        break
    fi
    echo "Number: $i"
done
```

- This loop prints numbers 1 and 2, then exits when `i` equals 3.

Example: Skipping an Iteration

```
#!/bin/bash
```

```
# Loop through numbers and skip when i equals 3
for i in {1..5}; do
    if [ $i -eq 3 ]; then
        continue
    fi
    echo "Number: $i"
done
```

- This loop prints all numbers from 1 to 5 except for 3.

6. Nested Loops

You can nest loops within loops to perform more complex operations, like processing multi-dimensional arrays.

Example: Multiplication Table

```
#!/bin/bash

# Generate a multiplication table
for i in {1..3}; do
    for j in {1..3}; do
        result=$((i * j))
        echo "$i * $j = $result"
    done
done
```

- This script prints a multiplication table for numbers 1 to 3.

7. Looping Through Command Output

You can loop through the output of a command using a `for` loop.

Example: Listing Files in a Directory

```
#!/bin/bash
```

```
# List all files in the current directory
for file in $(ls); do
    echo "File: $file"
done
```

- This script lists all files in the current directory.

Example: Reading a File Line by Line

```
#!/bin/bash

# Read a file line by line
filename="example.txt"

while read -r line; do
    echo "Line: $line"
done < "$filename"
```

- This script reads each line from `example.txt` and prints it.

Summary

- **`for` Loop**: Iterates over a list of items or a sequence of numbers.
- **`while` Loop**: Repeats as long as a condition is true.
- **`until` Loop**: Repeats until a condition becomes true.
- **C-style `for` Loop**: A more flexible loop, similar to those in C-like languages.
- **`break` and `continue`**: Control loop execution by exiting or skipping iterations.
- **Nested Loops**: Use loops within loops for more complex operations.
- **Looping Through Command Output**: Allows you to process the output of commands within loops.

Loops are fundamental in shell scripting for automating repetitive tasks, processing data, and efficiently managing workflows.