Python Basics Manual

Objective

The objective of this manual is to provide a comprehensive understanding of Loops. By the end of this manual, readers should be able to:

1. Iterate over sequences (lists and strings) using the `for` loop and `range()` function.
2. Understand and implement loops, Combine loops with conditional statements.
3. Use the break statement and the continue statement within loops.
4. Implement nested loops for handling complex iterations.

Table of Contents

1. For Loop

    - Using `range()`
    - Iterating Over Lists
    - Iterating Over Strings
    - Break and Continue Statement
    - Nested Loop

2. While Loop

    - Syntax of While Loop
    - Conditions in While Loop
    - Break Statement
    - Continue Statement
    - Nested Loops

1. For Loop

The `for` loop is used to iterate over a sequence (such as a list or a string) and execute a block of code for each element in the sequence.

1.1 Using `range()`

The `range()` function generates a sequence of numbers, which is commonly used with `for` loops.

Syntax:

```
for i in range(start, stop, step):
    # code to execute for each value of i
```

- `start`: The starting value of the sequence (inclusive).
- `stop`: The end value of the sequence (exclusive).
- `step`: The difference between each number in the sequence.

Example:

```
for i in range(1, 10, 2):
    print(i)
# Output: 1, 3, 5, 7, 9
```

1.2 Iterating Over Lists

You can use a `for` loop to iterate over each element in a list.

Example:

```
fruits = ['apple', 'banana', 'cherry']
for fruit in fruits:
    print(fruit)
# Output: apple, banana, cherry
```

1.3 Iterating Over Strings

Strings are sequences of characters, and you can use a `for` loop to iterate over each character in a string.

Example:

```
for char in 'hello':
    print(char)
# Output: h, e, l, l, o
```

1.4 Examples of For Loops

Example 1: Using `range()` with a Single Argument

```
for i in range(5):
    print(i)
# Output: 0, 1, 2, 3, 4
```

Example 2: Using `range()` with Start and Stop Arguments

```
for i in range(3, 8):
    print(i)
# Output: 3, 4, 5, 6, 7
```

Example 3: Iterating Over a List

```
colors = ['red', 'green', 'blue']
for color in colors:
    print(color)
# Output: red, green, blue
```

Example 4: Iterating Over a String

```
word = 'python'
for letter in word:
    print(letter)
# Output: p, y, t, h, o, n
```

1.5 Break Statement

The break statement terminates the loop immediately when encountered.

Examples of Break Statement

```
for i in range(10):
    if i == 7:
        break
    print(i)
```

1.6 Continue Statement

The continue statement skips the current iteration and moves to the next iteration of the loop.

Example: Skipping Even Numbers

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

1.7 Nested Loops

Nested loops are loops within loops. They are useful for performing repeated actions on a multidimensional scale.

Example 1: Multiplication Table

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

Example 2: Looping Through a Matrix

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

for row in matrix:
    for item in row:
        print(item, end=" ")
    print()
```

1.8 Practical Applications

Nested loops are often used in scenarios like matrix operations, traversing multi-dimensional arrays, and implementing algorithms that require repetitive iterations at multiple levels.

2. While Loop

Loops are fundamental constructs in programming that allow the execution of a block of code multiple times. This manual focuses on the while loop, break and continue statements, and nested loops, which are essential for creating efficient and effective code.

2.1 Syntax of While Loop

The while loop continues to execute a block of code as long as a specified condition is true. The syntax is as follows:

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

2.2 Conditions in While Loop

Conditions in a while loop determine how long the loop will continue to execute. The loop will only terminate when the condition evaluates to false.

Example 1: Basic While Loop

```
count = 0
while count < 5:
    print("Count is:", count)
    count += 1
```

Example 2: Using While Loop with User Input

```
user_input = ""
while user_input.lower() != "exit":
    user_input = input("Type 'exit' to stop: ")
```

2.3 Combining Loops and Conditionals

Loops can be combined with conditional statements to perform complex operations.

```
number = 0
while number < 10:
    if number % 2 == 0:
        print(number, "is even")
    else:
        print(number, "is odd")
    number += 1
```

Complex Example

Combining nested conditionals within loops for more intricate logic.

```
number = 1
while number <= 20:
    if number % 3 == 0 and number % 5 == 0:
        print("FizzBuzz")
    elif number % 3 == 0:
        print("Fizz")
    elif number % 5 == 0:
        print("Buzz")
    else:
        print(number)
    number += 1
```

2.4 Break Statement

The break statement terminates the loop immediately when encountered.

```
count = 0
while count < 10:
    print(count)
    if count == 5:
        break
    count += 1
```

Example 1: Breaking Out of a Loop

```
while True:
    response = input("Type 'quit' to exit: ")
    if response == 'quit':
        break
```

2.5 Continue Statement

The continue statement skips the current iteration and moves to the next iteration of the loop.

```
count = 0
while count < 10:
    count += 1
    if count % 2 == 0:
        continue
    print(count)
```

Example 1: Continue with User Input

```
while True:
    num = int(input("Enter a number (0 to skip, 9 to quit): "))
    if num == 9:
        break
    if num == 0:
        continue
    print("You entered:", num)
```

2.6 Nested Loops

Nested loops are loops within loops. They are useful for performing repeated actions on a multidimensional scale.

Example 1: Multiplication Table

```
i = 1  # Outer loop counter

while i <= 3:
    j = 1  # Inner loop counter
    while j <= 3:
        print(f"{i} * {j} = {i * j}")
        j += 1
    i += 1
    print()  # Blank line for better readability
```

3. Conclusion

Understanding loops is crucial for writing efficient and effective code for repetitive tasks. Mastery of these concepts allows for the creation of complex and dynamic programs. Practice with various examples and applications to solidify your understanding and improve your programming skills.