LOGIC - CONDITIONAL STATEMENTS AND WORKING WITH LOOPS IN PYTHON

CONDITIONAL STATEMENTS IN PYTHON

Conditional statements in Python allow us to check for certain conditions and perform actions based on the outcome of those checks. Conditional statements are an essential part of programming in Python, they allow you to make decisions based on the values of variables or the result of comparisons.

There are several types of conditional statements in Python, including:

- if statement
- else statement
- elif statement
- nested if statement

IF STATEMENT

In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly. Conditional statements give us this ability. The simplest form is the if statement, which has the genaral form:

```
if BOOLEAN EXPRESSION:
   STATEMENTS
```

The if statement checks to see if a condition is true after which it would execute the block of code if found true, if the condition is False, the code block will be skipped.

A few important things to know about if statements:

1. The colon (:) is significant and required. It separates the header of the compound statement from the body.
2. The line after the colon must be indented. It is standard in Python to use four spaces for indenting.
3. All lines indented the same amount after the colon will be executed whenever the BOOLEAN_EXPRESSION is true.

Example:

```
if x  >= 50:
  print("Number is greater than 50")
```

The boolean expression after the if statement is called the condition. If it is true, then all the indented statements get executed.

IF ELSE STATEMENT

The if else statement is used to execute one block of code if a condition is true, and another block of code if the condition is false. If the condition is true, the code block indented below the if statement will be executed, and the code block indented below the else statement will be skipped. If the condition is False, the code block indented below the else statement will be executed, and the code block indented below the if statement will be skipped.

Code example:

```
age = int(input("Tell us your age."))

if age >= 18:
print("You are old enough to vote.")
else:
print("You are not old enough to vote.")
```

In this example, we use an if-else statement to check if age is greater than or equal to 18. If it is, the message "You are old enough to vote." is printed. If it is not (that is, age is below 18), the message "You are not old enough to vote." is printed.

ELIF STATEMENT

The elif statement is used to check multiple conditions in sequence, and execute a specific blocks of code based on which condition is true. The elif statement is short for "else if", and can be used multiple times to check additional conditions.

Code example:

```
age = int(input("Tell us your age."))

if age >= 18:
   print("You are old enough to vote.")
elif age >= 16:
   print("You can drive but cannot vote.")
else:
   print("You cannot drive or vote yet.")
```

In this example, we use the comparison operators to check the ages.

If the age is greater than or equal to 18, the condition is True, and the code block indented below the if statement will be executed. It will print the message 'You are old enough to vote.'
If the age is greater than or equal to 16, the condition is False, and the code block indented below the else statement will be executed, printing the message "You can drive but cannot vote."

NESTED IF STATEMENT

The nested if else statement is used when we need to check a condition inside another condition.
Code example:

```
age = 18
gender = "female"

if age >= 18:
if gender == "male":
print("You are a male and old enough to vote.")
else:
print("You are a female and old enough to vote.")
else:
print("You are not old enough to vote yet.")
```

IMPORTANCE OF CONDITIONAL STATEMENTS IN PYTHON

1. Conditional statements in python provide a way to make decisions in your program and execute different code based on those decisions.
2. Conditional statements (if, else, and elif) are fundamental programming constructs that allow you to control the flow of your program based on conditions that you specify.
3. With conditionals you can create more powerful and versatile programs that can handle a wider range of tasks and scenarios.
4. Conditional statements help mathematicians and computer programmers make decisions based on the state of a situation.
5. They pivotal role in controlling the flow of your program.

LOOPS IN PYTHON

A loop is an instruction that repeats multiple times as long as some condition is met, a loop is a repetitive statement or task. The loop iterates as many times as the number of elements and prints the elements serially.

Python has two primitive loop commands:
1. while loops
2. for loops

While Loop Statement:

A while loop statement in Python is used to repeatedly execute a block of code as long as a condition is true. It is typically used when you don't know how many times the loop will run in advance.
Code Sample:

```
# Using while loop
i = 0

while i < 10:
    i += 1
    print(i)

# Print numbers from 1 to 5 using a while loop
count = 1
while count <= 5:
print(count)
count += 1
```

For Loop Statement:

A for-loop statement in Python is used to iterate over a sequence (such as a list, tuple, or string) and perform a certain action for each item in the sequence. The for loop does not require an indexing variable to set beforehand.

A for loop uses the *range()* function to iterate for a certain number of time. The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

The else keyword in a for loop specifies a block of code to be executed when the loop is finished. The else block will NOT be executed if the loop is stopped by a break statement.

Code Sample:

```
# Print each character in a string using a for loop
word = "Python"
for letter in word:
    print(letter)

# Using the end parameter
word = "Python"
for letter in word:
    print(letter, end = "*")

# Using for loops in a list
list1 = ['Python', 'SQL', 'Java', 'Excel', 'HTML']
for courses in list1:
     print(courses)

# Finding the sum of a list of numbers
list2 = [10, 20, 30, 40, 50, 60, 70, 80, 90]
sum  = 0
for num in list2:
    sum = sum + num
print(f"The sum of list2 is {sum}.")

# For loop using a tuple.
tuple1 = (10, 20, 30, 40, 50, 60, 70, 80, 90, 100)
sum  = 0
for num in tuple1:
    sum = sum + num
print(f"The sum of tuple1 is {sum}.")

# Using the range function
for num in range(1,5):
    print(num)
```

The break Statement

With the break statement we can stop the loop even if the while condition is true:

```
# Using the break statement
for i in range (1,20):
    if i == 5:
        break
    else:
        print(i)
```

The Continue Statement

With the continue statement we can stop the current iteration, and continue with the next:

```
# Using the continue statement
for i in range (1,9):
    if i == 5:
        continue
    else:
        print(i)
```

Range Function:

The `range()` function in Python is used to generate a sequence of numbers. It is commonly used in `for` loops to iterate over a sequence of numbers.

The `range()` function can take one, two, or three arguments:

1. range(stop)
2. range(start, stop)
3. range(start, stop, step)

1. range(stop)
This form generates numbers from 0 up to, but not including, `stop`.
Code Example:

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

In this example, `range(5)` generates numbers from 0 to 4. The `for` loop iterates over these numbers and prints each one.

2. range(start, stop)
This form generates numbers from `start` up to, but not including, `stop`.
Code Example:

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

Here, `range(2, 6)` generates numbers starting from 2 up to 5. The `for` loop prints each number in this range.

3. range(start, stop, step)
This form generates numbers from `start` up to, but not including, `stop`, incrementing by `step`. If `step` is negative, it decrements.
Code Example:

```
for num in range(1, 10, 2):
    print(num)
```

In this example, `range(1, 10, 2)` generates numbers starting from 1 up to 9, incrementing by 2. The `for` loop prints each number in this range.

An example with a negative step:

```
for num in range(10, 0, -2):
    print(num)
```

Here, `range(10, 0, -2)` generates numbers starting from 10 down to 2, decrementing by 2. The `for` loop prints each number in this range.

Practical Uses of `range()`
1. Generating Sequences:
You can use `range()` to generate a sequence of numbers for iteration:

```
for i in range(5):
    print(f"Iteration {i}")
```

2. Indexing Lists:
When you need to iterate over the indices of a list:

```
my_list = ['a', 'b', 'c', 'd']
for i in range(len(my_list)):
    print(f"Index {i} contains {my_list[i]}")`
```

3. Creating Lists:
Although not common with modern Python practices (since list comprehensions are preferred), you can create lists using `range()`:

```
numbers = list(range(10))
print(numbers)  # Outputs: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

4. Looping a Specific Number of Times:

When you want to execute a block of code a specific number of times:

```
for _ in range(3):
    print("Hello, World!")
```

Output:
Hello, World!
Hello, World!
Hello, World!

The `range()` function is a versatile tool for generating sequences of numbers. It can take one, two, or three arguments to control the start, stop, and step of the sequence. This function is particularly useful in `for` loops, enabling efficient and readable iteration over a sequence of numbers.

By understanding and using the `range()` function effectively, you can write more efficient and readable loops in your Python code.

Nested For Loops:
Nested loops are loops inside another loop. It can have multiple loops in it and more than one inner loop. The "inner loop" will be executed one time for each iteration of the "outer loop".
The preceding code executes as follows: The program first encounters the outer loop, performing its first iteration. This first iteration triggers the inner, nested loop, which then runs to completion. Then the program returns to the top of the outer loop, completing the second iteration and again triggers the nested loop. The nested loop runs to completion, and the program returns to the top of the outer loop until the sequence is complete.

```
# Using a nested loops in a list
list3 = ['Python', 'SQL', 'Java', 'Excel', 'HTML', 'Ruby']

for courses in list3:
    print(courses)
    for letters in courses:
        print(letters)
```

The pass Statement
for loops cannot be empty, but if you for some reason have a for loop with no content, put in the pass statement to avoid getting an error.

Advanced Techniques in Conditionals and Loops

1. Ternary Operators
Ternary operators offer a concise way to express conditional statements in a single line. This can improve code readability when used appropriately.
Code Sample:

```
# Syntax: value_if_true if condition else value_if_false
result = "Pass" if score >= 50 else "Fail"
```

The ternary operator is particularly useful for simple conditions where the intent is clear.

2. Pattern Matching (Python 3.10+)
Pattern matching is a powerful feature introduced in Python 3.10, allowing for more expressive and readable condition checks.

```
match command:
    case "start":
        print("Starting")
    case "stop":
        print("Stopping")
    case _:
        print("Invalid command")
```

Pattern matching simplifies complex conditional logic and makes the code easier to maintain.

Best Practices for Conditionals and Loops

1. Readability and Maintainability

Follow PEP 8 guidelines to ensure your code is readable and maintainable. Clear and consistent indentation, meaningful variable names, and comments are crucial.

```
# Good example
if temperature > 30:
    print("It's hot outside.")
else:
    print("It's cool outside.")
```

2. Using Functions for Complex Conditions

Break down complex conditions into functions to enhance readability and reusability.

Code Sample:

```
def is_eligible_for_discount(age, membership_duration):
    return age > 65 or membership_duration > 5

if is_eligible_for_discount(customer_age, customer_membership_duration):
    apply_discount()
```

Optimizations and Performance

1. List Comprehensions

List comprehensions offer a concise way to create lists and can be more efficient than traditional loops.

Code Sample:

```
squares = [x**2 for x in range(10)]
```

They are especially useful for simple transformations and filtering.

2. Generator Expressions

Generator expressions are similar to list comprehensions but are more memory-efficient as they generate items on-the-fly.

Code Sample:

```
sum_of_squares = sum(x**2 for x in range(10))
```

Use generator expressions for large datasets where memory efficiency is important.

Real-World Applications

1. Conditional Logic in Data Processing

Conditional statements are essential in data processing tasks, such as filtering and transforming data in pandas.

Code Sample:

```
import pandas as pd
df = pd.DataFrame({'score': [45, 78, 90, 66]})
df['status'] = df['score'].apply(lambda x: 'Pass' if x >= 50 else 'Fail')
```

2. Loop Optimization in Data Science

Efficient loops are critical in large-scale data processing. Utilizing libraries like NumPy can significantly enhance performance.

Code Sample:

```
import numpy as np
large_array = np.random.rand(1000000)
result = np.sum(large_array)
```

NumPy operations are optimized and often faster than pure Python loops.

3. Context Managers and Loops

Context managers can be used within loops for efficient resource management.

Code Sample:

```
with open('file.txt') as f:
    for line in f:
        print(line)
```

Using context managers ensures that resources are properly managed and released.

Note:

1. It's a convention in programming to use the letters i, j, and k for loop variables, unless there's a good reason to give the variable a more descriptive name.
2. Infifinite loops are never-ending loops.

REFERENCES

1. Simplilearn Python Loops Tutorial |
2. Freecodecamp How to Use Conditional Statements in Python
3. w3schools
4. Geek for geeks
5. Chat gpt
6. Youtube Resources
7.