1.What are the two values of the Boolean data type? How do you write them?

2. What are the three different types of Boolean operators?

3. Make a list of each Boolean operator's truth tables (i.e. every possible combination of Boolean values for the operator and what it evaluate ).

4. What are the values of the following expressions?

(5 > 4) and (3 == 5)

not (5 > 4)

(5 > 4) or (3 == 5)

not ((5 > 4) or (3 == 5))

(True and True) and (True == False)

(not False) or (not True)

5. What are the six comparison operators?

6. How do you tell the difference between the equal to and assignment operators?Describe a condition and when you would use one.

7. Identify the three blocks in this code:

spam = 0

if spam == 10:

print('eggs')

if spam > 5:

print('bacon')

else:

print('ham')

print('spam')

print('spam')

8. Write code that prints Hello if 1 is stored in spam, prints Howdy if 2 is stored in spam, and prints Greetings! if anything else is stored in spam.

9.If your programme is stuck in an endless loop, what keys you’ll press?

10. How can you tell the difference between break and continue?

11. In a for loop, what is the difference between range(10), range(0, 10), and range(0, 10, 1)?

12. Write a short program that prints the numbers 1 to 10 using a for loop. Then write an equivalent program that prints the numbers 1 to 10 using a while loop.

13. If you had a function named bacon() inside a module named spam, how would you call it after importing spam?

Answer:

1. The Boolean data type represents logical values that can be either true or false. In programming languages, these values are typically represented using the keywords "true" and "false". The Boolean values can also be expressed using numeric representations, where 1 is often used to represent true and 0 to represent false.

To summarize:

- Boolean values: true and false

- Keyword representation: true, false

- Numeric representation: 1 for true, 0 for false

2. The three different types of Boolean operators are:

AND operator: This operator is typically represented by the symbol "&&" (double ampersand). It returns true if both operands are true; otherwise, it returns false. For example:

- True && True = True

- True && False = False

- False && True = False

- False && False = False

OR operator: This operator is typically represented by the symbol "||" (double pipe). It returns true if at least one of the operands is true; otherwise, it returns false. For example:

- True || True = True

- True || False = True

- False || True = True

- False || False = False

NOT operator: This operator is typically represented by the exclamation mark "!" (negation). It reverses the logical state of its operand. If the operand is true, the NOT operator returns false, and if the operand is false, the NOT operator returns true. For example:

- !True = False

- !False = True

3. Certainly! Here are the truth tables for each Boolean operator:

AND Operator (&&):

|  |  |  |
| --- | --- | --- |
| Operand 1 | Operand 2 | Result |
| true | true | true |
| true | false | false |
| false | true | false |
| false | false | false |

OR Operator (||):

|  |  |  |
| --- | --- | --- |
| Operand 1 | Operand 2 | Result |
| true | true | true |
| true | false | true |
| false | true | true |
| false | false | false |

NOT Operator (!):

|  |  |
| --- | --- |
| Operand | Result |
| true | false |
| false | true |

These truth tables provide the possible combinations of Boolean values for each operator and the resulting evaluation.

4. Let's evaluate the given expressions:

1. . (5 > 4) and (3 == 5):

- (5 > 4) is true.

- (3 == 5) is false.

- The expression evaluates to false.

B) . not (5 > 4):

- (5 > 4) is true.

- Applying the "not" operator, the expression evaluates to false.

C) . (5 > 4) or (3 == 5):

- (5 > 4) is true.

- (3 == 5) is false.

- The expression evaluates to true.

D) . not ((5 > 4) or (3 == 5)):

- (5 > 4) is true.

- (3 == 5) is false.

- (5 > 4) or (3 == 5) is true.

- Applying the "not" operator, the expression evaluates to false.

E) . (True and True) and (True == False):

- True and True is true.

- True == False is false.

- The expression evaluates to false.

F). (not False) or (not True):

- not False is true.

- not True is false.

- The expression evaluates to true.

So, the values of the given expressions are:

A). False

B). False

C). True

D). False

E). False

F). True

5. The six comparison operators are as follows:

a. Equal to (==): Checks if two values are equal and returns true if they are, or false otherwise.

b. Not equal to (!=): Checks if two values are not equal and returns true if they are not, or false otherwise.

c. Greater than (>): Checks if the value on the left is greater than the value on the right, returning true if it is, or false otherwise.

d. Less than (<): Checks if the value on the left is less than the value on the right, returning true if it is, or false otherwise.

e. Greater than or equal to (>=): Checks if the value on the left is greater than or equal to the value on the right, returning true if it is, or false otherwise.

f. Less than or equal to (<=): Checks if the value on the left is less than or equal to the value on the right, returning true if it is, or false otherwise.

6. To differentiate between the equal to (==) operator and the assignment (=) operator, consider the following:

a. Equal to (==) operator:

- The equal to operator (==) is used for comparison in conditional statements or expressions.

- It checks if two values are equal and returns a Boolean value of true if they are equal, or false if they are not equal.

- It does not modify the values being compared; it simply evaluates their equality.

b. Assignment (=) operator:

- The assignment operator (=) is used for assigning a value to a variable.

- It takes the value on the right and assigns it to the variable on the left.

- It modifies the variable by replacing its existing value with the new assigned value.

Example:

Let's consider a simple scenario where we have a variable called "num" that stores a number. We want to check if the number is equal to 10.

Using the equal to (==) operator:

```python

num = 5

if num == 10:

print("The number is equal to 10.")

else:

print("The number is not equal to 10.")

```

In this case, the equal to operator (==) compares the value stored in "num" with 10, and based on the result, it executes the corresponding block of code.

Using the assignment (=) operator:

```python

num = 10

```

7. In the provided code, there are three blocks of code that can be identified based on indentation. Here they are:

Block 1:

```python

if spam == 10:

print('eggs')

```

This block is an if statement that checks if the variable "spam" is equal to 10. If it is, it will execute the indented line of code, which is `print('eggs')`.

Block 2:

```python

if spam > 5:

print('bacon')

```

This block is another if statement that checks if the variable "spam" is greater than 5. If it is, it will execute the indented line of code, which is `print('bacon')`.

Block 3:

```python

else:

print('ham')

print('spam')

print('spam')

```

This block consists of an else statement followed by three print statements that are not indented. The else statement is associated with the second if statement (Block 2). If the condition in Block 2 (`spam > 5`) is not true, it will execute the indented line of code, which is `print('ham')`. The three subsequent print statements (`print('spam')`, `print('spam')`) are not part of any if or else statement and will be executed regardless of the conditions.

8. Here's an example code that achieves the desired behavior:

```python

spam = 1

if spam == 1:

print("Hello")

elif spam == 2:

print("Howdy")

else:

print("Greetings!")

```

In this code, the variable `spam` is checked using conditional statements (`if`, `elif`, `else`) to determine its value. If `spam` is equal to 1, it will print "Hello". If `spam` is equal to 2, it will print "Howdy". If `spam` has any other value, it will print "Greetings!". You can modify the value of `spam` to see the different output based on its value.

9. If your program is stuck in an endless loop and you want to interrupt its execution, you can typically press the following key combination:

\*\*Ctrl + C\*\* (or \*\*Command + C\*\* on macOS): This key combination is commonly used to send an interrupt signal to the running program. It will terminate the program and return you to the command prompt or stop the execution within an integrated development environment (IDE).

Pressing Ctrl + C (or Command + C) should break the loop and allow you to regain control over your program. However, please note that if the program is unresponsive or stuck due to other issues, this key combination might not work. In such cases, you may need to terminate the program using other means, such as closing the terminal or stopping the execution within the IDE.

10. The `break` and `continue` statements are used in loops to alter the flow of execution. Here's how you can differentiate between them:

1. `break` statement:

- When encountered within a loop (such as `for` or `while`), the `break` statement immediately terminates the loop and continues executing the code after the loop.

- It effectively "breaks out" of the loop and skips any remaining iterations.

- After the `break` statement is executed, the program moves on to the next line of code outside the loop.

- It is commonly used when a specific condition is met, and there is no need to continue iterating through the loop.

Example:

```python

for i in range(5):

if i == 3:

break

print(i)

```

Output:

```

0

1

2

```

In this example, the loop is terminated when `i` becomes equal to 3 due to the `break` statement. The loop stops immediately, and the code continues executing after the loop.

2. `continue` statement:

- When encountered within a loop, the `continue` statement skips the remaining code inside the loop for the current iteration and moves on to the next iteration.

- It effectively "continues" to the next iteration without executing the subsequent code for the current iteration.

- After encountering the `continue` statement, the loop does not terminate; it continues with the next iteration.

Example:

```python

for i in range(5):

if i == 2:

continue

print(i)

```

Output:

```

0

1

3

4

```

11. In a `for` loop, `range(10)`, `range(0, 10)`, and `range(0, 10, 1)` are three different ways to specify the range of values to iterate over. Here's the difference between them:

1. `range(10)`: This specifies a range that starts from 0 (by default) and goes up to, but does not include, the specified end value. In this case, the range will include the values from 0 to 9 (total of 10 values). It steps through the range with a default increment of 1.

Example:

```python

for i in range(10):

print(i)

```

Output:

```

0

1

2

3

4

5

6

7

8

9

```

2. `range(0, 10)`: This specifies a range that starts from the provided start value (0 in this case) and goes up to, but does not include, the specified end value (10 in this case). It steps through the range with a default increment of 1.

Example:

```python

for i in range(0, 10):

print(i)

```

Output:

```

0

1

2

3

4

5

6

7

8

9

```

3. `range(0, 10, 1)`: This specifies a range that starts from the provided start value (0 in this case), goes up to, but does not include, the specified end value (10 in this case), and increments by the provided step value (1 in this case).

Example:

```python

for i in range(0, 10, 1):

print(i)

```

Output:

```

0

1

2

3

4

5

6

7

8

9

12. Here's a short program that prints the numbers 1 to 10 using a `for` loop:

```python

for i in range(1, 11):

print(i)

```

Output:

```

1

2

3

4

5

6

7

8

9

10

```

And here's an equivalent program that prints the numbers 1 to 10 using a `while` loop:

```python

i = 1

while i <= 10:

print(i)

i += 1

```

Output:

```

1

2

3

4

5

6

7

8

9

10

```

Both programs achieve the same result of printing the numbers 1 to 10. In the `for` loop version, the `range(1, 11)` function generates a sequence of numbers from 1 to 10 (inclusive) that are then iterated over. In the `while` loop version, a variable `i` is initialized with 1, and the loop continues as long as `i` is less than or equal to 10. Within the loop, `i` is printed, and its value is incremented by 1 using `i += 1` to ensure the loop eventually terminates.

13. If you have a function named `bacon()` inside a module named `spam`, you can call it after importing the `spam` module by using the dot notation.

Here's how you would call the `bacon()` function after importing the `spam` module:

```python

import spam

spam.bacon()

```

In the code above, `spam` is the module name, and `bacon()` is the function name. By using `spam.bacon()`, you access the `bacon()` function within the `spam` module and call it.