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

Answer :

The two values of the Boolean data type are:

1. **True**
2. **False**

In Python, they are written with an uppercase first letter, as True and False.

These are built-in constants in Python that represent the truth values for Boolean logic.

Example:

Code Below :

```
a = True

b = False
```

- True represents a truth value (equivalent to 1 in numeric contexts).
- False represents a false value (equivalent to 0 in numeric contexts).

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

Answer :

The three different types of Boolean operators are:

1. **AND** (and)
   - Returns **True** if both operands are **True**.
   - Example: **True and False** results in **False.**
2. **OR** (or)
   - Returns True if at least one of the operands is True.
   - Example: **True or False** results in **True**.
3. **NOT** (not)
   - Reverses the Boolean value of the operand.
   - Example: **not True** results in **False.**

These operators are used to create logical conditions in programming.

Q.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).

Answer :

Here are the truth tables for the three Boolean operators **(AND, OR, and NOT):**

1. <u>AND (and) Truth Table</u> :

A	**B**	**A and B**
True	True	True
True	False	False
False	True	False
False	False	False

2.<u>OR (or)</u> <u>Truth Table</u> :

A	**B**	**A or B**
True	True	True
True	False	True
False	True	True
False	False	False

3. **<u>NOT (not)</u>** <u>Truth Table</u> :

A	**not A**
True	False
False	True

These tables show how each Boolean operator evaluates for every combination of input values (**True and False**).

Q.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)

Answer :

Let's evaluate the Boolean expressions step by step:

1. **(5 > 4) and (3 == 5)**
   - (5 > 4) is **True**.
   - (3 == 5) is **False**.
   - True and False is **False**.

Result: False

2. **not (5 > 4)**
   - (5 > 4) is **True**.
   - not True is **False**.

Result: False

3. **(5 > 4) or (3 == 5)**
   - (5 > 4) is **True**.
   - (3 == 5) is **False**.
   - True or False is **True**.

Result: True

4. **not ((5 > 4) or (3 == 5))**
   - (5 > 4) is **True**.

- (3 == 5) is **False**.
- (True or False) is **True**.
- not True is **False**.

Result: False

5. **(True and True) and (True == False)**
    - (True and True) is **True**.
    - (True == False) is **False**.
    - True and False is **False**.

Result: False

6. **(not False) or (not True)**
    - not False is **True**.
    - not True is **False**.
    - True or False is **True**.

Result: True

<u>Summary of Results</u>:

1. **False**
2. **False**
3. **True**
4. **False**
5. **False**
6. **True**

Q.5 What are the six comparison operators?

Answer :

The six comparison operators in Python (and many other programming languages) are:

1. **==: Equal to**
    - Checks if the values of two operands are equal.
    - Example: 5 == 5 returns **True**.
2. **!=: Not equal to**
    - Checks if the values of two operands are not equal.
    - Example: 5 != 3 returns **True**.
3. **>: Greater than**
    - Checks if the value of the left operand is greater than the right operand.
    - Example: 5 > 3 returns **True**.
4. **<: Less than**
    - Checks if the value of the left operand is less than the right operand.
    - Example: 3 < 5 returns **True**.
5. **>=: Greater than or equal to**
    - Checks if the value of the left operand is greater than or equal to the right operand.
    - Example: 5 >= 5 returns **True**.
6. **<=: Less than or equal to**
    - Checks if the value of the left operand is less than or equal to the right operand.
    - Example: 3 <= 5 returns **True**.

These operators are commonly used in conditions and loops to compare values.

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

Answer:

In Python (and many other programming languages), the **equal to (==)** and **assignment (=)** operators serve different purposes:

1. Equal to (==):

- **Purpose**: It is a comparison operator used to check if two values are equal.
- **Usage**: It evaluates whether the left-hand side value is the same as the right-hand side value and returns a Boolean (True or False).

<u>**Example**</u> **:**

<u>**Code Below :**</u>

```
if x == 5:
   print("x is equal to 5")
```

In this case, == checks if the variable x is equal to 5. If x is indeed 5, the condition is True, and the code inside the if block will execute.

2. Assignment (=):

- **Purpose**: It is used to assign a value to a variable.
- **Usage**: The left-hand side is the variable, and the right-hand side is the value being assigned.

<u>**Example**</u>:

<u>**Code Below**</u> :

```
x = 5
```

Here, = assigns the value 5 to the variable x.

<u>Key Difference</u>:

- **==** compares two values to check if they are the same.
- **=** assigns a value to a variable.

<u>Condition Example</u>:

Suppose we are building a simple program to check if a person's age qualifies them for a discount:

<u>Code Below :</u>

```
age = 18  # Assignment operator (=)

if age == 18:  # Comparison operator (==)

    print("You qualify for a student discount!")
```

<u>In this case</u>:

- **age = 18** assigns the value **18** to the variable **age**.
- **age == 18** checks if the value of **age** is equal to **18.** If true, the message about the discount is printed.

Mistaking the **assignment (=)** operator for **equal to (==)** would cause errors in conditions and produce unexpected results.

Q.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')

Answer :

In the given code, there are three blocks of code. A **block** refers to a group of statements that are intended to be executed together based on a certain condition. Blocks are typically indicated by their indentation in Python.

Here is the corrected and structured version of the code, with the blocks indicated:

<u>**Code Below :**</u>

```
spam = 0  # Block 1


if spam == 10:  # Block 2 starts

   print('eggs')  # Indented code under 'if' is part of Block 2


if spam > 5:  # Block 3 starts

   print('bacon')  # Indented code under 'if' is part of Block 3

else:  # Part of Block 3

   print('ham')  # Indented code under 'else' is part of Block 3


print('spam')  # Outside all conditional blocks
print('spam')  # Outside all conditional blocks
```

<u>The Three Blocks</u>:

1. **Block 1**: The first line **spam = 0** is not part of any conditional block, so it stands alone.
2. **Block 2**: This block includes the **if spam == 10:** statement and the associated indented **print('eggs')** line. These two lines form a block because **print('eggs')** is only executed if the condition **spam == 10** is **True**.
3. **Block 3**: The third block includes the **if spam > 5:** and **else:** statements along with their indented print statements (**print('bacon'**) and **print('ham'))**. This block represents the conditional logic that prints either "bacon" if **spam > 5** or "ham" otherwise.

The final two **print('spam')** statements are not indented and are outside of all the conditional blocks, so they will always be executed, regardless of any conditions.

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.

Answer :

Here's the Python code that fulfills the condition:

```
spam = int(input("Enter a value for spam: "))  # Input to assign a value to spam

if spam == 1:

   print("Hello")

elif spam == 2:

   print("Howdy")

else:

   print("Greetings!")
```

Explanation:

- If spam is 1, it prints "Hello".
- If spam is 2, it prints "Howdy".
- For any other value of spam, it prints "Greetings!".

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

Answer :

If our program is stuck in an endless loop and you need to stop it, we can usually press the following key combinations, depending on the environment:

- **In most terminal/command prompt environments**:
    - Press **Ctrl + C**. This sends an interrupt signal to the program, which usually stops it.
- **In Jupyter Notebook**:

    - We can click on the "Stop" button (the square icon) in the toolbar or use the keyboard shortcut **Esc**, then **I, I** (press **I** twice).

- **In some IDEs (like PyCharm or VS Code)**:
    - We can typically use **Ctrl + C** in the terminal window or click the "Stop" button in the IDE's console.

Using these methods, We should be able to exit the endless loop safely.

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

Answer:

In Python, both **break** and **continue** are control flow statements used within loops, but they serve different purposes:

break

- **Purpose**: The **break** statement is used to exit the loop completely.
- **Usage**: When encountered, it immediately terminates the current loop, and the program execution continues with the next statement following the loop.

Example:

Code Below :

```
for i in range(5):
   if i == 3:
      break  # Exit the loop when i is 3
   print(i)


# Output:
# 0
# 1
# 2
```

Continue

- **Purpose**: The **continue** statement is used to skip the current iteration of the loop and move to the next iteration.
- **Usage**: When encountered, it stops the execution of the current loop iteration and jumps to the next iteration of the loop.

Example:

Code Below:

```
for i in range(5):
    if i == 3:
        continue  # Skip the iteration when i is 3
    print(i)


# Output:
# 0
# 1
# 2
# 4
```

Summary

- **break**: Exits the loop entirely.
- **continue**: Skips to the next iteration of the loop.

These statements are useful for controlling loop behavior based on specific conditions.

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

Answer:

In Python, the range() function generates a sequence of numbers and can be called with different parameters. Here's a breakdown of the three variations you mentioned: range(10), range(0, 10), and range(0, 10, 1).

1. range(10)

- **Description**: This creates a range object that starts from 0 and goes up to, but does not include, 10.
- **Equivalent to**: **range(0, 10)** (default start is 0, and the default step is 1).
- **Output**: The numbers generated will be **[0, 1, 2, 3, 4, 5, 6, 7, 8, 9].**

2. range(0, 10)

- **Description**: This explicitly defines the starting point as 0 and the endpoint as 10.
- **Equivalent to**: **range(10)** (start is 0 and the step is 1).
- **Output**: The same sequence of numbers will be generated: **[0, 1, 2, 3, 4, 5, 6, 7, 8, 9].**

3. range(0, 10, 1)

- **Description**: This specifies the starting point (0), the endpoint (10), and the step size (1).
- **Output**: This will generate the same sequence as the previous two: **[0, 1, 2, 3, 4, 5, 6, 7, 8, 9].**

<u>**Summary**</u>

- **range(10)** and **range(0, 10)** are equivalent in that they both start from 0 and go to 9 with a step of 1.
- **range(0, 10, 1)** explicitly defines all parameters but produces the same output.

These variations provide flexibility, allowing you to customize the start point, endpoint, and step size of the sequence generated by **range().**

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.

Answer:

The two short programs that print the numbers from 1 to 10: one using a **for** loop and the other using a **while** loop.

Using a For Loop

Code Below :

```
# Using a for loop to print numbers 1 to 10

for number in range(1, 11):

   print(number)
```

Using a While Loop

Code Below :

```
# Using a while loop to print numbers 1 to 10

number = 1

while number <= 10:

   print(number)

   number += 1  # Increment the number
```

Code Explanation:

- **For Loop**: The **for** loop iterates through a range of numbers from 1 to 10 (inclusive), printing each number.
- **While Loop**: The **while** loop starts with the variable **number** set to 1 and continues printing as long as **number** is less than or equal to 10. The variable is incremented by 1 after each iteration to eventually terminate the loop

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

Answer:

To call a function named **bacon()** that is defined inside a module named **spam**, you would first need to import the module and then use the dot notation to call the function. Here's how you can do it:

Code Below:

```
# Import the spam module

import spam


# Call the bacon() function from the spam module

spam.bacon()
```

Code Explanation:

- **Importing the Module**: import spam imports the entire module named spam.
- **Calling the Function**: spam.bacon() calls the bacon() function defined within the spam module using the dot notation, which is standard for accessing functions or variables in a module.