1.ANS

1.The Boolean data type has two values:

1. **True**: This represents a true or "on" state. It is typically written as "True" in many programming languages, including Python.
2. **False**: This represents a false or "off" state. It is typically written as "False" in many programming languages, including Python.

2.ANS

In Python, there are three primary Boolean operators, which are used for performing logical operations on Boolean values. These operators are:

1. \*\*AND Operator (`and`)\*\*: The `and` operator returns `True` if both of its operands are `True`. If at least one operand is `False`, the result is `False`.

2. \*\*OR Operator (`or`)\*\*: The `or` operator returns `True` if at least one of its operands is `True`. It returns `False` only if both operands are `False`. ```

3\*\*NOT Operator (`not`)\*\*: The `not` operator is a unary operator that negates the value of its operand. It returns `True` if the operand is `False`, and it returns `False` if the operand is `True`.

3.ANS

Here are the truth tables for each of the three primary Boolean operators: AND, OR, and NOT. These tables show all possible combinations of Boolean values and the results of the operators:

\*\*AND Operator (`and`):\*\*

| Operand 1 | Operand 2 | Result |

|-----------|-----------|--------|

| False | False | False |

| False | True | False |

| True | False | False |

| True | True | True |

\*\*OR Operator (`or`):\*\*

| Operand 1 | Operand 2 | Result |

|-----------|-----------|--------|

| False | False | False |

| False | True | True |

| True | False | True |

| True | True | True |

| False | True |

| True | False |

These truth tables illustrate the behavior of each Boolean operator for all possible combinations of its operands.

4.ANS

Let's evaluate the values of the given expressions step by step:

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

- `5 > 4` is `True`.

- `3 == 5` is `False`.

- `True and False` is `False`.

2. `not (5 > 4)`:

- `5 > 4` is `True`.

- `not True` is `False`.

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

- `5 > 4` is `True`.

- `3 == 5` is `False`.

- `True or False` is `True`.

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

- `(5 > 4) or (3 == 5)` is `True` (as shown in the previous evaluation).

- `not True` is `False`.

5. `(True and True) and (True == False)`:

- `True and True` is `True`.

- `True == False` is `False`.

- `True and False` is `False`.

6. `(not False) or (not True)`:

- `not False` is `True`.

- `not True` is `False`.

- `True or False` is `True`.

the values are given as below:

1. `False`

2. `False`

3. `True`

4. `False`

5. `False`

6. `True`

5.ANS

In Python, there are six comparison operators used to compare values. These operators are:

1. \*\*Equality Operator (`==`)\*\*: Checks if two values are equal.

2. \*\*Inequality Operator (`!=`)\*\*: Checks if two values are not equal.

3. \*\*Greater Than Operator (`>`)\*\*: Checks if the value on the left is greater than the value on the right.

4. \*\*Less Than Operator (`<`)\*\*: Checks if the value on the left is less than the value on the right.

5. \*\*Greater Than or Equal To Operator (`>=`)\*\*: Checks if the value on the left is greater than or equal to the value on the right.

6. \*\*Less Than or Equal To Operator (`<=`)\*\*: Checks if the value on the left is less than or equal to the value on the right.

6.ANS

In Python, you can differentiate between the equality operator (`==`) and the assignment operator (`=`) based on their usage and context:

1. \*\*Equality Operator (`==`):\*\*

- The equality operator is used to compare two values to check if they are equal.

- It is used in conditional statements to create conditions that test for equality.

- It returns a Boolean value, either `True` (if the values are equal) or `False` (if they are not).

Example of using the equality operator in a condition:

x = 5

if x == 5:

This condition checks if x is equal to 5 and performs some action if it is.

print("x is equal to 5")

2. Assignment Operator (`=`):

- The assignment operator is used to assign a value to a variable.

- It is not used for comparison; instead, it updates the value of a variable.

- It does not return a Boolean value but assigns the value on the right side to the variable on the left side.

Example of using the assignment operator to assign a value to a variable:

x = 5

This line assigns the value 5 to the variable x.

7.ANS

BLOCK 1

spam = 0

if spam == 10:

print('eggs')

This block sets the value of the variable spam to 0 and contains an if statement that checks if spam is equal to 10. If spam is equal to 10, it prints 'eggs'. However, since spam is 0, the condition is not met, and 'eggs' is not printed.

BLOCK 2

if spam > 5:

print('bacon')

This block contains another **if** statement that checks if **spam** is greater than 5. If **spam** is greater than 5, it prints 'bacon'. In this case, this condition is also not met because **spam** is 0, so 'bacon' is not printed.

BLOCK 3:

else:

print('ham')

print('spam')

print('spam')

This block includes an **else** statement, which is associated with the previous **if** statement. Since neither of the earlier conditions (in Block 1 and Block 2) were met, the code in the **else** block is executed. It prints 'ham', followed by two more print statements

1. The first **if** block that checks if **spam** is equal to 10 if spam not equal to ten
2. The second **if** block that checks if **spam** is greater than 5
3. The **else** block that is executed when neither of the previous conditions is met, printing 'ham' and two instances of 'spam'.

8.ANS

spam = 3 You can change the value of spam to test different cases

if spam == 1:spam = 3 # You can change the value of spam to test different cases

if spam == 1:

print('Hello')

elif spam == 2:

print('Howdy')

else:

print('Greetings!')

9.ANS

If your programme is stuck in an endless loop, we use break statement.

10.ANS

In Python, `break` and `continue` are two control flow statements used in loops (such as `for` and `while` loops) to change the flow of the program. They serve different purposes and have distinct behaviors:

1. `break`:

- The `break` statement is used to exit the current loop prematurely.

- When `break` is encountered, it immediately terminates the loop, and control is transferred to the code following the loop.

- It is typically used when a specific condition is met, and you want to exit the loop early.

Example:

for i in range(5):

if i == 3:

break

print(i)

In this example, the loop is terminated when `i` becomes 3, so only values 0, 1, and 2 are printed.

2. `continue`:

- The `continue` statement is used to skip the current iteration of the loop and move to the next iteration.

- When `continue` is encountered, the remaining code within the current iteration is skipped, and the loop proceeds to the next iteration.

- It is typically used when you want to bypass a particular iteration but continue the loop.

Example:

for i in range(5):

if i == 2:

continue

print(i)

In this example, the iteration for `i` equal to 2 is skipped, and the loop continues with values 0, 1, 3, and 4 being printed.

11.ANS

In Python, a `for` loop using `range()` can be defined with different arguments. However, in the specific context of your question, there is no practical difference between the following three variations:

1. `range(10)`: This generates a sequence of numbers starting from 0 (the default start value) up to, but not including, 10 (the specified stop value). It increments by 1 (the default step value), resulting in the sequence `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`.

2. `range(0, 10)`: This is essentially the same as `range(10)`. It specifies the start value (0, which is the default start value), the stop value (10), and uses the default step value of 1.

3. `range(0, 10, 1)`: Again, this is equivalent to the previous examples. It explicitly specifies the start value (0), the stop value (10), and the step value (1), but the step value of 1 is also the default.

So, in practice, all three variations will produce the same result: a sequence of numbers from 0 to 9. You can use any of them in a `for` loop to iterate over this sequence of numbers. The choice of which one to use may depend on your preference or the specific context in which you are writing your code.

12.ANS

i = 1

while i <= 10:

print(i)

i += 1

13.ANS

import spam

Call the bacon() function

spam.bacon()