***Python assgn 2***

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

**Ans.** The Boolean data type has two possible values:

**True:** This represents a true or affirmative condition.

**False:** This represents a false or negative condition.

In many programming languages, these values are written as "True" and "False" with the first letter capitalized. Some programming languages also allow the use of lowercase versions such as "true" and "false," but the case sensitivity can vary from language to language.

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

**Ans.** Boolean operators are used in search engines and programming to manipulate and combine Boolean values (true or false). There are three primary Boolean operators:

* **AND:** The AND operator returns true if both operands are true. In other words, it combines two or more conditions and only returns true if all of them are true. In Boolean algebra, it's represented as "∧" or sometimes as "&&" in programming languages. For example, in a search query, "cats AND dogs" would only return results that contain both "cats" and "dogs."
* **OR:** The OR operator returns true if at least one of the operands is true. It's represented as "∨" or "||" in programming languages. For example, in a search query, "cats OR dogs" would return results that contain either "cats" or "dogs" or both.
* **NOT:** The NOT operator is used to negate a Boolean value, which means it returns the opposite of the operand. If the operand is true, NOT returns false, and if the operand is false, NOT returns true. It's often represented as "¬" or "!" in programming languages. For example, "NOT cats" would return results that do not contain the word "cats."

These Boolean operators are fundamental for Boolean logic, which is the basis for many search queries, database queries, and programming conditions. They allow you to create complex and precise queries or conditions by combining and manipulating true and false values.

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

**Ans.** Certainly! Here are the truth tables for the basic Boolean operators: AND, OR, NOT, XOR (exclusive OR), and NAND (NOT AND).

* AND (Conjunction):

| **A** | **B** | **A AND B** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

* OR (Disjunction):

| **A** | **B** | **A OR B** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

* NOT (Negation):

| **A** | **NOT A** |
| --- | --- |
| 0 | 1 |
| 1 | 0 |

* XOR (Exclusive OR):

| **A** | **B** | **A XOR B** |
| --- | --- | --- |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

* NAND (NOT AND):

| **A** | **B** | **A NAND B** |
| --- | --- | --- |
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

These truth tables show all possible combinations of Boolean values for each operator and their corresponding results.

1. 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)

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

1. (5 > 4) and (3 == 5)
   * (5 > 4) evaluates to True
   * (3 == 5) evaluates to False
   * True and False evaluates to False
2. not (5 > 4)
   * (5 > 4) evaluates to True
   * not True evaluates to False
3. (5 > 4) or (3 == 5)
   * (5 > 4) evaluates to True
   * (3 == 5) evaluates to False
   * True or False evaluates to True
4. not ((5 > 4) or (3 == 5))
   * (5 > 4) evaluates to True
   * (3 == 5) evaluates to False
   * True or False evaluates to True
   * not True evaluates to False
5. (True and True) and (True == False)
   * True and True evaluates to True
   * True == False evaluates to False
   * True and False evaluates to False
6. (not False) or (not True)
   * not False evaluates to True
   * not True evaluates to False
   * True or False evaluates to True

So, the values of the given expressions are:

1. False
2. False
3. True
4. False
5. False
6. True
7. What are the six comparison operators?

**Ans.** The six comparison operators, also known as relational operators, are used in programming and mathematics to compare two values and determine the relationship between them. These operators are typically used in conditional statements to make decisions based on the comparison results. The six comparison operators are:

1. Equal to (==): This operator checks if two values are equal. If they are, it returns true; otherwise, it returns false.
2. Not equal to (!=): This operator checks if two values are not equal. If they are not equal, it returns true; otherwise, it returns false.
3. Greater than (>): This operator checks if the value on the left is greater than the value on the right. If it is, it returns true; otherwise, it returns false.
4. Less than (<): This operator checks if the value on the left is less than the value on the right. If it is, it returns true; otherwise, it returns false.
5. Greater than or equal to (>=): This operator checks if the value on the left is greater than or equal to the value on the right. If it is, it returns true; otherwise, it returns false.
6. Less than or equal to (<=): This operator checks if the value on the left is less than or equal to the value on the right. If it is, it returns true; otherwise, it returns false.

These operators are fundamental in programming for making decisions and controlling the flow of a program based on the comparison results.

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

**Ans.** The equal to operator (**==**) and the assignment operator (**=**) are two different operators used in programming for distinct purposes. Here's how you can tell the difference between them:

1. Equal To Operator (**==**):
   1. The double equals (**==**) operator is a comparison operator.
   2. It is used to compare two values or expressions to determine if they are equal.
   3. It returns a Boolean value, either **true** if the values are equal or **false** if they are not.

Use Case:

You would use the equal to operator in conditions to check if two values or expressions are equal or not. For example, when comparing user input with an expected value or checking if a condition is met.

1. Assignment Operator (=):

The single equals (=) operator is an assignment operator.

It is used to assign a value to a variable.

It does not perform a comparison; instead, it assigns the value on its right-hand side to the variable on its left-hand side.

Use Case:

You would use the assignment operator when you want to store a value in a variable or update the value of an existing variable.

To summarize, the equal to operator (**==**) is used for comparing values, and it returns a Boolean result. The assignment operator (**=**) is used for assigning values to variables. It's important not to confuse the two, as using the wrong operator in a particular context can lead to unexpected behavior or errors in your code.

1. 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’)

Ans. In the provided code, there are three blocks of code based on indentation and conditional structures. Here they are identified:

Block 1:

bash

spam = 0 if spam == 10: print('eggs')

This block sets the **spam** variable to 0 and checks if it's equal to 10. Since the condition is not met (spam is not equal to 10), the code inside this block is not executed.

Block 2:

bash

if spam > 5: print('bacon') else: print('ham')

This block contains an **if** statement that checks if **spam** is greater than 5. If it is, it prints 'bacon'; otherwise, it prints 'ham'. In this case, since **spam** is not greater than 5 (it's 0), 'ham' is printed.

Block 3:

bash

print('spam') print('spam')

This block consists of two **print** statements, and it will always be executed regardless of the conditions in the previous blocks. It prints 'spam' twice.

So, the three blocks in the code are as follows:

1. The first block with the **if spam == 10** condition.
2. The second block with the **if spam > 5** condition and the **else** branch.
3. The third block with the two **print** statements that always execute.
4. 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.

Ans. You can achieve this using an **if-elif-else** statement in Python. Here's the code to accomplish your task:

python

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

if spam == 1:

print("Hello")

elif spam == 2:

print("Howdy")

else:

print("Greetings!")

In this code:

* If the value stored in **spam** is 1, it will print "Hello."
* If the value is 2, it will print "Howdy."
* If the value is anything other than 1 or 2, it will print "Greetings!"

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

Ans. If a program is stuck in an endless loop and unresponsive, you can typically use a set of key combinations or commands to forcefully terminate the program or process. The exact keys or commands can vary depending on the operating system you're using. Here are some common methods for different operating systems:

1. **Windows**:
   * **Ctrl + Alt + Delete**: Press these keys simultaneously to open the Task Manager. From there, you can select the program that's stuck and click "End Task" to force it to close.
2. **macOS**:
   * **Option + Command + Esc**: Press these keys simultaneously to open the "Force Quit Applications" window. From there, you can select the unresponsive application and click "Force Quit" to terminate it.
3. **Linux** (using the terminal):
   * You can open a terminal and use the **kill** command with the process ID (PID) of the stuck program. First, find the PID by running the **ps aux | grep [program\_name]** command, and then use **kill -9 [PID]** to forcefully terminate it. For example:

perl

ps aux | grep firefox

kill -9 1234

In extreme cases, if your entire system becomes unresponsive due to a program running amok, you may need to reboot your computer using a hardware reset or power cycle. However, this should be done as a last resort since it can result in data loss and potential system instability.

Remember that forcibly terminating a program can lead to unsaved data loss and may leave your system in an inconsistent state, so it's always a good practice to save your work regularly and troubleshoot the root cause of the endless loop if possible.

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

Ans. In programming, "break" and "continue" are control flow statements used within loops (such as for loops or while loops) to alter the flow of the program. They serve different purposes and are used in distinct scenarios. Here's how you can tell the difference between them:

1. **Break**:
   * The "break" statement is used to exit or terminate a loop prematurely.
   * When encountered, "break" immediately exits the loop, regardless of whether the loop's condition or iteration steps have been completed.
   * It is often used when a certain condition is met, and you want to stop the loop from continuing to iterate.

Example (in Python):

python

for i in range(1, 10):

if i == 5:

break print(i)

In this example, the loop will print numbers from 1 to 4 and then exit when **i** becomes 5.

1. **Continue**:
   * The "continue" statement is used to skip the current iteration of a loop and move to the next iteration.
   * When encountered, "continue" causes the loop to jump immediately to the next iteration, bypassing any code that follows it in the current iteration.
   * It is often used when you want to skip specific iterations based on a condition, but you don't want to exit the loop entirely.

Example (in Python):

python

for i in range(1, 6):

if i == 3:

continue print(i)

In this example, the loop will print numbers 1, 2, 4, and 5, skipping the iteration when **i** is 3.

In summary, "break" is used to exit a loop prematurely, while "continue" is used to skip the current iteration and move to the next iteration of the loop. They are both useful control flow statements for controlling the flow of your loops in different ways.

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

Ans. In a for loop, there is no practical difference between using **range(10)**, **range(0, 10)**, and **range(0, 10, 1)** in Python. All three of these expressions will produce the same result.

The **range()** function in Python is quite flexible and allows you to specify a start value (inclusive), an end value (exclusive), and a step size. When you don't specify the start value, it defaults to 0, and when you don't specify the step size, it defaults to 1.

Here's how each of these expressions works:

* **range(10)**:
  + This generates a sequence of numbers from 0 to 9 (inclusive of 0 and exclusive of 10) with a default step size of 1.
* **range(0, 10)**:
  + This explicitly specifies the start value (0) and end value (10), but it still uses the default step size of 1. It generates the same sequence from 0 to 9.
* **range(0, 10, 1)**:
  + This explicitly specifies the start value (0), end value (10), and step size (1). Again, it generates the same sequence from 0 to 9.

In all cases, you can use any of these expressions interchangeably in a for loop when you want to iterate over the numbers from 0 to 9. The choice between them usually comes down to coding style and readability.

1. 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.

Ans. Sure, here's a Python program that prints the numbers 1 to 10 using both a for loop and a while loop:

Using a for loop:

python

for i in range(1, 11):

print(i)

Using a while loop:

python

i = 1

while i <= 10:

print(i)

i += 1

Both of these programs will produce the same output, printing the numbers from 1 to 10.

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

Ans.   
To call a function named **bacon()** that is defined inside a module named **spam** after importing the **spam** module, you would typically use the following syntax:

python

import spam

spam.bacon()

In this code:

1. **import spam** imports the **spam** module.
2. **spam.bacon()** calls the **bacon()** function from the **spam** module.

Make sure that the **spam** module is in the same directory as your Python script or is located in one of the directories listed in the Python path to be importable.