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

Ans.

A Boolean expression or data type is basically a statement that has one of the two possible values, which are either **'True'** or **'False'**. In other words, it would be a statement which can either be right or wrong. For example, 1 == 1 which is True and 3 < 1 is False. Also one thing to note would be that 'True' and 'False' are not strings but rather they are **keywords**.

And as for how do we write them, T and F should be capital or uppercase and the rest should be lowercase as python is a case sensitive language. Here's an example

```
# Assigning Boolean values to variables
is_raining = True
is_sunny = False

# Using Boolean values in conditional statements
if is_raining:
    print("It's raining!")

if not is_sunny:
    print("It's not sunny.")

# Performing a numerical operation with Boolean values
a = True
b = False
result = a + b
print(result)  # Output: 1 (True is treated as 1, and False as 0 in numerical operations)
```

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

Ans.

The different types of Boolean operators are:

1. **'and':** This 'and' operator would return the output as 'True' when **both** (in case of just two operands) the operands around the operator satisfy as true otherwise it would return as 'False'.
2. **'or':** This 'or' operator would return the output as 'True' if **either** one of the operands around the operator satisfies as true and when none of the operands satisfy as true, it will return as 'False'.
3. **'not':** This 'not' operator would return the **opposite** of the operand's value. If the operand is 'True', it returns 'False' and vice versa as illustrated in the below code

```
# Define variables with Boolean values
a = True
b = False
c = True

# Using "and" operator
print(a and b)  # Output: False (Both a and b are not true)
print(a and c)  # Output: True (Both a and c are true)

# Using "or" operator
print(a or b)   # Output: True (At least one of a or b is true)
print(b or c)   # Output: True (At least one of b or c is true)

# Using "not" operator
print(not a)    # Output: False (Negating the value of a, which is
True)
print(not b)    # Output: True (Negating the value of b, which is
False)
```

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).

Ans.

The truth tables for each Boolean operator are as follows, where the 'and' and 'or' operator has two operands and the 'not' has one:

1. **'and'** operator:

Operand 1	Operand 2	Output / Result
True	True	True
True	False	False
False	True	False
False	False	False

2. **'or'** operator:

Operand 1	Operand 2	Output / Result
True	True	True
True	False	True
False	True	True
False	False	False

3. **'not'** operator:

Operand	Output / Result
True	False
False	True

4. What are the values of the following expressions?

Problem	Output value
(5 > 4) and (3 == 5)	False
not (5 > 4)	False
(5 > 4) or (3 == 5)	True
not ((5 > 4) or (3 == 5))	False
(True and True) and (True == False)	False
(not False) or (not True)	True

5. What are the six comparison operators?

Ans.

In Python, comparison operators are used to compare two and return a Boolean value, which is either True or False. The six comparison operators are as follows:

1. Equal to: **('==')** This operator is responsible to check if the values on both sides of the operator are equal.
2. Not equal to: **('!=')** This one checks if the values on both sides of the operator are not equal.
3. Greater than: **('>')** Checks if the value on the left side of the operator is greater than the value on the right side.
4. Less than: **('<')** Checks if the value on the left side of the operator is less than the value on the right side.
5. Greater than or equal to: **('>=')** Checks if the value on the left side of the operator is greater than or equal to the value on the right side.
6. Less than or equal to: **('<=')** Checks if the value on the left side of the operator is less than or equal to the value on the right side.

These operators are mostly used in conditional statements and loops. For example:

```
a = 5
b = 10

print(a == b)   # Output: False
print(a != b)   # Output: True
print(a > b)    # Output: False
print(a < b)    # Output: True
print(a >= b)   # Output: False
print(a <= b)   # Output: True
```

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

Ans.

The comparison operator equal to '==' and the assignment operator '=' serve different purposes:

- **Equal to operator ('=='):**

  As mentioned earlier, it is used for comparison, where it checks if two values are equal and returns a Boolean value. It's commonly used in conditional statements as illustrated in the below example:

```
x = 5

y = 10


if x == y:

   print("x and y are equal.")

else:

   print("x and y are not equal.")
```

In this example, the code compares the values of '**x**' and '**y**' using the equal to operator ('**==**'). Since '**x**' is 5 and '**y**' is 10, the output will be "x and y are not equal."

- **Assignment operator ('='):**

  Now as for the assignment operator '=' with a single equal sign is used to assign a value to a variable. It takes the value on the right side and assigns it to the variable on the left side. It does not compare the values but instead, it performs an assignment operation. Example:

```
x = 5

y = x


print(y)  # Output: 5
```

Here the value of **'x'** (which is 5) is assigned to the variable **'y'**. The assignment operator **('=')** performs the assignment operation.

- **Condition:** In Python or programming in general, a condition is an expression or a statement that evaluates to either True or False. These conditions are fundamental

for controlling the flow of a program. Conditional statements are used to execute or skip blocks of code based on certain conditions. The **if, elif,** and **else** keywords are used to define these conditional statements.

Here's an example to illustrate the use of a condition in an **if** statement:

```
x = 10


if x > 5:
    print("x is greater than 5")
else:
    print("x is not greater than 5")
```

In this code, the condition **x > 5** evaluates to **True** because **x** is 10, which is indeed greater than 5. As a result, the first block of code will be executed, and the output will be "x is greater than 5."

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')
```

Ans.

The above code doesn't have indentation which is used to define a block of code, so assuming the code to be

```
spam = 0
if spam == 10:
    print('eggs')
if spam > 5:
    print('bacon')
else:
    print('ham')
print('spam')
print('spam')
```

In the given code, there are three blocks, each associated with an **'if'** or **'else'** statement identified by the indentation. They are:

- **Block 1:**

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

This is a conditional block that is executed if the value of the variable **spam** is equal to 10. Otherwise this block will be skipped.

- **Block 2:**

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

This is another conditional block with an **if-else** construct. If the value of **spam** is greater than 5, it will print 'bacon', and if it's not, it will print 'ham'.

- **Block 3:**

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

This block consists of two print statements which will be executed unconditionally, regardless of the previous 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.

Ans.

Code:

```
spam = int(input("Enter a single digit number, preferably between 1 to 3. "))


if spam == 1:
    print("Hello")


elif spam == 2:
    print("Howdy")


else:
    print("Greetings!")
```

One thing to note would be that we can also write this program without having to convert the string input from user into an integer, as the entered value does not undergo any calculations. So we can also write the code as:

```
spam = input("Enter a single digit number, preferably between 1 to 3. ")


if spam == '1':
    print("Hello")


elif spam == '2':
    print("Howdy")
```

```
else:
    print("Greetings!")
```

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

Ans.

A program while executed gets stuck in an endless or infinite loop when the condition given in a 'while' or 'for' loop is always true. As a result the loop will keep executing the code block repeatedly. An example of an infinite 'while' loop is shown below:

```
while True:
    print("This is what an infinite loop looks like.")
''' output: This is what an infinite loop looks like.
 This is what an infinite loop looks like.
 This is what an infinite loop looks like.
 This is what an infinite loop looks like...'''
```

The '**while True'** statement creates an infinite loop since the condition is always true (**'True'**). The loop will keep executing the code block inside it repeatedly, printing "This is an endless loop!" indefinitely.

To get out of this endless loop or cycle, or to terminate the program you can press **'Ctrl + C'** if you're on a Windows or Linux operating system and press **'Command + .'** (Command key followed by the period) if you're on macOS.

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

Ans.

'break' and 'continue' are two flow control statements in Python that are used inside loops to change or modify their behaviour. They change the flow of a loop based on certain conditions. And here's how you can tell the difference between 'break' and 'continue'.

- **'break'**: The '**break'** statement is used to exit the loop prematurely when a certain condition is met. And when it does, the **break** statement immediately terminates the loop. Example for 'break':

```
for i in range(5):

    if i == 3: # condition, that when true the break will
        break   # executed

    print(i)

# Output: 0 1 2
```

In this example, when 'i' becomes 3, the break is executed and the loop is terminated early or prematurely, so only 0, 1 and 2 are printed.

- **'continue':** The 'continue' statement is used to skip the current iteration of the loop when a certain condition is met. And when it does, the **continue** statement stops the current iteration and proceeds to the next. Example:

```
for i in range(5):

    if i == 2:      #The condition, when met the continue
        continue    # statement will be executed
    print(i)

# Output: 0 1 3 4
```

In this example, when 'i' becomes 2, the **continue** statement is executed, and the loop moves or skips to the following iteration without executing the rest of the code inside the loop for that particular iteration. Resulting in the value 2 being skipped.

11. 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 the outputs of range(10), range(0, 10) and range(0, 10, 1). All produce sequence of integers from 0 to 9 and not including 10.

- **'range(10)': range(stop)**
  This creates a sequence of integers starting from 0 (inclusive) and ending at 10 (exclusive) with a default step size of 1. The default start value is 0, and the step size is 1, which means it generates numbers from 0 to 9.

- **‘range(0, 10)’: range(start, stop)**
  This also creates a sequence of integers starting from 0 (inclusive) and ending at 10 (exclusive) with a default step size of 1. Specifying 0 as the start value explicitly has the same effect as the previous version.

- **‘range(0, 10, 1)’: range(start, stop, step)**
  Again, this creates the same sequence of integers starting from 0 (inclusive) and ending at 10 (exclusive) with a step size of 1. Specifying 1 as the step size explicitly has the same effect as the previous versions, where the step size defaults to 1.

To sum it up, again they all produce the same output but the use or overriding the parameters of range can be done where we can specify the start, end and the step or each iteration. This can be useful when you need to generate a sequence that starts from a different value, has a different range, or follows a specific pattern of increments.

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.

Ans.

- **‘for’ loop:**

```
# range(1, 11) to override the start parameter
for i in range(1, 11):

    # end = ' ', overriding the new line parameter
    print(i, end=' ')
```

In the above code, I used **‘range(1,11)’** to override the starting point from 1, failing to do so would have given me the output from 0 to 10. And in the print statement I used the **end= ‘ ‘** which is a parameter to override the default new line for every iteration which is completely optional and a matter of preference.

- **‘while’ loop:**

```
x = 1 # a variable which is basically a start point

while x <= 10:
# loop with a condition and 10 being the stop point

    print(x, end = ' ')
    x += 1 # this is basically the step or jump
```

In this code, I took a variable and assigned it as 1 which as per the problem is our starting point. Following which I took the while loop with the condition to stop or terminate the loop when the condition is dissatisfied or becomes False.

For more similarity between the ‘for’ loop and ‘while’ loop, you can also take the condition as **‘x < 11’,** as 11 being the stop point**.** Then there’s the print statement to print the value stored in **‘x’.** Then lastly, an expression to increment the value stored in **x** by 1 (step) which is crucial not just for the solution but also for not ending up in an infinite loop.

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

Ans.

If I have a function named ‘bacon()’ inside a module name ‘spam’, and I want to call the function after importing the module, I would use the following syntax:

```
import spam

spam.bacon()
```

1. **‘import spam’**: This statement imports the **spam** module, making all its functions accessible in the current Python script.
2. **‘spam.bacon()’**: After importing the **spam** module, you can call the **bacon()** function using the module name followed by the function name, separated by a dot (**.**).