1. Create an assert statement that throws an AssertionError if the variable spam is a negative integer.

ANS :- Certainly! Here's an example of an assert statement that throws an AssertionError if the variable `spam` is a negative integer:

```python

assert spam >= 0, "spam should not be a negative integer"

```

In this example, the `assert` statement checks if the condition `spam >= 0` is true. If the condition evaluates to False, an AssertionError is raised with the specified error message "spam should not be a negative integer". This helps in debugging and identifying issues in your code when an unexpected condition is encountered.

2. Write an assert statement that triggers an AssertionError if the variables eggs and bacon contain strings that are the same as each other, even if their cases are different (that is, 'hello' and 'hello' are considered the same, and 'goodbye' and 'GOODbye' are also considered the same).

ANS :- To compare the variables `eggs` and `bacon` while ignoring the case, you can use the `lower()` method to convert both strings to lowercase and then compare them. Here's an example of an assert statement that triggers an AssertionError if the lowercase versions of `eggs` and `bacon` are the same:

```python

assert eggs.lower() != bacon.lower(), "eggs and bacon should not be the same"

```

In this example, the `assert` statement checks if the condition `eggs.lower() != bacon.lower()` is true. If the condition evaluates to False, meaning the lowercase versions of `eggs` and `bacon` are the same, an AssertionError is raised with the specified error message "eggs and bacon should not be the same". This assertion helps ensure that the strings are not the same regardless of their case.

3. Create an assert statement that throws an AssertionError every time.

ANS :- To create an assert statement that always throws an AssertionError, you can use the `assert` keyword followed by a condition that is always false. Here's an example:

```python

assert False, "This assert statement always triggers an AssertionError"

```

In this example, the condition `False` is always false, so the assert statement will always trigger an AssertionError. The specified error message "This assert statement always triggers an AssertionError" will be displayed along with the AssertionError.

4. What are the two lines that must be present in your software in order to call logging.debug()?

ANS :- To call `logging.debug()` in your software, we need to have the following two lines:

```python

import logging

logging.basicConfig(level=logging.DEBUG, format='%(levelname)s: %(message)s')

```

The first line imports the logging module, which provides the debugging functionality. The second line configures the logging module with the desired log level (`DEBUG` in this case) and specifies the format of the log messages.

With these two lines in place, you can call `logging.debug()` to log debug-level messages in your software.

5. What are the two lines that your program must have in order to have logging.debug() send a logging message to a file named programLog.txt?

ANS :- To configure `logging.debug()` to send logging messages to a file named "programLog.txt", we need to have the following two lines in your program:

```python

import logging

logging.basicConfig(filename='programLog.txt', level=logging.DEBUG, format='%(levelname)s: %(message)s')

```

The first line imports the logging module, which provides the logging functionality. The second line configures the logging module to send messages to the specified file ("programLog.txt") using the `filename` parameter. It also sets the log level to `DEBUG` and specifies the format of the log messages.

With these two lines in place, you can use `logging.debug()` to log debug-level messages, and they will be written to the "programLog.txt" file.

6. What are the five levels of logging?

ANS :- The five levels of logging, in increasing order of severity, are:

1. DEBUG: Detailed information, typically used for debugging purposes.

2. INFO: General information about the program's execution.

3. WARNING: Indicates a potential issue or a warning that doesn't necessarily prevent the program from running.

4. ERROR: Indicates a more severe error or exception that may impact the program's functionality.

5. CRITICAL: Indicates a critical error or failure that may result in the program's termination.

These levels allow you to categorize and prioritize the logging messages based on their severity and importance.

7. What line of code would you add to your software to disable all logging messages?

ANS :- To disable all logging messages, you can add the following line of code at an appropriate place in your software:

```python

logging.disable(logging.CRITICAL)

```

This line sets the logging level to `CRITICAL`, which effectively disables all log messages with a severity level lower than or equal to `CRITICAL`. As a result, no log messages will be displayed or saved.

8.Why is using logging messages better than using print() to display the same message?

ANS :- Using logging messages is better than using print() because:

1. \*\*Flexibility\*\*: Logging allows you to control the verbosity of your program's output and choose different levels of logging (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL).

2. \*\*Granular control\*\*: You can selectively enable or disable specific log messages based on their importance or specific requirements.

3. \*\*Output destinations\*\*: Logging messages can be directed to various outputs like the console, log files, email, etc.

4. \*\*Timestamps and formatting\*\*: Logging provides built-in support for timestamps, log message formatting, and customization.

5. \*\*Integration\*\*: Logging seamlessly integrates with existing logging infrastructure and tools.

6. \*\*Performance\*\*: Logging is more efficient and optimized for handling large volumes of log messages.

9. What are the differences between the Step Over, Step In, and Step Out buttons in the debugger?

ANS :- The Step Over, Step In, and Step Out buttons are commonly used in debuggers to control the execution flow during debugging. Here are the differences between them:

1. \*\*Step Over\*\*: When we click the Step Over button, the debugger executes the current line of code and moves to the next line in the same function. If there is a function call on the current line, the entire function is executed, but the debugger does not step into the called function. It allows us to proceed with the execution without diving into the details of the called function.

2. \*\*Step In\*\*: Clicking the Step In button allows us to move the debugger to the next line of code, similar to Step Over. However, if there is a function call on the current line, the debugger steps into that function and allows us to debug its contents line by line. This is useful when we want to understand how a particular function works or track the execution flow within a function.

3. \*\*Step Out\*\*: The Step Out button is used to quickly exit from the current function being debugged. If we are currently inside a function and click Step Out, the debugger will execute the remaining lines of code in the current function and return to the calling function or the next line outside the current function. It helps us skip the detailed debugging within a function and move to the higher-level code.

10.After you click Continue, when will the debugger stop ?

ANS :- After clicking the Continue button in a debugger, the debugger will stop when one of the following conditions is met:

1. A breakpoint is encountered: If there is a breakpoint set at a specific line of code, the debugger will pause the execution when it reaches that line.

2. An exception is raised: If an exception occurs during the execution of the program, the debugger will stop at the line where the exception is raised, allowing you to inspect the error and the program state at that point.

3. The program terminates: When the program execution is completed and reaches the end, the debugger will stop.

11. What is the concept of a breakpoint?

ANS :- A breakpoint is a specific point in your code where the debugger pauses program execution, allowing you to inspect variables and step through the code line by line for debugging purposes.