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

**Ans.** Here's an assert statement that throws an AssertionError if the variable `spam` is a negative integer:

**assert spam >= 0 #spam should not be a negative integer**

This statement checks that `spam` is greater than or equal to 0. If `spam` is a negative integer, the assertion will fail and an `AssertionError` will be raised with the message "spam should not be a negative integer". If `spam` is a non-negative integer, the assertion will pass and execution will continue without any errors.

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.** Here's the assert statement that triggers an AssertionError if the variables `eggs` and `bacon` contain strings that are the same as each other, regardless of their case:

**assert eggs.lower() != bacon.lower() # eggs and bacon should not be the same**

This assertion compares the lowercase versions of `eggs` and `bacon` using the `!=` operator, and raises an AssertionError with the message "eggs and bacon should not be the same" if they are equal. The `.lower()` method is used to convert the strings to lowercase before comparing them.

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

**Ans.** One possible assert statement that throws an AssertionError every time is:

**assert False, "This assertion always fails."**

This statement checks whether the expression `False` evaluates to true, which is not the case, so an AssertionError is raised immediately. The optional second argument provides a custom error message that 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.** In order to call `logging.debug()` in software, two lines must be present:

**->** Import the logging module using `import logging` at the beginning of the file.

**->** Set up the logging configuration, including the desired log level and log file, using `logging.basicConfig()`.

For example:

**import logging**

**logging.basicConfig(level=logging.DEBUG, filename='example.log')**

**# Call logging.debug() to log debug-level messages**

**logging.debug('This is a debug message')**

The first line imports the `logging` module, while the second line sets up the logging configuration with a log level of `DEBUG` and a log file named `example.log`. Once the logging configuration is set up, the `logging.debug()` method can be called to log debug-level messages to the specified log file.

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.** In order to have the `logging.debug()` method send a logging message to a file named `programLog.txt`, your program must have the following two lines:

**import logging**

**logging.basicConfig(filename='programLog.txt', level=logging.DEBUG)**

The first line imports the `logging` module, while the second line sets up the logging configuration by specifying the name of the file to which the logs will be written (`programLog.txt` in this case) and the logging level (in this case, `logging.DEBUG`, which will enable logging messages at the `DEBUG` level and above to be written to the file). After these two lines, you can use the `logging.debug()` method (or other logging methods at different levels) to send messages to the file.

6. What are the five levels of logging?

**Ans.** The five levels of logging, in order of increasing severity, are:

**Debug:** This is the lowest level of logging and is used for messages that provide detailed information that may be useful for debugging purposes. These messages are typically only needed by developers during the development process.

**Info:** This level of logging is used for messages that provide information about normal program operation. These messages are typically used to track the progress of the application and provide insight into its behaviour.

**Warning:** This level of logging is used for messages that indicate a potential problem or issue that could lead to an error. These messages are used to alert developers and system administrators of potential issues that may require attention.

**Error:** This level of logging is used for messages that indicate an error has occurred in the application. These messages typically indicate a problem that needs to be addressed in order for the application to function properly.

**Critical:** This is the highest level of logging and is used for messages that indicate a critical error or failure that could cause the application to stop functioning. These messages are used to alert developers and system administrators of serious issues that require immediate attention.

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

**Ans.** To disable all logging messages in software, you can add the following line of code (in a language like Python) at the beginning of your program:

**import logging**

**logging.disable(logging.CRITICAL)**

This will disable all logging messages with a severity level of `CRITICAL` and above, effectively silencing all log output in your program. If you want to enable logging again, you can simply remove or comment out this line of code.

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

**Ans.** Using logging messages is better than using print() for displaying messages because:

**Flexibility:** Logging provides more flexibility in terms of where the messages are stored, their level of severity, and how they are formatted. This allows you to control the verbosity of the output, filter out noise, and route messages to different outputs based on their importance. With print() statements, you have limited control over the output and have to manually remove or modify the print statements if you want to change the logging behaviour.

**Maintainability:** With logging, you can disable or enable specific loggers or handlers, and modify the logging behaviour in runtime without changing the code. This makes it easier to debug and diagnose issues without having to reproduce the issue with a modified codebase. Print statements, on the other hand, are harder to disable or modify, and you may have to remove them manually after the debugging is done.

**Consistency:** Logging follows a standardized format and convention for displaying messages, making it easier to read and understand the output. In contrast, print statements may have different formatting, indentation, or verbosity, making it harder to compare and analyse output across different parts of the codebase.

**Performance:** When using print() statements, the output is sent directly to the standard output stream, which can have a negative impact on performance, especially when logging large amounts of data. With logging, you can route the messages to different streams, such as a file or a remote server, which can improve performance and reduce the impact on the end user.

Overall, using logging messages instead of print() statements provides more control, flexibility, and maintainability, making it easier to debug, diagnose, and maintain code in the long term.

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 found in a debugger, which is a tool used to identify and resolve errors in software code. Each of these buttons has a specific purpose and can be used to control the flow of the code during debugging.

**Step Over:** This button allows you to execute the current line of code and move to the next line without stepping into any function calls on that line. In other words, if the current line of code calls a function, the Step Over button will execute the function and return to the next line of code in the original function.

**Step In:** This button allows you to step into a function call on the current line of code. When you click the Step In button, the debugger will move to the first line of the function being called and allow you to debug that function as if it were a separate program.

**Step Out:** This button allows you to step out of the current function and return to the calling function. If you are currently debugging a function, the Step Out button will execute the remaining lines of the function and return to the line of code that called that function.

In summary, the Step Over button allows you to skip over function calls, the Step In button allows you to step into function calls, and the Step Out button allows you to step out of the current function and return to the calling function. By using these buttons together, you can effectively debug your code and identify and resolve errors.

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

**Ans.** The debugger will stop when the next breakpoint is reached.

11. What is the concept of a breakpoint?

**Ans.** In general, a breakpoint refers to a point at which something stops or changes direction. In various contexts, the concept of a breakpoint can have different meanings, but it often involves a critical moment, threshold, or turning point that triggers a significant shift or transition.

For example, in software development, a breakpoint is a marker set by a programmer that interrupts the execution of a program at a specific line of code, allowing them to inspect the program's state and variables. This helps with debugging and understanding the behaviour of the program.