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

We can use the assert statement in Python to validate a condition and throw an AssertionError if the condition is not met.

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

1. 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).

To compare two strings case-insensitively and trigger an AssertionError if they are the same, 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 checks if the variables eggs and bacon contain strings that are the same, regardless of case:

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

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

To create an assert statement that always throws an AssertionError, you can simply provide a condition that evaluates to False. Here's an example:

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

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

To use the logging.debug() function in your software, you need to include two lines: import the logging module and configure the logging level. Here's an example:

import logging

logging.basicConfig(level=logging.DEBUG)

1. 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?

To redirect logging.debug() messages to a file named programLog.txt, you need to include two additional lines along with the import of the logging module and configuration of the logging level.

import logging

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

1. What are the five levels of logging?

The five levels of logging in Python, from lowest to highest severity, are:

1. DEBUG: Detailed information, typically useful for debugging purposes. This level is the lowest severity and provides the most detailed log messages.
2. INFO: General informational messages that confirm the program is running as expected. These messages can help track the flow of the program.
3. WARNING: Indicates potential issues or situations that could lead to errors in the future but do not prevent the program from continuing execution.
4. ERROR: Indicates errors that caused the program to fail to perform a specific function or operation.
5. CRITICAL: Indicates critical errors or failures that may lead to the termination of the program. These errors require immediate attention.
6. What line of code would you add to your software to disable all logging messages?

logging.disable(logging.CRITICAL)

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

Using logging messages is generally considered better than using print() for displaying messages in software for several reasons:

1. Configurability: Logging provides a flexible and configurable logging system that allows you to control the level of messages, route them to different outputs, and format them in a desired way. You can easily adjust the logging level to show more or fewer messages without modifying the code. With print(), you would need to manually add or remove print statements or comment them out, which can be cumbersome.
2. Granularity: Logging allows you to log messages at different levels of severity, such as DEBUG, INFO, WARNING, ERROR, and CRITICAL. This enables you to provide detailed information during development and debugging (DEBUG), track program flow (INFO), and highlight potential issues or errors (WARNING, ERROR, CRITICAL). With print(), it's harder to differentiate the importance or severity of messages, and you may need to manually add conditional checks to control when certain print statements should execute.
3. Flexibility: Logging provides various handlers to route log messages to different destinations, such as the console, files, databases, or external services. This allows you to separate the concern of message output from the application logic. With print(), you are limited to displaying messages in the console, and redirecting or capturing the output requires additional code modifications.
4. Performance: Logging is generally more performant than print() statements, especially when dealing with a large volume of messages. Logging allows you to control the logging level, so you can avoid unnecessary computations or string formatting for messages that are not needed. On the other hand, print() statements are executed unconditionally, potentially impacting performance.
5. Production-readiness: Logging is designed for production environments and provides features like log rotation, log file size management, and log rotation based on time or size. These features help maintain log files efficiently and prevent them from consuming excessive disk space. With print(), you would need to implement such features manually.

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

1. Step Over:
   * The Step Over button allows you to execute the current line of code and move to the next line in the current function.
   * If the current line contains a function call, the debugger will execute the entire function without stepping into it, treating it as a single line.
   * Use Step Over when you want to execute the current line and move forward without stepping into the details of function calls.
2. Step In:
   * The Step In button allows you to step into the next line of code, even if it is a function call.
   * If the current line contains a function call, the debugger will step into that function, and execution will continue within the called function.
   * Use Step In when you want to delve into the details of a function call and follow the execution line-by-line within that function.
3. Step Out:
   * The Step Out button allows you to step out of the current function and continue execution at the line immediately following the function call that brought you into the current function.
   * If you are currently debugging within a function and want to quickly return to the calling context, you can use Step Out to skip the remaining lines in the current function.
   * Use Step Out when you want to quickly navigate back to the calling code and skip the remaining lines in the current function.

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

1. A breakpoint is encountered: If you have set breakpoints at specific lines of code, the debugger will pause execution when it reaches a breakpoint. This allows you to inspect variables and step through the code from that point onwards.
2. An exception is raised: If an exception occurs during the execution of your code and it is not caught and handled, the debugger will pause at the line where the exception is raised. This allows you to examine the traceback and investigate the cause of the exception.
3. The program finishes executing: If your program reaches its end and completes execution, the debugger will stop.

11. What is the concept of a breakpoint?

A breakpoint is a designated point in your code where the debugger will pause the execution of the program. It allows you to halt the program's execution at a specific line or lines of code to inspect variables, evaluate expressions, and analyze the program's state at that particular moment.

By setting breakpoints, you can interactively debug your code and step through it line by line, gaining insights into how the program behaves and identifying any issues or bugs. Breakpoints are particularly useful for understanding complex control flow, identifying incorrect variable values, or investigating unexpected behavior.