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

Ans:

Certainly! Here's how you can create an assert statement in Python to check if the variable spam is a negative integer:

assert spam >= 0, "spam should be a non-negative integer"

This assertion will raise an AssertionError if the value of spam is less than 0. The message accompanying the AssertionError will indicate that spam should be a non-negative integer.

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:

You can achieve this by converting both strings to lowercase (or uppercase) and then comparing them. Here's the assert statement:

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

This assertion will raise an AssertionError if the lowercase versions of eggs and bacon are equal, indicating that they should not be the same (case insensitive).

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

Ans:

To create an assert statement that throws an AssertionError every time, you can simply assert a condition that is always false. One common way to do this is by using the False keyword. Here's the assert statement:

assert False, "This assertion always triggers an AssertionError"

This will always raise an AssertionError with the specified message.

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

Ans:

To use logging.debug() in your Python code, you need to include two lines at the beginning of your script or module:

import logging

logging.basicConfig(level=logging.DEBUG)

The first line imports the logging module, and the second line configures the logging system to display debug messages (and messages of all levels above debug) to the console. You can adjust the logging level as needed, depending on the desired verbosity of your logs.

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 redirect logging messages from logging.debug() to a file named programLog.txt, you need to include two lines of code:

import logging

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

The first line imports the logging module, and the second line configures the logging system to write messages to the file programLog.txt at the DEBUG level and above. You can adjust the filename and logging level as needed for your specific requirements.

6. What are the five levels of logging?

Ans:

The Python logging module defines five standard levels of logging:

DEBUG: Detailed information, typically used for debugging purposes.

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

WARNING: Indicates a potential issue or unexpected behavior that does not necessarily interrupt the program's execution.

ERROR: Indicates a serious issue or error that has occurred during the program's execution but does not necessarily lead to termination.

CRITICAL: Indicates a critical error that may lead to the termination of the program or severe consequences if not addressed.

These levels allow developers to categorize and filter log 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 in your software, you can add the following line:

logging.disable(logging.CRITICAL)

This line disables all logging messages with severity levels of CRITICAL and below. As a result, no logging messages will be displayed or saved to any log files.

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

Ans:

Using logging messages is generally better than using print() for several reasons:

Logging Levels: Logging allows you to categorize messages by severity levels (DEBUG, INFO, WARNING, ERROR, CRITICAL), making it easier to filter and control the verbosity of messages based on the application's needs. print() statements don't provide this level of granularity.

Configurability: With logging, you can easily configure where and how log messages are handled, such as printing to the console, writing to files, sending emails, or even logging to a database. This flexibility is not available with print() statements, which are limited to outputting to the console.

Performance: Logging is generally more efficient than print() statements, especially in production environments. print() statements can significantly impact performance, especially in high-throughput applications, due to the overhead of writing to the console.

Maintenance: Using logging promotes cleaner and more maintainable code. It separates the concerns of generating output from the actual logic of your program. This makes it easier to modify or remove logging statements without affecting the program's functionality.

Debugging: Logging allows for more effective debugging by providing contextual information along with log messages, such as timestamps, module names, and line numbers. This can be invaluable in diagnosing issues, especially in complex systems.

Overall, logging provides a more robust and flexible solution for managing and analyzing program output compared to print() statements.

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

Ans:

Using logging messages is often preferred over using print() statements for several reasons:

Granular Control: With logging, you can control the verbosity of messages by setting different logging levels. This allows you to adjust the amount of detail in your logs without modifying the code.

Flexibility: Logging provides more flexibility in where and how log messages are output. You can easily redirect logs to files, streams, or other destinations without changing the application's code.

Filtering: Logging allows you to filter messages based on severity levels or other criteria, making it easier to focus on specific types of messages during debugging or troubleshooting.

Performance: While print() statements can be used for debugging, they are typically not removed from the code once debugging is complete. This can impact performance in production environments. In contrast, logging can be easily disabled or configured to minimize its impact on performance.

Standardization: Logging follows a standardized approach and integrates well with other logging libraries and frameworks. This makes it easier to collaborate with other developers and maintain consistency across projects.

Overall, using logging messages provides a more robust and maintainable approach to debugging and monitoring applications compared to using print() statements.

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

Ans:

After clicking "Continue" in a debugger, the debugger will not stop until one of the following conditions is met:

A breakpoint is encountered.

The program execution completes.

An unhandled exception occurs.

The program explicitly exits or terminates.

A specific condition specified in the debugger's configuration is met (such as hitting a certain line number or encountering a particular variable state).

In essence, clicking "Continue" allows the program to continue running without interruption until it reaches one of these stopping conditions.

11. What is the concept of a breakpoint?

Ans:

A breakpoint is a point in the code where program execution temporarily pauses, allowing the developer to inspect the program's state, variables, and control flow during debugging. Breakpoints are a fundamental tool in debugging because they enable developers to analyze the behavior of their code step by step.