Solutions

1. assert spam >= 0, 'spam should be a non-negative integer'.

2. assert eggs.lower() != bacon.lower(), "eggs and bacon strings are the same, even if their cases are different"

3. assert False, "This assertion will always fail."

4. import logging
   logging.basicConfig(level=logging.DEBUG)

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

6. The logging module in Python provides five standard levels of logging, listed below in order of increasing severity:

   DEBUG: Detailed information for debugging purposes.
   INFO: General information about what's happening in the program.
   WARNING: An indication that something unexpected or potentially problematic has happened.
   ERROR: An indication that an error or exceptional condition has occurred, but the program can still continue running.
   CRITICAL: An indication that a critical error or exceptional condition has occurred and the program may not be able to continue running.

   You can set the logging level in your application to control the amount of information that gets logged. For example, if you set the logging level to WARNING, only messages with a severity of WARNING, ERROR, or CRITICAL will be logged. Messages with a severity of DEBUG or INFO will be ignored.

7. logging.disable(logging.CRITICAL)

8.

Using logging messages is generally considered better than using print() statements to display messages in your code for several reasons:

Flexible configuration: Logging messages can be configured to be displayed in different ways (e.g., to a file or to the console), at different levels of detail, and for different parts of your codebase. With print() statements, you have less control over how and when messages are displayed.

Granular control: With logging, you can control which parts of your codebase log messages and at what level of detail. This allows you to fine-tune the amount of information that is logged and reduce noise in your logs. With print() statements, all messages are printed to the console indiscriminately.

Performance: Logging messages are generally faster than print() statements because they don't involve writing to the console, which can be slow. Additionally, logging messages can be disabled entirely, which can improve performance in cases where logging is not needed.

Separation of concerns: Logging is a separate concern from the functionality of your code. By separating logging from your code's main functionality, you can keep your code more organized and maintainable.

Debugging: Logging messages can provide more detailed information about the state of your code at runtime, which can be helpful for debugging purposes. print() statements can also provide this information, but logging provides a more flexible and granular way to log this information.

9. **Step Over**: This button executes the current line of code and then moves to the next line of code. If the current line of code contains a function call, the entire function is executed before moving to the next line of code. In other words, Step Over allows you to execute a line of code without diving into the details of any function calls within that line.

Step In: This button allows you to step into a function call and execute the code inside that function. If the current line of code does not contain a function call, Step In behaves the same as Step Over.

Step Out: This button allows you to step out of the current function and continue execution at the line of code that called the function. This can be useful when you are debugging a complex function and want to quickly return to the calling code to see how it interacts with the results of the function.

10. It depends on the debugger you are using and how it is configured. In general, when you click Continue in a debugger, it will continue executing the code until it reaches a breakpoint or an exception is thrown. If an exception is thrown, the debugger will stop at the line of code that caused the exception If there are no exceptions or breakpoints, the debugger will continue executing until the program completes or another breakpoint is encountered.

11. A breakpoint is a point in your code where the execution of your program can be paused during debugging. When you set a breakpoint, the debugger will stop the execution of your program when it reaches that point, allowing you to inspect the state of your program at that point in time.

    Setting a breakpoint is useful when you need to understand what's going on in your program at a particular point in time or when you want to isolate a particular section of code for debugging purposes. You can set breakpoints in your code using a debugger, and when the execution of the program reaches the breakpoint, the debugger will pause the program, allowing you to inspect variables, call stack, and other aspects of the program state.