1. Create an assert statement that throws an AssertionError if the variable spam is a negative integer.
   Solution 1: assert x >= 0, 'Negative numbers are not allowed'

   ```
   x = int(input("Enter a positive integer: "))
   assert x >= 0, 'Negative numbers are not allowed'
   print('Number accepted')
   ```

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).
   Solution 2: Two ways to do this are:
   - assert eggs.lower() != bacon.lower(), 'The eggs and bacon variables are the same!'
   - assert eggs.upper() != bacon.upper(), 'The eggs and bacon variables are the same!'

3. Create an assert statement that throws an AssertionError every time.
   Solution 3: assert False, 'This assertion always triggers.'

4. What are the two lines that must be present in your software in order to call logging.debug()?
   Solution 4:
   ```
   import logging
   logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
   %(levelname)s - %(message)s')
   ```

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?
   Solution 5:
   ```
   import logging
   logging.basicConfig(filename='programLog.txt', level=logging.DEBUG,
   format=' %(asctime)s - %(levelname)s - %(message)s')
   ```

6. What are the five levels of logging?
   Solution 6: The five levels of logging are:
   - DEBUG
   - INFO
   - WARNING
   - ERROR
   - CRITICAL

7. What line of code would you add to your software to disable all logging messages?
   Solution 7: logging.disable(logging.CRITICAL)

8. Why is using logging messages better than using print() to display the same message?
   Solution 8:
   - One of the advantages of using the logging module to track our codes is the ability to format the messages based on our needs.
   - We can disable logging messages without removing the logging function calls.
   - We can selectively disable lower-level logging messages.
   - We can create logging messages.
   - Logging messages provides a timestamp.

9. What are the differences between the Step Over, Step In, and Step Out buttons in the debugger?
   Solution 9: The Step button will move the debugger into a function call. The Over button will quickly execute the function call without stepping into it. The Out button will quickly execute the rest of the code until it steps out of the function it currently is in.
   - **Step over –** An action to take in the debugger that will step over a given line. If the line contains a function the function will be executed and the result returned without debugging each line.
   - **Step into –** An action to take in the debugger. If the line does not contain a function it behaves the same as "step over" but if it does the debugger will enter the called function and continue line-by-line debugging there.
   - **Step out –** An action to take in the debugger that returns to the line where the current function was called.

10. After you click Continue, when will the debugger stop?
    Solution 10: Continue execution, only stop when a breakpoint is encountered. In other words, Continue is an action to take in the debugger that will continue execution until **the next breakpoint is reached or the program exits**.

11. What is the concept of a breakpoint?
    Solution 11: A breakpoint is a setting on a line of code that causes the debugger to pause when the program execution reaches the line. To set a breakpoint in IDLE, we need to right-click the line and select Set Breakpoint from the context menu.