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

:- assert spam >= 0, '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).

:- assert eggs.lower() == bacon.lower(), 'eggs and bacon should have different values'

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

:- assert False, 'This assertion always fails'

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

:- 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?

:- 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?

:- DEBUG: Detailed information, typically of interest only when diagnosing problems.

INFO: General information about the program's execution.

WARNING: An indication that something unexpected or potentially problematic has happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.

ERROR: An indication that something more serious has gone wrong in the software. The software may still be able to continue running.

CRITICAL: A very serious error, indicating that the program itself may be unable to continue running.

7. 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?

:- Flexibility and configurability: With logging, you can easily control the output destination, formatting, and severity level of your messages without modifying the code.

Granularity and filtering: With logging, you can categorize your messages by severity level and/or by logger name, which allows you to selectively enable or disable certain types of messages based on your needs.

Performance and thread-safety: Logging is optimized for performance and thread-safety, which means that it has minimal impact on your application's performance and can handle multiple threads and processes simultaneously without conflicts or race conditions. In contrast, print() statements can slow down your program, especially if they are executed frequently or in tight loops, and may produce unexpected results in multithreaded or multiprocess environments.

Good programming practice: Using logging messages instead of print() statements is considered a good programming practice, as it separates the concerns of logging and application logic, and makes your code more modular, testable, and maintainable. It also makes it easier to integrate your application with third-party logging frameworks or tools, such as Splunk, Graylog, or ELK, which can provide advanced analytics, visualization, and alerting capabilities.

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

:-Step Over: This button executes the current line of code and moves to the next line in the same function. If the next line contains a function call, the function is executed, but the debugger does not step into it. Instead, the function is executed as a single step, and the debugger moves to the next line after the function call returns.

Step In: This button executes the current line of code and moves into the first line of the next function call, if there is one. This means that the debugger enters the called function and pauses at its first line, allowing you to debug its implementation.

Step Out: This button continues the execution of the current function until it returns, and then pauses at the next line in the calling function. This means that the debugger exits the current function and returns control to its caller, allowing you to step through the code that follows the function call. The Step Out button is useful for quickly skipping over the details of a function and returning to the higher-level code.

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

:- When you click "Continue" in a debugger, the debugger will continue executing your code until it hits another breakpoint, encounters an exception or error, or reaches the end of the program.

11. What is the concept of a breakpoint?

:- A breakpoint is a specific point in your program's code where you want the debugger to pause or stop execution. It's a useful tool for debugging because it allows you to examine the state of your program at a specific point in time, and to step through the code line by line to understand what's happening.