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

Ans:   
Certainly! Here's an example of an assert statement that throws an AssertionError if the variable spam is a negative integer:

spam = -5 assert spam >= 0, "spam should not be a negative integer"

In the above example, the assert statement checks if the value of the variable **spam** is greater than or equal to zero. If **spam** is a negative integer, the assert statement raises an **AssertionError** with the provided error message: "spam should not be a negative integer".

You can replace **-5** with the variable or expression you want to check for negativity. If the condition in the assert statement evaluates to **False**, the **AssertionError** will be raised, indicating that the assumption has been violated

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 : To compare the variables eggs and bacon while ignoring the case sensitivity, you can use the lower() method to convert the strings to lowercase before comparing them. Here's an example of an assert statement that triggers an AssertionError if the lowercase versions of eggs and bacon are the same:

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

In the above example, **eggs.lower()** and **bacon.lower()** convert the values of **eggs** and **bacon** to lowercase, respectively. The assert statement checks if the lowercase versions of the two variables are different. If they are the same, the assert statement raises an **AssertionError** with the provided error message: "eggs and bacon should not have the same value (ignoring case)".

By comparing the lowercase versions of the strings, the assert statement treats strings with different cases as equal, as specified in the requirements

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

Ans : If you want an assert statement that throws an AssertionError every time it is encountered, you can simply use the assert statement without any condition. Here's an example:

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

In the above example, the **assert** statement is followed by **False**, which is always evaluated as **False**. As a result, the **AssertionError** is raised, and the provided error message, "This assert statement always triggers an AssertionError", is displayed.

Keep in mind that using an **assert** statement that always raises an **AssertionError** can be useful for debugging or testing purposes but should generally be avoided in production code

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

Ans : To call the logging.debug() function in your software, you need to include the following two lines:

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

These lines are required to configure the logging system and set the desired logging level.

Explanation:

1. **import logging**: The **import** statement is used to import the **logging** module, which provides the logging functionality in Python.
2. **logging.basicConfig(level=logging.DEBUG)**: The **basicConfig()** function is called to configure the logging system. It sets the root logger's level to **DEBUG**, which enables logging at the **DEBUG** level and all higher levels (e.g., **INFO**, **WARNING**, **ERROR**, **CRITICAL**).

Once these two lines are present in your software, you can call **logging.debug()** to output log messages at the **DEBUG** level. These log messages can be captured and handled according to your logging configuration, such as writing to a file, displaying on the console, or sending to a logging server.

It's important to note that the **basicConfig()** function should typically be called once in your application, preferably at the beginning, to configure the logging system.

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 have logging.debug() send logging messages to a file named "programLog.txt," you need to include the following two lines in your program:

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

These lines configure the logging system to output messages at the **DEBUG** level and direct them to the specified file.

Explanation:

1. **import logging**: The **import** statement imports the **logging** module, which provides the logging functionality in Python.
2. **logging.basicConfig(filename='programLog.txt', level=logging.DEBUG)**: The **basicConfig()** function is called to configure the logging system. By providing the **filename** argument, you specify the file to which the logging messages will be directed. In this case, it's set to "programLog.txt". The **level** argument is set to **logging.DEBUG** to enable logging at the **DEBUG** level and all higher levels (e.g., **INFO**, **WARNING**, **ERROR**, **CRITICAL**).

Once these two lines are present in your program, you can use **logging.debug()** to send log messages, and they will be written to the "programLog.txt" file.

It's important to note that the **basicConfig()** function should typically be called once in your application, preferably at the beginning, to configure the logging system.

6. What are the five levels of logging?

Ans : The five levels of logging in Python, in increasing order of severity, are as follows:

1. **DEBUG**: The **DEBUG** level is used for detailed diagnostic information. It is typically used for troubleshooting and providing the most detailed logging output.
2. **INFO**: The **INFO** level is used to confirm that things are working as expected. It provides informational messages that indicate the progress and status of the application.
3. **WARNING**: The **WARNING** level is used to indicate potential issues or unexpected situations that are not necessarily errors. It signifies a condition that may require attention but does not prevent the program from functioning.
4. **ERROR**: The **ERROR** level is used to report errors that caused a particular operation to fail or prevented the program from continuing. These messages indicate a problem that should be addressed.
5. **CRITICAL**: The **CRITICAL** level is the most severe level of logging. It is used to indicate critical errors that may lead to the termination of the application or significant failure in its functionality. These messages highlight errors that require immediate attention.

These logging levels provide a way to categorize and prioritize the severity of log messages. By setting the logging level appropriately, you can control which messages are displayed or saved based on their 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 of code:

import logging logging.disable(logging.CRITICAL)

The **logging.disable(level)** function is used to disable logging messages at or below the specified **level**. By passing **logging.CRITICAL**, which is the highest level, all logging messages, including critical messages, errors, warnings, info, and debug messages, will be disabled.

After adding this line of code, no logging messages will be displayed or saved. Keep in mind that this should be used cautiously and only in specific situations where logging is not needed or desired.

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

Ans :   
Using logging messages instead of print() statements offers several advantages:

1. Granular control: Logging provides different levels of severity, such as DEBUG, INFO, WARNING, ERROR, and CRITICAL. By using the appropriate logging level, you can control the verbosity of the output. In contrast, print() statements don't offer built-in levels of severity, so it can be more challenging to selectively display or filter messages based on their importance.
2. Flexible configuration: With logging, you can easily configure the behavior of logging messages, such as directing them to different outputs (e.g., console, file, network), formatting the messages, and adjusting the log levels during runtime. This flexibility allows you to adapt the logging behavior to different environments and scenarios without modifying your code.
3. Easy enable/disable: Logging messages can be easily enabled or disabled by changing the logging configuration, such as adjusting the log level or directing the output to a different location. This makes it convenient to control the amount of output and selectively enable or disable logging messages during development, testing, and production stages, without modifying the code itself.
4. Diagnostic context: Logging provides additional contextual information, such as timestamps, module names, line numbers, and stack traces (if configured), which can be valuable for troubleshooting and debugging purposes. This context helps in understanding the flow and behavior of the application, especially in complex scenarios.
5. Integration with libraries and frameworks: Many libraries and frameworks utilize logging extensively, and they often provide hooks and mechanisms for customizing the logging behavior. By using the logging module, you can seamlessly integrate with these libraries and frameworks and benefit from their logging features and configurations.

Overall, using logging messages instead of print() statements provides better control, configurability, and flexibility for managing and analyzing application output, making it a more robust and maintainable approach for logging and debugging purposes.

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

Ans : The "Step Over," "Step In," and "Step Out" buttons are commonly found in debugger interfaces and serve different purposes when stepping through code during debugging sessions. Here are their differences:

1. Step Over:
   * The "Step Over" button allows you to execute the current line of code and then move to the next line, without diving into the details of any function calls on that line.
   * If the current line contains a function call, the entire function is executed as a single step, and the debugger moves to the next line in the current scope.
   * This button is useful for quickly moving through code without stepping into function calls or examining their internal workings.
   * Shortcut key: F10
2. Step In:
   * The "Step In" button allows you to execute the current line of code and move into the details of any function calls present on that line.
   * If the current line contains a function call, the debugger will step into that function and allow you to debug through its internal code.
   * This button is useful when you want to closely examine the execution of a specific function and step through its statements line by line.
   * Shortcut key: F11
3. Step Out:
   * The "Step Out" button allows you to continue stepping through code until you exit the current function being debugged.
   * If you are currently inside a function, the debugger will execute the remaining lines of that function without stepping into any further function calls within it.
   * Once the current function is fully executed, the debugger returns to the caller function or the line of code that invoked the current function.
   * This button is useful when you want to quickly move out of the current function and return to the higher-level context.
   * Shortcut key: Shift+F11

These buttons provide control over the flow of execution during debugging and help in understanding and troubleshooting code behavior. The specific behavior of these buttons may vary slightly depending on the debugger and the programming language being used.

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

Ans : After clicking the "Continue" button in the debugger, the debugger will stop at the next breakpoint encountered in the code or when the program execution completes.

Breakpoints are markers that you set in your code to pause the execution at a specific line or condition for inspection and debugging. When the debugger encounters a breakpoint during the "Continue" operation, it will pause execution and allow you to examine the program state and step through the code from that point onward.

If there are no further breakpoints set, the debugger will continue running until the program reaches its end or encounters an error. At that point, the debugger will automatically halt and display the relevant information for analysis.

It's important to note that the behavior of the "Continue" operation may vary slightly depending on the specific debugger and its configuration, as well as the programming language being used. However, the general principle remains the same: the debugger will run continuously until encountering a breakpoint or the end of the program's execution.

11. What is the concept of a breakpoint?

Ans:   
A breakpoint is a marker or a specific point in your code where the debugger pauses program execution, allowing you to inspect and analyze the program's state at that particular moment. When the debugger encounters a breakpoint during debugging, it stops execution, giving you an opportunity to examine variables, step through the code, and investigate the program's behavior.

Breakpoints are used as a debugging tool to help identify and resolve issues in code by providing a controlled and interactive environment for troubleshooting. They allow you to examine the state of the program, inspect variable values, and step through the code line by line or at specified intervals. By pausing the program's execution at a breakpoint, you can observe the program's behavior, identify any unexpected values or errors, and gain insights into the flow of execution.

When you set a breakpoint, you typically indicate a specific line of code or a condition that triggers the breakpoint. The debugger then monitors the execution and halts when it reaches the specified breakpoint. Once paused, you can use the debugger's features to inspect variables, modify values, step through the code, and gather information to diagnose and fix issues.

Breakpoints are powerful tools for understanding and debugging code, allowing developers to gain insights into the program's runtime behavior and identify the source of problems more effectively.