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

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

spam = -5

assert spam >= 0, "Spam must be a positive integer, but got {}".format(spam)

This assert statement checks if spam >= 0. If the condition is True, the program continues normally. If the condition is False, an AssertionError is raised with the message "Spam must be a positive integer, but got -5". The message can be customized to provide additional information about the error.

**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).**

Here's an example of an assert statement that triggers an AssertionError if the variables eggs and bacon contain strings that are the same, even if their cases are different:

eggs = "Hello"

bacon = "hello"

assert eggs.lower() != bacon.lower(), "Eggs and bacon cannot contain the same string, even if their cases are different"

This assert statement checks if the lowercase versions of the eggs and bacon variables are not equal to each other. If the lowercase versions are not equal, the program continues normally. If the lowercase versions are equal, an AssertionError is raised with the message "Eggs and bacon cannot contain the same string, even if their cases are different".

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

Here's an example of an assert statement that will always throw an AssertionError:

assert False, "This assert statement will always trigger an AssertionError"

This assert statement checks the condition False, which is always False. As a result, the AssertionError with the message "This assert statement will always trigger an AssertionError" will always be raised.

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

To call logging.debug() in Python, two lines must be present in your software:

1)Import the logging module:

import logging

2)Configure the logging system:

logging.basicConfig(level=logging.DEBUG)

The basicConfig method sets up the logging system with the desired log level, which in this case is DEBUG. With this setup in place, calls to logging.debug() will log messages at the DEBUG level.

**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?**

To have logging.debug() send a logging message to a file named programLog.txt, two lines must be present in your program:

1)Import the logging module:

import logging

2) Configure the logging system:

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

The basicConfig method sets up the logging system with the desired log level, which in this case is DEBUG, and the file name, which in this case is programLog.txt. With this setup in place, calls to logging.debug() will log messages at the DEBUG level to the specified file.

**6. What are the five levels of logging?**

In Python, the five levels of logging are:

1. DEBUG
2. INFO
3. WARNING
4. ERROR
5. CRITICAL

These levels provide a way to categorize log messages based on their severity or importance. For example, DEBUG messages are meant for developers to use when debugging their code, while ERROR messages are meant to signal that a serious problem has occurred. The higher the level, the more serious the message. When configuring the logging system, you can specify the minimum level of messages that you want to log, and messages of that level or higher will be recorded.

**7. What line of code would you add to your software to disable all logging messages?**

In Python, you can use the logging module to control the logging messages. To disable all logging messages, you can set the logging level to "CRITICAL", which will only show log messages that are critical in nature:

import logging

logging.basicConfig(level=logging.CRITICAL)

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

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

Separation of Concerns: Logging is designed to provide information about what a software is doing, while print() statements are intended to provide information to the developer during development. By using logging, you can separate the debug information from the main code, making it easier to manage and maintain.

Flexibility: The logging module provides a lot of flexibility in terms of what to log, where to log, and how to log. You can log messages to different handlers, such as a file, syslog, or the console, and you can choose different logging levels, such as DEBUG, WARNING, or ERROR, to control the verbosity of your logs.

Scalability: When using print() statements, it can become difficult to manage large amounts of debugging information. The logging module, on the other hand, can easily handle large amounts of data, and you can configure it to log only the information that you need.

Persistence: Logging messages are usually stored in a file, making it easy to review them later. This is especially useful when debugging an issue that may not be immediately apparent.

Standardization: The logging module provides a standard way of logging messages, which makes it easier for developers to understand what is happening in a codebase, especially when working on large projects with multiple contributors.

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

The "Step Over", "Step In", and "Step Out" buttons in the debugger in Python refer to different ways of stepping through the code. These buttons are used when debugging to control the execution of the code and inspect its behavior.

"Step Over": This button allows you to step over a function call, meaning that the debugger will execute the entire function without stopping and will return to the next line of code after the function call.

"Step In": This button allows you to step into a function call, meaning that the debugger will stop at the first line of the called function and allow you to step through the function line by line.

"Step Out": This button allows you to step out of the current function and continue execution at the next line of code after the current function returns.

By using these buttons, you can control the execution of the code and inspect its behavior, making it easier to identify and fix bugs in your code.

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

The debugger will stop when it encounters a breakpoint in the code. A breakpoint is a line of code where the execution of the program will stop and the debugger will become active, allowing you to inspect the state of the program and step through the code.

You can set a breakpoint in the code by clicking in the margin next to the line of code or by using the breakpoint() function in the code. When the debugger reaches a breakpoint, it will stop the execution of the code and allow you to inspect the state of the program and step through the code using the "Step Over", "Step In", and "Step Out" buttons.

Once you have finished debugging and want to continue the execution of the code, you can click the "Continue" button, and the debugger will resume the execution of the code until it reaches the next breakpoint or the end of the program.

**11. What is the concept of a breakpoint?**

In Python, a breakpoint is a feature of the debugger that allows you to pause the execution of the program at a specific line of code. When the program reaches a breakpoint, the execution will stop and the debugger will become active, allowing you to inspect the state of the program, step through the code line by line, and inspect the values of variables.

Breakpoints are useful for finding and fixing bugs in your code by allowing you to inspect the behavior of the program at a specific point in time. You can set a breakpoint by clicking in the margin next to the line of code in your development environment or by using the breakpoint() function in the code.

Once you have finished debugging, you can continue the execution of the code by clicking the "Continue" button in the debugger. The program will then continue to run until it reaches the next breakpoint or the end of the program.