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

assert spam >= 0, "spam must 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 strings cannot be the same"

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

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

**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)

**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)

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

DEBUG - It is used to provide detailed information and only use it when there is diagnosing problems.

INFO - It provides the information regarding that things are working as we want.

WARNING - It is used to warn that something happened unexpectedly, or we will face the problem in the upcoming time.

ERROR - It is used to inform when we are in some serious trouble, the software hasn't executed some programs.

CRITICAL - It specifies the serious error, the program itself may be incapable of remaining executing.

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

import logging

logging.disable(logging.CRITICAL)

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

* We can configure the logging system to output messages to different files without modifying your code.
* The logging module offers different log levels that allow you to categorize and filter messages based on their severity. With print() we need to add conditional statements manually to control the verbosity which can be error-prone.
* With logging we can selectively enable or disable log messages based on the desired level of detail or the specific components of code. In contrast, print() statements are typically scattered throughout the code and may require manual modification to disable or enable specific debug output.

**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 are commonly found in debuggers and are used for controlling the execution flow during debugging.

1. Step Over:

- When we click the Step Over the debugger executes the current line of code and moves to the next line.

- If the current line contains a function call, the debugger does not step into the function and instead executes it as a single step.

- This button is useful when we want to move forward in the code execution without diving into the details of each function call.

2. Step In:

- The Step In button allows us to dive into the details of a function call.

- When we click Step In, the debugger moves to the next line of code, just like Step Over. However, if the current line contains a function call, the debugger steps into that function and switches to its code execution.

- This button is handy when we want to analyze the internal workings of a function or investigate the behavior at a more granular level.

3. Step Out:

- The Step Out button is used to quickly execute the remaining lines of the current function and return to the calling function.

- If we are already inside a function and want to skip the detailed execution of that function, we can click Step Out.

- The debugger continues executing the lines of the current function until it reaches the `return` statement or the end of the function. It then moves to the line of code that called the function.

- This button is helpful when we are inside a function but want to quickly return to the higher-level context without going through all the details of the function.

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

After clicking the Continue button in a debugger, the debugger will stop when it reaches one of the following conditions:

Breakpoint: If there is an active breakpoint set at a specific line of code, the debugger will stop when it reaches that line. Breakpoints are markers that pause the execution of the program at a specific location, allowing us to examine the program's state and variables.

Exception: If an unhandled exception occurs during the program's execution, the debugger will stop and display the details of the exception. This allows us to inspect the state of the program when the exception was raised and analyze the cause of the error.

Program completion: The debugger will also stop when the program execution completes.

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

In software development and debugging, a breakpoint is a specific point in code where we want the program's execution to pause temporarily.

When a breakpoint is set, the debugger will halt the execution of the program when it reaches that line of code or condition specified by the breakpoint. Once paused at the breakpoint, we can inspect variables, evaluate expressions, step through the code, and analyze the program's behavior in a controlled manner.