**Assignment – 11**

Q1. 1. Create an assert statement that throws an AssertionError if the variable spam is a negative Integer.

Sol1.

**Code:**

assert spam >= 0, "The variable 'spam' cannot be a negative integer."

In this statement, the expression spam >= 0 checks if the value of spam is greater than or equal to zero. If it evaluates to False, indicating that spam is a negative integer, an AssertionError will be raised with the provided error message: "The variable 'spam' cannot be a negative integer."

It's important to note that assert statements are typically used for debugging and testing purposes. They can be disabled globally, so they may not always be reliable for handling errors in production code.

Q2. 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).

Sol. Certainly! Here's an example of an assert statement that triggers an Assertion Error if the variables eggs and bacon contain strings that are the same as each other, regardless of case sensitivity:

assert eggs.lower() != bacon.lower(), "The variables 'eggs' and 'bacon' cannot contain the same string."

In this statement, the **.**lower() method is used to convert both eggs and bacon to lowercase. Then, the expression eggs.lower() != bacon.lower() compares the lowercase versions of the strings to check if they are different. If they are the same, an AssertionError will be raised with the provided error message: "The variables 'eggs' and 'bacon' cannot contain the same string."

By using the .lower() method, the assert statement considers strings with different cases (e.g., "hello" and "HELLO") as the same, fulfilling the requirement specified in the question.

Q3. Create an assert statement that throws an AssertionError every time.

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

In this statement, the expression **False** is used, which always evaluates to **False**. As a result, the assert statement will always raise an AssertionError. The provided error message, "This assert statement always triggers an AssertionError," is optional but can be included to provide additional information about why the AssertionError is being raised.

Q4. What are the two lines that must be present in your software in order to call logging.debug()?

Sol. import logging

logging.basicConfig(level=logging.DEBUG)

The first line import logging imports the logging module, which provides the necessary functions and classes for logging in Python.

The second line logging.basicConfig(level=logging.DEBUG) configures the logging system to display debug-level messages. By setting the level parameter to logging.DEBUG, you ensure that debug-level log messages will be processed and displayed. You can adjust the logging level based on your requirements (e.g., logging.INFO, logging.WARNING, logging.ERROR, etc.).

Q5. What are the two lines that your program must have in order to have logging.debug() send an logging message to a file named programLog.txt?

Sol. import logging

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

The first line import logging imports the logging module, which provides the necessary functions and classes for logging in Python.

The second line logging.basicConfig(filename='programLog.txt',level=logging.DEBUG) configures the logging system to log messages to a file named "programLog.txt." The filename parameter specifies the name of the file where the log messages will be stored. By setting the level parameter to logging.DEBUG, you ensure that debug-level log messages will be processed and written to the file.

Q6. What are the five levels of logging?

Sol. The five levels of logging, in increasing order of severity, are:

DEBUG: This level is used for detailed debugging information. It is typically used during development to provide granular information about the program's execution.

INFO: This level is used to convey general information about the program's execution. It is often used to indicate important milestones or significant events that occur during the program's operation.

WARNING: This level is used to indicate potential issues or situations that could lead to problems in the future. It is typically used to highlight non-critical issues or exceptions that were handled gracefully.

ERROR: This level is used to report errors or exceptions that occur during the program's execution. It indicates that a specific problem occurred, but the program can still continue running.

CRITICAL: This level is used to indicate critical errors that prevent the program from functioning properly. It denotes severe issues that require immediate attention and may cause the program to terminate.

Q7. What line of code would you add to your software to disable all logging messages?

Sol. logging.disable(logging.CRITICAL)

Q8.Why is using logging messages better than using print() to display the same message?

Sol. Using logging messages is generally considered better than using **print()** statements for displaying messages in software for several reasons:

**Log Levels and Filtering**: The logging module provides different log levels (such as DEBUG, INFO, WARNING, ERROR, CRITICAL) that allow you to categorize and prioritize messages based on their severity. This allows you to control the verbosity of the output and filter messages based on their importance. With **print()**, you would need to manually implement similar filtering logic, resulting in less efficient and more error-prone code.

**Configurability**: The logging module offers extensive configuration options, allowing you to control the output format, destination, and behavior of the logging messages. You can easily redirect log messages to different outputs (e.g., console, file, network), set the log level globally or on a per-handler basis, and format log records in a consistent manner. With **print()**, you have limited control over the formatting and behavior of the output.

**Runtime Impact**: Logging messages can be dynamically enabled or disabled without modifying the code itself. This means you can leave the logging statements in your code and control their visibility or verbosity at runtime. In contrast, **print()** statements are typically used for immediate output and require manual modification or removal when no longer needed, leading to cluttered code and potential oversight.

**Flexibility**: The logging module supports multiple handlers, allowing you to send log messages to different destinations simultaneously. You can log messages to files, databases, email, or even external logging services. This flexibility is not readily available with **print()** statements, which are limited to the standard output (console).

**Integration with Libraries and Frameworks**: The logging module is widely used and supported by various libraries and frameworks. This means that logging messages from different components can be consolidated and coordinated, providing a unified logging experience. Many libraries also utilize the logging module, allowing you to capture and manage their log messages seamlessly. **print()** statements, on the other hand, are independent and cannot be easily integrated into a unified logging infrastructure.

**Debugging and Production Use**: Logging is more suitable for long-term software development, debugging, and production environments. It allows you to log messages during development and leave them in the code without impacting performance. In production, you can selectively enable or disable logging levels to capture relevant information for debugging or monitoring purposes. In contrast, **print()** statements are often used for quick and temporary output during development and debugging, and they may be overlooked or forgotten when deploying the code to production.

Overall, logging messages provide more control, configurability, and flexibility compared to **print()** statements. They are designed specifically for logging purposes, offering a standardized and efficient approach to capturing and managing messages in software.

Q9. What are the differences between the Step Over, Step In, and Step Out buttons in the debugger?

Sol.   
The Step Over, Step In, and Step Out buttons are commonly found in debuggers and are used for controlling the execution flow during debugging. Here are the differences between these buttons:

**Step Over**: The Step Over button allows you to execute the next line of code without stepping into any function calls. If the current line contains a function call, the debugger will execute the entire function and move to the next line in the current scope. This allows you to skip the internal details of the function and focus on the high-level execution flow.

**Step In**: The Step In button allows you to step into the next line of code, even if it is a function call. If the current line contains a function call, the debugger will enter the function and pause at the first line of that function. This enables you to dive into the details of the function and debug its internal behavior, examining variable values and stepping through each line.

**Step Out**: The Step Out button is used to quickly step out of the current function and return to the calling function. If the debugger is currently paused inside a function, clicking Step Out will execute the remaining lines of the current function and pause at the line where the function was called. This is helpful when you have stepped into a function and want to quickly return to the higher-level code execution.

Q10. After you click Continue, when will the debugger stop ?

Sol. After clicking the Continue button in the debugger, the debugger will stop when one of the following conditions is met:

**Breakpoint**: If there are any active breakpoints set in the code, the debugger will stop when the program execution reaches a line of code where a breakpoint is set. Breakpoints are specific points in the code where you want the debugger to pause and allow you to inspect the program's state.

**Exception**: If an unhandled exception is encountered during program execution, the debugger will stop and pause at the line of code where the exception occurred. This allows you to examine the exception details and debug the issue that caused the exception.

**Program Completion**: If the program execution reaches the end of the code without encountering any breakpoints or exceptions, the debugger will stop. This indicates that the program has finished running, and the debugging session will end.

Q11. What is the concept of a breakpoint?

Sol.   
The concept of a breakpoint is a fundamental aspect of debugging in software development. A breakpoint is a designated point in the source code where you instruct the debugger to pause the program's execution. When the program reaches a line of code with an active breakpoint, it will halt, allowing you to examine the program's state, variables, and perform further analysis.

Breakpoints serve several purposes:

**Code Inspection**: Breakpoints allow you to inspect the state of the program at a specific point in its execution. By pausing the program at a breakpoint, you can examine the values of variables, check the flow of control, and verify if the program is behaving as expected.

**Debugging and Troubleshooting**: Breakpoints are an invaluable tool for identifying and debugging issues in your code. By setting breakpoints strategically in areas of interest or suspected problems, you can isolate and analyze specific sections of code more effectively, helping to locate and fix bugs or unexpected behavior.

**Flow Control**: Breakpoints provide control over the execution flow of your program. They allow you to step through the code line by line, examine the changes in variable values, and observe the program's behavior in a controlled manner. This is particularly useful for understanding the sequence of events and identifying the cause of complex issues.

**Conditional Breakpoints**: Some debuggers allow the setting of conditional breakpoints. These breakpoints are triggered only when a specified condition is met, such as a certain variable value or an expression evaluation. Conditional breakpoints help narrow down debugging efforts by stopping the program's execution only when the specified condition is satisfied.

Breakpoints are typically set within integrated development environments (IDEs) or debugger tools that support debugging capabilities. They provide a powerful means of gaining insight into the program's execution and locating and resolving bugs, ultimately enhancing the efficiency and effectiveness of the software development process.