Python Assignment - 11

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

=> spam = -1

assert spam >= 0

**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).**

=> eggs = “Hello”

bacon = “hello”

assert eggs.lower() != bacon.lower()

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

=> assert False

**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, you need to include the following two lines in your software:

1. Importing the logging module:

‘ Import logging ‘

2. Configuring the logging settings:

‘ 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?**

=>

1. Importing the logging module and defining the logging configuration:

‘ import logging ‘

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

2. Calling logging.debug() to log messages:

‘ logging.debug('This is a debug message') ‘

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

=> The five levels of logging are:

**1. DEBUG**: Detailed information, typically used for debugging purposes. It provides the most detailed level of logging and is typically only enabled during development or troubleshooting.

**2. INFO**: General information about the program's execution. It gives confirmation that things are working as expected and provides a high-level overview of the program's progress.

**3. WARNING**: Indicates a potential issue or something that could cause problems in the future but does not prevent the program from continuing its execution. It highlights situations that may require attention.

**4. ERROR**: Indicates an error that caused a particular functionality to fail or a problem that prevented the program from completing a specific operation. It indicates more severe issues that need attention but do not necessarily terminate the entire program.

**5. CRITICAL**: Indicates a critical error or a severe issue that may prevent the program from continuing its execution. It highlights errors that require immediate attention as they can potentially lead to the termination of the program or cause significant issues.

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

=> The code to be added for disabling the all logging messages in our software is:

Logging.disable(logging.CRITICAL)

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

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

**1. Flexibility** **and** **configurability**: Logging allows you to configure different levels of verbosity and selectively control which messages should be displayed.

**2. Granularity and filtering**: Logging provides different levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) that allow you to categorize and prioritize your messages. This granularity allows you to filter and handle different types of messages separately.

**3. Standardization and consistency**: Logging follows a standardized format and provides consistent message formatting, timestamps, and other details.

**4. Integration and extensibility**: Logging integrates well with other tools and libraries in the Python ecosystem.

**5. Performance and optimization**: Logging can be optimized for performance, allowing you to control the volume and frequency of logging messages. By adjusting the logging level and using appropriate logging techniques (e.g., lazy evaluation of log messages), you can minimize the impact on performance.

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

=> The differences between step over, step in and step out are:

1. **Step Over**: When you click the Step Over button, 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 enter the function but instead executes it as a single step, treating it as a black box.

**2. Step In**: Clicking the Step In button causes the debugger to enter the next line of code, even if it is a function call. If the current line contains a function call, the debugger enters that function and pauses at the first line inside the function.

**3. Step Out**: The Step Out button (or associated keyboard shortcut) is used to execute the remaining lines of the current function and pause when the function returns. If you are currently inside a function while debugging and want to quickly return to the calling context, you can click Step Out.

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

=> After clicking the continue the debugger will stop when one of the following condition is met:

1. **Breakpoint**: If there is a breakpoint set at a specific line of code, the debugger will stop when the execution reaches that line. Breakpoints are useful for pausing the execution at specific points in your code to inspect variables, evaluate expressions, or analyze the program's state.

**2. Exception**: If an unhandled exception occurs during the execution, the debugger will pause at the point where the exception is raised. This allows you to investigate the exception and the state of the program at the time of the error.

**3. Program** **completion**: The debugger will stop when the program completes its execution. In other words, if the program reaches its end without encountering any breakpoints or exceptions, the debugger will pause at the end of the program.

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

=> A breakpoint is a specific point in your code where you instruct the debugger to pause the program's execution for inspection and analysis. It is a debugging tool that allows you to temporarily halt the program's flow at a designated location, enabling you to examine variables, evaluate expressions, and understand the state of the program at that particular point.

By setting breakpoints, you can selectively pause the execution of your code to gain insights into its behavior, identify issues, and debug problems. When the debugger encounters a breakpoint during program execution, it suspends the program and provides you with a debugging environment, typically with access to variables, stack traces, and other debugging features. Breakpoints can be set at various locations in your code, such as specific lines, function calls, or conditions.