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

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

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

6. What are the five levels of logging?

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

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

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

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

11. What is the concept of a breakpoint?

Answers

1. **Assert Statement for Negative Integer**:
   * You can use an assert statement to check if a variable is a negative integer. If it’s not, an AssertionError will be raised.
   * Example:
   * def check\_positive(num):
   * assert num >= 0, "Variable 'spam' must be non-negative"
   * # Rest of your code
   * spam = -5
   * check\_positive(spam)
2. **Assert Statement for String Equality (Case-Insensitive)**:
   * To check if two strings are equal (case-insensitive), use an assert statement.
   * Example:
   * eggs = 'Hello'
   * bacon = 'hello'
   * assert eggs.lower() == bacon.lower(), "Strings 'eggs' and 'bacon' must be equal"
3. **Always Failing Assert Statement**:
   * If you want an assert statement that always fails, simply provide a condition that evaluates to False.
   * Example:
   * assert False, "This assertion always fails"
4. **Logging Debug Messages**:
   * To use logging.debug(), you need to set up logging with the following two lines:
   * import logging
   * logging.basicConfig(level=logging.DEBUG)
5. **Logging to a File (programLog.txt)**:
   * To send logging.debug() messages to a file named programLog.txt, add these lines:
   * import logging
   * logging.basicConfig(filename='programLog.txt', level=logging.DEBUG)
6. **Five Levels of Logging**:
   * The five standard logging levels are (in increasing order of severity):
     + DEBUG: Detailed information for debugging purposes.
     + INFO: General information about the program’s operation.
     + WARNING: Indicates a potential issue or something unexpected.
     + ERROR: Indicates an error that caused the program to fail.
     + CRITICAL: A critical error that might lead to application failure.
     + Example:
     + import logging
     + logging.debug("Debug message")
     + logging.info("Info message")
     + logging.warning("Warning message")
     + logging.error("Error message")
     + logging.critical("Critical message")
7. **Disabling All Logging Messages**:
   * To disable all logging messages, set the logging level to a higher level (e.g., logging.WARNING or logging.CRITICAL).
   * import logging
   * logging.disable(logging.CRITICAL) # Disables all logging messages
8. **Advantages of Logging over print()**:
   * Logging provides more flexibility (levels, handlers, formatters).
   * You can selectively enable/disable logging without modifying code.
   * Log messages can go to files, databases, or other destinations.
   * Example:
   * import logging
   * logging.info("This is an informational message")
9. **Debugger Buttons (Step Over, Step In, Step Out)**:
   * These buttons are used during debugging:
     + **Step Over**: Executes the current line and moves to the next line (skips function calls).
     + **Step In**: Steps into a function (if the current line is a function call).
     + **Step Out**: Steps out of the current function (finishes executing the current function).
     + Use them to navigate through code during debugging.
10. **Debugger Stop After Continue**:
    * The debugger will stop again when it encounters another breakpoint or an exception.
    * It continues until the next breakpoint or the end of the program.
11. **Concept of a Breakpoint**:
    * A breakpoint is a designated point in your code where the debugger pauses execution.
    * It allows you to inspect variables, step through code, and diagnose issues.
    * Set breakpoints using your IDE or explicitly with pdb.set\_trace().