**Python Assignment 11**

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

Ans. Sure, here's an example assert statement in Python that will raise an AssertionError if the variable spam is a negative integer:

assert spam >= 0, "spam must be a non-negative integer"

This statement will check if the value of spam is greater than or equal to zero. If spam is negative, the assertion will fail and raise an AssertionError with the message "spam must be a non-negative integer".

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).

Ans. Here's an example assert statement that will raise an AssertionError if the variables eggs and bacon contain strings that are the same as each other, even if their cases are different:

assert eggs.lower() != bacon.lower(), "eggs and bacon must have different values"

This assertion compares the lower-case versions of the two variables (eggs and bacon) using the lower() method to ensure that the assertion is case-insensitive. If the lower-case strings are equal, the assertion will fail and raise an AssertionError with the message "eggs and bacon must have different values".

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

Ans. Sure, here's an example assert statement in Python that will always raise an AssertionError:

assert False, "This assertion always fails"

This assertion simply checks if the value False is true, which is never the case. Therefore, this assert statement will always fail and raise an AssertionError with the message "This assertion always fails".

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

Ans. To call logging.debug() in your software, you must have the following two lines:

import logging

logging.basicConfig(level=logging.DEBUG)

The first line imports the logging module into your Python script, while the second line configures the logging system to output debug messages. The basicConfig() method is used to set the logging level to logging.DEBUG, which enables debug messages to be output to the console or a log file.

With these two lines present in your software, you can then call logging.debug() to output debug messages in your code.

Q5. 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?

Ans. To send logging messages to a file named programLog.txt using logging.debug(), you need to add the following two lines to your program:

import logging

# Set up logging to a file

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

The basicConfig() method sets up the logging system with a filename argument to specify the name of the log file to which log messages will be written. By default, the basicConfig() method will write messages to the console, but with the filename argument set, log messages will be written to a file named programLog.txt instead. The level argument is also set to logging.DEBUG, which allows debug messages to be written to the log file.

With these two lines present in your program, you can then call logging.debug() to send debug messages to the log file.

Q6. What are the five levels of logging?

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

**DEBUG**: - Detailed information for diagnosing problems. Typically only enabled during development.

**INFO**: - General information about the program's operation. This is the default level.

**WARNING**: - An indication that something unexpected happened or indicative of some problem in the near future (e.g., 'disk space low'). The software is still working as expected.

**ERROR**: - Due to a more serious problem, the software has not been able to perform some function.

**CRITICAL**: - A very serious error, indicating that the program itself may be unable to continue running.

You can set the level of logging messages that your program outputs by calling the basicConfig() method of the logging module, and specifying the desired level with the level argument. For example, setting the level to logging.DEBUG will enable debug, info, warning, error, and critical messages to be output, while setting the level to logging.ERROR will only output error and critical messages.

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

Ans. To disable all logging messages in your software, you can add the following line of code:

logging.disable(logging.CRITICAL)

This will disable all logging messages with a severity level of logging.CRITICAL or lower. Since CRITICAL is the highest severity level, all logging messages will be disabled. You must first import the logging module for this code to work.

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

Ans. Using logging messages is generally better than using print() to display the same message for several reasons:

**Control over message severity and verbosity**: - With logging, you can control the level of detail in the messages that are output, as well as the severity level of the message. This allows you to output messages that are appropriate for the situation, and to filter out messages that are not important or relevant.

**Output to multiple destinations**: - With logging, you can output messages to multiple destinations, such as the console, a file, a database, or a remote server. This makes it easy to log messages for later analysis, or to monitor the status of a running program in real-time.

**Efficient output**: - Logging can be more efficient than using print() for outputting messages, especially when dealing with large volumes of data or when performance is critical. This is because logging can be configured to buffer messages and output them in batches, which can reduce I/O overhead and improve performance.

**Easier to maintain and debug**:- Logging can make it easier to maintain and debug code over time, as it provides a record of the program's behavior that can be used to diagnose problems or to understand how the code is working. This is especially useful when dealing with complex or long-running programs, where print() statements can become difficult to manage or understand.

Overall, using logging messages provides more control, flexibility, and efficiency than using print() statements for outputting messages in your code.

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

Ans. The Step Over, Step In, and Step Out buttons are used in the debugger to execute code one line at a time and to help you understand how the program is running. Here are the differences between these buttons:

**Step Over**: - This button is used to execute the current line of code and then move to the next line. If the current line contains a function call, the function is executed and the result is returned, but the debugger does not step into the function. This is useful for quickly executing a block of code without diving into the details of any functions that are called.

**Step In**: - This button is used to execute the current line of code and then step into any function calls that are made on that line. If the current line does not contain a function call, Step In behaves the same as Step Over. This is useful for diving into the details of how a function works, and for stepping through code one line at a time.

**Step Out**: - This button is used to execute the current function call and return to the line that called the function. If there are multiple function calls in the current stack frame, Step Out will execute all of them until it reaches the outermost stack frame. This is useful for quickly returning from a function call and continuing execution of the code that called the function.

Overall, these buttons provide different levels of control for stepping through code and understanding how it works, and are essential tools for debugging and troubleshooting code.

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

Ans. After you click the Continue button in the debugger, the debugger will continue running the program without interruption until one of the following events occurs:

**A breakpoint is encountered**: - If you have set a breakpoint in the code, the debugger will stop when it reaches that breakpoint. This allows you to inspect the state of the program at that point in the code and to step through the remaining lines of code.

**An exception is raised**: - If an exception is raised during program execution, the debugger will stop and display the exception information, allowing you to diagnose and fix the problem.

**The program exits**: - If the program completes its execution normally, the debugger will stop and display the final state of the program. This allows you to inspect the output of the program and to verify that it has completed correctly.

In general, the Continue button is used when you want to let the program run without interruption, and to only stop the debugger when a specific event occurs. This can be useful for testing and debugging code in a realistic environment, where you want to see how the code behaves under normal conditions.

Q11. What is the concept of a breakpoint?

Ans. In software development, a breakpoint is a debugging tool that allows you to temporarily pause the execution of a program at a specific point in the code. When a breakpoint is encountered during program execution, the program is paused and the debugger takes over, allowing you to inspect the current state of the program, view the values of variables, and step through the remaining lines of code.

Breakpoints are useful for several reasons:

**Debugging complex code**: - When you are working with complex code that has many lines of code, it can be difficult to understand what is happening at any given point. Breakpoints allow you to pause the program at specific points and inspect the state of the program, making it easier to understand what is happening.

**Identifying errors**: - Breakpoints can be used to identify errors in your code by allowing you to step through the code one line at a time, examining the state of the program at each step. This can help you to identify the point in the code where an error occurs and to diagnose and fix the problem.

**Testing code**: - Breakpoints can be used to test code by allowing you to pause the program at specific points and examine the state of the program. This can help you to verify that the program is behaving correctly and to identify any issues before the code is released.

Overall, breakpoints are an essential tool for debugging and testing code, and can help you to identify and fix errors quickly and efficiently.