**Python Assignment 11**

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

Here's an example of an assert statement that throws an AssertionError if the variable spam is a negative integer:

spam = -5

assert spam >= 0, "spam cannot be a negative integer"

In this example, the assert statement checks if the condition spam >= 0 is True. If the condition is False, an AssertionError is raised, and the message "spam cannot be a negative integer" is included in the error message. Replace the value of spam with the variable we want to check.

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).

Here's an assert statement that triggers an AssertionError if the variables eggs and bacon contain strings that are the same as each other, regardless of their cases:

eggs = "hello"

bacon = "Hello"

assert eggs.lower() != bacon.lower(), "eggs and bacon strings cannot be the same"

In this example, just like in the previous response, the assert statement checks if the condition eggs.lower() != bacon.lower() is True. The .lower() method is used to convert both strings to lowercase before comparison, so the assertion will fail if the strings are the same regardless of their case. If the condition is False, an AssertionError is raised with the specified message.

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

We can create an assert statement that always triggers an AssertionError by providing a condition that evaluates to False. Here's an example:

assert False, "This assert statement will always raise an AssertionError"

In this example, the condition False will never be True, so the assert statement will always raise an AssertionError with the specified message. Keep in mind that using such assert statements in real code is not common and may not be a good practice, as assertions are typically used to check conditions that should be true during the normal execution of our program.

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

To use the logging.debug() function in your software, you need to perform two steps:

Import the logging module:

we need to import the logging module at the beginning of our script to access the logging functionality.

import logging

Configure the logging level:

Before we can use logging.debug() or any other logging function, we need to configure the logging level. By default, the logging level is set to WARNING, which means that debug messages won't be displayed. we can change the logging level to show debug messages.

logging.basicConfig(level=logging.DEBUG)

Here's an example of the two lines that should be present in our software to call logging.debug():

import logging

logging.basicConfig(level=logging.DEBUG)

# Now we can use logging.debug() to log debug messages

logging.debug("This is a debug message")

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?

To use logging.debug() to send logging messages to a file named programLog.txt, we need to set up the logging configuration to specify the file as the output destination. Here are the two lines we need to include in our program:

Import the logging module:

Import the logging module at the beginning of our script to access the logging functionality.

import logging

Configure logging to write to a file:

Configure the logging module to write messages to the specified file. we can use the basicConfig() function with the filename parameter to specify the file name. Additionally, we can set the desired log level, format, and other options.

import logging

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

# Now we can use logging.debug() to log debug messages to the file

logging.debug("This is a debug message written to programLog.txt")

6. What are the five levels of logging?

The logging module in Python provides five levels of logging, ranging from most critical to least critical. These levels are used to categorize the importance of log messages and control which messages are displayed based on the configured log level. The five levels of logging, in decreasing order of severity, are:

CRITICAL (50):

The highest level of severity. Used for critical errors that may lead to application failure.

ERROR (40):

Used for errors that do not necessarily cause the program to fail but indicate a serious problem that needs attention.

WARNING (30):

Used for potential issues or unexpected behavior that is not necessarily an error but might require investigation.

INFO (20):

Used for informational messages about the program's operation. Provides insights into the program's progress.

DEBUG (10):

The lowest level of severity. Used for detailed debug messages that provide insights into the program's internal workings for troubleshooting and debugging purposes.

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

To disable all logging messages, we can set the logging level to a level higher than the most critical level, which effectively suppresses all log messages. The highest level in the logging hierarchy is CRITICAL, so we can set the logging level to this value to disable all logging messages. Here's the line of code we would add to achieve this:

import logging

logging.disable(logging.CRITICAL)

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

Using logging messages is generally better than using print() statements for displaying messages in your code for several reasons:

Control and Flexibility:

Logging provides more control and flexibility over where and how messages are displayed. we can configure different log handlers to output messages to various destinations, such as console, files, email, or external logging services. This allows us to control the behavior of logging messages without modifying our code.

Log Levels:

Logging messages can be categorized into different severity levels (e.g., DEBUG, INFO, ERROR, CRITICAL). we can set a global log level to control which messages are displayed. This is useful for filtering out less important messages in production environments and focusing on critical issues.

Filtering and Analysis:

Logging messages can be filtered and analyzed more effectively. we can search, filter, and analyze logs based on different criteria, such as severity, timestamps, and messages. This is particularly useful for troubleshooting and debugging.

Non-Intrusiveness:

Logging messages can be added and removed without modifying the program's logic. This allows us to add detailed diagnostic information during development and remove it in production without changing the code structure.

Performance:

In some cases, logging may be more efficient than printing, especially when dealing with large volumes of messages. Logging messages can be optimized and processed by log handlers in a more efficient manner.

Standardization:

Logging follows a standardized approach for displaying and managing messages. This makes it easier for developers to understand and maintain the codebase, especially in collaborative projects.

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 to control the execution of a program while debugging. These buttons allow us to move through the code one step at a time and inspect the program's state at each step. The differences between these buttons are as follows:

Step Over:

When we click the "Step Over" button (or use the corresponding keyboard shortcut), the debugger will execute the current line of code and move to the next line. If the current line contains a function call, the debugger will not enter the function; it will simply execute the entire function and move to the next line after the function call. This is useful for quickly moving through the code without diving into the details of function calls.

Step In:

Clicking the "Step In" button (or using the keyboard shortcut) causes the debugger to move into the function call on the current line. It will pause execution at the first line inside the called function, allowing us to inspect the behavior of that function in detail. This is useful when we want to step into a function and understand how it works.

Step Out:

The "Step Out" button (or keyboard shortcut) is used to step out of the current function. If we are already inside a function and we want to return to the calling function, clicking "Step Out" will continue execution until the current function returns and then pause at the line immediately after the function call. This is useful when we are done inspecting the behavior of the current function and want to move back to the caller.

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

When we click the "Continue" button (sometimes labeled "Resume" or represented by a play icon) in a debugger, the debugger will continue the execution of the program from the current point, allowing it to run uninterrupted until it reaches one of the following stopping conditions:

Breakpoint:

If we have set one or more breakpoints in our code, the debugger will stop when the execution reaches a line with a breakpoint. This allows us to inspect the program's state and variables at that specific point.

Exception:

If an unhandled exception is raised during the execution, the debugger will stop when the exception occurs. This allows us to investigate the cause of the exception and the program's state when it happened.

End of Program:

The debugger will stop when the program's execution is completed, meaning it has reached the end of the code. This is the natural stopping point for the debugger when the program has finished running.

Manual Interruption:

we can manually interrupt the execution by clicking the "Pause" button or the equivalent option in the debugger interface. This will immediately pause the program's execution and allow us to inspect its current state.

User Interaction:

If the program prompts for user input while running, the debugger might stop when it reaches such a point and waits for the user's response.

Keep in mind that the behavior of the "Continue" button might be influenced by the settings and configurations of the debugger and the specific integrated development environment (IDE) we are using. It's important to be aware of the stopping conditions to effectively use the debugger for troubleshooting and debugging our code.

11. What is the concept of a breakpoint?

A breakpoint is a specific point in our code where we instruct the debugger to pause the execution of the program so that we can inspect the program's state, variables, and control flow. Breakpoints are a fundamental tool in debugging and are used to help developers identify and diagnose issues in their code.

When a breakpoint is encountered during the program's execution, the debugger halts the program and gives us the opportunity to:

Inspect Variables: we can view the values of variables and data structures at that particular point in the code. This helps us understand how the program's state changes as it runs.

Step Through Code: we can step through the code line by line, executing each line individually, and observing how the program's behavior changes.

Evaluate Expressions: we can evaluate and execute specific expressions in the current context to check their values or behavior.

Set Watches: Watches are expressions that we can monitor as we step through the code. The debugger will notify us when the value of a watched expression changes.

Modify Variables: Some debuggers allow us to modify variable values during runtime, which can help us to test different scenarios without restarting the program.

Breakpoints can be set in various ways, such as through the debugger's graphical interface, by clicking on the line number in our code editor, or by using specific debugging commands. we can set multiple breakpoints throughout our code to focus on specific areas of interest.

Here's an example of setting a breakpoint in Python using the built-in pdb debugger:

import pdb

def my\_function(x, y):

result = x + y

pdb.set\_trace() # Set a breakpoint here

return result

print(my\_function(2, 3))

In this example, the pdb.set\_trace() line sets a breakpoint. When the program execution reaches this line, the debugger will pause, allowing us to interactively inspect and step through the code.

Breakpoints are essential for effective debugging, as they allow us to examine the program's behavior at specific moments, helping us locate and resolve issues more efficiently.