**Assignment-11**

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

You can use the assert statement to check if a variable spam is a negative integer. If the condition is not met, an AssertionError will be raised.

assert spam >= 0, "Variable spam should not be a negative integer."

1. 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).

assert eggs.lower() != bacon.lower(), "Variables eggs and bacon should have different string values."

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

If you want to create an assert statement that throws an AssertionError every time, you can simply use the assert statement without specifying a condition.

assert False, "This assert statement always throws an AssertionError."

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

* Import the logging module:

import logging

* Configure the logging level to include the debug level logging.basicConfig(level=logging.DEBUG)

1. 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?

* Import the logging module:

import logging

* Configure the logging to write messages to a file:

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

1. What are the five levels of logging?

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

* DEBUG: Detailed information, typically useful for debugging purposes. It provides the most detailed level of logging.
* INFO: General information about the program's execution. It gives an overview of the program's progress.
* WARNING: Indicates a potential issue or something that may cause problems in the future, but the program can still continue running.
* ERROR: Indicates a more serious issue or error that prevents a specific part of the program from functioning correctly.
* CRITICAL: Indicates a critical error or failure that may result in the program being unable to continue running. It represents the highest level of severity.

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

logging.disable(logging.CRITICAL)

1. 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 due to the following reasons:

* Configurability: Logging provides a more flexible and configurable way to manage messages. You can easily control the logging level to determine which messages are displayed based on their severity. This allows you to selectively enable or disable different levels of logging, making it easier to debug and analyze your code in different environments.
* Granularity: Logging allows you to categorize and prioritize messages based on their importance and severity. With different logging levels (e.g., DEBUG, INFO, WARNING, ERROR, CRITICAL), you can fine-tune the amount of information displayed. This helps you focus on relevant information while filtering out unnecessary details.
* Persistence: Logging messages can be logged to different outputs, such as a file, console, or network stream. This allows you to retain a record of messages for future analysis or reference. With print(), the messages are typically only displayed in the console, and you would need to redirect them manually to a file if you wanted to persist them.
* Flexibility: Logging provides additional features like log formatting, log rotation, and log filtering. You can customize the format of log messages, add timestamps, include contextual information, and even define your own handlers to handle log messages in specific ways. This flexibility allows for more advanced logging capabilities beyond simple printing.
* Integration: Logging integrates well with existing logging frameworks and libraries. It is a standardized approach for managing log messages across different modules or libraries in your software. By using the logging module, you can benefit from consistent logging practices and easily incorporate third-party logging extensions or tools.
* Production-readiness: Logging is designed to be suitable for production environments. It provides a structured and reliable way to handle messages, allowing you to monitor and troubleshoot your application effectively. Logging can handle logging in multi-threaded or multi-process scenarios, offers performance optimizations, and supports various log levels and handlers to meet specific production requirements.

Overall, using logging messages provides a more robust, flexible, and manageable approach for handling messages in your code compared to using print(). It offers better control, configurability, and integration possibilities while allowing for easier debugging and monitoring of your software.

1. 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 flow during debugging. Here are the differences between these buttons:

* Step Over: The Step Over button allows you to execute the current line of code and move to the next line. If the current line contains a function call, the entire function is executed without stepping into it. It is useful when you want to skip the detailed execution within a function and move to the next line in the current scope.
* Step In: The Step In button allows you to step into the next line of code, even if it's a function call. If the current line contains a function call, the debugger will enter the function and start debugging from the first line of that function. It is useful when you want to examine the detailed execution within a function or method.
* Step Out: The Step Out button is used to quickly step out of the current function and return to the calling function. It allows you to continue execution until the current function returns, and then the debugger will stop at the next line after the function call. It is useful when you want to skip the remaining execution within the current function and return to the higher-level context.

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

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

* A breakpoint is encountered: If there is a breakpoint set in the code, the debugger will stop when it reaches that breakpoint. Breakpoints allow you to pause the execution at a specific line or condition to inspect variables and step through the code.
* An exception/error occurs: If an unhandled exception or error occurs during the execution of the program, the debugger will automatically stop at the line where the exception occurred. This allows you to examine the state of the program at the point of the exception and understand the cause of the error.
* The program reaches its end: If the program execution reaches the end of the script or the specified main entry point, the debugger will stop. This is the normal termination point, and the debugger will no longer have any lines to execute.

1. What is the concept of a breakpoint?

A breakpoint is a designated point in your code where you want the debugger to pause the execution of your program. When the debugger encounters a breakpoint during runtime, it temporarily suspends the program's execution, allowing you to inspect variables, step through the code line by line, and analyze the program's state at that particular moment.

Breakpoints serve as a powerful tool in the debugging process, enabling you to:

* Examine program state: By pausing the program's execution at specific breakpoints, you can inspect the values of variables, check the state of objects, and verify the flow of control in your code.
* Identify bugs or issues: Breakpoints allow you to narrow down the scope of your investigation by focusing on specific sections of code. You can carefully examine the execution path and analyze the behavior of your program to identify and resolve bugs or unexpected behaviors.
* Step through code: Once the debugger hits a breakpoint, you can step through the code line by line, observing the changes in variables and understanding how the program's logic unfolds. This step-by-step analysis can provide insights into the program's behavior and help pinpoint the cause of issues.
* Modify variables: Some debuggers allow you to modify variable values while the program is paused at a breakpoint. This capability can be useful for testing different scenarios or correcting erroneous values to observe the impact on the program's execution.

Setting breakpoints is typically done through an integrated development environment (IDE) or a debugger interface. You can place breakpoints on specific lines of code, on conditional statements, or even inside loops, depending on your debugging needs. By strategically placing breakpoints, you can focus your debugging efforts and gain better insights into the inner workings of your program.