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

assert spam >= 0, 'spam must be a non-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).**

Here's 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:

```

assert eggs.lower() != bacon.lower(), "eggs and bacon are the same, even if their cases are different"

```

In this statement, the `lower()` method is used to convert both `eggs` and `bacon` to lowercase strings, so that their cases are ignored. The `!=` operator is used to check whether the two lowercase strings are different, and if they are the same, an AssertionError is triggered with the specified error message.

Note that if either `eggs` or `bacon` is not a string, this assert statement will raise a TypeError. If you need to check that both variables are strings before comparing them, you can add an additional assert statement like this:

```

assert isinstance(eggs, str) and isinstance(bacon, str), "eggs and/or bacon is not a string"

```

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

Here's an example assert statement that throws an AssertionError every time:

```

assert False, 'This assertion always fails.'

```

This assert statement always evaluates to `False`, which means that an AssertionError will always be raised. The error message associated with the AssertionError is the string `'This assertion always fails.'`.

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

In order to use the `logging.debug()` function in Python, you need to perform two steps:

1. Import the `logging` module

2. Configure the logging system

Here are the two lines of code you need:

```

import logging

logging.basicConfig(level=logging.DEBUG)

```

The first line imports the `logging` module, which provides the `debug()` function along with other functions for logging messages at different levels of severity.

The second line configures the logging system to log messages at the `DEBUG` level or above. You can adjust the logging level to a higher or lower severity level (e.g., `INFO`, `WARNING`, `ERROR`, `CRITICAL`) depending on the level of detail you want in your logs.

Once you have performed these two steps, you can call `logging.debug()` to log debug-level messages in your code. For example:

```

import logging

logging.basicConfig(level=logging.DEBUG)

def my\_function():

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

# rest of function code

```

In this example, the `my\_function()` function logs a debug message using `logging.debug()`. The message will only be logged if the logging level is set to `DEBUG` or lower.

**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 have `logging.debug()` send a logging message to a file named `programLog.txt`, the following two lines of code are needed:

```python

import logging

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

```

The first line imports the `logging` module, which provides the functionality for creating log messages. The second line sets up the logging configuration by calling the `basicConfig()` method of the `logging` module, which specifies the output file name (`filename='programLog.txt'`) and the minimum log level (`level=logging.DEBUG`) for messages to be written to the file. Once this configuration is set up, any subsequent calls to `logging.debug()` will write a message to the specified file.

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

The Python `logging` module defines five standard levels of severity for log messages, in increasing order of severity:

1. `DEBUG`: Detailed information, typically of interest only when diagnosing problems.

2. `INFO`: General information about the program's operation.

3. `WARNING`: An indication that something unexpected or undesirable has happened, or indicative of some problem in the near future (e.g., ‘disk space low’).

4. `ERROR`: An error occurred which caused the program to behave in an unexpected or incorrect way.

5. `CRITICAL`: A very serious error occurred, indicating that the program may be unable to continue running.

When you log a message at a certain level of severity, all messages at that level and higher levels will be recorded in the log output. For example, if you set the logging level to `WARNING`, only messages with severity of `WARNING`, `ERROR`, or `CRITICAL` will be recorded in the log output.

You can also define custom log levels beyond the five standard levels if you need more fine-grained control over the severity of your log messages.

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

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

```python

logging.disable(logging.CRITICAL)

```

This sets the logging level to `logging.CRITICAL`, which is the highest level of severity. Any messages with a lower severity level than `CRITICAL` will not be displayed or saved to the log file. This effectively disables all logging messages.

It's worth noting that disabling all logging messages should generally only be done for debugging purposes, as logging is an important tool for troubleshooting and monitoring applications in production environments.

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

Using logging messages is generally considered better than using `print()` to display messages in code for several reasons:

1. \*\*Granular control over the output\*\*: With logging, you have the ability to control the level of detail of the messages that are output based on the severity of the issue being logged. This can be helpful for debugging and troubleshooting by providing more context and information about the issue, without overwhelming the user with unnecessary information.

2. \*\*Separation of concerns\*\*: Logging is a separate concern from the actual functionality of the code, allowing developers to more clearly separate the logging code from the core functionality of the program. This can make it easier to maintain and update the code over time.

3. \*\*Configurability\*\*: Logging can be configured to output to various destinations, such as a file, a database, or a remote server, and the level of detail of the messages can be adjusted based on the needs of the user. This can be helpful for large applications or distributed systems where multiple components may need to log messages to a central location for monitoring and analysis.

4. \*\*Performance overhead\*\*: `print()` statements can be expensive, especially if they are executed frequently or in a loop. Logging messages, on the other hand, are only output if the log level is set to a level that is equal to or more severe than the level of the log message being emitted. This means that logging can be more performant than using `print()` for outputting messages.

In summary, logging provides more fine-grained control over the output of messages, better separation of concerns, and more configurability than using `print()`, making it a better option for logging messages in production code.

**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 used during the debugging process to execute code line by line and analyze program behavior.

Here are the differences between these buttons:

1. Step Over:

When you click the Step Over button, the debugger executes the current line of code and then moves to the next line. If the current line contains a function call, the function is executed, but the debugger does not stop inside the function. Instead, it executes the function and stops at the next line after the function call.

2. Step In:

When you click the Step In button, the debugger moves to the next line of code, but if the current line contains a function call, the debugger stops inside the function and allows you to step through each line of the function.

3. Step Out:

When you click the Step Out button, the debugger runs the rest of the code in the current function and stops at the line after the function call that initiated the current function. This button is useful when you are debugging a function and want to quickly move to the next line of code in the calling function without stepping through all the lines in the current function.

In summary, Step Over executes the current line and moves to the next, Step In allows you to step through each line of a function, and Step Out executes the rest of the current function and moves to the next line in the calling function.

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

When you click the Continue button in a debugger, the program will resume execution and continue running until one of the following occurs:

1. The program reaches a breakpoint: If a breakpoint is set, the program will stop at the breakpoint, allowing you to inspect the program's state at that point.

2. An unhandled exception is thrown: If the program throws an unhandled exception, the debugger will stop at the line of code that caused the exception, allowing you to inspect the exception and the program's state.

3. The program finishes executing: If the program completes executing without any issues, the debugger will stop once the program has finished running.

It's important to note that the debugger will not stop at every line of code while the program is running after you click Continue. Instead, it will only stop at breakpoints or when an unhandled exception is thrown.

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

A breakpoint is a tool used in debugging that allows a programmer to pause the execution of a program at a specific line of code, in order to inspect and analyze the program's state at that point.

When a breakpoint is set, the program will execute normally until it reaches the line of code specified by the breakpoint. At this point, the program will pause, allowing the programmer to examine the program's state and variables, step through the code line by line, and make any necessary corrections or adjustments.

Breakpoints can be extremely helpful in identifying and fixing bugs in a program, as they allow a programmer to isolate specific sections of code and analyze them in detail. They are particularly useful when dealing with complex or hard-to-find bugs, as they allow the programmer to examine the program's state at a specific point in the execution, which can provide important clues as to the cause of the bug.

In most modern integrated development environments (IDEs), setting a breakpoint is as simple as clicking on the line of code where you want to pause execution. Once the breakpoint is set, the program will execute normally until it reaches that line, at which point it will pause and wait for further instructions from the programmer.