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

An `**assert` statement** in Python is a **debugging aid** that tests a condition as an **internal self check** in your code. When the **assertion is true**, the **program continues to execute normally**. However, if the assertion is **false**, then the **program raises an AssertionError exception** with an **optional error message.** The **syntax** for an assert statement is as follows:

**assert condition, message**

**Purpose:** to **find bugs more quickly** and **easily** during development phase of a program. Allows to check certain conditions in code during runtime without having to manually inspect output of program or use a debugger.

Now an **assert statement** that throws an Assertion Error if the variable spam is a negative integer

**assert spam>=0, “spam must be a non 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(), "eggs and bacon must be different"**

This assert statement uses the lower() method to convert both strings to lowercase before comparing them. If the two strings are the same, regardless of case, the assertion will fail and an AssertionError will be raised with the message "eggs and bacon must be different".

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

**assert 1 == 2, "1 is not equal to 2"**

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

To call **logging.debug()** in your software, you must **first import** the **logging module** and then **set the logging level** **to logging.DEBUG**. The two lines that must be present are:

**import logging**

**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 logging**

**logging.basicConfig(filename='programLog.txt', level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')**

The first line imports the logging module, and the second line configures the logger to write log messages to the programLog.txt file at the DEBUG level. The format parameter specifies the format of the log messages, including the timestamp, log level, and message.

1. What are the five levels of logging?

* **DEBUG**: **Detailed information**, typically of interest only when diagnosing problems.
* **INFO**: **General information** about the program's operation.
* **WARNING**: An indication that **something unexpected or potentially problematic** 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.**

1. 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:

**logging.disable(logging.CRITICAL)**

This will **disable all logging messages** with a **level** of **logging.CRITICAL** and **below**, which **includes all levels** of logging (CRITICAL, ERROR, WARNING, INFO, and DEBUG).

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

* Logging messages are more **flexible** and **customizable** than print statements.
* Logging messages can be **selectively enabled or disabled** based on their level and source.
* Logging messages **can be directed to different outputs**, such as files, streams, or external logging services.
* Logging messages can **include contextual information**, such as the timestamp, severity level, and source of the message.
* Logging messages can **help diagnose issues** in production environments where print statements may not be visible or accessible.
* Logging messages can **improve performance by reducing the overhead of printing messages** to the console.
* Logging messages can provide a **structured** and **standardized way to log messages across multiple modules and components.**

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 **debugger controls** used to **execute code line-by-line for debugging purposes**. 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, it will execute the entire function and move to the next line after the function call returns.

**Step In**: The Step In button allows you to move into a function call and execute its code line-by-line. If the current line contains a function call, Step In will take you to the first line of the function call.

**Step Out**: The Step Out button **allows** you to **execute** **code** **until** you **return** **from** the **current function.** If you are currently inside a function call, Step Out will **execute the remaining lines of the function call** and **return to the line that called the function.**

In summary, Step Over is used to execute the current line of code and move to the next line, Step In is used to move into a function call and execute its code, and Step Out is used to execute the remaining lines of the current function and return to the line that called the function.

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

When you click the "Continue" button in the debugger, it will **resume executing** the **code** **without stopping until it reaches** the **next breakpoint**, an **exception** is **raised**, or the program finishes executing. The debugger will stop at the next breakpoint it encounters or when an exception is raised. If there are no more breakpoints or exceptions, the program will run until completion.

11. What is the concept of a breakpoint?

A breakpoint is a designated point in your code where the debugger pauses execution, allowing you to inspect the state of the program at that point. When the debugger reaches a breakpoint, it stops executing the code and gives you the opportunity to examine the values of variables, step through the code line by line, or modify the code to correct any errors.

Setting a breakpoint can be useful for debugging because it allows you to stop the program at a specific point and examine the state of the program at that point. This can help you identify the cause of any errors or unexpected behavior in your code. Breakpoints can be set in the code editor or debugger interface, and can be removed or disabled as needed.