**Python Basic Assignment 11**

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

**integer.**

Ans:- In Python, you can use an `assert` statement to raise an `AssertionError` if a condition is not met. To throw an `AssertionError` if the variable `spam` is a negative integer, you can write the following assert statement:

```python

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

```

In this code:

- `spam` is the variable you want to check.

- `spam >= 0` is the condition you are asserting. If `spam` is not greater than or equal to 0, the assertion will fail.

- `"spam should not be a negative integer"` is an optional error message that will be displayed when the assertion fails. You can customize this message to provide more context about the assertion.

If `spam` is a negative integer when this `assert` statement is encountered in your code, it will raise an `AssertionError` with the specified error message. Otherwise, if `spam` is non-negative, the `assert` statement will have no effect, and the program will continue executing.

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

Ans:- To create an `assert` statement that throws an `AssertionError` every time, you can use a condition that is always `False`. Here's an example:

```python

assert False, "This assertion always raises an AssertionError"

```

In this code:

- The condition `False` is always `False`, so the `assert` statement will always raise an `AssertionError`.

- The second argument is an optional error message that will be displayed when the assertion fails. You can customize this message to provide more context about why the assertion is always failing.

When this `assert` statement is encountered in your code, it will immediately raise an `AssertionError` with the specified error message. This can be useful for testing and debugging purposes to identify and handle unexpected conditions in your code.

**4. 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 Python software, you need to include two essential lines of code:

1. Import the `logging` module :

- You must import the `logging` module at the beginning of your Python script or module. This line allows you to access the `logging` functionality.

```python

import logging

```

2. Configure the logging :

- Before you can use `logging.debug()`, you should configure the logging system to specify where and how log messages should be recorded. This configuration should typically be done at the beginning of your script.

For a basic configuration that sends log messages to the console, you can use:

```python

logging.basicConfig(level=logging.DEBUG)

```

This sets the logging level to `DEBUG`, which means that messages at or above the `DEBUG` level will be recorded. You can adjust the logging level as needed (e.g., to `INFO`, `WARNING`, `ERROR`, or `CRITICAL`) depending on the desired verbosity of your logs.

After these two lines are present in your code, you can call `logging.debug("your message")` to log debug messages. Here's an example:

```python

import logging

# Configure logging to show DEBUG level messages on the console

logging.basicConfig(level=logging.DEBUG)

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

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

```

With these lines in place, calling `logging.debug("your message")` will record a debug message, and it will be displayed on the console if the logging level is set to `DEBUG`.

**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?**

Ans:-- To have `logging.debug()` send a logging message to a file named `programLog.txt`, you need to include two essential lines of code in your program:

1. Import the `logging` module :

- Import the `logging` module at the beginning of your Python script or module. This line allows you to access the `logging` functionality.

```python

import logging

```

2. Configure the logging to write to a file :

- You must configure the `logging` module to write log messages to the specified file. To do this, use the `logging.basicConfig()` function with a `filename` parameter set to the name of the log file (`programLog.txt`) and a `level` parameter to specify the logging level (e.g., `DEBUG`, `INFO`, `WARNING`, `ERROR`, or `CRITICAL`).

```python

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

```

With these two lines in place, you can use `logging.debug("your message")` to log debug messages, and they will be written to the `programLog.txt` file. Here's an example:

```python

import logging

# Configure logging to write to programLog.txt and set the logging level to DEBUG

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

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

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

```

The `logging.debug("your message")` call in this code will record a debug message in the `programLog.txt` file if the logging level is set to `DEBUG`.

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

Ans:-- In most logging systems, including Python's `logging` module, there are five standard levels of logging, ordered from lowest severity to highest severity. These levels allow you to categorize and prioritize log messages based on their importance and impact on your application:

1. DEBUG :

- The `DEBUG` level is the lowest severity level.

- It is typically used for messages that provide detailed information for debugging and troubleshooting purposes.

- Debug messages are usually not suitable for production environments because they can generate a large volume of logs.

2. INFO :

- The `INFO` level is used for informational messages that describe the progress or status of an application.

- These messages are generally used for general application events and to indicate that everything is running as expected.

3. WARNING :

- The `WARNING` level indicates potential issues or anomalies in the application.

- It is used for situations where something unexpected has occurred but has not caused the application to fail.

- Warning messages serve as a heads-up for potential problems that should be investigated.

4. ERROR :

- The `ERROR` level is used to report errors or exceptions that have occurred during the execution of the application.

- These messages indicate that something has gone wrong, but the application may still be able to continue running.

5. CRITICAL :

- The `CRITICAL` level is the highest severity level.

- It is used for messages that indicate a critical error or failure that prevents the application from continuing to run correctly.

- Critical messages usually require immediate attention as they can lead to the application's failure.

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

Ans:- To disable all logging messages in your software using Python's `logging` module, you can add the following line of code:

```python

logging.disable(logging.CRITICAL)

```

The `logging.disable(level)` method sets the logging level threshold to the specified level or higher. In this case, it sets the threshold to `CRITICAL`, which is the highest level. As a result, all log messages with levels lower than or equal to `CRITICAL` (i.e., `CRITICAL`, `ERROR`, `WARNING`, `INFO`, and `DEBUG`) will be effectively disabled, and they won't be logged or displayed.

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

Ans:-- Using logging messages is better than using `print()` for displaying messages in your software because logging offers granular control, configurability, consistency, runtime control, logging levels, production readiness, integration, debugging support, non-disruptiveness, and security features, making it suitable for various application scenarios, including production environments and large-scale applications. `print()` lacks these advantages and is less flexible and maintainable for handling messages.

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

Ans::-- In a debugger, such as the one provided by many integrated development environments (IDEs) and programming languages, you often have three important debugging actions represented by buttons or keyboard shortcuts: Step Over, Step In, and Step Out. These actions help you navigate through your code while debugging, and they have distinct purposes:

1. Step Over (or Next) :

- Purpose : The Step Over command allows you to execute the current line of code and move to the next line in your code.

- Use Case : It is useful when you want to execute a function call or a block of code without stepping into the details of that function or block. If the current line contains a function call, the debugger will execute that function and stop at the next line outside the function call.

2. Step In (or Step Into) :

- Purpose : The Step In command allows you to enter and debug the details of a function or method called on the current line.

- Use Case : Use Step In when you want to dive into the code of a function or method to inspect its behavior, variables, and execution flow. It's useful for debugging inside library or framework code or when you want to understand what's happening within a specific function.

3. Step Out :

- Purpose : The Step Out command is used to continue execution until the current function or method is completed and return to the line where the function was called.

- Use Case : Step Out is handy when you have stepped into a function using Step In and want to quickly return to the caller's context without executing every line of the function's code. It allows you to "step out" of the current function and move up the call stack.

Here's a simple analogy to help understand the differences:

- Step Over : It's like walking past a closed door without entering. You execute the current line and move to the next line.

- Step In : It's like opening the door and entering a room. You dive into a function or method to inspect its details.

- Step Out : It's like exiting the room and returning to the hallway. You leave the current function's context and return to the caller's context.

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

Ans:-- When you click "Continue" in a debugger, the debugger will allow your program to run without interruption until it encounters a breakpoint, raises an exception, completes its execution, or until you manually stop it. It resumes normal program execution until one of these conditions is met.

A breakpoint is a debugging feature that allows you to pause the execution of a program at a specific point or line of code during debugging. It is a tool used by developers to inspect the program's state, variables, and behavior at that particular moment. The concept of a breakpoint is fundamental in the debugging process and offers the following key functionalities:

1. Pause Execution : When a breakpoint is encountered, the program's execution is temporarily halted. This gives you the opportunity to examine the program's current state and behavior at that specific point in the code.

2. Inspect Variables : While paused at a breakpoint, you can inspect the values of variables and data structures in the program. This allows you to check the values of variables and understand how they change as the program runs.

3. Step Through Code : Breakpoints are often used in conjunction with step-by-step debugging actions like "Step Into," "Step Over," and "Step Out." These actions enable you to move through the code one line at a time, allowing you to understand the flow and behavior of the program.

4. Identify Issues : Breakpoints are especially useful for identifying bugs and issues in the code. By examining the program's state at a breakpoint, you can determine whether variables have unexpected values or if the code is following an incorrect logic path.

5. Conditional Breakpoints : Some debuggers allow you to set conditional breakpoints. These breakpoints will only pause the program if a specified condition is met. For example, you can set a breakpoint to trigger when a variable reaches a specific value.

6. Temporary Debugging Aid : Breakpoints are often used as temporary aids during the debugging process. Once the issue is identified and resolved, you can remove or disable the breakpoint.

Here's how to set a breakpoint in a typical debugging environment:

1. Open your code in the debugger.

2. Locate the line of code where you want to set the breakpoint.

3. Click in the margin next to the line of code, or use a keyboard shortcut to set the breakpoint.

4. Run your program in debug mode.

5. When the program reaches the line with the breakpoint, it will pause, and you can start inspecting the program's state.

In summary, breakpoints are essential tools for debugging software. They allow you to control the flow of your program, inspect variables, and identify and resolve issues more effectively by providing a way to pause the execution and examine the code at specific points of interest.