Name :- Deepawali . B. Mhaisagar
Assignment no 11

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

ANS

Certainly! 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, "Variable '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, indicating that spam is a negative integer, an AssertionError will be raised, and the specified error message "Variable 'spam' cannot be a negative integer." will be displayed.

You can replace the value of spam with any integer value to test the assertion. If spam is not a negative integer, the assertion will pass silently. However, if spam is a negative integer, the assertion will fail and raise an AssertionError.

Assertions are useful for verifying expected conditions during program execution. They help with debugging and ensuring that certain assumptions are met.

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).

ANS

To trigger an AssertionError if the variables eggs and bacon contain strings that are the same as each other, regardless of their cases, you can use the lower() method to convert the strings to lowercase and then compare them. Here's an example of the assert statement:

```
eggs = 'Hello'

bacon = 'hello'

assert eggs.lower() != bacon.lower(), "Variables 'eggs' and 'bacon' cannot have the same string value (case-insensitive)."
```

In this example, the assert statement checks if the lowercase versions of eggs and bacon are not equal using the != operator. If the lowercase versions of the strings are the same, an AssertionError will be raised, and the specified error message will be displayed.

The lower() method is used to convert both eggs and bacon to lowercase before comparison. This ensures that strings like 'hello' and 'Hello' are considered the same.

You can modify the values of eggs and bacon to test the assertion. If the strings are the same (case-insensitive), the assertion will fail and raise an AssertionError.

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

ANS

If you want to create an assert statement that throws an AssertionError every time, you can simply provide a condition that is always False. Here's an example:

assert False, "This assert statement will always throw an AssertionError."

In this case, the condition False will always evaluate to False, triggering the AssertionError. The error message specified after the comma will be displayed when the assertion fails.

This can be useful in situations where you want to explicitly raise an AssertionError to indicate a certain condition that should never occur or to mark a section of code that is still under development.

Keep in mind that it's important to use assertions judiciously and ensure they are meaningful for your specific use case. Assertions are primarily used for debugging and should not be relied upon for error handling or user-facing error messages. They can be disabled in a production environment, so they are most commonly used during development and testing.

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

ANS
To call the logging.debug() function in your software, you need to include the following two lines:

import logging

logging.basicConfig(level=logging.DEBUG)

Explanation:

import logging: This line imports the logging module, which provides the necessary functions and classes for logging in Python.
logging.basicConfig(level=logging.DEBUG): This line configures the logging system to output debug-level messages. It sets the logging level to DEBUG, which is the lowest level for logging. By setting the logging level to DEBUG, it enables the logging of debug messages and any messages with higher severity levels, such as INFO, WARNING, ERROR, and CRITICAL. The basicConfig() function sets up the default configuration for the logging system.

After including these two lines, you can call logging.debug() to log debug-level messages in your software. For example:

```
import logging

logging.basicConfig(level=logging.DEBUG)

# Your code...

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

This will log the debug message to the console or any other configured output destination, depending on the logging configuration.

You can customize the logging configuration further by specifying additional parameters in the basicConfig() function, such as the format of the log messages, the output file, or the logging destination.

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 configure the logging.debug() function to send logging messages to a file named "programLog.txt," you need to include the following two lines in your program:

```
import lOgging

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

Explanation:

import logging: This line imports the logging module, which provides the necessary functions and classes for logging in Python.
logging.basicConfig(filename='programLog.txt', level=logging.DEBUG): This line configures the logging system to output debug-level messages to a file named "programLog.txt". The basicConfig() function sets up the default configuration for the logging system. By specifying the filename parameter, you indicate the file to which the log messages should be written. The level

parameter is set to logging.DEBUG to enable logging of debug messages and any messages with higher severity levels.

After including these two lines, you can call logging.debug() to log debug-level messages in your program, and they will be sent to the "programLog.txt" file.

Here's an example of how you can use it:

```
import logging

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

# Your code...

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

This will log the debug message to the "programLog.txt" file.

You can customize the logging configuration further by specifying additional parameters in the basicConfig() function, such as the format of the log messages, the logging level, or other options.

6. What are the five levels of logging?

ANS

The Python logging module supports five standard levels of logging, listed in ascending order of severity:

DEBUG: Detailed information, typically used for debugging purposes. It provides the most detailed logging information.
INFO: General information about the program's operation. It confirms that things are working as expected.
WARNING: Indicates potential issues or situations that might lead to problems later. The program can continue running, but it's advisable to check the log and take corrective actions.
ERROR: Indicates a specific error or problem that caused the program to fail to perform a function or operation. The program can still continue running with possible workarounds.
CRITICAL: Indicates a severe error that might prevent the program from functioning correctly. It usually leads to the termination of the program.

When setting the logging level using logging.basicConfig(level=...) or when configuring the logger's level in the configuration, messages with a severity level equal to or higher than the specified level will be logged. For example, if you set the level to logging.DEBUG, all five levels of logging will be recorded in the log file or

output, while setting it to logging.ERROR will only record ERROR and CRITICAL messages.

Here's an example of how to configure the logging level:

```
import logging

logging.basicConfig(level=logging.DEBUG)  # Set the logging level to DEBUG

# Your code...

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

logging.info('This is an info message.')

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

logging.error('This is an error message.')

logging.critical('This is a critical message.')
```

In this example, all five levels of logging are used, but only the messages with a level equal to or higher than DEBUG will be recorded in the log output, depending on the configuration.

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, you can add the following line of code:

```
import logging
logging.disable(logging.CRITICAL)
```

This line imports the logging module and calls the disable() function, passing logging.CRITICAL as the argument. This effectively disables all logging messages with a severity level of CRITICAL or lower. As a result, no log messages will be displayed or written to the log file.

It's important to note that once the logging is disabled, it cannot be re-enabled within the same process. Therefore, it's recommended to use this line of code early in your program, preferably at the beginning, to ensure that logging is disabled from the start.

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

Using logging messages instead of print() offers several advantages:

Configurability: With logging, you can easily configure the log output based on different levels (e.g., DEBUG, INFO, WARNING, ERROR). You can enable or disable specific log levels based on your needs. This flexibility allows you to control the verbosity of your program's output without modifying the code
.
Granular control: Logging allows you to categorize and filter messages based on their severity or log level. You can selectively display or store only the important messages while ignoring less critical ones. This helps in debugging and troubleshooting by focusing on relevant information.

Persistence: Logging messages can be directed to various outputs, such as console, files, or external services. By logging to a file, you can retain a record of events and easily review them later. This is particularly useful for long-running programs or server applications where real-time monitoring may not be feasible.

Scalability: Logging provides a standardized approach to handling program output. It supports logging from multiple modules or components within a larger system. This allows you to gather log messages from different parts of your codebase and analyze them collectively.

Flexibility: The logging module in Python offers a wide range of features and options, including log formatting, log rotation, log filtering, and integration with external logging systems. These features make it easier to manage and analyze logs in a structured and organized manner.

Overall, using logging messages instead of print() statements provides a more robust and flexible approach to handling program output, making it easier to maintain and debug your code in the long run.

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

In a debugger, the Step Over, Step In, and Step Out buttons are used to control the execution of the program during debugging. Here are the differences between these buttons:

1. Step Over: The Step Over button allows you to execute the next line of code in the current function or method being debugged. If the current line contains a function call, the debugger will execute the entire function without stepping into it. It is useful when you want to quickly move through a function call without diving into its implementation details.

2. Step In: The Step In button allows you to step into the next line of code, even if it contains a function call. If the current line contains a function call, the debugger will move into the called function, allowing you to debug through its statements. This is helpful when you want to investigate the behavior or variables within a specific function.

3. Step Out: The Step Out button is used to execute the remaining lines of the current function and return to the calling function. It allows you to quickly exit the current function and continue the execution of the program. This is useful when you have stepped into a function and want to quickly return to the higher-level code.

In summary, the Step Over button executes the current line and moves to the next line, the Step In button allows you to step into a function call and debug its statements, and the Step Out button helps you quickly exit the current function and continue the execution at the caller's level. These buttons provide control and flexibility when navigating through the code during debugging.

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

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

1. A breakpoint is encountered: If you have set breakpoints in your code, the debugger will stop when it reaches a line of code with a breakpoint. Breakpoints are specific lines in your code where you want the debugger to pause and allow you to inspect variables and step through the code.

2. An exception is raised: If an unhandled exception occurs during the execution of the program, the debugger will stop at the line of code where the exception is raised. This allows you to examine the state of the program at the point of the exception and debug the issue.

3. The program execution completes: If the program reaches its natural end and all statements have been executed, the debugger will stop. This typically happens when the main function or the script itself has finished executing.

It's important to note that the behavior of the debugger can vary depending on the specific debugger and its configuration settings. The above conditions are general guidelines, but the exact behavior may differ in different debugging environments.

11. What is the concept of a breakpoint?
ANS

In programming, a breakpoint is a designated point in your code where you want the debugger to pause the execution of the program. When the program reaches a line of code with a breakpoint, it temporarily stops running, allowing you to inspect the

state of the program, examine variable values, and step through the code line by line.

Breakpoints are useful for debugging and understanding the flow of execution in your program. By setting breakpoints at strategic locations in your code, you can observe the program's behavior, check the values of variables at specific points, and identify any issues or unexpected behavior.

Once the program execution is paused at a breakpoint, you can use the debugging tools provided by the debugger to analyze the program's state, step through the code, modify variable values, and gather information to troubleshoot and fix any problems.

Breakpoints can be set and managed within integrated development environments (IDEs) or debugging tools. Typically, you can set breakpoints by clicking on the desired line of code or using specific debugging commands provided by the debugging environment.

It's worth noting that breakpoints are temporary and are meant for debugging purposes. Once you have identified and resolved the issues in your code, you can remove the breakpoints to allow the program to run normally without interruption.