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

Ans: To create a program that searches a folder tree for files with a specific file extension and then copies these files to a new folder, you can use the `os` and `shutil` modules in Python. The `os` module is used to traverse the folder tree and find files with the desired extension, while the `shutil` module is used to copy the files to the new folder. Here's a sample program to achieve this:

import os

import shutil

def find\_and\_copy\_files(src\_folder, dst\_folder, file\_extension):

# Ensure the source folder exists

if not os.path.exists(src\_folder):

print(f"Source folder '{src\_folder}' does not exist.")

return

# Ensure the destination folder exists, create if not

os.makedirs(dst\_folder, exist\_ok=True)

# Traverse the source folder tree and find files with the desired extension

for root, dirs, files in os.walk(src\_folder):

for file in files:

if file.endswith(file\_extension):

# Get the full path of the source and destination files

src\_file\_path = os.path.join(root, file)

dst\_file\_path = os.path.join(dst\_folder, file)

# Copy the file to the destination folder

shutil.copy(src\_file\_path, dst\_file\_path)

print(f"Copied: {src\_file\_path} -> {dst\_file\_path}")

if \_\_name\_\_ == "\_\_main\_\_":

source\_folder = "/path/to/source\_folder" # Replace with the actual source folder path

destination\_folder = "/path/to/destination\_folder" # Replace with the actual destination folder path

extension\_to\_copy = ".pdf" # Specify the desired file extension

find\_and\_copy\_files(source\_folder, destination\_folder, extension\_to\_copy)

Replace `/path/to/source\_folder` with the path of the folder where you want to start the search, and `/path/to/destination\_folder` with the path of the folder where you want to copy the matching files. Set `extension\_to\_copy` to the desired file extension (e.g., `".pdf"` or `".jpg"`).

When you run the program, it will traverse the folder tree starting from the source folder and copy all files with the specified extension to the destination folder. If the destination folder does not exist, the program will create it..

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).

Ans: To compare the strings in a case-insensitive manner and trigger an AssertionError if the variables `eggs` and `bacon` contain strings that are the same (ignoring the case), you can use the `lower()` method to convert both strings to lowercase and then use the `assert` statement to check their equality. Here's how you can do it:

eggs = 'hello'

bacon = 'HELLO'

# Convert both strings to lowercase and check for equality

assert eggs.lower() != bacon.lower(), "eggs and bacon strings are the same (case-insensitive)"

In this example, `eggs.lower()` and `bacon.lower()` convert both strings to lowercase, and then the `assert` statement checks whether they are different. If they are the same (ignoring the case), the `assert` statement will raise an `AssertionError` with the given message.You can use this assert statement to ensure that `eggs` and `bacon` contain different strings, considering the case-insensitive comparison. If they are the same (ignoring the case), the program will raise the AssertionError, indicating the unexpected situation.

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

Ans: To create an `assert` statement that throws an `AssertionError` every time, you can simply use the `assert` keyword followed by `False`. Since `False` is a boolean expression that evaluates to `False`, the `assert` statement will always raise an `AssertionError`. Here's how you can do it:

assert False, "This assert statement always raises an AssertionError"

When you run this code, it will immediately raise an `AssertionError` with the given message, indicating that the assert statement failed. This can be useful during development and testing to check whether certain conditions are met, and if not, trigger an error to catch potential bugs or incorrect assumptions in the code.

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

Ans: To use the `logging.debug()` function in your software, you need to import the `logging` module and configure the logging level to allow `DEBUG` messages to be displayed. Here are the two lines that must be present in your software:

import logging

logging.basicConfig(level=logging.DEBUG)

A) `import logging`: This line imports the `logging` module, which provides the logging functionality in Python.

B) `logging.basicConfig(level=logging.DEBUG)`: This line sets up the basic configuration for logging. In this example, it configures the logging level to `DEBUG`, which allows `debug()` messages to be displayed. Other logging levels include `INFO`, `WARNING`, `ERROR`, and `CRITICAL`, in increasing order of severity. If you want to see messages at a specific level or above, you can set the `level` parameter accordingly.

Once these two lines are present in your software, you can use `logging.debug()` to log debug 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 your code

In this example, the debug message will only be displayed if the logging level is set to `DEBUG`. If you set the logging level to a higher level like `WARNING`, the debug message will not be displayed.

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?

Ans: To have `logging.debug()` send logging messages to a file named `programLog.txt`, you need to configure the logging module with a FileHandler that writes the messages to the desired file. Here are the two lines that must be present in your program:

import logging

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

1. `import logging`: This line imports the `logging` module, which provides the logging functionality in Python.

2. `logging.basicConfig(filename='programLog.txt', level=logging.DEBUG)`: This line sets up the basic configuration for logging, specifying the filename as `'programLog.txt'`, and configures the logging level to `DEBUG`. This creates a FileHandler that writes messages with a level of `DEBUG` and above to the file specified.

Once these two lines are present in your program, you can use `logging.debug()` to log debug messages, and they will be written to the `programLog.txt` file. Other log messages with levels equal to or higher than `DEBUG` will also be written to the same file.

1. What are the five levels of logging?

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

a. DEBUG: The lowest level. Used for detailed information, mainly useful for debugging purposes. Typically, these messages are only relevant during development and should not be enabled in production.

b. INFO: Used to confirm that things are working as expected. Provides general operational information about the program's execution, but not necessarily indicative of a problem.

c. WARNING: Indicates that something unexpected happened or a potential issue has been detected. The program is still functioning correctly, but attention may be required.

d. ERROR: Indicates a more serious problem or error that caused the program to fail to perform a specific task. The program might be able to continue running, but the particular operation may not have completed successfully.

e. CRITICAL: The highest level of severity. Indicates a critical error or problem that may lead to the program's termination or unexpected termination. This level is reserved for very severe issues.

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 set the logging level to `logging.NOTSET`. This level effectively disables all logging messages because it is below all the standard logging levels. Here's the line of code you can add to achieve this:

import logging

# Set the logging level to disable all logging messages

logging.basicConfig(level=logging.NOTSET)

With this line of code, all logging messages, including messages at the `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL` levels, will be suppressed, and no logging output will be displayed.It's important to note that while disabling logging messages can be useful in certain situations, it's generally not recommended to do this in production code as logging can be essential for debugging and troubleshooting issues. Instead, you can set the logging level to the appropriate level based on your deployment environment and requirements.

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 several reasons:

i. \*\*Log Levels and Filtering\*\*: The logging module allows you to use different log levels (DEBUG, INFO, WARNING, ERROR, CRITICAL) to categorize and prioritize log messages. You can set the logging level to control which messages get displayed. This enables you to log more detailed information during development (e.g., DEBUG), but in production, you can set the level to a higher value, such as WARNING or ERROR, to only see important messages. With print(), you would need to manually comment out or remove print statements to control the output.

ii. \*\*Flexible Output\*\*: The logging module allows you to configure the output destination for log messages, such as writing them to a file, sending them to a syslog server, or streaming them to a remote logging service. This provides greater flexibility and control over where the logs are stored and how they are managed.

iii. \*\*Granular Control\*\*: With logging, you can log messages from different parts of your codebase and have different loggers for different modules or components. This makes it easier to isolate issues and debug problems in specific areas of your application.

iv. \*\*Performance\*\*: Logging is generally more efficient than print() for large-scale applications because logging is a built-in part of Python and optimized for performance. Print() statements can be slow, especially in situations where there is a high volume of output.

v. \*\*Structured Messages\*\*: Logging messages can be more structured and include additional information like timestamps, log levels, and module names. This structured format makes it easier to analyze and process log data automatically.

vi. \*\*Ease of Maintenance\*\*: Logging statements are easier to manage and maintain compared to print() statements. When you need to remove or adjust the logging messages, you can simply change the logging level or configuration rather than searching for and modifying print() statements throughout the codebase.

Overall, using the logging module offers more control, flexibility, and performance compared to print() statements, making it a better choice for handling log messages in your Python applications.

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

Ans: The Step Over, Step In, and Step Out buttons are commonly found in debugger interfaces, including IDEs and debuggers like pdb (Python Debugger). They allow you to control the execution flow of your program when it is paused at a breakpoint. Here are the differences between these three debugging actions:

A.] \*\*Step Over (also known as Next or Continue):\*\*

- Function: The Step Over button allows you to execute the current line of code and move to the next line.

- Execution: If the current line contains a function call, the Step Over action will execute the entire function and move to the next line in the current context.

- Use case: Use Step Over when you are not interested in diving into the details of the function call and just want to continue execution without exploring the called function's code.

B.] \*\*Step In (also known as Step Into or Dive):\*\*

- Function: The Step In button allows you to enter the function call and start debugging within the called function's context.

- Execution: If the current line contains a function call, the Step In action will "dive" into the function and pause at the first line of the called function.

- Use case: Use Step In when you want to explore the details and execution flow inside the called function. This is helpful when you suspect an issue within the function itself.

C.] \*\*Step Out (also known as Step Return):\*\*

- Function: The Step Out button allows you to finish debugging the current function and return to its caller function.

- Execution: The Step Out action is used when you have stepped into a function using Step In and want to quickly return to the calling function without stepping through every line of the called function.

- Use case: Use Step Out when you have finished debugging within a function and want to return to the higher-level context where the function was called.

To illustrate these concepts, imagine a situation where function `A()` calls function `B()`, and function `B()` calls function `C()`. When debugging, you might encounter a breakpoint inside function `B()`. Here's how you would use each button in this scenario:

- \*\*Step Over:\*\* When you click Step Over, the debugger will execute the entire function `B()` and move to the next line in function `A()`.

- \*\*Step In:\*\* When you click Step In, the debugger will "dive" into function `B()` and pause at the first line inside function `B()`.

- \*\*Step Out:\*\* When you click Step Out, the debugger will finish debugging function `B()` and return to function `A()`.

Using these debugging actions, you can effectively control the flow of execution and explore different parts of your code during debugging to identify and fix issues.

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

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

A. \*\*Breakpoint is encountered:\*\* If there are any breakpoints set in the code, the debugger will stop when it reaches a line with a breakpoint. Breakpoints are specific locations in the code where you want the debugger to pause so that you can inspect the program's state.

B. \*\*Exception is raised:\*\* If an unhandled exception occurs during program execution, the debugger will stop at the line that raises the exception. This allows you to examine the state of the program at the point of the exception and understand the cause of the error.

C. \*\*Program completes execution:\*\* If the program completes its execution successfully without encountering any breakpoints or exceptions, the debugger will stop when the program finishes running.

D. \*\*User interrupt:\*\* You can manually stop the debugger by sending an interrupt signal, such as pressing Ctrl+C in the terminal or using the appropriate interrupt command in the debugger interface.

In general, the "Continue" action allows the program to run freely until one of the above conditions is met. It is useful when you want to observe the program's behavior without stepping through every line of code. When one of the stopping conditions occurs, the debugger will pause execution, and you can examine variables, inspect the call stack, and perform further debugging actions as needed.

11. What is the concept of a breakpoint?

Ans: The concept of a breakpoint is a fundamental feature in debugging tools that allows developers to pause the execution of a program at a specific line of code or at certain conditions during runtime. When a breakpoint is encountered, the program execution is temporarily halted, and the developer gains the ability to inspect the program's state, variables, and memory, as well as step through the code to understand its behavior and find bugs. Some of the keypoints are as follows:

a. \*\*Setting Breakpoints:\*\* Breakpoints can be set at specific lines of code in the source file or at certain conditions (conditional breakpoints). Developers can set breakpoints using debugging tools provided by Integrated Development Environments (IDEs) or command-line debuggers.

b. \*\*Pausing Execution:\*\* When the program execution reaches a line with a breakpoint, it is paused, and the debugger takes control. The program is held at that point until the developer continues the execution or takes other debugging actions.

c. \*\*Inspecting State:\*\* While at the breakpoint, developers can inspect the values of variables, examine the call stack (the sequence of function calls leading to the current point), and analyze the current state of the program.

d. \*\*Stepping Through Code:\*\* After pausing at a breakpoint, the developer can step through the code to move line by line or dive into functions (Step In) or return from functions (Step Out) to observe the program's flow and behavior.

e. \*\*Conditional Breakpoints:\*\* Developers can set breakpoints based on certain conditions. The breakpoint will only be triggered when the specified condition evaluates to true. This is helpful for debugging specific scenarios or problematic areas in the code.

f. \*\*Error Identification:\*\* By setting breakpoints strategically, developers can identify the point at which an error occurs, observe the state of variables, and understand the program's behavior leading up to the error.