Q1. Describe three applications for exception processing.

1. Error Handling: Exception processing in Python allows for the graceful handling of errors that may occur during program execution. By using try-except blocks, specific exceptions can be caught and appropriate actions taken, such as displaying an error message or executing alternative code paths.
2. Input Validation: Exception processing can be utilized to validate user input. By raising exceptions when input data doesn't meet certain criteria, such as incorrect data types or invalid values, developers can ensure the program handles such situations and prompts the user for valid input.

Q2. What happens if you don't do something extra to treat an exception?

If an exception is not handled or treated in any way, it will result in the program terminating abruptly and displaying an error message. This behavior is known as an "unhandled exception." The error message typically includes information about the type of exception that occurred, along with a traceback that indicates the sequence of function calls leading to the exception.

Q3. What are your options for recovering from an exception in your script?

When recovering from an exception in a Python script, you have several options:

1. Handling the Exception: Use a try-except block to catch the exception and execute alternative code or display an appropriate error message.

2. Retry Mechanism: Implement a retry mechanism by enclosing the code in a loop and attempting the operation again after an exception occurs.

3. Logging: Log the exception details to a file or console for analysis and troubleshooting purposes.

4. Graceful Termination: If the exception is critical and cannot be recovered, terminate the script gracefully by exiting the program or notifying the user about the issue.

Q4. Describe two methods for triggering exceptions in your script.

1. Explicitly Raising Exceptions: You can trigger exceptions explicitly in your script by using the `raise` keyword followed by an exception type. For example, `raise ValueError("Invalid input")` raises a `ValueError` exception with a specific error message.

2. Handling Built-in Exceptions: Certain operations or functions in Python may raise exceptions by default. For instance, dividing a number by zero (`1 / 0`) triggers a `ZeroDivisionError`. By performing such operations or invoking functions that may raise exceptions, you can trigger them in your script.

Q5. Identify two methods for specifying actions to be executed at termination time, regardless of whether or not an exception exists.

1. Finally Block: You can use a `finally` block in combination with a `try-except` block to specify actions that will be executed regardless of whether an exception occurs or not. The code within the `finally` block will be executed before the control leaves the `try-except` block, ensuring cleanup or finalization tasks are performed.

2. Context Managers: By using context managers with the `with` statement, you can define actions to be executed at termination time. The `\_\_exit\_\_()` method of the context manager will be called, allowing you to define the necessary cleanup or finalization operations. Context managers provide a convenient way to handle resources and guarantee their proper release.