**Q1. Describe three applications for exception processing.**

**Solution -** Exception processing in Python is a powerful mechanism for handling and responding to exceptional situations or errors that may occur during the execution of a program. Here are three common applications of exception processing in Python:

**Error handling and graceful recovery**: Exception processing allows you to handle errors and exceptions in a structured and controlled manner. You can use try-except blocks to catch specific exceptions and provide alternative code paths or error handling routines. This is particularly useful when interacting with external resources, such as files or network connections, where errors can occur.

**Input validation and user interaction**: Exception processing can be used for input validation, ensuring that the user provides valid data. For example, if your program expects a numeric input, you can use a try-except block to catch a ValueError if the user enters a non-numeric value.

**Resource management and cleanup:** Exception processing is instrumental in handling resource management and cleanup tasks. For instance, if your program is interacting with a database or opening a file, exceptions can be caught to ensure that the resources are properly released and cleaned up, even if an error occurs during their usage.

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

**Solution -** When an exception occurred, if you don't handle it, the program terminates abruptly and the code past the line that caused the exception will not get executed

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

**Solution - When an exception occurs in a script, there are several options for recovering from the exception and continuing the execution of the script. Here are some common approaches:**

**Try-Except block:** The most common way to recover from an exception is by using a try-except block. You enclose the code that may raise an exception within the try block, and then specify one or more except blocks to handle specific exceptions. In the except block, you can include code to handle the exception, log the error, display an error message, or take any other appropriate action. By catching the exception, you can prevent the script from abruptly terminating and continue with the remaining code.

**Retry mechanism:** In some cases, it may be appropriate to retry the operation that raised the exception. For example, if you're making a network request and encounter a connection error, you can implement a retry mechanism that attempts to reconnect and retry the operation a certain number of times before giving up.

**Graceful termination:** In certain situations, it may not be possible or desirable to recover from an exception. Instead, you can perform cleanup tasks and gracefully terminate the script.

**Exception chaining:** Exception chaining allows you to catch an exception, perform some handling or recovery actions, and then raise a new exception or re-raise the original exception with additional information.

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

**Solution -** In Python, you can intentionally trigger exceptions in your script using various methods to handle specific scenarios or to test error handling mechanisms. Here are two common methods for triggering exceptions:

**Raise an exception explicitly**: You can raise an exception explicitly using the raise statement. This allows you to create and raise exceptions of any type or use built-in exceptions. By raising an exception, you can indicate that a certain condition or scenario has occurred that requires special handling. For example, you can raise a ValueError if a function receives invalid input or raise an IOError if there's an issue with file I/O.

**Use built-in functions that raise exceptions:** Python provides built-in functions that can raise specific exceptions based on certain conditions. For instance, the open() function is used to open files, but if the file does not exist, it raises a FileNotFoundError. Similarly, the int() function is used to convert a string to an integer, but if the string cannot be converted to an integer, it raises a ValueError.

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

**Solution -** In Python, there are a couple of methods for specifying actions to be executed at termination time, regardless of whether or not an exception occurs. These methods ensure that certain cleanup or finalization tasks are performed before the program terminates. Here are two common approaches:

**Using the finally block:** The finally block allows you to specify code that will be executed regardless of whether an exception is raised or not. It is typically used in conjunction with a try-except block. The code inside the finally block will always be executed, regardless of whether an exception is caught or propagated.

**Using the atexit module:** The atexit module provides a way to register functions that will be called when the program is about to exit, regardless of whether an exception is raised or not. You can use the atexit.register() function to register a function that should be executed at program termination.