Q1. Describe three applications for exception processing.

A1. Exception processing is a powerful feature of Python that allows programmers to handle errors in their programs gracefully. Here are three applications of exception processing:

1. Input validation: When a program prompts a user for input, there is always the possibility that the user will enter an invalid value. By using exception handling, a programmer can detect and handle invalid input without crashing the program.
2. File handling: When working with files, there are many things that can go wrong, such as missing files, incorrect permissions, and unexpected file formats. By using exception handling, a programmer can handle these errors without causing the program to crash.
3. Network programming: When communicating over a network, there are many things that can go wrong, such as connection failures, timeouts, and incorrect data. By using exception handling, a programmer can handle these errors and take appropriate action, such as retrying the connection or notifying the user.

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

A2. If we don't handle an exception in our code, it will cause the program to terminate abruptly with an error message. This error message typically includes a traceback that indicates where in the code the exception was raised. In some cases, this is fine, as it may be an expected outcome that the user should be aware of (e.g., if the program is unable to access a required file or network resource). However, in most cases, it is better to handle exceptions explicitly in our code to prevent these unhandled exceptions from causing our program to crash and to provide a more user-friendly error message or alternative behavior.

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

A3. When an exception occurs in a Python script, there are several options available for recovering from it. Some of these options include:

1. Using a try-except block: This is a common way to handle exceptions in Python. The try block contains the code that might raise an exception, and the except block contains the code that will be executed if an exception is raised.
2. Raising a custom exception: We can define our own exception classes and raise them in our code when an error condition is detected. This allows us to provide more information about the error and handle it more effectively.
3. Logging: We can log the exception to a file or console, which can be useful for debugging purposes.
4. Graceful degradation: If the code can still continue despite the exception, we can handle the exception and continue executing the rest of the code.
5. Program termination: If the exception is severe and the program cannot continue, we can terminate the program using the **sys.exit()** function or let the exception propagate to the top level and crash the program.

The appropriate option depends on the nature of the exception, the context in which it occurs, and the desired behavior of the program.

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

A4. There are several ways to trigger exceptions in a Python script:

1. Raise an exception manually: We can use the **raise** statement to manually raise an exception. For example, **raise ValueError("Invalid input")** will raise a **ValueError** exception with the message "Invalid input".
2. Call a method that may raise an exception: If we call a method that may raise an exception, we can wrap the call in a try-except block to catch the exception.

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

A5. In Python, we can use the **finally** clause and context managers to specify actions to be executed at termination time, regardless of whether or not an exception exists.

1. The **finally** clause is used to specify a block of code that is executed irrespective of whether an exception occurs or not. It is typically used to perform cleanup operations, such as closing files or network connections. The **finally** clause is always executed, even if the program is terminated by a **KeyboardInterrupt** or a system error.
2. Context managers are used to specify a block of code that is executed before and after a section of code is run. They are typically used to acquire and release resources, such as files or network connections. Context managers are implemented using the **with** statement, which automatically calls the **\_\_enter\_\_** and **\_\_exit\_\_** methods of an object when entering and exiting a context. The **\_\_exit\_\_** method is always called, even if an exception is raised within the context.