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

Ans- Exception processing is an important feature of most programming languages that allows for the detection and handling of errors and exceptional conditions in a program. Here are three common applications for exception processing:

1. Error handling: One of the most common applications of exception processing is for error handling. When an error occurs during program execution, such as a divide-by-zero error or a file-not-found error, an exception can be raised to signal the error. The exception can then be caught and handled by the program, allowing it to recover from the error or terminate gracefully.
2. Resource management: Another important application of exception processing is for resource management. In many programs, resources such as files, network sockets, and database connections need to be opened and closed properly. If an error occurs while using a resource, such as a file handle that cannot be closed, an exception can be raised to signal the error and ensure that the resource is properly released.
3. Input validation: Exception processing can also be used for input validation. When a user inputs data into a program, it is important to validate the input to ensure that it is valid and meets certain criteria. For example, if a user is asked to input a number, the program should raise an exception if the input is not a valid number. By raising an exception, the program can ensure that the input is valid and prevent errors from occurring later in the program.

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

Ans-If an exception is raised in a program and there is no code to handle the exception, the program will terminate abruptly and an error message will be displayed in the console or output window.

This is because when an exception is raised, the program's normal flow of control is interrupted and the interpreter searches for an exception handler to handle the exception. If no exception handler is found, the program terminates and the exception is displayed as an error message.

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

Ans- When an exception is raised in a script, there are several options for recovering from the exception and allowing the program to continue running. Here are some common ways to recover from exceptions:

1. Catch the exception: The most common way to recover from an exception is to catch it using a try/except block. By catching the exception, you can handle it gracefully and continue running the program.
2. Retry the operation: In some cases, it may be possible to recover from an exception by retrying the operation that caused the exception. For example, if a network connection fails, you could try to reconnect and retry the operation. It's important to be careful with retries, however, as retrying too many times could cause the program to get stuck in a loop.
3. Provide a default value: If an exception occurs while retrieving a value from a dictionary or database, you could provide a default value instead of raising an exception. This can be useful when the missing value is not critical to the program's operation.
4. Log the error: If you are unable to recover from an exception, you can log the error to a file or database for later analysis. This can help you identify and fix the underlying issue that caused the exception.

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

Ans- In Python, there are several ways to trigger exceptions in your script. Here are two common methods:

1. Raise an exception: You can explicitly raise an exception using the raise statement. This is useful when you want to trigger an exception based on a certain condition or error.
2. Call a function that raises an exception: Many built-in Python functions and methods can raise exceptions if something goes wrong.

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

Ans-In Python, there are two ways to specify actions to be executed at termination time, regardless of whether or not an exception exists:

1. Using the ‘**try/finally’** block: The **‘finally’** block is always executed before leaving the ‘try/except’ block, regardless of whether an exception was raised or not. This makes it useful for cleaning up resources or releasing locks.
2. Using the ‘**atexit’** module: The **‘atexit’** module provides a way to register functions to be called when the Python interpreter is about to exit. This can be useful for performing cleanup tasks or saving data.

Using either of these methods can help ensure that important tasks are always executed at termination time, even if an exception is raised.