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

Absolutely! Here are three applications for exception processing in Python, along with explanations and examples:

**1. Robust Error Handling**

* **Purpose:** Exception processing prevents your program from crashing unexpectedly due to runtime errors. By using try...except blocks, you can gracefully handle potential problems and provide meaningful information to the user.
* **Example:**

Python

try:

result = 10 / int(input("Enter a number: "))

print("Result:", result)

except ZeroDivisionError:

print("Error: You cannot divide by zero.")

except ValueError:

print("Error: Please enter a valid number.")

* **Explanation:** In this code, we anticipate potential errors: the user might try to divide by zero or enter a non-numeric value. The try...except blocks catch these specific exceptions and provide customized error messages.

**2. Managing Resource Cleanup**

* **Purpose:** Ensures proper release of resources like files, network connections, and database handles. Even if errors occur, exception processing combined with the finally block guarantees that resources are always released.
* **Example:**

Python

try:

file = open("my\_file.txt", "r")

data = file.read()

# Process the data here

except FileNotFoundError:

print("Error: File not found.")

finally:

if file:

file.close()

* **Explanation:** We attempt to open a file within a try block. If the file doesn't exist, the FileNotFoundError is caught. Regardless of whether the file is opened successfully or not, the finally block ensures that the file object is closed, preventing resource leaks.

**3. Validating User Input and Enforcing Constraints**

* **Purpose:** Exception processing helps you guide the user and maintain data integrity by catching and responding to invalid input.
* **Example:**

Python

def get\_age():

while True:

try:

age = int(input("Enter your age: "))

if age < 0:

raise ValueError("Age cannot be negative.")

return age

except ValueError as e:

print("Error:", e)

age = get\_age()

print("Your age is:", age)

* **Explanation:** This code continuously prompts the user for their age until valid input is entered. It uses a ValueError to enforce the rule that age cannot be negative. The custom exception message aids the user in providing correct input.

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

In Python, if you don't do anything to handle an exception, the following happens:

1. **Traceback:** The Python interpreter stops the current execution of your program and prints a "traceback" to the console. This traceback includes:
   * The type of exception that occurred (e.g., ZeroDivisionError, TypeError, FileNotFoundError).
   * A descriptive error message explaining the issue.
   * The line number where the exception occurred.
   * The call stack, showing the chain of functions that led to the exception.
2. **Program Termination:** If the exception remains unhandled, your program will terminate prematurely. This means the rest of your code won't execute.

**Example**

Python

result = 10 / 0 # Attempt to divide by zero

**Output:**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero

**Why Exception Handling is Important**

* **Prevent Crashes:** Handling exceptions stops your program from crashing unexpectedly, making it more robust.
* **Informative Error Messages:** You can provide custom error messages to guide the user or log meaningful error details for debugging.
* **Improved User Experience:** Your application can recover gracefully from errors instead of simply terminating, giving a smoother user experience.

**How to Handle Exceptions**

Use the try...except blocks to catch and handle exceptions:

Python

try:

# Code that might raise an exception

except ExceptionType:

# Code to handle the exception

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

Here are your primary options for recovering from an exception in your Python script:

**1. Retry the Operation**

* **Best for:** Transient errors that might resolve themselves if you try again. Examples include network glitches, temporary file locks, or resource contention.
* **Implementation:**

Python

for attempts in range(3): # Try up to 3 times

try:

result = perform\_unreliable\_operation()

break # Exit the loop if successful

except TemporaryError:

print("Error occurred. Retrying...")

else: # If all retries failed

print("Failed to complete operation after multiple attempts.")

**2. Use a Default Value**

* **Best for:** When you can accept a reasonable substitute in case of an error.
* **Implementation**

Python

try:

user\_input = int(input("Enter a number: "))

except ValueError:

user\_input = 0 # Use default value of 0 on input error

**3. Log the Error and Continue**

* **Best for:** Non-critical errors where execution can proceed, but you want a record of the issue for later troubleshooting.
* **Implementation:**

Python

import logging

try:

# Some code that might raise an exception

except Exception as e:

logging.error("Exception occurred: %s", e)

# Continue with the rest of your program

**4. Propagate the Exception**

* **Best for:** When the current function can't meaningfully recover, but a higher-level function in the call stack might be able to handle it.
* **Implementation:** Let the exception raise normally, or use the raise keyword to re-raise it.

**5. Clean Up and Exit Gracefully**

* **Best for:** Fatal errors where recovery is impossible. Ensure resources are released properly.
* **Implementation:**

Python

try:

# Code that might raise an exception

except CriticalError:

# Do necessary cleanup (close files, release connections, etc.)

print("Critical error. Exiting...")

exit(1) # Exit with an error code

**Important Considerations**

* **Balance Specificity and Generality:** Be specific enough with your except clauses to handle different error types appropriately, but avoid being so specific that you miss related errors.
* **Don't Overuse Exception Handling:** Don't catch exceptions you don't intend to handle meaningfully. Let some exceptions propagate naturally so you can identify underlying problems.
* **Test Your Recovery Logic:** Thoroughly test how your script handles different exceptions to confirm it responds as intended.

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

In Python, here are two methods for triggering exceptions in your script:

**1. Using the raise keyword**

* **Purpose:** Explicitly signal that an error or abnormal condition has occurred, forcing an exception to be raised. This is useful for enforcing constraints, validating input, or signaling problems during complex operations.
* **Example:**

Python

def check\_positive(number):

if number <= 0:

raise ValueError("Number must be positive.")

num = int(input("Enter a positive number: "))

check\_positive(num)

* **Explanation:** The check\_positive function raises a ValueError if the provided number is not positive. This allows you to create custom error conditions and control how your code responds to them.

**2. Implicitly by Python Operations**

* **Purpose:** Many built-in operations and functions in Python automatically raise exceptions when they encounter invalid conditions. This is how Python's internal error-checking mechanisms work.
* **Examples:**
  + **ZeroDivisionError:** Attempting to divide by zero.

Python

result = 10 / 0

* + **TypeError:** Trying to perform an operation on incompatible data types.

Python

"hello" + 10

* + **IndexError:** Accessing an index that's out of bounds in a list or sequence.

Python

my\_list = [1, 2, 3]

print(my\_list[5])

**Important Considerations**

* **raise for Custom Errors:** Use raise when you need to signal an exception based on the specific logic and conditions within your program.
* **Implicit Exceptions from Functions:** Be familiar with the potential exceptions that can be raised by the various Python functions and operations that you use.

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

In Python, here are two methods for specifying actions to be executed at program termination, even if there are exceptions:

**1. The finally Clause**

* **Purpose:** The finally block associated with a try...except suite *always* executes, whether an exception was raised or not. This makes it ideal for essential cleanup tasks related to resource management.
* **Example:**

Python

try:

file = open("my\_data.txt", "r")

# Process file contents

except FileNotFoundError:

print("Error: File not found")

finally:

if file:

file.close() # Ensure file is closed

**Explanation:**

* The try block attempts to open a file.
* If the file isn't found, the FileNotFoundError is caught.
* Regardless of whether an exception happened or not, the finally block **always** executes, closing the file if it was opened successfully.

**2. The atexit Module**

* **Purpose:** Register functions to be executed when your Python program terminates normally (not due to an unhandled fatal exception).
* **Example:**

Python

import atexit

def cleanup\_task():

print("Performing cleanup at program exit")

atexit.register(cleanup\_task)

**Explanation:**

* The atexit.register function adds the cleanup\_task to a list of functions to be called when the interpreter exits.

**Key Points**

* **finally:** Best for cleanup actions related to resources opened in a try block. Executes even if there are exceptions.
* **atexit:** Best for global cleanup tasks at normal program termination. Be aware that functions registered with atexit might not execute if the program crashes due to a severe error.