1.

i)File Handling:

When working with files, various issues may arise, such as missing files, permission errors, or unexpected data formats. Exception processing is used to handle these situations in a controlled manner.

For example, when reading a file, you can use a try block and catch specific exceptions, such as '**FileNotFoundError'** or '**PermissionError'**, to notify the user or take appropriate corrective actions. This prevents the program from crashing and provides meaningful feedback to the user.

```
try:
   with open("file.txt", "r") as file:
      data = file.read()
except FileNotFoundError:
   print("File not found.")
except PermissionError:
   print("Permission denied.")
except Exception as e:
   print("An unexpected error occurred:", str(e))
```

ii)Web Scraping:

In web scraping, developers fetch data from websites. However, websites can change their structure or become temporarily unavailable, leading to potential errors. Exception handling is useful in such scenarios to gracefully handle errors and continue processing.

For example, catching '**HTTPError'**, '**ConnectionError'**, or **'TimeoutError** 'can help manage network-related issues, and catching '**AttributeError'** can handle cases where expected data on a webpage is not found.

```
import requests

try:
   response = requests.get("https://example.com")
   response.raise_for_status()  # Raises an exception for non-200 status codes
   data = response.json()
except requests.exceptions.HTTPError as http_err:
   print("HTTP error occurred:", http_err)
except requests.exceptions.ConnectionError as conn_err:
   print("Connection error:", conn_err)
except requests.exceptions.Timeout as timeout_err:
   print("Request timed out:", timeout_err)
except requests.exceptions.RequestException as req_err:
   print("An unexpected error occurred:", req_err)
```

iii) Database Operations:

When working with databases, exceptions can be thrown due to connection issues, query errors, or constraint violations. Exception handling is vital in such cases to manage database interactions effectively.

For instance, catching **sqlite3.Error'** in '**SQLite' or 'psycopg2.Error'** in PostgreSQL can help address specific database-related errors gracefully.

```
import sqlite3

try:
   connection = sqlite3.connect("database.db")
   cursor = connection.cursor()
   cursor.execute("SELECT * FROM table_name")
   data = cursor.fetchall()
   connection.commit()
except sqlite3.Error as db_err:
   print("Database error occurred:", db_err)
finally:
   if connection:
      connection.close()
```

2.

If you don't do something extra to treat an exception, your program will crash. This is because the Python interpreter will not know how to handle the error, and it will simply stop executing your program. The default behavior of Python is to terminate the program and display an error message, also known as an **unhandled exception**. This can lead to unexpected program termination and a potentially poor user experience
Here's an example to illustrate what happens when an exception is not handled:

```
def divide_by_zero():
   return 1 / 0

try:
   divide_by_zero()
except ZeroDivisionError:
   print("Error: cannot divide by zero")
```

# This will print the error message and then crash

As you can see, the ‘**divide_by_zero()’** function raises a ‘**ZeroDivisionError’** exception. The **try** block catches this exception and prints an error message. However, the **except** block does not do anything else, so the program crashes.

Here is how to handle an exception :

```
def divide_by_zero():
   return 1 / 0
try:
   divide_by_zero()
except ZeroDivisionError:
   print("Error: cannot divide by zero.")
   return None
```

# This will print the error message and then return None

3.

When an exception occurs in your Python script, you have several options for recovering from it and handling the exceptional situation. These options include :

i) Using 'try' and 'except' Blocks:
Enclose the code that may raise an exception inside a try block, and then you define one or more except blocks to handle specific types of exceptions. When an exception occurs within the try block, Python will jump to the corresponding except block that matches the exception's type, allowing you to execute recovery code or provide error messages.

```
try:
   # Code that may raise an exception
   result = divide(5, 0)
   print("Result:", result)
except ZeroDivisionError as zd_err:
   print("Error:", zd_err)
   # Recovery code or user feedback
```

ii) **Using 'else' Block with 'try' :**
The code inside the else block will only execute if no exception occurs in the try block. This can be useful when you want to execute some code specifically when there are no exceptions.

```
try:
   # Code that may raise an exception
   result = divide(5, 2)
except ZeroDivisionError as zd_err:
   print("Error:", zd_err)
   # Recovery code or user feedback
else:
   print("Result:", result)
```

iii) **Using 'finally' Block:**
The finally block is used to define code that should be executed regardless of whether an exception occurred or not. It is often used to perform cleanup tasks, such as closing files or releasing resources, after executing the try and except blocks.

```
try:
   # Code that may raise an exception
   file = open("data.txt", "r")
   data = file.read()
except FileNotFoundError:
   print("File not found.")
   # Recovery code or user feedback
finally:
   if file:
      file.close()  # Always close the file, even if an exception occurred
```

iv) Using Multiple 'except' Blocks:
You can have multiple except blocks to handle different types of exceptions differently. This allows you to tailor your recovery actions based on the specific type of exception raised.

```
try:
    # Code that may raise different types of exceptions
    result = divide(5, "hello")
    print("Result:", result)
except ZeroDivisionError as zd_err:
    print("Division by zero:", zd_err)
except TypeError as type_err:
    print("Type error:", type_err)
    # Recovery code or user feedback
```

4.

i)Using the raise statement:
The "**raise"** statement allows you to explicitly raise an exception of a specified type. You can use this method when you encounter a situation that requires exceptional handling. To trigger an exception using the raise statement, you can follow this syntax:

```
def example_function(number):
    if number <= 0:
        raise ValueError("The input number must be greater than 0.")
    # Rest of the function's code

# Example usage:
try:
    input_number = int(input("Enter a positive number: "))
    example_function(input_number)
except ValueError as ve:
    print(f"Error: {ve}")
```

In this example, if the user enters a non-positive number, the '**ValueError'** exception will be raised with a custom error message

ii)Using built-in functions that raise exceptions:
Python provides several built-in functions that can raise exceptions based on specific conditions. For example:
int() function: If you try to convert a non-integer string to an integer using int(), it will raise a '**ValueError'**
open() function: When attempting to open a file that doesn't exist, the open() function will raise a '**FileNotFoundError.'**
assert statement: The assert statement raises an '**AssertionError'** if its condition evaluates to False.

Here's an example of using the int() function to trigger a ValueError:

```
try:
    age = int(input("Enter your age: "))
except ValueError as ve:
    print(f"Invalid input: {ve}")
```

If the user enters a non-integer value, like "abc," the **int()** function will raise a **ValueError**, which will be caught by the except block

5.

The 'try-finally' block:
The try-finally block is used to ensure that some code is always executed, regardless of whether an exception occurred or not. The code inside the finally block will be executed after the try block, regardless of whether an exception was raised or not.

```
def example_function():
    try:
        # Code that might raise an exception
        result = 10 / 0  # This will raise a ZeroDivisionError
    finally:
        # Code that will always be executed
        print("Finally block: Cleaning up resources.")

# Example usage:
try:
    example_function()
except ZeroDivisionError:
    print("Error: Division by zero occurred.")
```

In this example, even though the '**ZeroDivisionError'** is raised, the code inside the finally block will still be executed, printing "Finally block: Cleaning up resources."

The atexit module:
The '**atexit'** module provides a way to register functions to be executed at program termination, regardless of whether an exception was raised or not. You can use the '**atexit.register()'** function to register one or more functions to be called when the program exits normally

```
import atexit

def cleanup_function():
    print("Cleanup function: Performing cleanup tasks.")

def example_function():
    try:
        # Code that might raise an exception
        result = 10 / 0  # This will raise a ZeroDivisionError
    finally:
        atexit.register(cleanup_function)

# Example usage:
try:
    example_function()
except ZeroDivisionError:
    print("Error: Division by zero occurred.")
```

In this example, even if an exception is raised, the **cleanup_function()** will still be executed when the program exits, printing **"Cleanup function: Performing cleanup tasks."**