Use Case

- **User Input Validation**: Validate user inputs and handle errors gracefully, prompting users to provide correct information.
- **File Operations**: Manage errors that occur during file operations, such as file not found, permission denied, or read/write errors.
- **Network Communication**: Handle network-related errors like timeouts, connection issues, and invalid responses from servers.
- **API Interactions**: Gracefully handle errors when making API requests, such as dealing with unavailable services, rate limits, or invalid API keys.
- **Database Access**: Manage exceptions that occur during database operations, like connection failures, query errors, or transaction issues.
- **Data Parsing**: Handle errors that arise when parsing data from various formats, such as JSON, XML, or CSV, ensuring robust data processing.
- **Arithmetic Operations**: Catch exceptions in mathematical computations, such as division by zero or overflow errors, to maintain program stability.
- **Resource Management**: Ensure that resources (files, network connections, etc.) are properly closed or released even if an error occurs during their use.
- **Logging Errors**: Record detailed error information to log files for debugging and maintenance purposes.
- **User Authentication**: Handle exceptions during user authentication processes, such as invalid credentials or expired tokens.

Error List

- **`ArithmeticError`**: The base class for all errors related to arithmetic operations.
  - `ZeroDivisionError`: Raised when dividing by zero.
  - `OverflowError`: Raised when the result of an arithmetic operation is too large to be represented.
  - `FloatingPointError`: Raised when a floating-point operation fails.
- **`AttributeError`**: Raised when an attribute reference or assignment fails.
- **`EOFError`**: Raised when the `input()` function hits an end-of-file condition (EOF) without reading any data.
- **`ImportError`**: Raised when an import statement fails to find the module definition or when a `from...import` fails to find a name that is to be imported.

    - `ModuleNotFoundError`: A subclass of `ImportError` raised when a module cannot be found.
- **`IndexError`**: Raised when a sequence subscript is out of range.
- **`KeyError`**: Raised when a dictionary key is not found.
- **`KeyboardInterrupt`**: Raised when the user hits the interrupt key (Ctrl+C or Delete).
- **`MemoryError`**: Raised when an operation runs out of memory.
- **`NameError`**: Raised when a local or global name is not found.
    - `UnboundLocalError`: A subclass of `NameError` that is raised when a local variable is referenced before it has been assigned.
- **`OSError`**: The base class for operating system-related errors.
    - `FileNotFoundError`: Raised when a file or directory is requested but doesn't exist.
    - `PermissionError`: Raised when trying to run an operation without the necessary access rights.
    - `IsADirectoryError`: Raised when a file operation is requested on a directory.
- **`RuntimeError`**: Raised when an error is detected that doesn't fall in any other category.
    - `NotImplementedError`: Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.
- **`SyntaxError`**: Raised when the parser encounters a syntax error.
    - `IndentationError`: Raised when there's an indentation issue.
    - `TabError`: Raised when tabs and spaces are mixed inconsistently in indentation.
- **`TypeError`**: Raised when an operation or function is applied to an object of inappropriate type.
- **`ValueError`**: Raised when a function receives an argument of the correct type but an inappropriate value.
- **`StopIteration`**: Raised by the `next()` function to indicate that there are no further items produced by the iterator.
- **`ZeroDivisionError`**: Raised when dividing or performing modulo operations by zero.

⬛ Simple Try-Except

```python
try:
    # Attempt to divide by zero
    result = 10 / 0
except ZeroDivisionError:
    # Handle the division by zero error
    print("Error: Cannot divide by zero.")
```

⬛ Handling Multiple Exceptions

```python
try:
    # Attempt to read a non-existent file and divide by a non-numeric value
    with open("non_existent_file.txt") as file:
        content = file.read()
    result = 10 / int(content)  # Assuming the content should be an integer
except FileNotFoundError:
    # Handle the file not found error
    print("Error: File not found.")
except ValueError:
    # Handle the error if content is not a valid integer
    print("Error: File content is not a valid number.")
except ZeroDivisionError:
    # Handle the division by zero error
    print("Error: Cannot divide by zero.")
```

⬛ Catching All Exceptions

```python
try:
    # Code that may raise an exception
    with open("example.txt") as file:
        content = file.read()
    result = 10 / int(content)
except Exception as e:
    # Handle any exception
    print(f"An error occurred: {e}")
```

⬛ Finally Block

```python
try:
    # Code that may raise an exception
    with open("example.txt") as file:
        content = file.read()
    result = 10 / int(content)
except ZeroDivisionError:
    # Handle the division by zero error
    print("Error: Cannot divide by zero.")
finally:
    # This block will be executed no matter what
    print("Execution completed.")
```

⬛ Finally Block

```
try:
    # Code that may raise an exception
    with open("example.txt") as file:
        content = file.read()
    result = 10 / int(content)
except ZeroDivisionError:
    # Handle the division by zero error
    print("Error: Cannot divide by zero.")
except FileNotFoundError:
    # Handle the file not found error
    print("Error: File not found.")
except ValueError:
    # Handle the error if content is not a valid integer
    print("Error: File content is not a valid number.")
else:
    # This block will be executed if no exceptions occur
    print("Division successful, result is:", result)
finally:
    # This block will be executed no matter what
    print("Execution completed.")
```

⬛ Combined Example

```
try:
    with open("example.txt", "r") as file:
        content = file.read()
    # Attempting to convert content to integer and perform division
    number = int(content.strip())
    result = 10 / number
except FileNotFoundError:
    print("Error: File not found.")
except ValueError:
    print("Error: File content is not a valid number.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
else:
    print("File read and division successful, result is:", result)
finally:
    print("Execution completed.")
```