Exception Handling

Exceptions

- Exceptions are unforeseen errors or problems that can occur during the execution of a Python program.
- These errors can cause the program to terminate abruptly if not handled properly. Python provides a robust mechanism called "**exception handling**" to manage such errors gracefully.

ZeroDivisionError,
TypeError,
ValueError,
IOError,

IndexError,
KeyError,
NameError

Types of Exception

- ZeroDivisionError,
- TypeError,
- ValueError,
- IOError,
- IndexError,
- KeyError,
- NameError

Example of Exception

- 102/0 **ZeroDivisionError**
- Open(xyz.txt). //xyz.txt not present in the folder **IOError**
- Int(“Hello”) **TypeError**
- date_str = "2023-07-32"
- date_obj = datetime.strptime(date_str, "%Y-%m-%d") **ValueError**

Try and Except Statement

- Try and except statements are used to catch and handle exceptions in Python.
- Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Handling Exception

```
try:
    # Code that may raise an exception
    result = 10 / 0  # Raises ZeroDivisionError
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

Find Exception type

ZeroDivisionError: This exception is raised when an attempt is made to divide a number by zero.

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

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
```

```
numerator = 10
denominator = 0
result = numerator/denominator
print(result)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-4-09c4a9b60a05> in <cell line: 3>()
      1 numerator = 10
      2 denominator = 0
----> 3 result = numerator/denominator
      4 print(result)

ZeroDivisionError: division by zero
```

0s

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.
```

```
Error: Denominator cannot be 0.
```

TypeError: This exception is raised when an operation or function is applied to an object of the wrong type.

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

```
try:
    num = int(input("Enter a number: "))
except TypeError:
    print("Error: Invalid input. Please enter an integer.")
```

IOError: This exception is raised when an I/O operation, such as reading or writing a file, fails due to an input/output error.

```
with open("non_existent_file.txt", "r") as file:
        content = file.read()
```

```
try:
    with open("non_existent_file.txt", "r") as file:
        content = file.read()
except IOError:
    print("Error: File not found.")
```

IndexError: This exception is raised when an index is out of range for a list, tuple, or other sequence types.

```
my_list = [1, 2, 3]
print(my_list[5])
```

```
try:
    my_list = [1, 2, 3]
    print(my_list[5])
except IndexError:
    print("Error: Index out of range.")
```

KeyError: This exception is raised when a key is not found in a dictionary.

```
my_dict = {"name": "John", "age": 30}
print(my_dict["gender"])
```

```
try:
    my_dict = {"name": "John", "age": 30}
    print(my_dict["gender"])
except KeyError:
    print("Error: Key not found in the dictionary.")
```

NameError: This exception is raised when a variable or function name is not found in the current scope.

print(x) //x is not defined previously

```
try:
    print(x)
except NameError:
    print("Error: Variable x is not defined.")
```

result = 10 + "5"

```
try:
    result = 10 + "5"
except TypeError:
    print("Error: Cannot add integer and string.")
```

result = "Hello" / 2

```
try:
    result = "Hello" / 2
except TypeError:
    print("Error: Unsupported operator for strings.")
```

result = sum(range(1, "5")

```
try:
    result = sum(range(1, "5"))
except TypeError:
    print("Error: Incorrect argument type for range().")
```

num = int("123", base=2)

```
try:
    num = int("123", base=2)
except ValueError:
    print("Error: Invalid base for int() conversion.")
```

value = float("12.3.4")

```
try:
    value = float("12.3.4")
except ValueError:
    print("Error: Incorrect float format.")
```

Multiple error possibility

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a valid integer.")
```

Finally: Something that must execute

- The finally block, if present, will always be executed, whether an exception occurs or not.
- It is used for cleanup operations like closing files or releasing resources.

```
try:
    # Code that may raise an exception
    file = open("example.txt", "r")
    # Perform some file operations
finally:
    file.close()  # File will be closed regardless of exceptions
```

0s

```
try:
    numerator = 10
    denominator = 0

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

finally:
    print("This is finally block.")
```

```
Error: Denominator cannot be 0.
This is finally block.
```

Else clause

- The else clause is executed if no exceptions occur inside the try block.
- It is typically used to specify code that should be executed when the try block runs without any exceptions.

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Invalid input. Please enter a valid integer.")
else:
    print("Division result:", result)
```

Custom Exception

- Instead of using the built-in exception handling mechanisms, it demonstrates how to create and use a custom exception.
- Instead of catching a general IOError, which could signify various issues (like permission errors, file not found, etc.), the custom exception specifies that the error is related to a missing file.

```
class FileNotFoundError(Exception):
    def __init__(self, file_name):
        self.file_name = file_name

    def __str__(self):
        return f"File not found: {self.file_name}"

def read_file(file_name):
    try:
        with open(file_name, "r") as file:
            content = file.read()
            print("File content:", content)
    except IOError:
        raise FileNotFoundError(file_name)

if __name__ == "__main__":
    try:
        file_name = "example.txt"
        read_file(file_name)
    except FileNotFoundError as e:
        print(f"Error occurred: {e}")
```

Custom Exception:

```
Error occurred: File not found: example.txt
```

- **Explanation:**
- **Custom Exception Class:**
  - Defines a custom exception FileNotFoundError that inherits from the base Exception class.
  - The __init__ method initializes the exception with the file name.
  - The __str__ method provides a custom error message.
- **Function to Read File:**
  - read_file attempts to open and read a file.
  - If successful, it prints the content.
  - If an IOError occurs (e.g., the file does not exist), it raises a FileNotFoundError.
- **Main Execution Block:**
  - The if __name__ == "__main__": block ensures the code runs only when the script is executed directly.
  - It tries to read a file named example.txt.
  - If a FileNotFoundError is raised, it prints an error message.

Problem

- You are working on a project to build a custom text processing tool that reads input from various sources, processes the text data, and stores the results in an output file. As part of this project, you need to implement a robust exception handling mechanism to handle potential errors that may arise during the text processing.
- The tool needs to perform the following steps:

1. Read the input data from a file specified by the user.
2. Process the text data by performing various operations, such as counting words, calculating character frequencies, and generating word clouds.
3. Store the processed results in an output file.

Task

- Your task is to design a Python program that incorporates appropriate exception handling to handle the following situations:

1. File Not Found Error: If the user provides an invalid file path or the input file is not found, your program should raise a custom exception FileNotFoundError with a suitable error message.
2. Invalid Input Data: During text processing, if any unexpected input data is encountered (e.g., non-string values or missing data), your program should raise a custom exception InvalidInputDataError with relevant details.
3. Disk Space Full: If the output file cannot be written due to insufficient disk space, your program should raise a custom exception DiskSpaceFullError

Additionally

You can use Python's built-in modules, such as os for file handling and logging for logging exceptions

- The program should have the following features:
- The custom exception classes should inherit from the base Exception class and provide meaningful error messages.
- Proper logging should be implemented to capture details about the exceptions that occur during text processing.
- The program should provide a user-friendly interface, allowing the user to enter the input file path and choose the desired text processing operations.
- The processed results should be stored in an output file with a suitable format (e.g., JSON, CSV, or plain text).