DATE : 17 june 2024

DAY : Monday

TOPICS : Exception Handling, File Handling

Introduction to Exception Handling

Exception handling in Python is a mechanism to respond to runtime errors, preventing the program from crashing and allowing the program to handle errors gracefully. It helps in debugging, maintaining clean code, and providing user-friendly error messages.

Key Concepts

1. Exception: An exception is an error that occurs during the execution of a program. When an exception is raised, the normal flow of the program is interrupted.

2. Try Block: The code that might raise an exception is placed inside a try block.

3. Except Block: The code that handles the exception is placed inside an except block.

4. Else Block: The code inside the else block is executed if no exceptions are raised.

5. Finally Block: The code inside the finally block is executed regardless of whether an exception is raised or not.

6. Raise: Used to raise an exception manually.

try:

```
 # code that may raise an exception
```

except ExceptionType:

```
 # code that runs if the exception occurs
```

else:

```
 # code that runs if no exception occurs
```

finally:

```
 # code that runs no matter what
```

Example 1: Handling Division by Zero

```
def divide(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        return "Cannot divide by zero!"
    else:
        return result
    finally:
        print("Execution of divide function complete.")

print(divide(10, 2))  # Output: 5.0
print(divide(10, 0))  # Output: Cannot divide by zero!
```

```
Execution of divide function complete.
5.0
Execution of divide function complete.
Cannot divide by zero!
```

Example 2: Handling File Operations

```
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            data = file.read()
    except FileNotFoundError:
        return "File not found!"
    except IOError:
        return "Error reading file!"
    else:
        return data
    finally:
        print("Execution of read_file function complete.")

print(read_file("existing_file.txt"))  # Output: (contents of the file)
print(read_file("nonexistent_file.txt"))  # Output: File not found!
```

```
Execution of read_file function complete.
File not found!
Execution of read_file function complete.
File not found!
```

Example 3: Handling Multiple Exceptions

```
def process_input(value):
    try:
        result = int(value)
    except ValueError:
        return "Invalid input! Please enter a number."
    except TypeError:
        return "Invalid type! Please enter a valid input."
    else:
        return f"Valid input: {result}"
    finally:
        print("Execution of process_input function complete.")

print(process_input("10"))  # Output: Valid input: 10
print(process_input("abc"))  # Output: Invalid input! Please enter a number.
print(process_input(None))   # Output: Invalid type! Please enter a valid input.
```

```
Execution of process_input function complete.
Valid input: 10
Execution of process_input function complete.
Invalid input! Please enter a number.
Execution of process_input function complete.
Invalid type! Please enter a valid input.
```

Example 4: Custom Exception

```
class NegativeValueError(Exception):
    def __init__(self, value):
        self.value = value
        self.message = f"Negative value error: {value}"
        super().__init__(self.message)

def check_positive(value):
    try:
        if value < 0:
            raise NegativeValueError(value)
        return "Value is positive."
    except NegativeValueError as e:
        return str(e)
    finally:
        print("Execution of check_positive function complete.")

print(check_positive(10))   # Output: Value is positive.
print(check_positive(-5))   # Output: Negative value error: -5
```

```
Execution of check_positive function complete.
Value is positive.
Execution of check_positive function complete.
Negative value error: -5
```

Common Built-in Exceptions

1. IndexError
2. KeyError
3. ValueError
4. TypeError

5. ZeroDivisionError
6. FileNotFoundError
7. IOError
8. ImportError
9. AttributeError
10. RuntimeError

1. IndexError Scenario: Accessing an invalid index in a list.

```
def get_list_element(lst, index):
    try:
        return lst[index]
    except IndexError as e:
        return f"IndexError: {e}"

my_list = [1, 2, 3]
print(get_list_element(my_list, 2))  # Output: 3
print(get_list_element(my_list, 5))  # Output: IndexError: list index out of range
```

```
3
IndexError: list index out of range
```

2. KeyError Scenario: Accessing a non-existent key in a dictionary.

```
def get_dict_value(d, key):
    try:
        return d[key]
    except KeyError as e:
        return f"KeyError: {e}"

my_dict = {'a': 1, 'b': 2}
print(get_dict_value(my_dict, 'a'))  # Output: 1
print(get_dict_value(my_dict, 'c'))  # Output: KeyError: 'c'
```

```
1
KeyError: 'c'
```

3. ValueError Scenario: Converting an invalid string to an integer.

```
def convert_to_int(value):
    try:
        return int(value)
    except ValueError as e:
        return f"ValueError: {e}"

print(convert_to_int("123"))  # Output: 123
print(convert_to_int("abc"))  # Output: ValueError: invalid literal for int() with base 10:
```

```
123
ValueError: invalid literal for int() with base 10: 'abc'
```

4. TypeError Scenario: Performing an invalid operation on incompatible types.

```
def add_numbers(a, b):
    try:
        return a + b
    except TypeError as e:
        return f"TypeError: {e}"

print(add_numbers(10, 5))
print(add_numbers(10, "five"))
```

```
15
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

5. ZeroDivisionError Scenario: Dividing a number by zero.

```
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError as e:
        return f"ZeroDivisionError: {e}"

print(divide(10, 2))
print(divide(10, 0))
```

```
5.0
ZeroDivisionError: division by zero
```

6. FileNotFoundError Scenario: Trying to open a non-existent file.

```
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            return file.read()
    except FileNotFoundError as e:
        return f"FileNotFoundError: {e}"

print(read_file("existing_file.txt"))
print(read_file("nonexistent_file.txt"))
```

```
FileNotFoundError: [Errno 2] No such file or directory: 'existing_file.txt'
FileNotFoundError: [Errno 2] No such file or directory: 'nonexistent_file.txt'
```

7. IOError Scenario: Error occurs during input/output operation.

```
def write_file(file_path, content):
    try:
        with open(file_path, 'w') as file:
            file.write(content)
    except IOError as e:
        return f"IOError: {e}"

print(write_file("/path/to/readonly_file.txt", "Some content"))
```

```
IOError: [Errno 2] No such file or directory: '/path/to/readonly_file.txt'
```

8. ImportError Scenario: Importing a non-existent module.

```
try:
    import non_existent_module
except ImportError as e:
    print(f"ImportError: {e}")
```

```
ImportError: No module named 'non_existent_module'
```

9. AttributeError Scenario: Accessing an invalid attribute of an object.

```
class MyClass:
    def __init__(self, value):
        self.value = value

obj = MyClass(10)
try:
    print(obj.non_existent_attribute)
except AttributeError as e:
    print(f"AttributeError: {e}")
```

```
AttributeError: 'MyClass' object has no attribute 'non_existent_attribute'
```

10. RuntimeError Scenario: General runtime error not covered by other categories.

```
def raise_runtime_error():
    try:
        raise RuntimeError("This is a runtime error")
    except RuntimeError as e:
        return f"RuntimeError: {e}"

print(raise_runtime_error())
```

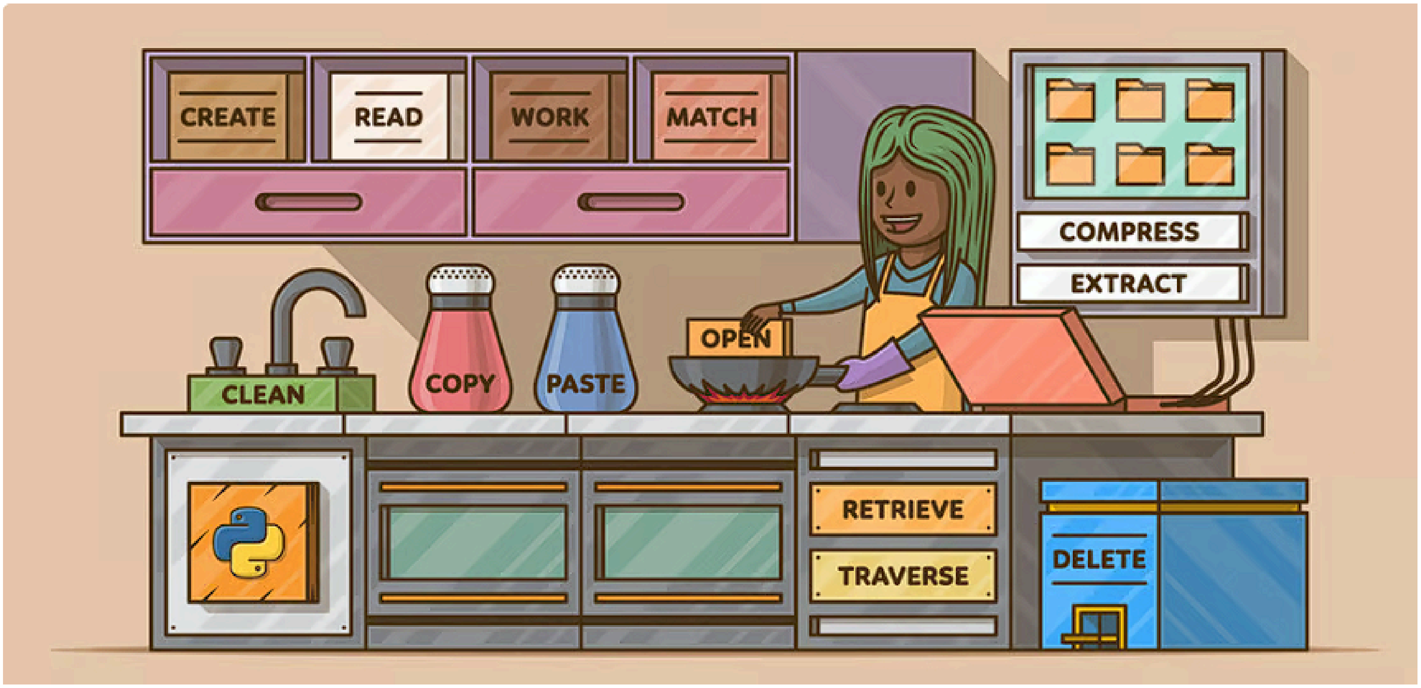
```
RuntimeError: This is a runtime error
```

Introduction to File Handling

File handling is an essential aspect of programming that involves reading from and writing to files. Python provides built-in functions and modules to handle files, allowing you to create, read, write, and manipulate files in various ways.

It is a powerful and versatile tool that can be used to perform a wide range of operations. However, it is important to carefully consider the advantages and disadvantages of file handling when writing Python programs, to ensure that the code is secure, reliable, and performs well.

Python supports file handling and allows users to handle files i.e., to read and write files, along with many other file handling options, to operate on files. The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, like other concepts of Python, this concept here is also easy and short. Python treats files differently as text or binary and this is important. Each line of code includes a sequence of characters, and they form a text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun. Let’s start with the reading and writing files.



Key Concepts

File Modes: Modes specify the purpose of opening a file. Common modes include:

-'r': Read (default mode). Opens a file for reading.

-'w': Write. Opens a file for writing (creates a new file or truncates an existing file).

-'a': Append. Opens a file for appending (creates a new file if it doesn't exist).

-'b': Binary. Used with other modes to handle binary files.

-'+': Update. Opens a file for reading and writing.

File Methods:

open(): Opens a file and returns a file object.

read(): Reads the entire content of a file.

readline(): Reads a single line from a file.

readlines(): Reads all lines from a file and returns them as a list.

write(): Writes a string to a file.

writelines(): Writes a list of strings to a file.

close(): Closes the file.

With Statement: Ensures proper acquisition and release of resources, automatically closing the file when the block inside the with statement is exited.

Basic File Operations

Creating a file

```
# Python code to create a file
file = open('eid.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

Scenario: Reading the entire content of a file.

```
file = open('eid.txt','w')
file.write("Aarzoo, Plaha, Alisha, and Harhdeep are present in the class.")

file.close()
```

```
def read_file(file_path):
    try:
        with open(file_path, 'r') as file:
            content = file.read()
            return content
    except FileNotFoundError as e:
        return f"FileNotFoundError: {e}"
    except IOError as e:
        return f"IOError: {e}"

# Usage
file_content = read_file("/content/eid.txt")
print(file_content)
```

```
Aarzoo, Plaha, Alisha, and Harhdeep are present in the class.
```

Example 2: Writing to a File

Scenario: Writing a string to a file.

```
def write_file(file_path, content):
    try:
        with open(file_path, 'w') as file:
            file.write(content)
            return "Write successful"
    except IOError as e:
        return f"IOError: {e}"

# Usage
result = write_file("/content/vid.txt", "Hello, World!")
print(result)
```

```
Write successful
```

Example 3: Appending to a File

Scenario: Appending a string to an existing file.

```
def append_file(file_path, content):
    try:
        with open(file_path, 'a') as file:
            file.write(content)
            return "Append successful"
    except IOError as e:
        return f"IOError: {e}"

# Usage
result = append_file("/content/eid.txt", "\nAppended text.")
print(result)  # Output: Append successful
```

```
Append successful
```

Example 4: Reading a File Line by Line

Scenario: Reading a file line by line and printing each line.

```
def read_file_by_line(file_path):
    try:
        with open(file_path, 'r') as file:
            for line in file:
                print(line.strip())
    except FileNotFoundError as e:
        print(f"FileNotFoundError: {e}")
    except IOError as e:
        print(f"IOError: {e}")

# Usage
read_file_by_line("/content/eid.txt")  # Output: (each line of example.txt)
```

```
Aarzoo, Plaha, Alisha, and Harhdeep are present in the class.
Appended text.
```

Example 5: Working with Binary Files

Scenario: Writing and reading binary data.

```
def write_binary_file(file_path, data):
    try:
        with open(file_path, 'wb') as file:
            file.write(data)
            return "Write successful"
    except IOError as e:
        return f"IOError: {e}"

def read_binary_file(file_path):
    try:
        with open(file_path, 'rb') as file:
            data = file.read()
            return data
    except FileNotFoundError as e:
        return f"FileNotFoundError: {e}"
    except IOError as e:
        return f"IOError: {e}"

# Usage
binary_data = bytes([104, 101, 108, 108, 111])  # Binary data for "hello"
write_result = write_binary_file("example.bin", binary_data)
print(write_result)

read_result = read_binary_file("example.bin")
print(read_result)
```

```
Write successful
b'hello'
```

Advantages of File Handling in Python

Versatility: File handling in Python allows you to perform a wide range of operations, such as creating, reading, writing, appending, renaming, and deleting files.

Flexibility: File handling in Python is highly flexible, as it allows you to work with different file types (e.g. text files, binary files, CSV files, etc.), and to perform different operations on files (e.g. read, write, append, etc.).

User–friendly: Python provides a user-friendly interface for file handling, making it easy to create, read, and manipulate files.

Cross-platform: Python file-handling functions work across different platforms (e.g. Windows, Mac, Linux), allowing for seamless integration and compatibility

Disadvantages of File Handling in Python

- **Error-prone:**File handling operations in Python can be prone to errors, especially if the code is not carefully written or if there are issues with the file system (e.g. file permissions, file locks, etc.).
- **Security risks**: File handling in Python can also pose security risks, especially if the program accepts user input that can be used to access or modify sensitive files on the system.
- **Complexity:** File handling in Python can be complex, especially when working with more advanced file formats or operations. Careful attention must be paid to the code to ensure that files are handled properly and securely.