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.

Common Built-in Exceptions

1. IndexError
2. KeyError
3. ValueError
4. TypeError
5. ZeroDivisionError
6. FileNotFoundError
7. IOError
8. ImportError
9. AttributeError
10. RuntimeError

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!
```

```
print(10/0)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-4-fe01563e1bc6> in <cell line: 1>()
----> 1 print(10/0)

ZeroDivisionError: division 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
```

Best Practices for Exception Handling

Catch Specific Exceptions: Always catch specific exceptions instead of a generic Exception to handle errors more precisely.

Use Finally Block: Ensure that necessary cleanup (e.g., closing files or releasing resources) is performed by using the finally block.

Avoid Silent Failures: Do not use empty except blocks; always provide some logging or error message.

Log Exceptions: Use logging to record exceptions for future debugging and monitoring.

Use Custom Exceptions: Define custom exceptions for specific error conditions in your application to provide more meaningful error handling.

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))
```

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"))
```

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"))
```

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")
```