01 Exception Handling

Exception Handling in Python

Table of Contents

---

1. Introduction to Exception Handling

What are Exceptions?

Exceptions are events that occur during program execution that disrupt the normal flow of instructions. When Python encounters an error, it creates an exception object. If not handled properly, the program terminates abruptly.

Common built-in exceptions include:

- `ZeroDivisionError`: Occurs when dividing by zero
- `TypeError`: Occurs when an operation is performed on an inappropriate data type
- `ValueError`: Occurs when a function receives an argument of the correct type but inappropriate value
- `NameError`: Occurs when a local or global name is not found
- `IndexError`: Occurs when trying to access an index that is out of range
- `KeyError`: Occurs when a dictionary key is not found
- `FileNotFoundError`: Occurs when trying to open a file that doesn't exist
- `ImportError`: Occurs when an import statement fails
- `AttributeError`: Occurs when an attribute reference or assignment fails

Why Exception Handling?

Exception handling allows you to:

- **Gracefully manage errors** without crashing your program
- **Separate error-handling code** from regular code
- **Provide meaningful feedback** to users
- **Clean up resources** (files, network connections) even when errors occur
- **Make code more robust and maintainable**

Common Built-in Exceptions

Before diving deep, let's understand some common exceptions:

```python
# ZeroDivisionError - Division by zero
result = 10 / 0

# TypeError - Wrong type operation
result = "hello" + 5

# ValueError - Right type, wrong value
num = int("abc")

# KeyError - Key doesn't exist in dictionary
my_dict = {'a': 1}
value = my_dict['b']

# IndexError - Index out of range
my_list = [1, 2, 3]
item = my_list[10]

# FileNotFoundError - File doesn't exist
file = open('nonexistent.txt', 'r')

# AttributeError - Attribute doesn't exist
result = "hello".nonexistent_method()
```

---

2. Understanding try-except-else-finally Blocks

Basic try-except Structure

The `try-except` block is the fundamental mechanism for catching and handling exceptions.

Syntax:

```python
try:
    # Code that might raise an exception
    risky_operation()
except ExceptionType:
    # Code to handle the exception
    handle_error()
```

Examples: Basic Exception Handling

Example 1: Handling Division by Zero

PYTHON

```python
# Without exception handling - Program crashes
def divide_unsafe(a, b):
    return a / b

# This would crash the program
# result = divide_unsafe(10, 0)  # ZeroDivisionError!

# With exception handling - Program continues
def divide_safe(a, b):
    try:
        result = a / b
        return result
    except ZeroDivisionError:
        print("Error: Cannot divide by zero!")
        return None

# Program continues running
result = divide_safe(10, 0)
print(f"Result: {result}")  # Output: Result: None
result = divide_safe(10, 2)
print(f"Result: {result}")  # Output: Result: 5.0
```

Example 2: Handling User Input

PYTHON

```python
def get_integer_input():
    try:
        user_input = input("Enter a number: ")
        number = int(user_input)
        return number
    except ValueError:
        print("That's not a valid integer!")
        return None

# Usage
num = get_integer_input()
if num is not None:
    print(f"You entered: {num}")
```

Multiple except Blocks

You can handle different exceptions differently by using multiple `except` blocks.

Example 3: Multiple Exception Types

PYTHON

```python
def process_data(data_list, index):
    try:
        # Multiple operations that could fail
        value = data_list[index]
        result = 100 / value
        return result
    except IndexError:
        print("Error: Index is out of range!")
        return None
    except ZeroDivisionError:
        print("Error: Value at index is zero, cannot divide!")
        return None
    except TypeError:
        print("Error: Invalid data type!")
        return None

# Testing different scenarios
print(process_data([1, 2, 3], 1))     # Works: 100/2 = 50.0
print(process_data([1, 2, 3], 10))    # IndexError
print(process_data([1, 0, 3], 1))     # ZeroDivisionError
print(process_data("not a list", 0))  # TypeError
```

Catching Multiple Exceptions Together

When you want to handle multiple exceptions the same way, group them in a tuple.

Example 4: Grouping Exceptions

PYTHON

```python
def safe_calculation(a, b, c):
    try:
        result = (a + b) / c
        return result
    except (TypeError, ValueError, ZeroDivisionError) as e:
        print(f"Calculation error occurred: {type(e).__name__}")
        print(f"Error message: {e}")
        return None

# Testing
print(safe_calculation(10, 20, 5))    # Works: 6.0
print(safe_calculation(10, 20, 0))    # ZeroDivisionError
print(safe_calculation("10", 20, 5))  # TypeError
```

The `as` Keyword - Accessing Exception Object

Use `as` to capture the exception object and access its details.

In Python, **exceptions are objects**—instances of classes that inherit from the built-in `BaseException` class (most commonly from `Exception`). When an error occurs (like trying to open a file that doesn't exist), Python **raises** an exception by creating an instance of a specific exception class (e.g., `FileNotFoundError`).

```python
except FileNotFoundError as e:
```

you're **catching** that exception object and assigning it to the variable `e`. This object contains useful information about what went wrong.

Example 5: Exception Object Details

```python
def detailed_error_handling(filename):
    try:
        with open(filename, 'r') as file:
            content = file.read()
            return content
    except FileNotFoundError as e:
        print(f"Error Type: {type(e).__name__}")
        print(f"Error Message: {e}")
        print(f"Error Args: {e.args}")
        return None
    except PermissionError as e:
        print(f"Permission denied: {e}")
        return None

# Testing
content = detailed_error_handling("nonexistent.txt")
```

Bare except Clause (Not Recommended)

A bare `except` catches ALL exceptions, including system exits and keyboard interrupts.

Example 6: Bare except (Use with Caution)

```python
# NOT RECOMMENDED - Too broad
def risky_bare_except():
    try:
        # Some operation
        value = 10 / 0
    except:
        print("Something went wrong!")

# BETTER - Catch Exception base class
def better_broad_except():
    try:
        value = 10 / 0
    except Exception as e:
        print(f"An error occurred: {e}")
        # This won't catch KeyboardInterrupt or SystemExit

# Testing
better_broad_except()
```

The else Clause

The `else` block executes **only if no exception was raised** in the try block.

Example 7: Using else Clause

```python
def divide_with_else(a, b):
    try:
        result = a / b
    except ZeroDivisionError:
        print("Cannot divide by zero!")
        return None
    else:
        # This runs ONLY if no exception occurred
        print("Division successful!")
        return result

# Testing
print(divide_with_else(10, 2))  # Prints "Division successful!" then 5.0
print(divide_with_else(10, 0))  # Prints "Cannot divide by zero!" then None
```

Example 8: else with File Operations

```python
def read_file_safely(filename):
    try:
        file = open(filename, 'r')
    except FileNotFoundError:
        print(f"File {filename} not found!")
    else:
        # Only executes if file opened successfully
        print(f"File {filename} opened successfully!")
        content = file.read()
        file.close()
        return content

    return None

# Testing
content = read_file_safely("example.txt")
```

The finally Clause

The `finally` block **always executes**, regardless of whether an exception occurred or not. It's perfect for cleanup operations.

Example 9: finally for Cleanup

PYTHON

```python
def demonstrate_finally():
    try:
        print("1. Trying to open file...")
        file = open('data.txt', 'r')
        print("2. File opened successfully")
        content = file.read()
        # Simulate an error
        result = 10 / 0
    except FileNotFoundError:
        print("3. File not found!")
    except ZeroDivisionError:
        print("3. Division by zero error!")
    else:
        print("4. No errors occurred")
    finally:
        print("5. Finally block - This ALWAYS executes!")
        # Cleanup code here
        try:
            file.close()
            print("6. File closed")
        except:
            print("6. No file to close")

# Testing
demonstrate_finally()
```

Example 10: finally for Resource Management

PYTHON

```python
def database_operation(db_name):
    connection = None
    try:
        print(f"Connecting to {db_name}...")
        # Simulate database connection
        connection = f"Connection to {db_name}"
        print("Connection established")

        # Simulate operation that might fail
        if db_name == "bad_db":
            raise ValueError("Invalid database!")

        print("Performing operations...")
        return "Success"

    except ValueError as e:
        print(f"Error: {e}")
        return "Failed"
    finally:
        # This cleanup ALWAYS happens
        if connection:
            print(f"Closing connection...")
            connection = None

# Testing
print(database_operation("good_db"))
print("\n---\n")
print(database_operation("bad_db"))
```

Complete try-except-else-finally Example

Example 11: All Components Together

PYTHON

```python
def comprehensive_example(filename, divisor):
    """
    Demonstrates try-except-else-finally with all components
    """
    file = None
    try:
        print("=== TRY BLOCK ===")
        # Operation 1: File opening
        file = open(filename, 'r')
        print(f"File '{filename}' opened successfully")

        # Operation 2: Read and convert
        content = file.read().strip()
        number = int(content)
        print(f"Read number: {number}")

        # Operation 3: Division
        result = number / divisor
        print(f"Division result: {result}")

    except FileNotFoundError:
        print("=== EXCEPT BLOCK (FileNotFoundError) ===")
        print(f"Error: File '{filename}' not found!")
        return None

    except ValueError:
        print("=== EXCEPT BLOCK (ValueError) ===")
        print("Error: File content is not a valid number!")
        return None

    except ZeroDivisionError:
        print("=== EXCEPT BLOCK (ZeroDivisionError) ===")
        print("Error: Cannot divide by zero!")
        return None

    else:
        print("=== ELSE BLOCK ===")
        print("All operations completed successfully!")
        return result

    finally:
        print("=== FINALLY BLOCK ===")
        if file:
            file.close()
            print("File closed successfully")
```

```
        print("Cleanup completed\n")

# Create a test file
with open('test_number.txt', 'w') as f:
    f.write('42')

# Testing different scenarios
print("Scenario 1: Everything works")
result = comprehensive_example('test_number.txt', 2)
print(f"Final result: {result}\n")

print("Scenario 2: File not found")
result = comprehensive_example('missing.txt', 2)
print(f"Final result: {result}\n")

print("Scenario 3: Division by zero")
result = comprehensive_example('test_number.txt', 0)
print(f"Final result: {result}\n")

# Cleanup
import os
os.remove('test_number.txt')
```

Nested try-except Blocks

You can nest try-except blocks for fine-grained error handling.

Example 12: Nested Exception Handling

PYTHON

```python
def nested_exception_handling():
    print("Outer try block starting ... ")

    try:
        # Outer try block
        print("Attempting outer operation ... ")
        outer_list = [1, 2, 3]

        try:
            # Inner try block
            print("Attempting inner operation ... ")
            value = outer_list[1]
            result = 10 / value
            print(f"Inner operation successful: {result}")

        except ZeroDivisionError:
            print("Inner except: Division by zero")

        except IndexError:
            print("Inner except: Index error")

        # This will cause an error
        problematic = outer_list[10]

    except IndexError:
        print("Outer except: Caught index error from outer operation")

    except Exception as e:
        print(f"Outer except: Caught unexpected error: {e}")

    finally:
        print("Outer finally: Cleanup complete")

# Testing
nested_exception_handling()
```

---

3. Raising Exceptions

The raise Statement

You can manually trigger exceptions using the `raise` statement. This is useful for:

- **Input validation**
- **Enforcing business rules**

- **Creating custom error conditions**

Syntax:

```python
raise ExceptionType("Error message")
```

Basic raise Examples

Example 13: Simple raise

```python
def check_positive(number):
    if number < 0:
        raise ValueError("Number must be positive!")
    return number

# Testing
try:
    result = check_positive(10)
    print(f"Valid number: {result}")

    result = check_positive(-5)  # This raises ValueError
    print(f"Valid number: {result}")  # This won't execute

except ValueError as e:
    print(f"Error caught: {e}")
```

Example 14: Input Validation

```python
def create_user(username, age):
    """Create a user with validation"""

    # Validate username
    if not username:
        raise ValueError("Username cannot be empty!")

    if len(username) < 3:
        raise ValueError("Username must be at least 3 characters!")

    # Validate age
    if not isinstance(age, int):
        raise TypeError("Age must be an integer!")

    if age < 0:
        raise ValueError("Age cannot be negative!")

    if age < 18:
        raise ValueError("User must be at least 18 years old!")

    return {"username": username, "age": age}

# Testing
try:
    user1 = create_user("john_doe", 25)
    print(f"User created: {user1}")

    user2 = create_user("ab", 30)  # Too short username
except ValueError as e:
    print(f"Validation error: {e}")
except TypeError as e:
    print(f"Type error: {e}")
```

Re-raising Exceptions

You can catch an exception, do some processing, and then re-raise it.

Example 15: Re-raising Exceptions

DESIGN:

This is a classic **separation of concerns**:
Log in one layer, **handle** in another.

The design has **two responsibilities**:

1. `process_with_logging()`:
   → Do the core work
   → **Log** the error (for debugging/auditing)
   → But **don't handle** the error — let someone else deal with it
2. `main_function()`:
   → Call the worker function
   → **Actually handle** the error (e.g., show user-friendly message, recover, etc.)

PYTHON

```python
def process_with_logging(data):
    try:
        result = 100 / data
        return result
    except ZeroDivisionError:
        print("Logging: Division by zero attempted")
        # Log to file, database, etc.
        raise  # Re-raise the same exception

def main_function():
    try:
        result = process_with_logging(0)
    except ZeroDivisionError:
        print("Main: Handling the re-raised exception")

# Testing
main_function()
```

`raise` (bare)

- **Re-raises the exact same exception object** that was caught
- **Preserves the original traceback** (stack trace)
- This is **almost always what you want** when re-raising

Example 16: Transforming Exceptions

```python
def read_config_file(filename):
    try:
        with open(filename, 'r') as f:
            content = f.read()
            return content
    except FileNotFoundError:
        # Transform into a more specific exception
        raise RuntimeError(f"Configuration file '{filename}' is missing!") from
None

def initialize_app():
    try:
        config = read_config_file('config.ini')
    except RuntimeError as e:
        print(f"Application initialization failed: {e}")

# Testing
initialize_app()
```

raise from - Exception Chaining

Use `raise ... from ...` to show the relationship between exceptions.

Example 17: Exception Chaining

```python
def parse_config(config_string):
    try:
        # Simulate parsing JSON
        if '{' not in config_string:
            raise ValueError("Invalid JSON format")
    except ValueError as e:
        # Chain exceptions to show causation
        raise RuntimeError("Configuration parsing failed") from e

def load_application_config():
    try:
        config = "invalid config"
        parse_config(config)
    except RuntimeError as e:
        print(f"Error: {e}")
        print(f"Caused by: {e.__cause__}")

# Testing
load_application_config()
```

Example 18: Advanced Exception Chaining

```python
class ConfigError(Exception):
    """Custom exception for configuration errors"""
    pass

def validate_database_config(config):
    try:
        host = config['host']
        port = int(config['port'])

        if port < 1 or port > 65535:
            raise ValueError(f"Invalid port number: {port}")

    except KeyError as e:
        raise ConfigError(f"Missing required config key: {e}") from e
    except ValueError as e:
        raise ConfigError(f"Invalid configuration value") from e

# Testing
try:
    config = {'host': 'localhost', 'port': '99999'}
    validate_database_config(config)
except ConfigError as e:
    print(f"Configuration Error: {e}")
    print(f"Original cause: {type(e.__cause__).__name__}: {e.__cause__}")
```

---

4. User-Defined Exceptions

Why Create Custom Exceptions?

Custom exceptions allow you to:

- **Create domain-specific error types**
- **Provide more meaningful error messages**
- **Organize exception handling by category**
- **Add custom attributes and methods**

Basic Custom Exception

Example 19: Simple Custom Exception

PYTHON

```python
class InvalidAgeError(Exception):
    """Raised when age is invalid"""
    pass

def set_age(age):
    if age < 0:
        raise InvalidAgeError("Age cannot be negative!")
    if age > 150:
        raise InvalidAgeError("Age seems unrealistic!")
    return age

# Testing
try:
    age = set_age(200)
except InvalidAgeError as e:
    print(f"Custom error caught: {e}")
```

Custom Exception with Attributes

Example 20: Exception with Additional Data

```python
class InsufficientFundsError(Exception):
    """Raised when account has insufficient funds"""

    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        self.shortage = amount - balance
        message = f"Insufficient funds: Need ${amount}, but only ${balance}
available"
        super().__init__(message)

class BankAccount:
    def __init__(self, balance):
        self.balance = balance

    def withdraw(self, amount):
        if amount > self.balance:
            raise InsufficientFundsError(self.balance, amount)
        self.balance -= amount
        return self.balance

# Testing
account = BankAccount(100)

try:
    print(f"Balance: ${account.balance}")
    account.withdraw(50)
    print(f"After withdrawal: ${account.balance}")

    account.withdraw(100)  # This will fail

except InsufficientFundsError as e:
    print(f"\nError: {e}")
    print(f"Current balance: ${e.balance}")
    print(f"Attempted withdrawal: ${e.amount}")
    print(f"Shortage: ${e.shortage}")
```

Exception Hierarchy in Custom Exceptions

Example 21: Creating an Exception Hierarchy

PYTHON

```python
# Base exception for all application errors
class ApplicationError(Exception):
    """Base class for all application exceptions"""
    pass

# Category: Authentication errors
class AuthenticationError(ApplicationError):
    """Base class for authentication errors"""
    pass

class InvalidCredentialsError(AuthenticationError):
    """Invalid username or password"""
    pass

class AccountLockedError(AuthenticationError):
    """Account is locked"""
    pass

class SessionExpiredError(AuthenticationError):
    """Session has expired"""
    pass

# Category: Database errors
class DatabaseError(ApplicationError):
    """Base class for database errors"""
    pass

class ConnectionError(DatabaseError):
    """Cannot connect to database"""
    pass

class QueryError(DatabaseError):
    """Error executing query"""
    pass

# Usage example
def login(username, password):
    if username == "":
        raise InvalidCredentialsError("Username cannot be empty")
    if password == "":
        raise InvalidCredentialsError("Password cannot be empty")
    if username == "locked_user":
        raise AccountLockedError("This account has been locked")
    return "Login successful"
```

```
    def execute_query(query):
        if not query:
            raise QueryError("Query cannot be empty")
        if "DROP" in query.upper():
            raise QueryError("DROP operations are not allowed")
        return "Query executed"

    # Testing - Can catch specific or general exceptions
    try:
        login("locked_user", "password123")
    except InvalidCredentialsError as e:
        print(f"Credentials error: {e}")
    except AccountLockedError as e:
        print(f"Account locked: {e}")
    except AuthenticationError as e:  # Catches all authentication errors
        print(f"Authentication error: {e}")

    try:
        execute_query("DROP TABLE users")
    except DatabaseError as e:  # Catches all database errors
        print(f"Database error: {e}")
    except ApplicationError as e:  # Catches all application errors
        print(f"Application error: {e}")
```

Advanced Custom Exception Example

Example 22: Feature-Rich Custom Exception

PYTHON

```python
import datetime

class ValidationError(Exception):
    """Advanced validation exception with multiple features"""

    def __init__(self, field_name, value, reason, suggestions=None):
        self.field_name = field_name
        self.value = value
        self.reason = reason
        self.suggestions = suggestions or []
        self.timestamp = datetime.datetime.now()

        # Construct detailed message
        message = f"Validation failed for '{field_name}'"
        message += f"\nValue: {value}"
        message += f"\nReason: {reason}"

        if self.suggestions:
            message += f"\nSuggestions: {', '.join(self.suggestions)}"

        super().__init__(message)

    def log(self):
        """Log the error details"""
        print(f"[{self.timestamp}] ValidationError")
        print(f"  Field: {self.field_name}")
        print(f"  Value: {self.value}")
        print(f"  Reason: {self.reason}")

class UserValidator:
    @staticmethod
    def validate_email(email):
        if '@' not in email:
            raise ValidationError(
                field_name="email",
                value=email,
                reason="Email must contain '@' symbol",
                suggestions=["user@example.com", "name@domain.com"]
            )
        if '.' not in email.split('@')[1]:
            raise ValidationError(
                field_name="email",
                value=email,
                reason="Email domain must contain a dot",
                suggestions=["user@example.com"]
```

```
            )

    @staticmethod
    def validate_password(password):
        if len(password) < 8:
            raise ValidationError(
                field_name="password",
                value="*" * len(password),  # Hide actual password
                reason="Password must be at least 8 characters",
                suggestions=["Use a mix of letters, numbers, and symbols"]
            )

# Testing
try:
    UserValidator.validate_email("invalid-email")
except ValidationError as e:
    print(e)
    print("\n--- Error Details ---")
    e.log()

print("\n" + "="*50 + "\n")

try:
    UserValidator.validate_password("short")
except ValidationError as e:
    print(e)
```

Context Managers with Custom Exceptions

Example 23: Custom Exception in Context Manager

This concept will be covered later!

PYTHON

```python
class FileProcessingError(Exception):
    """Raised when file processing fails"""
    pass

class SafeFileProcessor:
    """Context manager with custom exception handling"""

    def __init__(self, filename):
        self.filename = filename
        self.file = None

    def __enter__(self):
        try:
            self.file = open(self.filename, 'r')
            return self
        except FileNotFoundError:
            raise FileProcessingError(
                f"Cannot process '{self.filename}': File not found"
            )
        except PermissionError:
            raise FileProcessingError(
                f"Cannot process '{self.filename}': Permission denied"
            )

    def __exit__(self, exc_type, exc_val, exc_tb):
        if self.file:
            self.file.close()

        # Transform any unhandled exception
        if exc_type is not None and exc_type != FileProcessingError:
            raise FileProcessingError(
                f"Error processing '{self.filename}': {exc_val}"
            ) from exc_val

        return False  # Don't suppress exceptions

    def read(self):
        if self.file:
            return self.file.read()
        return None

# Testing
try:
    with SafeFileProcessor('data.txt') as processor:
        content = processor.read()
```

```
            print(content)
    except FileProcessingError as e:
        print(f"Processing failed: {e}")
```

1. `exc_type` → **The exception *class***

- Type: `type` (a class)
- Example: `<class 'FileNotFoundError'>`
- This tells you **what kind** of error occurred.

```
if exc_type is FileNotFoundError:
    print("File not found!")
```

PYTHON

2. `exc_val` → **The exception *instance***

- Type: **an instance of the exception class**
- Example: `FileNotFoundError(2, 'No such file or directory')`
- This is the **actual error object** — same as what you'd get in `except ... as e`.

3. `exc_tb` → **The *traceback* object**

- Type: `traceback`
- This is the **stack trace** — a low-level object that records **where the exception happened** (file, line number, function calls).

✅ Normally, you don't use it directly — but it's used by:

- `traceback.format_exception()`
- Python's default error printer
- Debuggers

```
import traceback
if exc_tb:
    print("".join(traceback.format_tb(exc_tb)))
```

PYTHON

---

5. Exception Hierarchy

Understanding Python's Exception Hierarchy

Python exceptions are organized in a hierarchy. Understanding this helps you catch exceptions at the right level.

```
BaseException
├── SystemExit
├── KeyboardInterrupt
├── GeneratorExit
└── Exception
    ├── StopIteration
    ├── ArithmeticError
    │   ├── FloatingPointError
    │   ├── OverflowError
    │   └── ZeroDivisionError
    ├── AssertionError
    ├── AttributeError
    ├── BufferError
    ├── EOFError
    ├── ImportError
    │   └── ModuleNotFoundError
    ├── LookupError
    │   ├── IndexError
    │   └── KeyError
    ├── MemoryError
    ├── NameError
    │   └── UnboundLocalError
    ├── OSError
    │   ├── FileNotFoundError
    │   ├── PermissionError
    │   └── TimeoutError
    ├── RuntimeError
    │   ├── NotImplementedError
    │   └── RecursionError
    ├── TypeError
    ├── ValueError
    │   └── UnicodeError
    └── Warning
        ├── DeprecationWarning
        ├── UserWarning
        └── FutureWarning
```

Example 24: Exploring Exception Hierarchy

```python
def show_exception_hierarchy(exception_class, indent=0):
    """Recursively display exception hierarchy"""
    print("  " * indent + exception_class.__name__)
    for subclass in exception_class.__subclasses__():
        show_exception_hierarchy(subclass, indent + 1)

# Show the hierarchy
print("Exception Hierarchy from Exception base class:")
show_exception_hierarchy(Exception)
```

Catching at Different Hierarchy Levels

Example 25: Hierarchy in Action

```python
def demonstrate_hierarchy():
    """Show how exception catching works with hierarchy"""

    operations = [
        ("Index out of range", lambda: [1, 2, 3][10]),
        ("Key not found", lambda: {'a': 1}['b']),
        ("Division by zero", lambda: 10 / 0),
        ("Type error", lambda: "hello" + 5),
    ]

    for description, operation in operations:
        print(f"\n--- {description} ---")

        # Try 1: Catch specific exception
        try:
            operation()
        except IndexError as e:
            print(f"Caught IndexError: {e}")
        except KeyError as e:
            print(f"Caught KeyError: {e}")
        except ZeroDivisionError as e:
            print(f"Caught ZeroDivisionError: {e}")
        except TypeError as e:
            print(f"Caught TypeError: {e}")

# Testing
demonstrate_hierarchy()
```

Example 26: Catching Parent Exceptions

```python
def catch_at_different_levels(operation_type):
    """Demonstrate catching at different hierarchy levels"""

    try:
        if operation_type == "index":
            result = [1, 2, 3][10]
        elif operation_type == "key":
            result = {'a': 1}['b']
        elif operation_type == "divide":
            result = 10 / 0

    except LookupError as e:
        # LookupError is parent of IndexError and KeyError
        print(f"Caught LookupError (parent): {type(e).__name__} - {e}")

    except ArithmeticError as e:
        # ArithmeticError is parent of ZeroDivisionError
        print(f"Caught ArithmeticError (parent): {type(e).__name__} - {e}")

    except Exception as e:
        # Exception is parent of most exceptions
        print(f"Caught Exception (grandparent): {type(e).__name__} - {e}")

# Testing
catch_at_different_levels("index")  # Caught by LookupError
catch_at_different_levels("key")    # Caught by LookupError
catch_at_different_levels("divide") # Caught by ArithmeticError
```

Order Matters in Exception Handling

Example 27: Exception Catching Order

PYTHON

```python
def wrong_order_example():
    """Demonstrates why order matters"""
    try:
        result = [1, 2, 3][10]
    except Exception as e:
        # This catches everything!
        print(f"Caught by Exception: {e}")
    except IndexError as e:
        # This will NEVER execute (unreachable)
        print(f"Caught by IndexError: {e}")

def correct_order_example():
    """Correct order: specific to general"""
    try:
        result = [1, 2, 3][10]
    except IndexError as e:
        # Specific exception first
        print(f"Caught by IndexError: {e}")
    except LookupError as e:
        # More general
        print(f"Caught by LookupError: {e}")
    except Exception as e:
        # Most general last
        print(f"Caught by Exception: {e}")

# Testing
print("Wrong order:")
wrong_order_example()

print("\nCorrect order:")
correct_order_example()
```

Best Practices for Exception Handling

1. **Be Specific with Exception Types**: Catch specific exceptions rather than using a generic `except` clause to avoid masking bugs.
2. **Don't Silence Exceptions Unnecessarily**: Only catch exceptions you can properly handle. Don't use empty `except` blocks without good reason.
3. **Use finally for Cleanup**: Use the `finally` clause for cleanup operations like closing files and connections.
4. **Log Exceptions Properly**: Include contextual information and proper stack traces when logging exceptions.
5. **Create Custom Exceptions for Domain Logic**: Define custom exceptions that extend from built-in exceptions for application-specific error handling.

6. **Clean up Resources Properly**: Use context managers (`with` statements) to ensure resources are properly cleaned up.
7. **Centralize Error Handling**: Implement error handling at the appropriate levels of your application. Consider using middleware or decorators for repetitive error handling.
8. **Use Exception Chaining**: Use `raise ... from exc` to preserve the original exception information.
9. **Only Handle Exceptions That You Can Actually Recover From**: If you can't recover from an error, it's often better to let it propagate up to a higher level.
10. **Separate Business Logic from Error Handling**: Keep your code clean by separating normal business logic from error handling code.

Conclusion

Proper exception handling is a critical aspect of writing robust Python code. By understanding the exception hierarchy, using appropriate try-except-else-finally blocks, and following best practices, you can create code that gracefully handles errors and continues to function in the face of unexpected situations.