

Python Exception Handling

A Comprehensive Guide

Table of Contents

1. [Introduction to Exceptions](#)
2. [Types of Exceptions](#)
3. [Basic Exception Handling](#)
4. [Multiple Exception Handling](#)
5. [The else Clause](#)
6. [The finally Clause](#)
7. [Creating Custom Exceptions](#)
8. [Best Practices for Exception Handling](#)
9. [Common Pitfalls](#)
10. [Advanced Exception Handling](#)
11. [Assignments](#)

Introduction to Exceptions

Exceptions are events that occur during the execution of a program that disrupt the normal flow of instructions. In Python, exceptions are objects that represent errors.

What are Exceptions?

An exception is an event that occurs during the execution of a program that disrupts the normal flow of the program's instructions. When a Python script encounters a situation it can't handle, it raises an exception.

Why Handle Exceptions?

- Prevents program crashes
- Provides meaningful error messages
- Allows graceful degradation
- Improves user experience
- Makes debugging easier

Types of Exceptions

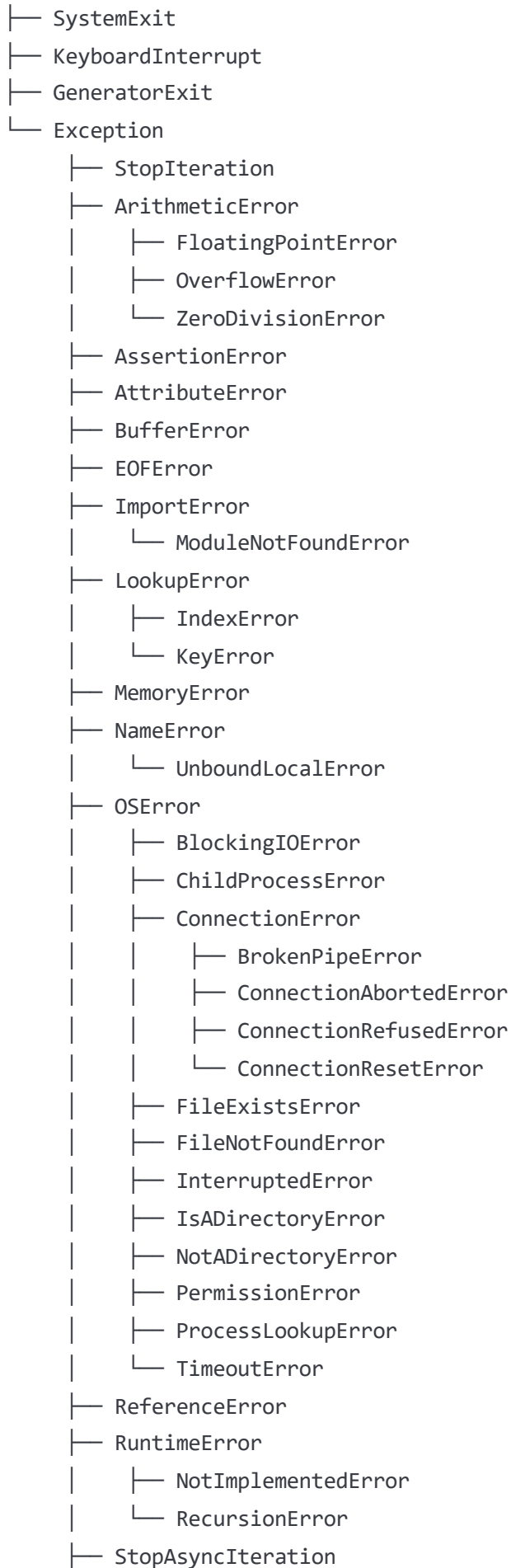
Python has many built-in exceptions. Here are some of the most common ones:

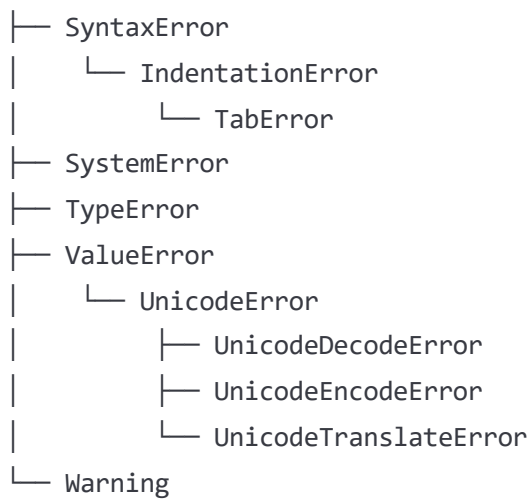
Common Built-in Exceptions

- `SyntaxError`: Raised when there is a syntax error in the code
- `TypeError`: Raised when an operation is performed on an inappropriate data type
- `ValueError`: Raised when a function receives an argument of the correct type but inappropriate value
- `NameError`: Raised when a local or global name is not found
- `IndexError`: Raised when an index is out of range
- `KeyError`: Raised when a dictionary key is not found
- `FileNotFoundError`: Raised when a file or directory is requested but doesn't exist
- `IOError`: Raised when an I/O operation (such as opening a file) fails
- `ZeroDivisionError`: Raised when division or modulo by zero is performed
- `ImportError`: Raised when an import statement fails
- `AttributeError`: Raised when an attribute reference or assignment fails

Exception Hierarchy

BaseException





Basic Exception Handling

The try-except Block

The basic structure for exception handling in Python is the `try-except` block:

```
python

try:
    # Code that might raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Code to handle the exception
    print("Error: Division by zero")
```

Catching General Exceptions

You can catch all exceptions with a general except clause, but this is generally discouraged:

```
python

try:
    # Code that might raise an exception
    result = int("abc")
except Exception as e:
    # Code to handle the exception
    print(f"An error occurred: {e}")
```

Multiple Exception Handling

You can handle different exceptions in different ways:

python

```
try:
    # Code that might raise multiple exceptions
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    # Handles ValueError (e.g., if input is not a number)
    print("Error: Please enter a valid number")
except ZeroDivisionError:
    # Handles ZeroDivisionError
    print("Error: Cannot divide by zero")
```

Catching Multiple Exceptions in One Block

You can handle multiple exceptions in the same way:

python

```
try:
    # Code that might raise multiple exceptions
    file = open("non_existent_file.txt", "r")
    contents = file.read()
    file.close()
except (FileNotFoundError, PermissionError) as e:
    # Handles both FileNotFoundError and PermissionError
    print(f"File error: {e}")
```

The else Clause

The `else` clause runs if no exceptions were raised in the `try` block:

python

```
try:
    # Code that might raise an exception
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Error: Please enter a valid number")
except ZeroDivisionError:
    print("Error: Cannot divide by zero")
else:
    # This will only run if no exceptions were raised
    print(f"Result: {result}")
```

The finally Clause

The `finally` clause runs regardless of whether an exception was raised:

python

```
try:
    file = open("sample.txt", "r")
    content = file.read()
except FileNotFoundError:
    print("Error: File not found")
finally:
    # This will always execute, even if an exception was raised
    # It's a good place for cleanup code (like closing files)
    try:
        file.close()
    except:
        pass # If file was never opened, closing will fail
    print("Execution completed")
```

Creating Custom Exceptions

You can create your own exception classes by inheriting from `Exception`:

python

```
class InsufficientFundsError(Exception):
    """Raised when a withdrawal from an account would result in a negative balance"""
    def __init__(self, balance, amount):
        self.balance = balance
        self.amount = amount
        self.message = f"Cannot withdraw ${amount}. Account balance is ${balance}"
        super().__init__(self.message)

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

try:
    new_balance = withdraw(100, 150)
except InsufficientFundsError as e:
    print(e)
```

Best Practices for Exception Handling

1. Be Specific

Catch specific exceptions rather than using a bare except clause.

python

```
# Good
try:
    # Code
except ValueError:
    # Handle ValueError

# Avoid
try:
    # Code
except: # This will catch all exceptions, including KeyboardInterrupt, SystemExit, etc.
    # Handle exception
```

2. Keep try Blocks Small

Only wrap the specific code that might raise an exception.

python


Good

```
def process_file(filename):
    try:
        file = open(filename, "r")
    except FileNotFoundError:
        print(f"Error: {filename} not found.")
        return None

    # Process the file (outside the try block since this part isn't expected to raise FileNotFoundError)
    contents = file.read()
    file.close()
    return contents
```

Avoid

```
def process_file(filename):
    try:
        file = open(filename, "r")
        contents = file.read() # This could raise other exceptions not related to file opening
        file.close()
        return contents
    except FileNotFoundError:
        print(f"Error: {filename} not found.")
        return None
```



3. Use finally for Cleanup

Use the finally clause for cleanup code that should always run.

python

```
def read_file(filename):
    file = None
    try:
        file = open(filename, "r")
        return file.read()
    except FileNotFoundError:
        print(f"Error: {filename} not found.")
        return None
    finally:
        if file is not None:
            file.close()
```


4. Don't Suppress Exceptions Unnecessarily

Only catch exceptions you can handle properly.

```
python

# Good - Only catching specific exceptions we know how to handle
try:
    data = json.loads(json_string)
except json.JSONDecodeError:
    data = {} # Fallback to empty dict on invalid JSON

# Avoid - Suppressing all exceptions without handling them properly
try:
    data = json.loads(json_string)
except Exception:
    data = {} # This could hide other unexpected errors
```

5. Re-raise Exceptions When Appropriate

If you can't handle an exception completely, consider re-raising it.

```
python

def process_data(data):
    try:
        result = complex_operation(data)
        return result
    except ValueError as e:
        # Log the error
        logging.error(f"Error processing data: {e}")
        # Re-raise so caller knows something went wrong
        raise
```

6. Use Context Managers (with Statement)

For resource management, use context managers when possible.

python

Good - File will automatically be closed even if an exception occurs

```
with open("file.txt", "r") as file:
```

```
    data = file.read()
```

```
    # Process data
```

Avoid

```
file = open("file.txt", "r")
```

```
try:
```

```
    data = file.read()
```

```
    # Process data
```

```
finally:
```

```
    file.close()
```

Common Pitfalls

1. Catching Exception Base Class

Catching the base `Exception` class can hide bugs and make debugging difficult:

python

Problematic

```
try:
```

```
    # Some code
```

```
except Exception:
```

```
    pass # Silently ignoring all exceptions
```

2. Bare except Clause

Using a bare `except` clause is even worse than catching `Exception`:

python

Very problematic

```
try:
```

```
    # Some code
```

```
except:
```

```
    pass # This will catch absolutely everything, including SystemExit, KeyboardInterrupt, etc
```

3. Passing in except Blocks

Silently ignoring exceptions:

```
python

# Problematic
try:
    result = 10 / 0
except ZeroDivisionError:
    pass # User gets no indication something went wrong
```

4. Not Closing Resources

Forgetting to close files, network connections, etc.:

```
python

# Problematic
try:
    file = open("data.txt", "r")
    data = file.read()
    # No file.close() if an exception occurs while processing data
    process_data(data)
except FileNotFoundError:
    print("File not found")
```

5. Over-Catching Exceptions

Catching exceptions at a level where you cannot properly handle them:

```
python

# Problematic
def complex_function():
    try:
        # Hundreds of lines of code that could raise many different exceptions
        pass
    except Exception as e:
        print(f"An error occurred: {e}") # Not very helpful for debugging
```

Advanced Exception Handling

1. Exception Chaining

You can chain exceptions to preserve the original exception's context:

python

```
def process_data(data):  
    try:  
        return parse_data(data)  
    except ValueError as e:  
        # Chain a new exception with the original as the cause  
        raise RuntimeError("Data processing failed") from e
```

2. Traceback Information

You can access detailed traceback information:

python

```
import traceback  
  
try:  
    # Code that might raise an exception  
    result = 1 / 0  
except Exception as e:  
    # Print the full traceback  
    traceback.print_exc()  
  
    # Or get the traceback as a string  
    error_msg = traceback.format_exc()  
    log_error(error_msg)
```

3. Context Managers (Creating Your Own)

You can create your own context managers for resource management:

python

```
class DatabaseConnection:
    def __init__(self, connection_string):
        self.connection_string = connection_string
        self.connection = None

    def __enter__(self):
        # Set up resource
        self.connection = connect_to_db(self.connection_string)
        return self.connection

    def __exit__(self, exc_type, exc_val, exc_tb):
        # Clean up resource
        if self.connection:
            self.connection.close()
        # Return False to propagate exceptions, True to suppress
        return False

# Usage
with DatabaseConnection("db://localhost") as conn:
    data = conn.query("SELECT * FROM users")
```

4. Assertion-Based Exception Handling

Using assertions for internal checks:

python

```
def calculate_average(numbers):
    # Check precondition
    assert len(numbers) > 0, "Cannot calculate average of empty list"

    return sum(numbers) / len(numbers)
```

Assignments

Assignment 1: File Processor

Objective: Create a program that reads a CSV file containing numeric data, performs calculations, and handles potential exceptions.

Requirements:

1. Write a function `process_csv` that takes a filename as input.
2. The function should read the file and calculate the sum and average of each row of numbers.
3. Handle at least the following exceptions:
 - File not found
 - Invalid data (non-numeric values)
 - Empty file
4. Use appropriate exception handling techniques including try-except blocks, else clause, and finally clause.
5. Create a custom exception called `InvalidCSVFormatError` for when the CSV format is invalid.
6. Include proper cleanup code to ensure files are closed properly.

Template:

```
python

class InvalidCSVFormatError(Exception):
    """Raised when the CSV file has an invalid format"""
    pass

def process_csv(filename):
    # Your code here
    pass

# Test your function with different files
# process_csv("valid_data.csv")
# process_csv("non_existent_file.csv")
# process_csv("invalid_data.csv")
```

Assignment 2: Banking System

Objective: Implement a simple banking system with proper exception handling.

Requirements:

1. Create a `BankAccount` class with methods for deposit, withdraw, and transfer.
2. Implement the following custom exceptions:
 - `InsufficientFundsError`: Raised when trying to withdraw more money than the account balance.
 - `NegativeAmountError`: Raised when trying to deposit or withdraw a negative amount.

- `MaxBalanceExceededError`: Raised when the account balance would exceed a maximum limit.
3. Each method should use try-except blocks to handle potential exceptions.
 4. Include proper validation and error messages.
 5. Create a simple menu-driven interface for users to interact with the banking system.
 6. Demonstrate how exceptions are propagated and caught at different levels.

Template:

python

```
class InsufficientFundsError(Exception):
    """Raised when a withdrawal would result in a negative balance"""
    pass

class NegativeAmountError(Exception):
    """Raised when a negative amount is provided for a transaction"""
    pass

class MaxBalanceExceededError(Exception):
    """Raised when a deposit would exceed the maximum allowed balance"""
    pass

class BankAccount:
    def __init__(self, account_number, holder_name, initial_balance=0, max_balance=100000):
        # Your code here
        pass

    def deposit(self, amount):
        # Your code here
        pass

    def withdraw(self, amount):
        # Your code here
        pass

    def transfer(self, target_account, amount):
        # Your code here
        pass

# Menu-driven interface
def main():
    # Your code here
    pass

if __name__ == "__main__":
    main()
```