# ****W3D4 -- Python Error Handling: Building Robust Programs****

JTC Program: Tech Pathways  
Cohort: S25  
Lesson Plan: Python Error Handling - Building Robust Programs  
Type: Lesson Plan  
Week / Day: W3D4  
Version Date: 05/27/2025

## ****Focus Concepts****

* Understanding what errors are and why they occur in Python programs
* Learning the difference between syntax errors and runtime exceptions
* Implementing try-except blocks to handle errors gracefully
* Using multiple exception types and handling them appropriately
* Creating robust programs that continue running despite encountering errors
* Building user-friendly error messages and recovery mechanisms

## ****Learning Objectives****

By the end of this session, fellows will be able to:

* Identify common types of errors in Python programs
* Implement basic try-except blocks to catch and handle exceptions
* Use multiple except blocks to handle different error types appropriately
* Create programs that provide helpful error messages to users
* Build input validation systems using error handling techniques
* Apply else and finally blocks in appropriate situations
* Write defensive code that anticipates potential problems

## ****Out-of-Scope Objectives****

* Advanced exception handling patterns and decorators
* Creating complex custom exception hierarchies
* Performance implications of exception handling
* Debugging techniques and tools beyond basic error handling
* Exception handling in multi-threaded applications
* Advanced logging frameworks and configuration

## ****Required Competencies****

* Understanding of basic Python syntax and data types (from W1-W2)
* Familiarity with functions, loops, and conditional statements
* Experience with user input and basic file operations
* Comfort with Python data structures (lists, dictionaries)
* Understanding of basic programming concepts like variables and scope

## ****Technical Requirements****

* Python 3.x installed
* Code editor (VS Code, PyCharm, or similar)
* Jupyter Notebook (optional but recommended)
* Access to Python interpreter/terminal
* Sample text files for file handling exercises

## ****Prerequisites****

* Completion of W1: Python Basics
* Completion of W2: Data Structures and Control Flow
* Completion of W3D1-D3: Functions and File Handling
* Understanding of basic programming logic and flow control

## ****Assigned Reading & Pre-Class Learning****

Estimated Time: 25 minutes

Resources:

* [Python Exception Handling (Real Python)](https://realpython.com/python-exceptions/) - Understanding Python's exception system - 15 minutes
* [Common Python Errors and How to Fix Them](https://www.freecodecamp.org/news/python-common-errors-how-to-fix-traceback/) - Practical error examples - 10 minutes

## ****Before-Class Mini Quiz Questions (5 questions)****

1. What happens when your Python program encounters an error and you don't handle it?
   * A) The program continues running but skips the problematic line
   * \*B) The program crashes and stops executing
   * C) Python automatically fixes the error
   * D) The error is ignored and execution continues
2. Which of these is a syntax error that would prevent your program from running?
   * A) Trying to divide by zero
   * \*B) Missing a closing parenthesis in a function call
   * C) Accessing a list index that doesn't exist
   * D) Converting a string that contains letters to an integer
3. What is the purpose of a try-except block?
   * A) To make your code run faster
   * B) To fix errors automatically
   * \*C) To catch errors and handle them gracefully without crashing
   * D) To prevent all errors from occurring
4. Which exception would you catch when trying to convert user input to an integer?
   * A) IndexError
   * B) FileNotFoundError
   * \*C) ValueError
   * D) KeyError
5. What's the benefit of handling errors in your programs?
   * A) It makes your code shorter
   * B) It prevents errors from ever happening
   * \*C) It makes your programs more user-friendly and robust
   * D) It makes your programs run faster

## ****Key Terms****

* **Exception**: An error that occurs during program execution that can be handled
* **Try Block**: Code section where you expect an error might occur
* **Except Block**: Code that runs when a specific exception occurs
* **Finally Block**: Code that always runs, regardless of whether an exception occurred
* **Else Block**: Code that runs only if no exception occurred in the try block
* **Raise**: Keyword used to manually trigger an exception
* **Traceback**: The error message that shows where and why an error occurred
* **Stack Trace**: Detailed information about the function calls leading to an error
* **Error Handling**: The practice of anticipating and managing errors in code
* **Graceful Degradation**: Allowing a program to continue operating when errors occur
* **Input Validation**: Checking user input to ensure it meets expected criteria
* **Defensive Programming**: Writing code that anticipates potential problems
* **Exception Hierarchy**: The organization of different exception types in Python
* **Custom Exception**: User-defined exception classes for specific error scenarios
* **Error Message**: Human-readable description of what went wrong
* **Robust Code**: Code that handles unexpected situations without crashing
* **Fault Tolerance**: A program's ability to continue operating despite errors
* **Exception Propagation**: How exceptions travel up through function calls
* **Catching vs Raising**: The difference between handling exceptions and creating them

## ****Lesson Schedule & Detailed Script****

### ****6:30 PM -- 6:45 PM: Interactive Check-In****

**Instructor Script:** "Welcome to Week 3, Day 4! Over the past few weeks, we've learned Python basics, data structures, and functions. Today we're going to learn about something crucial for building real-world applications: error handling. Every program encounters unexpected situations, and knowing how to handle these gracefully is what separates beginner code from professional-quality software. By the end of today, you'll know how to write programs that don't crash when things go wrong."

**Admin Tasks:**

* Take attendance
* Ensure everyone has Python running and can execute basic code
* Check for any issues with previous assignments

**Prompting Questions:**

* "Can anyone share a time when a program or app you were using crashed or gave you an unhelpful error message?"
* "What's the most frustrating error message you've encountered while programming?"

**Poll Questions:**

* "On a scale of 1-5, how often do you encounter errors while coding?"
* "When your code doesn't work, what's your first instinct: Google the error, re-read your code, or ask for help?"

### ****6:45 PM -- 7:15 PM: Session 1 -- Understanding Errors and Exceptions****

**Objective:** Learn what errors are, why they occur, and the different types of errors in Python.

**Instructor Script:** "Before we can handle errors, we need to understand what they are and why they occur. Python has different types of errors, and knowing the difference will help us write better error-handling code."

#### ****Types of Errors in Python:****

# 1. Syntax Errors - These prevent your program from running at all

print("Hello World" # Missing closing parenthesis - SyntaxError

# 2. Runtime Errors (Exceptions) - These occur while the program is running

print("=== Common Runtime Errors ===")

# Example 1: ZeroDivisionError

try:

result = 10 / 0

except ZeroDivisionError:

print("Cannot divide by zero!")

# Example 2: ValueError

try:

number = int("hello")

except ValueError:

print("Cannot convert 'hello' to an integer!")

# Example 3: IndexError

try:

my\_list = [1, 2, 3]

print(my\_list[10])

except IndexError:

print("List index out of range!")

# Example 4: KeyError

try:

my\_dict = {"name": "Alice", "age": 25}

print(my\_dict["height"])

except KeyError:

print("Key 'height' not found in dictionary!")

# Example 5: FileNotFoundError

try:

with open("nonexistent\_file.txt", "r") as file:

content = file.read()

except FileNotFoundError:

print("File not found!")

# Example 6: TypeError

try:

result = "5" + 5

except TypeError:

print("Cannot add a string and an integer!")

#### ****Understanding Tracebacks:****

# Let's see what happens without error handling

print("=== Understanding Tracebacks ===")

def divide\_numbers(a, b):

return a / b

def calculate\_average(numbers):

total = sum(numbers)

count = len(numbers)

return divide\_numbers(total, count)

def main():

# This will cause an error

empty\_list = []

average = calculate\_average(empty\_list)

print(f"Average: {average}")

# Uncomment the next line to see the traceback

# main()

print("The traceback shows:")

print("1. Where the error occurred (line number)")

print("2. What type of error it was")

print("3. The chain of function calls that led to the error")

print("4. A description of what went wrong")

#### ****The Cost of Unhandled Errors:****

print("=== What Happens Without Error Handling ===")

# Scenario: A simple calculator that crashes on invalid input

def bad\_calculator():

print("Simple Calculator (Version 1.0 - Crashes easily!)")

# Get user input (commented out for demonstration)

# num1 = float(input("Enter first number: "))

# operation = input("Enter operation (+, -, \*, /): ")

# num2 = float(input("Enter second number: "))

# Simulate user input

num1 = 10

operation = "/"

num2 = 0 # This will cause division by zero

if operation == "+":

result = num1 + num2

elif operation == "-":

result = num1 - num2

elif operation == "\*":

result = num1 \* num2

elif operation == "/":

result = num1 / num2 # This line will crash!

print(f"Result: {result}")

print("Thanks for using the calculator!") # This line never executes

print("Problems with the bad calculator:")

print("- Crashes on division by zero")

print("- Crashes on invalid number input")

print("- Provides unhelpful error messages")

print("- Doesn't give users a chance to correct their input")

#### ****Introduction to Exception Handling:****

print("=== Basic Exception Handling Concept ===")

# The basic structure of exception handling

def demonstrate\_try\_except():

print("Demonstrating try-except structure:")

try:

# Code that might cause an error

risky\_operation = 10 / 0

print("This line won't execute")

except ZeroDivisionError:

# Code that runs if the error occurs

print("Caught a division by zero error!")

print("The program continues running...")

print("This line executes normally")

demonstrate\_try\_except()

# Multiple operations with individual error handling

def demonstrate\_multiple\_operations():

print("\nHandling different operations:")

operations = [

lambda: 10 / 2, # Success

lambda: 10 / 0, # ZeroDivisionError

lambda: int("abc"), # ValueError

lambda: [1, 2][5], # IndexError

]

for i, operation in enumerate(operations):

try:

result = operation()

print(f"Operation {i + 1}: Success! Result = {result}")

except ZeroDivisionError:

print(f"Operation {i + 1}: Cannot divide by zero")

except ValueError:

print(f"Operation {i + 1}: Invalid value for conversion")

except IndexError:

print(f"Operation {i + 1}: Index out of range")

print("All operations completed (with error handling)!")

demonstrate\_multiple\_operations()

**Key Learning Points:**

* Syntax errors prevent programs from running; runtime errors occur during execution
* Unhandled exceptions crash programs and provide poor user experience
* Exception handling allows programs to continue running when errors occur
* Different error types require different handling strategies

### ****7:15 PM -- 7:45 PM: Session 2 -- Basic Try-Except Implementation****

**Objective:** Learn to implement basic try-except blocks and handle common exceptions.

**Instructor Script:** "Now let's dive into the practical side of error handling. We'll start with basic try-except blocks and build up to more sophisticated error handling patterns."

#### ****Basic Try-Except Structure:****

print("=== Basic Try-Except Examples ===")

# Example 1: User input validation

def get\_age\_safely():

while True:

try:

age\_input = input("Please enter your age: ")

age = int(age\_input)

if age < 0:

print("Age cannot be negative. Please try again.")

continue

elif age > 150:

print("Age seems unrealistic. Please try again.")

continue

return age

except ValueError:

print("Please enter a valid number.")

print("Try again.")

# Demonstration (commented for file usage)

# user\_age = get\_age\_safely()

# print(f"You are {user\_age} years old.")

# Example 2: Safe division function

def safe\_divide(numerator, denominator):

try:

result = numerator / denominator

return result

except ZeroDivisionError:

print("Error: Cannot divide by zero!")

return None

# Test the safe division

print("Testing safe division:")

print(f"10 / 2 = {safe\_divide(10, 2)}")

print(f"10 / 0 = {safe\_divide(10, 0)}")

print(f"7 / 3 = {safe\_divide(7, 3)}")

#### ****Handling Different Exception Types:****

print("\n=== Handling Multiple Exception Types ===")

def process\_user\_data():

# Simulate different types of user input

test\_inputs = [

("25", 0), # Valid age, division by zero

("abc", 5), # Invalid age, valid divisor

("30", 3), # Valid age, valid divisor

("-5", 2), # Negative age, valid divisor

]

for age\_str, divisor in test\_inputs:

print(f"\nProcessing: age='{age\_str}', divisor={divisor}")

try:

# Convert age to integer

age = int(age\_str)

# Check for valid age

if age < 0:

print("Warning: Negative age detected")

# Perform calculation

result = age / divisor

print(f"Success: {age} / {divisor} = {result:.2f}")

except ValueError:

print("Error: Age must be a valid number")

except ZeroDivisionError:

print("Error: Cannot divide by zero")

except Exception as e:

print(f"Unexpected error: {e}")

process\_user\_data()

#### ****Building a Robust Calculator:****

print("\n=== Building a Robust Calculator ===")

def robust\_calculator():

print("Robust Calculator (Version 2.0 - Now with error handling!)")

def get\_number(prompt):

while True:

try:

return float(input(prompt))

except ValueError:

print("Please enter a valid number.")

def get\_operation():

while True:

operation = input("Enter operation (+, -, \*, /): ").strip()

if operation in ['+', '-', '\*', '/']:

return operation

else:

print("Please enter a valid operation: +, -, \*, or /")

# For demonstration, we'll simulate user input

def simulate\_calculation(num1, operation, num2):

print(f"Calculating: {num1} {operation} {num2}")

try:

if operation == "+":

result = num1 + num2

elif operation == "-":

result = num1 - num2

elif operation == "\*":

result = num1 \* num2

elif operation == "/":

if num2 == 0:

raise ZeroDivisionError("Cannot divide by zero")

result = num1 / num2

print(f"Result: {result}")

return result

except ZeroDivisionError:

print("Error: Division by zero is not allowed.")

return None

except Exception as e:

print(f"Unexpected error occurred: {e}")

return None

# Test different scenarios

test\_cases = [

(10, "+", 5),

(10, "/", 0), # Division by zero

(7, "\*", 3),

(15, "-", 8)

]

for num1, op, num2 in test\_cases:

simulate\_calculation(num1, op, num2)

print("-" \* 30)

robust\_calculator()

#### ****File Handling with Error Management:****

print("\n=== Safe File Operations ===")

def read\_file\_safely(filename):

try:

with open(filename, 'r') as file:

content = file.read()

print(f"Successfully read {len(content)} characters from {filename}")

return content

except FileNotFoundError:

print(f"Error: The file '{filename}' was not found.")

return None

except PermissionError:

print(f"Error: Permission denied to read '{filename}'.")

return None

except Exception as e:

print(f"Unexpected error reading file: {e}")

return None

def write\_file\_safely(filename, content):

try:

with open(filename, 'w') as file:

file.write(content)

print(f"Successfully wrote content to {filename}")

return True

except PermissionError:

print(f"Error: Permission denied to write to '{filename}'.")

return False

except Exception as e:

print(f"Unexpected error writing file: {e}")

return False

# Test file operations

print("Testing file operations:")

content = read\_file\_safely("nonexistent.txt") # This will fail gracefully

success = write\_file\_safely("test\_output.txt", "Hello, World!") # This might succeed

**Key Learning Points:**

* Try-except blocks prevent programs from crashing
* Specific exception types should be caught with appropriate handling
* User input validation is a common use case for error handling
* File operations are prone to errors and should always be wrapped in try-except

### ****7:45 PM -- 8:10 PM: Capstone Work Session****

**Activity:** Work on capstone project, applying error handling concepts to existing code.

### ****8:10 PM -- 8:20 PM: Break****

10-minute break

### ****8:20 PM -- 8:50 PM: Session 3 -- Advanced Error Handling Patterns****

**Objective:** Learn about else and finally blocks, multiple exception handling, and creating user-friendly error messages.

**Instructor Script:** "Now that we understand basic try-except blocks, let's explore more sophisticated error handling patterns. We'll learn about else and finally blocks, and how to create error handling that provides excellent user experience."

#### ****Using Else and Finally Blocks:****

print("=== Else and Finally Blocks ===")

def demonstrate\_complete\_structure():

print("Complete try-except-else-finally structure:")

def process\_file(filename, operation="read"):

file\_handle = None

try:

print(f"Attempting to {operation} file: {filename}")

if operation == "read":

file\_handle = open(filename, 'r')

content = file\_handle.read()

return content

elif operation == "write":

file\_handle = open(filename, 'w')

file\_handle.write("Sample content")

return "Write successful"

else:

raise ValueError("Invalid operation")

except FileNotFoundError:

print(f"Error: File '{filename}' not found")

return None

except PermissionError:

print(f"Error: Permission denied for '{filename}'")

return None

except ValueError as e:

print(f"Error: {e}")

return None

else:

print("File operation completed successfully!")

return "Success"

finally:

if file\_handle:

file\_handle.close()

print("File handle closed")

print("Cleanup completed")

# Test different scenarios

scenarios = [

("existing\_file.txt", "read"), # Will likely fail

("test.txt", "write"), # Might succeed

("any\_file.txt", "invalid"), # Will cause ValueError

]

for filename, operation in scenarios:

print(f"\n--- Testing {operation} on {filename} ---")

result = process\_file(filename, operation)

print(f"Result: {result}")

print("-" \* 40)

demonstrate\_complete\_structure()

#### ****Handling Multiple Exceptions Elegantly:****

print("\n=== Multiple Exception Handling Strategies ===")

def data\_processor(data\_list):

"""Process a list of data with comprehensive error handling"""

results = []

errors = []

for i, item in enumerate(data\_list):

try:

# Try to convert to number and perform calculation

number = float(item)

# Perform some calculation that might fail

if number == 0:

raise ZeroDivisionError("Cannot process zero values")

result = 100 / number

results.append({"index": i, "input": item, "result": result})

except ValueError:

error\_msg = f"Item {i}: '{item}' is not a valid number"

errors.append(error\_msg)

print(f"Warning: {error\_msg}")

except ZeroDivisionError as e:

error\_msg = f"Item {i}: {e}"

errors.append(error\_msg)

print(f"Warning: {error\_msg}")

except Exception as e:

error\_msg = f"Item {i}: Unexpected error - {e}"

errors.append(error\_msg)

print(f"Error: {error\_msg}")

return results, errors

# Test with various data types

test\_data = [

"10", # Valid number

"0", # Will cause division by zero

"abc", # Invalid number

"25.5", # Valid float

"", # Empty string

"2" # Valid number

]

print("Processing test data:")

results, errors = data\_processor(test\_data)

print(f"\nProcessing complete!")

print(f"Successful operations: {len(results)}")

print(f"Errors encountered: {len(errors)}")

print("\nSuccessful results:")

for result in results:

print(f" {result['input']} -> {result['result']:.2f}")

#### ****Creating User-Friendly Error Messages:****

print("\n=== User-Friendly Error Messages ===")

class UserFriendlyProcessor:

def \_\_init\_\_(self):

self.error\_messages = {

ValueError: "The value you entered is not in the correct format.",

ZeroDivisionError: "Mathematical error: cannot divide by zero.",

FileNotFoundError: "The file you're looking for doesn't exist.",

PermissionError: "You don't have permission to access this resource.",

KeyError: "The information you requested is not available.",

IndexError: "You're trying to access data that doesn't exist.",

TypeError: "The data types you're using are incompatible."

}

def get\_friendly\_message(self, exception):

"""Convert technical exceptions to user-friendly messages"""

exception\_type = type(exception)

if exception\_type in self.error\_messages:

return self.error\_messages[exception\_type]

else:

return "An unexpected error occurred. Please try again."

def safe\_operation(self, operation\_name, operation\_func, \*args, \*\*kwargs):

"""Execute an operation with user-friendly error handling"""

try:

print(f"Attempting {operation\_name}...")

result = operation\_func(\*args, \*\*kwargs)

print(f"✓ {operation\_name} completed successfully!")

return result, None

except Exception as e:

friendly\_message = self.get\_friendly\_message(e)

print(f"✗ {operation\_name} failed: {friendly\_message}")

# Optional: log technical details for developers

print(f" Technical details: {type(e).\_\_name\_\_}: {e}")

return None, friendly\_message

# Demonstration

processor = UserFriendlyProcessor()

# Test various operations

def divide\_numbers(a, b):

return a / b

def access\_list\_item(lst, index):

return lst[index]

def convert\_to\_int(value):

return int(value)

# Run tests

operations = [

("Division", divide\_numbers, 10, 2),

("Division by zero", divide\_numbers, 10, 0),

("List access", access\_list\_item, [1, 2, 3], 1),

("Invalid list access", access\_list\_item, [1, 2, 3], 10),

("String conversion", convert\_to\_int, "123"),

("Invalid conversion", convert\_to\_int, "abc"),

]

for name, func, \*args in operations:

result, error = processor.safe\_operation(name, func, \*args)

print("-" \* 30)

#### ****Input Validation Patterns:****

print("\n=== Input Validation Patterns ===")

class InputValidator:

@staticmethod

def get\_integer\_in\_range(prompt, min\_val=None, max\_val=None, max\_attempts=3):

"""Get integer input with range validation and attempt limiting"""

attempts = 0

while attempts < max\_attempts:

try:

user\_input = input(f"{prompt} (Attempt {attempts + 1}/{max\_attempts}): ")

value = int(user\_input)

# Range validation

if min\_val is not None and value < min\_val:

print(f"Value must be at least {min\_val}")

attempts += 1

continue

if max\_val is not None and value > max\_val:

print(f"Value must be no more than {max\_val}")

attempts += 1

continue

return value

except ValueError:

print("Please enter a valid integer")

attempts += 1

print("Maximum attempts reached. Using default value.")

return None

@staticmethod

def validate\_email(email):

"""Basic email validation with specific error messages"""

try:

if not email or not isinstance(email, str):

raise ValueError("Email cannot be empty")

email = email.strip()

if '@' not in email:

raise ValueError("Email must contain @ symbol")

if email.count('@') != 1:

raise ValueError("Email must contain exactly one @ symbol")

username, domain = email.split('@')

if not username:

raise ValueError("Email must have a username before @")

if not domain:

raise ValueError("Email must have a domain after @")

if '.' not in domain:

raise ValueError("Email domain must contain at least one dot")

return True, "Valid email address"

except ValueError as e:

return False, str(e)

# Test input validation

print("Testing input validation:")

# Test email validation

test\_emails = ["user@example.com", "invalid.email", "@domain.com", "user@", ""]

for email in test\_emails:

is\_valid, message = InputValidator.validate\_email(email)

status = "✓" if is\_valid else "✗"

print(f"{status} '{email}': {message}")

**Key Learning Points:**

* Else blocks run only when no exceptions occur
* Finally blocks always run, making them perfect for cleanup
* User-friendly error messages improve the user experience significantly
* Input validation should provide clear guidance on what went wrong and how to fix it

### ****8:50 PM -- 9:15 PM: Session 4 -- Building Robust Applications****

**Objective:** Apply error handling concepts to build a complete, robust application.

**Instructor Script:** "Let's put everything together and build a real application that demonstrates professional-level error handling. We'll create a contact management system that gracefully handles all the errors we might encounter."

#### ****Complete Application: Contact Manager****

print("=== Building a Robust Contact Manager ===")

import json

import os

from datetime import datetime

class ContactManager:

def \_\_init\_\_(self, filename="contacts.json"):

self.filename = filename

self.contacts = {}

self.load\_contacts()

def load\_contacts(self):

"""Load contacts from file with error handling"""

try:

if os.path.exists(self.filename):

with open(self.filename, 'r') as file:

self.contacts = json.load(file)

print(f"✓ Loaded {len(self.contacts)} contacts from {self.filename}")

else:

print(f"ℹ No existing contact file found. Starting with empty contacts.")

self.contacts = {}

except json.JSONDecodeError:

print(f"⚠ Contact file is corrupted. Starting with empty contacts.")

self.backup\_corrupted\_file()

self.contacts = {}

except PermissionError:

print(f"✗ Permission denied to read {self.filename}")

self.contacts = {}

except Exception as e:

print(f"✗ Unexpected error loading contacts: {e}")

self.contacts = {}

def backup\_corrupted\_file(self):

"""Create backup of corrupted file"""

try:

backup\_name = f"{self.filename}.backup\_{datetime.now().strftime('%Y%m%d\_%H%M%S')}"

os.rename(self.filename, backup\_name)

print(f"ℹ Corrupted file backed up as {backup\_name}")

except Exception as e:

print(f"⚠ Could not backup corrupted file: {e}")

def save\_contacts(self):

"""Save contacts to file with error handling"""

try:

with open(self.filename, 'w') as file:

json.dump(self.contacts, file, indent=2)

print(f"✓ Contacts saved to {self.filename}")

return True

except PermissionError:

print(f"✗ Permission denied to write to {self.filename}")

return False

except Exception as e:

print(f"✗ Error saving contacts: {e}")

return False

def validate\_contact\_data(self, name, email, phone):

"""Validate contact information"""

errors = []

# Name validation

if not name or not name.strip():

errors.append("Name cannot be empty")

elif len(name.strip()) < 2:

errors.append("Name must be at least 2 characters long")

# Email validation

if email:

email = email.strip()

if '@' not in email or email.count('@') != 1:

errors.append("Email must be a valid email address")

elif not email.split('@')[1] or '.' not in email.split('@')[1]:

errors.append("Email domain must be valid")

# Phone validation

if phone:

phone\_digits = ''.join(filter(str.isdigit, phone))

if len(phone\_digits) < 10:

errors.append("Phone number must contain at least 10 digits")

return errors

def add\_contact(self, name, email="", phone=""):

"""Add a new contact with validation"""

try:

# Validate input

errors = self.validate\_contact\_data(name, email, phone)

if errors:

print("✗ Cannot add contact due to validation errors:")

for error in errors:

print(f" - {error}")

return False

# Check if contact already exists

name\_clean = name.strip().lower()

if name\_clean in [existing.lower() for existing in self.contacts.keys()]:

print(f"⚠ Contact '{name}' already exists. Use update\_contact() to modify.")

return False

# Add the contact

self.contacts[name.strip()] = {

"email": email.strip() if email else "",

"phone": phone.strip() if phone else "",

"created": datetime.now().isoformat(),

"updated": datetime.now().isoformat()

}

print(f"✓ Added contact: {name}")

return True

except Exception as e:

print(f"✗ Unexpected error adding contact: {e}")

return False

def get\_contact(self, name):

"""Retrieve a contact with error handling"""

try:

if not name or not name.strip():

print("✗ Contact name cannot be empty")

return None

# Case-insensitive search

name\_clean = name.strip().lower()

for contact\_name, contact\_info in self.contacts.items():

if contact\_name.lower() == name\_clean:

return {

"name": contact\_name,

\*\*contact\_info

}

print(f"ℹ Contact '{name}' not found")

return None

except Exception as e:

print(f"✗ Error retrieving contact: {e}")

return None

def update\_contact(self, name, email=None, phone=None):

"""Update existing contact with validation"""

try:

contact = self.get\_contact(name)

if not contact:

return False

# Validate new data

current\_email = email if email is not None else contact.get("email", "")

current\_phone = phone if phone is not None else contact.get("phone", "")

errors = self.validate\_contact\_data(name, current\_email, current\_phone)

if errors:

print("✗ Cannot update contact due to validation errors:")

for error in errors:

print(f" - {error}")

return False

# Update the contact

contact\_name = contact["name"]

if email is not None:

self.contacts[contact\_name]["email"] = email.strip()

if phone is not None:

self.contacts[contact\_name]["phone"] = phone.strip()

self.contacts[contact\_name]["updated"] = datetime.now().isoformat()

print(f"✓ Updated contact: {contact\_name}")

return True

except Exception as e:

print(f"✗ Error updating contact: {e}")

return False

def delete\_contact(self, name):

"""Delete a contact with confirmation"""

try:

contact = self.get\_contact(name)

if not contact:

return False

contact\_name = contact["name"]

del self.contacts[contact\_name]

print(f"✓ Deleted contact: {contact\_name}")

return True

except Exception as e:

print(f"✗ Error deleting contact: {e}")

return False

def list\_contacts(self):

"""List all contacts with error handling"""

try:

if not self.contacts:

print("ℹ No contacts found")

return

print(f"\n=== Contact List ({len(self.contacts)} contacts) ===")

for name, info in sorted(self.contacts.items()):

email = info.get("email", "No email")

phone = info.get("phone", "No phone")

print(f"📇 {name}")

print(f" Email: {email}")

print(f" Phone: {phone}")

print(f" Updated: {info.get('updated', 'Unknown')[:10]}")

print()

except Exception as e:

print(f"✗ Error listing contacts: {e}")

# Demonstration of the Contact Manager

def demonstrate\_contact\_manager():

print("=== Contact Manager Demonstration ===")

# Create a contact manager instance

cm = ContactManager("demo\_contacts.json")

# Test adding valid contacts

print("\n--- Adding Valid Contacts ---")

cm.add\_contact("Alice Johnson", "alice@email.com", "555-0123")

cm.add\_contact("Bob Smith", "bob@email.com", "555-0456")

# Test adding invalid contacts

print("\n--- Testing Invalid Contacts ---")

cm.add\_contact("", "invalid@email", "123") # Empty name, invalid email, short phone

cm.add\_contact("X", "not-an-email", "") # Short name, invalid email

# Test duplicate contact

print("\n--- Testing Duplicate Contact ---")

cm.add\_contact("Alice Johnson", "different@email.com", "555-9999")

# Test retrieving contacts

print("\n--- Retrieving Contacts ---")

alice = cm.get\_contact("Alice Johnson")

if alice:

print(f"Found: {alice['name']} - {alice['email']}")

missing = cm.get\_contact("Charlie Brown")

# Test updating contact

print("\n--- Updating Contact ---")

cm.update\_contact("Bob Smith", email="robert@newdomain.com", phone="555-7890")

# List all contacts

print("\n--- Listing All Contacts ---")

cm.list\_contacts()

# Save contacts

print("\n--- Saving Contacts ---")

cm.save\_contacts()

return cm

# Run the demonstration

contact\_manager = demonstrate\_contact\_manager()

#### ****Advanced Error Handling Patterns:****

print("\n=== Advanced Error Handling Patterns ===")

# Pattern 1: Retry mechanism with exponential backoff

import time

import random

def retry\_operation(operation, max\_attempts=3, base\_delay=1):

"""Retry an operation with exponential backoff"""

for attempt in range(max\_attempts):

try:

result = operation()

print(f"✓ Operation succeeded on attempt {attempt + 1}")

return result

except Exception as e:

if attempt == max\_attempts - 1:

print(f"✗ Operation failed after {max\_attempts} attempts: {e}")

raise

delay = base\_delay \* (2 \*\* attempt) + random.uniform(0, 1)

print(f"⚠ Attempt {attempt + 1} failed: {e}")

print(f"ℹ Retrying in {delay:.2f} seconds...")

time.sleep(delay)

# Example: Simulated network operation

def unreliable\_network\_call():

if random.random() < 0.7: # 70% chance of failure

raise ConnectionError("Network timeout")

return "Data retrieved successfully"

print("Testing retry mechanism:")

try:

result = retry\_operation(unreliable\_network\_call, max\_attempts=3)

print(f"Final result: {result}")

except Exception as e:

print(f"Operation ultimately failed: {e}")

# Pattern 2: Context manager for resource handling

class SafeFileHandler:

def \_\_init\_\_(self, filename, mode='r'):

self.filename = filename

self.mode = mode

self.file = None

def \_\_enter\_\_(self):

try:

self.file = open(self.filename, self.mode)

print(f"✓ Opened file: {self.filename}")

return self.file

except Exception as e:

print(f"✗ Failed to open file {self.filename}: {e}")

raise

def \_\_exit\_\_(self, exc\_type, exc\_val, exc\_tb):

if self.file:

self.file.close()

print(f"✓ Closed file: {self.filename}")

if exc\_type:

print(f"⚠ Exception occurred: {exc\_type.\_\_name\_\_}: {exc\_val}")

return False # Don't suppress exceptions

# Example usage of context manager

print("\nTesting context manager:")

try:

with SafeFileHandler("demo\_file.txt", "w") as f:

f.write("This is a test file")

print("✓ File written successfully")

except Exception as e:

print(f"File operation failed: {e}")

# Pattern 3: Logging errors for debugging

import logging

# Configure logging

logging.basicConfig(

level=logging.INFO,

format='%(asctime)s - %(levelname)s - %(message)s',

handlers=[

logging.FileHandler('error\_log.txt'),

logging.StreamHandler()

]

)

logger = logging.getLogger(\_\_name\_\_)

def logged\_operation(operation\_name, operation\_func, \*args, \*\*kwargs):

"""Execute operation with comprehensive logging"""

logger.info(f"Starting operation: {operation\_name}")

try:

result = operation\_func(\*args, \*\*kwargs)

logger.info(f"Operation '{operation\_name}' completed successfully")

return result

except Exception as e:

logger.error(f"Operation '{operation\_name}' failed: {type(e).\_\_name\_\_}: {e}")

logger.debug(f"Operation details - Args: {args}, Kwargs: {kwargs}")

raise

# Example of logged operations

def risky\_calculation(x, y):

return x / y

print("\nTesting logged operations:")

try:

result1 = logged\_operation("Safe division", risky\_calculation, 10, 2)

print(f"Result: {result1}")

result2 = logged\_operation("Risky division", risky\_calculation, 10, 0)

except Exception as e:

print(f"Logged operation failed: {e}")

**Key Learning Points:**

* Real applications require comprehensive error handling at every level
* User experience is greatly improved with thoughtful error messages
* Logging helps with debugging and monitoring application health
* Advanced patterns like retry mechanisms and context managers provide robust solutions

### ****9:15 PM -- 9:25 PM: Session 5 -- Best Practices and Wrap-Up****

**Objective:** Consolidate learning and discuss error handling best practices.

**Instructor Script:** "Let's wrap up by discussing best practices for error handling and reviewing what we've learned today. These practices will help you write more professional, maintainable code."

#### ****Error Handling Best Practices:****

print("=== Error Handling Best Practices ===")

# Best Practice 1: Be specific with exception types

print("1. Be Specific with Exception Types")

# Good: Specific exception handling

def good\_input\_validation(value):

try:

number = int(value)

return number

except ValueError:

print("Error: Please enter a valid integer")

return None

# Avoid: Catching all exceptions

def avoid\_broad\_exceptions(value):

try:

number = int(value)

return number

except Exception: # Too broad - might hide other issues

print("Something went wrong")

return None

# Best Practice 2: Provide helpful error messages

print("\n2. Provide Helpful Error Messages")

def helpful\_error\_messages(age\_str):

try:

age = int(age\_str)

if age < 0:

raise ValueError("Age cannot be negative")

if age > 150:

raise ValueError("Please enter a realistic age")

return age

except ValueError as e:

if "invalid literal" in str(e):

print(f"Error: '{age\_str}' is not a valid number. Please enter digits only.")

else:

print(f"Error: {e}")

return None

# Best Practice 3: Use finally for cleanup

print("\n3. Use Finally for Cleanup")

def proper\_resource\_handling(filename):

file\_handle = None

try:

file\_handle = open(filename, 'r')

content = file\_handle.read()

return content

except FileNotFoundError:

print(f"File {filename} not found")

return None

finally:

if file\_handle:

file\_handle.close()

print("File properly closed")

# Best Practice 4: Don't ignore exceptions

print("\n4. Don't Ignore Exceptions")

# Good: Handle or log exceptions

def good\_exception\_handling():

try:

result = risky\_operation()

return result

except SpecificError as e:

logger.error(f"Known issue occurred: {e}")

return default\_value

except Exception as e:

logger.critical(f"Unexpected error: {e}")

raise # Re-raise if we can't handle it

# Avoid: Silent failures

def avoid\_silent\_failures():

try:

result = risky\_operation()

return result

except:

pass # Never do this - hides problems!

def risky\_operation():

return "success"

# Best Practice 5: Validate early and often

print("\n5. Validate Early and Often")

def validate\_early\_example(user\_data):

# Validate at the start

if not isinstance(user\_data, dict):

raise TypeError("User data must be a dictionary")

required\_fields = ['name', 'email', 'age']

for field in required\_fields:

if field not in user\_data:

raise ValueError(f"Missing required field: {field}")

# Continue with processing only if validation passes

return process\_valid\_data(user\_data)

def process\_valid\_data(data):

return f"Processing {data['name']}"

# Test validation

try:

result = validate\_early\_example({'name': 'John', 'email': 'john@email.com', 'age': 30})

print(f"✓ {result}")

except (TypeError, ValueError) as e:

print(f"✗ Validation failed: {e}")

#### ****Common Pitfalls to Avoid:****

print("\n=== Common Pitfalls to Avoid ===")

print("1. Don't catch exceptions you can't handle meaningfully")

print("2. Don't use bare 'except:' clauses")

print("3. Don't ignore exceptions silently")

print("4. Don't make error messages too technical for users")

print("5. Don't forget to clean up resources")

print("6. Don't assume operations will always succeed")

print("7. Don't over-engineer error handling for simple scripts")

# Example of what NOT to do

def bad\_error\_handling\_example():

try:

# Multiple operations that could fail differently

file\_content = open("file.txt").read()

number = int(file\_content)

result = 100 / number

return result

except: # Too broad

return "error" # Unhelpful message

# Better approach

def good\_error\_handling\_example():

try:

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

file\_content = f.read().strip()

try:

number = int(file\_content)

except ValueError:

print("Error: File content is not a valid number")

return None

if number == 0:

print("Error: Cannot divide by zero (file contains 0)")

return None

result = 100 / number

return result

except FileNotFoundError:

print("Error: Could not find the input file 'file.txt'")

return None

except PermissionError:

print("Error: Permission denied to read 'file.txt'")

return None

#### ****Summary and Key Takeaways:****

print("\n=== Summary: What We've Learned Today ===")

summary\_points = [

"1. Errors are inevitable - plan for them from the start",

"2. Use specific exception types rather than broad catches",

"3. Provide helpful, user-friendly error messages",

"4. Always clean up resources using finally or context managers",

"5. Validate input early and often",

"6. Log errors for debugging while showing friendly messages to users",

"7. Don't ignore exceptions - handle them or let them propagate",

"8. Test your error handling as thoroughly as your main functionality",

"9. Consider the user experience when designing error responses",

"10. Use error handling to make your programs more robust, not just to avoid crashes"

]

for point in summary\_points:

print(f"✓ {point}")

print("\n=== Your Error Handling Toolkit ===")

toolkit = {

"Basic Pattern": "try-except blocks for common errors",

"Specific Handling": "Multiple except blocks for different error types",

"User Experience": "Friendly error messages with guidance",

"Resource Management": "finally blocks and context managers",

"Input Validation": "Check data before processing",

"Logging": "Record errors for debugging",

"Retry Patterns": "Handle temporary failures gracefully",

"Testing": "Test error conditions as thoroughly as success conditions"

}

for tool, description in toolkit.items():

print(f"🔧 {tool}: {description}")

print("\nRemember: Good error handling is what separates professional code from beginner scripts!")

### ****9:25 PM -- 9:30 PM: Wrap-Up & Final Questions****

**Instructor Script:** "Today we've covered the fundamentals of error handling in Python. We've learned how to anticipate problems, handle them gracefully, and create programs that provide excellent user experiences even when things go wrong. Error handling is a skill that will serve you well in every programming project, from simple scripts to complex applications."

**Review Key Points:**

* Errors are normal and should be planned for, not feared
* Different types of errors require different handling strategies
* User experience is greatly improved by thoughtful error handling
* Professional code includes comprehensive error handling from the design phase
* Testing error conditions is just as important as testing success scenarios

**Prompting Question:** "What's one error handling technique you're excited to apply in your own projects?"

## ****After-Class Quiz (5 questions)****

1. What's the main advantage of using try-except blocks in your programs?
   * A) They make your code run faster
   * B) They prevent all errors from occurring
   * \*C) They allow your program to continue running when errors occur
   * D) They automatically fix errors in your code
2. When should you use a finally block?
   * A) Only when you expect an error to occur
   * B) Only when you want to prevent errors
   * \*C) When you need code to run regardless of whether an error occurred
   * D) Only when handling file operations
3. What's wrong with using a bare except: clause?
   * A) It's too slow
   * B) It doesn't work in Python 3
   * \*C) It catches all exceptions, potentially hiding important errors
   * D) It can only catch one type of error
4. Which approach is better for user experience?
   * A) Showing technical error messages like "ValueError: invalid literal for int()"
   * \*B) Showing friendly messages like "Please enter a valid number"
   * C) Not showing any error messages
   * D) Crashing the program when errors occur
5. What should you do when you catch an exception that you can't handle meaningfully?
   * A) Ignore it completely
   * B) Print a generic "error occurred" message
   * \*C) Log the error and re-raise it or return an appropriate default
   * D) Always return None

## ****Homework Assignment****

Create a **Personal Budget Tracker** program that demonstrates comprehensive error handling:

### ****Requirements:****

1. **File Operations**: Save/load budget data with error handling for missing files, permission errors, and corrupted data
2. **Input Validation**: Handle invalid user input for amounts, categories, and dates
3. **Data Validation**: Ensure budget entries are logical (positive amounts, valid categories, reasonable dates)
4. **User Experience**: Provide helpful error messages and recovery options
5. **Logging**: Log errors to a file for debugging while showing friendly messages to users

### ****Core Features:****

* Add income and expense entries
* View budget summary
* Save and load data from a JSON file
* Handle all potential errors gracefully

### ****Bonus Challenges:****

* Implement a retry mechanism for file operations
* Create custom exception classes for budget-specific errors
* Add data backup and recovery features
* Implement input validation with multiple attempts

**Submission:** Submit your Python file along with a brief report describing the error handling strategies you implemented and why you chose them.

## ****Additional Resources****

* [Python Exception Handling - Real Python](https://realpython.com/python-exceptions/)
* [Python Error Handling Best Practices](https://docs.python.org/3/tutorial/errors.html)
* [Logging in Python](https://realpython.com/python-logging/)
* [Writing Robust Python Code](https://peps.python.org/pep-0008/)

This lesson plan is designed to take students from basic error understanding to professional-level error handling practices, with hands-on examples and real-world applications throughout.