Exception Handling in Practice & the Robust Registration System

AiCE Software Engineering Foundations — Python Programming (Intermediate)

This closing lesson on exception handling applies everything from this unit to two of the most common real-world situations — file operations and validating user input — then brings it all together in a complete, production-quality registration system that never crashes on bad input.

1. Exception Handling With File Operations

File operations fail constantly in the real world — files get moved, deleted, or become unreadable due to permissions. Combining the with statement from the previous unit with try-except produces genuinely robust file handling:

```
def load_student_data(filename):
    """Load student data from a file, handling common failure
cases."""
    try:
        with open(filename, "r") as file:
            lines = file.readlines()
    except FileNotFoundError:
        print(f"Error: '{filename}' does not exist. Creating a
new file.")
        with open(filename, "w") as file:
            pass    # Create an empty file
        return []
    except PermissionError:
        print(f"Error: No permission to read '{filename}'.")
        return []
    else:
        students = [line.strip() for line in lines if
line.strip()]
        print(f"Loaded {len(students)} student(s) from
'{filename}'.")
        return students
```

```
students = load_student_data("students.txt")
more_students = load_student_data("missing_file.txt")
```

The function handles a missing file by creating a new empty one and returning an empty list — rather than crashing, the program adapts and continues. The else clause processes the loaded lines only when the file was read successfully.

2. Input Validation With Exception Handling

A powerful pattern combines exception handling with a while loop — keep asking the user for input until they provide something valid, catching invalid entries along the way instead of crashing:

```
def get_valid_age():
    """Keep asking until a valid age is entered."""
    while True:
        try:
            age = int(input("Enter your age: "))
            if age < 0 or age > 120:
                raise ValueError("Age must be between 0 and
120.")
            return age
        except ValueError as e:
            if str(e) == "invalid literal for int() with base
10: ''" or "invalid literal" in str(e):
                print("Please enter a valid whole number.")
            else:
                print(f"Invalid age: {e}")

age = get_valid_age()
print(f"Age recorded: {age}")
```

This function never returns until it gets a genuinely valid age — typing 'abc', then -5, then 200, then finally 25 would reject the first three and accept the fourth. Notice that both a conversion failure (ValueError from int()) and a business rule violation (a raised

ValueError for out-of-range ages) are caught by the same except clause, since both are ValueError.

> **The Loop-Try-Except Pattern**
> *This loop-try-except combination is one of the most useful patterns in this entire unit. It appears constantly in real applications: login forms, checkout flows, configuration wizards — anywhere a program needs to guarantee it eventually receives valid data before proceeding.*

3. Common Python Exceptions — Reference

Exception	Common Cause
ValueError	Right type, wrong value — int('abc'), or a value outside an expected range
TypeError	Operation on incompatible types — 'text' + 5
ZeroDivisionError	Dividing by zero
FileNotFoundError	Opening a file that doesn't exist in read mode
PermissionError	Insufficient permission to read or write a file
IndexError	Accessing a list index that doesn't exist
KeyError	Accessing a dictionary key that doesn't exist
AttributeError	Calling a method that doesn't exist on an object
NameError	Using a variable or function name that hasn't been defined

4. Bringing It All Together — The AiCE Robust Registration System

A capstone that never crashes, no matter what the user enters — combining every concept from this unit:

```
import os

REGISTRY_FILE = "robust_registry.txt"
```

```
def get_valid_name():
    while True:
        name = input("Full name: ").strip()
        if name:
            return name.title()
        print("Name cannot be empty. Please try again.")

def get_valid_age():
    while True:
        try:
            age = int(input("Age: "))
            if 16 <= age <= 100:
                return age
            print("Age must be between 16 and 100.")
        except ValueError:
            print("Please enter a whole number.")

def get_valid_score():
    while True:
        try:
            score = float(input("Previous exam score (0-100):
"))
            if 0 <= score <= 100:
                return score
            print("Score must be between 0 and 100.")
        except ValueError:
            print("Please enter a valid number.")

def save_registration(name, age, score):
    try:
        with open(REGISTRY_FILE, "a") as file:
            file.write(f"{name},{age},{score}\n")
    except PermissionError:
        print("Error: Could not save — permission denied.")
        return False
    else:
        return True
    finally:
```

```
        print("Save operation complete.")

def register_student():
    print("=== AiCE Robust Registration ===")
    name = get_valid_name()
    age = get_valid_age()
    score = get_valid_score()

    saved = save_registration(name, age, score)
    if saved:
        print(f"\n{name} registered successfully!")
        print(f"Age: {age}, Score: {score}")
    else:
        print("\nRegistration could not be saved. Please
contact support.")

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

Try to break this program — enter letters for age, negative numbers, empty names, scores above 100. It handles every case gracefully, never crashing, always guiding the user toward valid input. This is the standard every professional program should meet before it reaches real users.

Key Takeaways

Combining try-except with file operations lets a program adapt to missing files or permission issues instead of crashing, and the loop-try-except pattern guarantees a program eventually receives valid input rather than giving up at the first mistake. The Robust Registration System demonstrates the standard every professional program should meet: no matter what a user types, the program responds calmly and keeps working. With exception handling complete, the next unit introduces list comprehensions — Python's most expressive syntax for building and transforming lists in a single, readable line.