else, finally & Raising Exceptions

AiCE Software Engineering Foundations — Python Programming (Intermediate)

With try and except established, this lesson completes the full exception-handling structure with else and finally, then flips the perspective entirely: instead of only catching exceptions Python raises automatically, you'll learn to raise your own — deliberately enforcing rules in the code you write.

1. The else Clause

The else clause in a try-except block runs only if no exception was raised in the try block. It's the place for code that should execute only on success — code that itself shouldn't be inside try, because you don't want its own exceptions caught by the same except:

```
def get_student_score():
    try:
        score = float(input("Enter exam score (0-100): "))
    except ValueError:
        print("Invalid input. Score must be a number.")
        return None
    else:
        # Only runs if no ValueError was raised
        if 0 <= score <= 100:
            print(f"Score accepted: {score}")
            return score
        else:
            print(f"Score {score} is out of range (0-100).")
            return None
```

Test this three ways: with a valid score like 85, an invalid entry like 'eighty', and an out-of-range value like 150. The else block only runs when the input was successfully converted to a float — the range check inside else is safe from the ValueError catch above it.

2. The finally Clause

The finally clause runs no matter what — whether the try block succeeded, whether an exception was caught, or even whether an unhandled exception occurred. It is guaranteed to execute.

The primary use of finally is cleanup — closing files, releasing connections, clearing temporary data — actions that must happen regardless of whether the operation succeeded:

```
def read_student_file(filename):
    file = None
    try:
        file = open(filename, "r")
        content = file.read()
        print(f"File read successfully ({len(content)}
characters).")
        return content
    except FileNotFoundError:
        print(f"Error: File '{filename}' not found.")
        return None
    finally:
        # This ALWAYS runs — close the file if it was opened
        if file:
            file.close()
            print("File connection closed.")
```

Test with an existing file and a missing one — the finally block runs in both cases, success and failure. In practice, you'd use the with statement from the previous unit for file handling, which does this automatically. But finally is essential for other kinds of resources — database connections, network sockets, locks — where no equivalent auto-close mechanism exists.

> **try-except-else-finally Together**
> *All four clauses can be combined in a single try block: try for the risky code, except for catching specific errors, else for success-only code, and finally for guaranteed cleanup. You don't need all four every time — use only what the situation requires.*

3. The Complete try-except-else-finally Structure

```
try:
    # Risky code — might raise an exception
    result = risky_operation()
except SpecificError as e:
    # Runs only if SpecificError was raised
    print(f"Caught error: {e}")
except AnotherError:
    # Runs only if AnotherError was raised
    print("Another error occurred.")
else:
    # Runs only if NO exception was raised
    print(f"Success: {result}")
finally:
    # ALWAYS runs — exception or not
    print("Cleanup complete.")
```

Clause	When It Runs	Typical Use
try	Always — it is the code to protect	Any code that might fail
except	Only if a matching exception was raised in try	Handle the specific error
else	Only if NO exception was raised in try	Success-only follow-up code
finally	Always — regardless of exceptions	Cleanup and resource release

4. Raising Exceptions

You aren't limited to catching exceptions Python raises automatically. You can raise them yourself using the raise keyword — deliberately triggering an exception when your code detects an invalid condition.

This is how you enforce rules in your code. If a function receives an argument that violates its requirements, raise an exception rather than silently proceeding with bad data:

```
def enroll_student(name, age, score):
    """Enroll a student after validating all inputs."""
    if not name or not name.strip():
        raise ValueError("Student name cannot be empty.")
    if not isinstance(age, int) or age < 16 or age > 60:
        raise ValueError(f"Age {age} is invalid. Must be
between 16 and 60.")
    if not (0 <= score <= 100):
        raise ValueError(f"Score {score} is out of range. Must
be 0-100.")
    return {
        "name": name.strip().title(), "age": age,
        "score": score, "status": "Enrolled"
    }

# Test with invalid age
try:
    student = enroll_student("Kofi Mensah", 12, 75)
except ValueError as e:
    print(f"Enrollment failed: {e}")
```

This produces:

```
Enrollment failed: Age 12 is invalid. Must be between 16 and
60.
```

The function raises ValueError with a clear, specific message for each violation. The calling code catches the exception and handles it — logging the error, displaying a message, or prompting the user to try again. This raise-and-catch pattern is how professional Python code enforces data integrity.

> **Crashes Are a Choice**
> *Errors are inevitable. Crashes are a choice. Every unhandled exception in a production system is a decision not to plan for reality. Handle what you can predict. Let the unexpected ones surface — and fix them when they do.*

Key Takeaways

else runs only when the try block succeeds, keeping success-only logic separate from the risky code being protected; finally always runs, exception or not, making it the right place for guaranteed cleanup. raise lets you trigger an exception deliberately, exactly the mechanism professional code uses to enforce its own rules rather than silently accepting bad data. The next lesson applies all of this to file operations and input validation loops, closing with a complete, production-quality Robust Registration System.