Python Programming

Unit 03 – Lecture 02: Errors vs Exceptions and Exception Handling

Tofik Ali

School of Computer Science, UPES Dehradun

February 14, 2026

Repository: `https://github.com/tali7c/Python-Programming`

Quick Links

Core Concepts Demo Interactive Summary

Agenda

Learning Outcomes

- Explain the difference between **errors** and **exceptions**

Learning Outcomes

- Explain the difference between **errors** and **exceptions**
- Use `try/except/else/finally` for safe programs

Learning Outcomes

- Explain the difference between **errors** and **exceptions**
- Use `try/except/else/finally` for safe programs
- Handle multiple exceptions correctly and specifically

Learning Outcomes

- Explain the difference between **errors** and **exceptions**
- Use `try/except/else/finally` for safe programs
- Handle multiple exceptions correctly and specifically
- Use `raise` and `assert` appropriately

Errors vs Exceptions

- **Syntax errors**: program cannot start (invalid code)

Errors vs Exceptions

- **Syntax errors**: program cannot start (invalid code)
- **Exceptions**: runtime problems (division by zero, bad input, missing file)

Errors vs Exceptions

- **Syntax errors**: program cannot start (invalid code)
- **Exceptions**: runtime problems (division by zero, bad input, missing file)
- A robust program anticipates exceptions and handles them cleanly

The Exception Model (Big Picture)

- An exception is **raised** when something goes wrong

The Exception Model (Big Picture)

- An exception is **raised** when something goes wrong
- If not handled, it **propagates** up and stops the program

The Exception Model (Big Picture)

- An exception is **raised** when something goes wrong
- If not handled, it **propagates** up and stops the program
- `try/except` lets you catch and recover

Basic try/except

```
try:
    n = int(input("Enter n: "))
    print(10 / n)
except ValueError:
    print("Please enter an integer.")
except ZeroDivisionError:
    print("Division by zero is not allowed.")
```

else and finally

- else: runs only if **no exception** occurred

```
try:
    f = open("data.txt", "r")
    data = f.read()
except FileNotFoundError:
    print("File not found")
else:
    print("Length:", len(data))
finally:
    # runs even if exception happens
    try:
        f.close()
    except Exception:
        pass
```

else and finally

- else: runs only if **no exception** occurred
- finally: runs **always** (cleanup)

```
try:
    f = open("data.txt", "r")
    data = f.read()
except FileNotFoundError:
    print("File not found")
else:
    print("Length:", len(data))
finally:
    # runs even if exception happens
    try:
        f.close()
    except Exception:
        pass
```

Multiple Exceptions

- Catch **specific** exceptions first

```
try:
    x = int(input("x: "))
    y = int(input("y: "))
    print(x // y)
except (ValueError, ZeroDivisionError) as e:
    print("Error:", e)
```

Multiple Exceptions

- Catch **specific** exceptions first
- Avoid catching everything unless you re-raise or log

```
try:
    x = int(input("x: "))
    y = int(input("y: "))
    print(x // y)
except (ValueError, ZeroDivisionError) as e:
    print("Error:", e)
```

raise and assert

- raise creates your own exception with a message

```
age = int(input("Age: "))
if age < 0:
    raise ValueError("Age must be non-negative")

assert age >= 0
```

raise and assert

- raise creates your own exception with a message
- assert is mainly for debugging assumptions

```
age = int(input("Age: "))
if age < 0:
    raise ValueError("Age must be non-negative")

assert age >= 0
```

Demo: Safe Input + Safe File Read

- File: `demo/exception_handling_demo.py`

Demo: Safe Input + Safe File Read

- File: `demo/exception_handling_demo.py`
- Shows:

Demo: Safe Input + Safe File Read

- File: `demo/exception_handling_demo.py`
- Shows:
  - repeated input until valid integer

Demo: Safe Input + Safe File Read

- File: `demo/exception_handling_demo.py`
- Shows:
  - repeated input until valid integer
  - safe division

Demo: Safe Input + Safe File Read

- File: `demo/exception_handling_demo.py`
- Shows:
  - repeated input until valid integer
  - safe division
  - safe file reading (missing file handling)

Checkpoint 1

Question: When does the `else` block of a `try` statement run?

Checkpoint 2

Question: Why is it better to catch `ValueError` than to catch `Exception` for user input conversion?

Think-Pair-Share

Discuss:

- Where should you place `try/except` blocks?
- Around the entire program OR only around the risky lines?

Key Takeaways

- Exceptions are runtime issues; syntax errors stop the program immediately

Key Takeaways

- Exceptions are runtime issues; syntax errors stop the program immediately
- Use `try/except` to handle expected failures

Key Takeaways

- Exceptions are runtime issues; syntax errors stop the program immediately
- Use `try/except` to handle expected failures
- `else` runs only on success; `finally` runs always

Key Takeaways

- Exceptions are runtime issues; syntax errors stop the program immediately
- Use `try/except` to handle expected failures
- `else` runs only on success; `finally` runs always
- Catch specific exceptions and keep error messages user-friendly

Exit Question

Write a snippet that reads an integer and prints it. If the input is invalid, print `"Invalid integer"` without crashing.