5. Exception Handling

❖Introduction to exceptions and how to handle them using try, except, and finally.

What Are Exceptions?

- **Exceptions** are runtime anomalies—like dividing by zero or accessing an undefined variable—in otherwise syntactically correct code.
- Python provides built-in exception classes (e.g., ZeroDivisionError, TypeError, IOError) and lets you define your own by subclassing Exception .

The try … except Structure :

```
try:
    risky_operation()
except SpecificException as e:
   handle_error(e)
```

- The try block runs code that may raise an exception.
- If an exception occurs and matches the named type in except, that handler runs; otherwise it’s propagated.
- You can chain multiple except blocks to handle different exception types:

Syntax:

```
try:
    ...
except ZeroDivisionError:
    ...
except (TypeError, ValueError):
    ...
else:
    ...
finally:
```

❖Understanding multiple exceptions and custom exceptions.

1. Catching Multiple Exceptions :

You can catch several built-in exceptions when the handling is identical:

```
try:
    result = int(user_input)
except (ValueError, TypeError) as e:
    print("Invalid input:", e)
```

Or use separate `except` blocks for different responses—just ensure you catch more specific exceptions first, otherwise they'll be swallowed by a broad `Exception` block.

2. Custom Exceptions :

When built-ins aren't expressive enough, define your own:

```
class OutOfStockError(Exception):
    def __init__(self, product_id):
        super().__init__(f"Product {product_id} is out of stock")
        self.product_id = product_id
```

This is especially helpful in domain-specific logic—it improves clarity, maintainability, and debugging.

Structuring Custom Exceptions :

- Create a base exception (e.g., class MyAppError(Exception)), then subclass for more nuanced errors (SettingsError, DatabaseError, etc.).
- You can nest them inside classes too, similar to Django's Model.DoesNotExist, but that's less common.