📄 Error and Exception Handling in Python (with Decorators)

1. Introduction

Errors are inevitable in programming. Python provides exceptions to handle unexpected situations gracefully. Along with that, decorators can be used to wrap functions with reusable error-handling logic.

---

2. Types of Errors

1. Syntax Errors → detected before execution.

   ```
   print("Hello"  # SyntaxError
   ```

2. Runtime Errors (Exceptions) → occur during execution.

   ```
   print(10 / 0)  # ZeroDivisionError
   ```

---

3. Basic Exception Handling

```
try:
    num = int("abc")
except ValueError:
    print("Invalid input!")
```

---

4. Multiple Except Blocks

```
try:
    x = 10 / 0
except ZeroDivisionError:
    print("Division by zero!")
except ValueError:
    print("Invalid value!")
```

---

5. Catching Multiple Exceptions

```
try:
    num = int("abc")
```

```
except (ValueError, TypeError) as e:
    print("Error:", e)
```

---

6. Else and Finally

```
try:
    num = int("42")
except ValueError:
    print("Invalid number!")
else:
    print("Conversion successful:", num)
finally:
    print("Execution finished")
```

---

7. Raising Exceptions

```
def divide(a: float, b: float) -> float:
    if b == 0:
        raise ZeroDivisionError("Cannot divide by zero")
    return a / b
```

---

8. Custom Exceptions

```
class NegativeNumberError(Exception):
    """Raised when a negative number is used where not allowed"""
    pass

def factorial(n: int) -> int:
    if n < 0:
        raise NegativeNumberError("Factorial not defined for negatives")
    return 1 if n == 0 else n * factorial(n-1)
```

---

9. Assertions

```
x = -5
assert x >= 0, "x must be non-negative"
```

---

10. Using `with` for Safe Resources

```
try:
    with open("file.txt", "r") as f:
        data = f.read()
except FileNotFoundError:
    print("File does not exist")
```

---

11. Decorators in Error Handling

Decorators allow wrapping functions with extra functionality such as logging, retrying, or catching exceptions.

Example 1: Logging Exceptions

```
from functools import wraps

def log_exceptions(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except Exception as e:
            print(f"Error in {func.__name__}: {e}")
    return wrapper

@log_exceptions
def risky_division(a: int, b: int) -> float:
    return a / b

risky_division(5, 0)  # Error in risky_division: division by zero
```

Example 2: Retrying Function on Exception

```
def retry(times: int = 3):
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for attempt in range(times):
                try:
                    return func(*args, **kwargs)
                except Exception as e:
                    print(f"Attempt {attempt+1} failed: {e}")
```

```
            raise Exception(f"{func.__name__} failed after {times} retries")
        return wrapper
    return decorator

@retry(times=3)
def unstable_func(x: int) -> float:
    if x == 0:
        raise ValueError("x cannot be zero")
    return 10 / x

unstable_func(0)
```

Example 3: Validating Inputs with Decorators

```
def ensure_positive(func):
    @wraps(func)
    def wrapper(n: int):
        if n < 0:
            raise ValueError("Input must be positive")
        return func(n)
    return wrapper

@ensure_positive
def square_root(n: int) -> float:
    return n ** 0.5

print(square_root(16))  # 4.0
# square_root(-9) → raises ValueError
```

---

12. Best Practices

- Catch specific exceptions, not generic `Exception`.
- Use `finally` or `with` for resource cleanup.
- Define custom exceptions for clarity.
- Use decorators to centralize logging, retrying, and validation.

📝 25 Exercises on Error & Exception Handling (with Decorators)

Beginner

1. Handle division by zero error in a function.
2. Catch `ValueError` when converting input to `int`.
3. Handle file not found error when opening a file.
4. Write code that handles `IndexError` in a list.
5. Catch multiple exceptions in a single `try-except`.

Intermediate

6. Use `else` block to print success when no exception occurs.
7. Use `finally` to ensure a closing message always prints.
8. Handle `KeyError` when accessing dictionary keys.
9. Write a program that handles `TypeError` in addition.
10. Implement safe file reading with `with` and `try-except`.

Advanced

11. Raise a `ValueError` if age entered is negative.
12. Define a `PasswordError` custom exception and use it in validation.
13. Use `assert` to validate that a number is positive.
14. Create a decorator that logs exceptions in functions.
15. Implement a retry decorator that retries failed function calls.

Applications

16. Create a function to divide two numbers with custom error messages.
17. Handle exceptions when parsing JSON data.
18. Write a `BankAccount` class that raises exceptions on overdraft.
19. Define a `TemperatureError` for invalid temperature inputs.
20. Wrap risky functions with a decorator to catch errors.

Challenge

21. Write a decorator that retries a function 3 times before failing.
22. Write a decorator that ensures function arguments are positive.
23. Create a decorator that converts exceptions into return values instead of raising them.
24. Implement a file manager class that handles all I/O exceptions.
25. Combine decorators: logging + retry + validation in one function.
