AI ASSISTED CODING
LAB-ASSIGNMENT 5.1 & 6.1

A.Srichala

2303A52435

BT-31

Question 1: Employee Data

Task:

Create Python code that defines a class named `Employee` with the following

attributes: `empid`, `empname`, `designation`, `basic_salary`, and `exp`.

Implement a method `display_details()` to print all employee details. Implement

another method `calculate_allowance()` to determine additional allowance based

on experience:

- If `exp > 10 years` → allowance = 20% of `basic_salary`

- If `5 ≤ exp ≤ 10 years` → allowance = 10% of `basic_salary`

- If `exp < 5 years` → allowance = 5% of `basic_salary`

Finally, create at least one instance of the `Employee` class, call the

`display_details()` method, and print the calculated allowance.

Prompt:

Create a Python class Employee with attributes empid, empname, designation,

basic_salary, and exp. Add methods to display employee details and calculate

allowance based on years of experience with 20% for >10 years, 10% for 5-10 years, and 5% for <5 years.

Code:

```python
class Employee:
    def __init__(self, empid, empname, designation, basic_salary, exp):
        self.empid = empid
        self.empname = empname
        self.designation = designation
        self.basic_salary = basic_salary
        self.exp = exp

    def display_details(self):
        print(f"Employee ID: {self.empid}")
        print(f"Name: {self.empname}")
        print(f"Designation: {self.designation}")
        print(f"Basic Salary: {self.basic_salary}")
        print(f"Experience: {self.exp} years")

    def calculate_allowance(self):
        if self.exp > 10:
            allowance = self.basic_salary * 0.20
        elif self.exp >= 5:
            allowance = self.basic_salary * 0.10
        else:
```

```python
        allowance = self.basic_salary * 0.05

        return allowance


    def display_with_allowance(self):

        self.display_details()

        allowance = self.calculate_allowance()

        total_salary = self.basic_salary + allowance

        print(f"Allowance: {allowance}")

        print(f"Total Salary: {total_salary}")


# Example usage
if __name__ == "__main__":

    emp1 = Employee(101, "John Doe", "Senior Developer", 50000, 12)

    emp1.display_with_allowance()
```

Output(O/P):

Employee ID: 101

Name: John Doe

Designation: Senior Developer

Basic Salary: 50000

Experience: 12 years

Allowance: 10000.0

Total Salary: 60000.0

Explanation:

This class demonstrates object-oriented programming principles by encapsulating

employee data and behavior. The calculate_allowance() method uses conditional

statements to determine allowance percentage based on experience tiers. The

implementation follows clean code practices with clear method names and proper

data encapsulation.

Question 2: Electricity Bill Calculation

Task:

Electricity Bill Calculation- Create Python code that defines a class named

`ElectricityBill` with attributes: `customer_id`, `name`, and `units_consumed`.

Implement a method `display_details()` to print customer details, and a method

`calculate_bill()` where:

- Units ≤ 100 → ₹5 per unit

- 101 to 300 units → ₹7 per unit

- More than 300 units → ₹10 per unit

Create a bill object, display details, and print the total bill amount.

Prompt:

Create Python code that defines a class named `ElectricityBill` with attributes:

`customer_id`, `name`, and `units_consumed`. Implement a method

`display_details()` to print customer details, and a method `calculate_bill()`

where:

- Units ≤ 100 → ₹5 per unit

- 101 to 300 units → ₹7 per unit

- More than 300 units → ₹10 per unit

Create a bill object, display details, and print the total bill amount.

Code:

```python
class ElectricityBill:
    def __init__(self, customer_id, name, units_consumed):
        self.customer_id = customer_id
        self.name = name
        self.units_consumed = units_consumed

    def display_details(self):
        print(f"Customer ID: {self.customer_id}")
        print(f"Name: {self.name}")
        print(f"Units Consumed: {self.units_consumed}")

    def calculate_bill(self):
        if self.units_consumed <= 100:
            bill = self.units_consumed * 5
        elif self.units_consumed <= 300:
            bill = self.units_consumed * 7
        else:
            bill = self.units_consumed * 10
        return bill

# Create a bill object
bill = ElectricityBill("C001", "John Doe", 250)
```

Display details

bill.display_details()

Print total bill amount

print(f"Total Bill Amount: {bill.calculate_bill()}")

Output(O/P):

Customer ID: C001

Name: John Doe

Units Consumed: 250

Total Bill Amount: 1750

Explanation:

The calculate_bill() method implements tiered pricing logic commonly used in utility billing systems. This progressive pricing structure encourages energy conservation by charging higher rates for excessive consumption. The implementation correctly handles boundary conditions between pricing tiers.

Question 3: Product Discount Calculation

Task:

Create Python code that defines a class named `Product` with attributes:

`product_id`, `product_name`, `price`, and `category`. Implement a method `display_details()` to print product details. Implement another method `calculate_discount()` where:

- Electronics → 10% discount

- Clothing → 15% discount

- Grocery → 5% discount

Create at least one product object, display details, and print the final price after discount.

Prompt:

Create a Python class Product with category-based discount calculation: 10% for

Electronics, 15% for Clothing, and 5% for Grocery. Include methods to display

product information and calculate discounted price.

Code:

```python
class Product:
    # Discount rates by category
    DISCOUNTS = {
        "Electronics": 0.10,
        "Clothing": 0.15,
        "Grocery": 0.05
    }

    def __init__(self, name, category, price):
        self.name = name
        self.category = category
        self.price = price
```

```python
    def display_info(self):
        """Display product information"""
        print(f"Product: {self.name}")
        print(f"Category: {self.category}")
        print(f"Price: ${self.price:.2f}")

    def calculate_discounted_price(self):
        """Calculate and return the discounted price"""
        discount_rate = self.DISCOUNTS.get(self.category, 0)
        discount_amount = self.price * discount_rate
        discounted_price = self.price - discount_amount
        return discounted_price

    def display_discount_info(self):
        """Display product info with discount details"""
        self.display_info()
        discount_rate = self.DISCOUNTS.get(self.category, 0)
        discounted_price = self.calculate_discounted_price()
        print(f"Discount: {discount_rate * 100:.0f}%")
        print(f"Discounted Price: ${discounted_price:.2f}")

# Example usage
if __name__ == "__main__":
    product1 = Product("Laptop", "Electronics", 1000)
    product1.display_discount_info()
    print()
```

```python
    product2 = Product("T-Shirt", "Clothing", 50)

    product2.display_discount_info()

    print()


    product3 = Product("Milk", "Grocery", 5)

    product3.display_discount_info()
```

Output(O/P):

Product: Laptop

Category: Electronics

Price: $1000.00

Discount: 10%

Discounted Price: $900.00

Product: T-Shirt

Category: Clothing

Price: $50.00

Discount: 15%

Discounted Price: $42.50

Product: Milk

Category: Grocery

Price: $5.00

Discount: 5%

Discounted Price: $4.75

Explanation:

This class demonstrates polymorphic behavior through category-based discount calculation.

Question 4: Book Late Fee Calculation

Task:

Create Python code that defines a class named `LibraryBook` with attributes:

`book_id`, `title`, `author`, `borrower`, and `days_late`. Implement a method

`display_details()` to print book details, and a method `calculate_late_fee()`

where:

- Days late ≤ 5 → ₹5 per day

- 6 to 10 days late → ₹7 per day

- More than 10 days late → ₹10 per day

Create a book object, display details, and print the late fee.

Prompt:

Create a Python class LibraryBook to calculate late return fees with progressive

rates: ₹5/day for ≤5 days, ₹7/day for 6-10 days, and ₹10/day for >10 days late.

Include book details and display functionality.

Code:

```
class LibraryBook:
    def __init__(self, title, author, isbn, due_date):
        self.title = title
        self.author = author
        self.isbn = isbn
```

```python
        self.due_date = due_date

    def calculate_late_fee(self, days_late):
        """Calculate late return fee based on progressive rates"""
        if days_late <= 0:
            return 0
        elif days_late <= 5:
            return days_late * 5
        elif days_late <= 10:
            return days_late * 7
        else:
            return days_late * 10

    def display_details(self):
        """Display book details"""
        print(f"Title: {self.title}")
        print(f"Author: {self.author}")
        print(f"ISBN: {self.isbn}")
        print(f"Due Date: {self.due_date}")

    def display_with_late_fee(self, days_late):
        """Display book details with late fee calculation"""
        self.display_details()
        fee = self.calculate_late_fee(days_late)
        print(f"Days Late: {days_late}")
        print(f"Late Fee: ₹{fee}")
```

```python
# Example usage
if __name__ == "__main__":
    book = LibraryBook("The Great Gatsby", "F. Scott Fitzgerald",
                "978-0743273565", "2025-01-15")

    print("Book returned 3 days late:")
    book.display_with_late_fee(3)
    print()

    print("Book returned 8 days late:")
    book.display_with_late_fee(8)
    print()

    print("Book returned 15 days late:")
    book.display_with_late_fee(15)
```

Output(O/P):

Book returned 3 days late:

Title: The Great Gatsby

Author: F. Scott Fitzgerald

ISBN: 978-0743273565

Due Date: 2025-01-15

Days Late: 3

Late Fee: ₹15

Book returned 8 days late:

Title: The Great Gatsby

Author: F. Scott Fitzgerald

ISBN: 978-0743273565

Due Date: 2025-01-15

Days Late: 8

Late Fee: ₹56

Book returned 15 days late:

Title: The Great Gatsby

Author: F. Scott Fitzgerald

ISBN: 978-0743273565

Due Date: 2025-01-15

Days Late: 15

Late Fee: ₹150

Explanation:

The late fee calculation implements a progressive penalty structure that incentivizes timely returns while being fair to borrowers with minor delays. The method correctly handles edge cases including on-time returns (zero days late). The implementation uses clear conditional logic to apply appropriate fee rates based on delay duration.

Question 5: Student Grade Calculation

Task:

Create Python code that defines a class named `Student` with attributes: `roll_no`, `name`, and a list of `marks` for five subjects. Implement a method `display_details()` to print student details, and a method `calculate_grade()` that:

- Calculates the average marks

- Returns grade: A (≥90), B (80-89), C (70-79), D (60-69), F (<60)

Create at least one student object, display details, and print the grade.

Prompt:

Create a Python class Student with methods to calculate average marks and assign

grades (A for ≥90, B for 80-89, C for 70-79, D for 60-69, F for <60). Include

student details and display functionality.

Code:

```python
class Student:
    def __init__(self, roll_no, name, marks):
        """
        Initialize Student object
        Args:
            roll_no: Student roll number
            name: Student name
            marks: List of marks for five subjects
        """
        self.roll_no = roll_no
        self.name = name
        self.marks = marks

    def calculate_average(self):
```

```python
        """Calculate average of all marks"""
        return sum(self.marks) / len(self.marks)

    def calculate_grade(self):
        """Calculate grade based on average marks"""
        avg = self.calculate_average()
        if avg >= 90:
            return 'A'
        elif avg >= 80:
            return 'B'
        elif avg >= 70:
            return 'C'
        elif avg >= 60:
            return 'D'
        else:
            return 'F'

    def display_details(self):
        """Display student details with marks, average, and grade"""
        print(f"Roll No: {self.roll_no}")
        print(f"Name: {self.name}")
        print(f"Marks: {self.marks}")
        print(f"Average: {self.calculate_average():.2f}")
        print(f"Grade: {self.calculate_grade()}")

# Example usage
```

```python
if __name__ == "__main__":
    # Create student objects with different performance levels
    student1 = Student("S001", "Alice Johnson", [95, 92, 88, 94, 91])
    student2 = Student("S002", "Bob Smith", [78, 82, 75, 80, 85])
    student3 = Student("S003", "Charlie Brown", [65, 58, 70, 62, 55])

    print("Student 1 Details:")
    student1.display_details()
    print()

    print("Student 2 Details:")
    student2.display_details()
    print()

    print("Student 3 Details:")
    student3.display_details()
```

Output(O/P):

Student 1 Details:

Roll No: S001

Name: Alice Johnson

Marks: [95, 92, 88, 94, 91]

Average: 92.00

Grade: A

Student 2 Details:

Roll No: S002

Name: Bob Smith

Marks: [78, 82, 75, 80, 85]

Average: 80.00

Grade: B

Student 3 Details:

Roll No: S003

Name: Charlie Brown

Marks: [65, 58, 70, 62, 55]

Average: 62.00

Grade: D

Explanation:

The Student class encapsulates academic performance tracking by storing marks

and calculating grades based on average performance. The calculate_average()

method uses Python's built-in sum() function for clean calculation. The grading

system uses cascading if-elif statements ordered from highest to lowest grade,

ensuring the correct grade is assigned. This implementation demonstrates

effective use of class methods to separate concerns: data storage, calculation,

and display.

Question 6: Banking Transaction System

Task:

Create Python code that defines a class named `BankAccount` with attributes:

`account_number`, `holder_name`, and `balance`. Implement methods:

- `deposit(amount)` to add money

- `withdraw(amount)` to subtract money (with balance check)

- `display_details()` to print account information

Create an account object and perform multiple transactions.

Prompt:

Create a Python class BankAccount with deposit and withdrawal methods that

maintain balance integrity. Include account details display and proper balance

checking before withdrawals.

Code:

```
class BankAccount:

    def __init__(self, account_number, holder_name, initial_balance=0):

        """

        Initialize bank account

        Args:

            account_number: Unique account identifier

            holder_name: Account holder's name

            initial_balance: Starting balance (default 0)

        """

        self.account_number = account_number

        self.holder_name = holder_name

        self.balance = initial_balance


    def deposit(self, amount):

        """
```

```python
        Deposit money into account
        Args:
            amount: Amount to deposit
        Returns:
            bool: True if successful, False otherwise
        """

        if amount > 0:
            self.balance += amount
            print(f"Deposited: ₹{amount}")
            print(f"New Balance: ₹{self.balance}")
            return True
        else:
            print("Error: Deposit amount must be positive")
            return False

    def withdraw(self, amount):
        """
        Withdraw money from account
        Args:
            amount: Amount to withdraw
        Returns:
            bool: True if successful, False otherwise
        """
        if amount <= 0:
            print("Error: Withdrawal amount must be positive")
            return False
```

```python
        elif amount > self.balance:

            print(f"Error: Insufficient funds. Current balance: ₹{self.balance}")

            return False

        else:

            self.balance -= amount

            print(f"Withdrawn: ₹{amount}")

            print(f"New Balance: ₹{self.balance}")

            return True


    def display_details(self):

        """Display account information"""

        print(f"Account Number: {self.account_number}")

        print(f"Holder Name: {self.holder_name}")

        print(f"Current Balance: ₹{self.balance}")


# Example usage
if __name__ == "__main__":

    # Create account

    account = BankAccount("ACC001", "John Doe", 1000)


    print("Initial Account Details:")

    account.display_details()

    print()


    print("Performing Deposit:")

    account.deposit(500)
```

```python
    print()

    print("Performing Withdrawal:")
    account.withdraw(300)
    print()

    print("Attempting to overdraw:")
    account.withdraw(2000)
    print()

    print("Final Account Details:")
    account.display_details()
```

Output(O/P):

Initial Account Details:

Account Number: ACC001

Holder Name: John Doe

Current Balance: ₹1000

Performing Deposit:

Deposited: ₹500

New Balance: ₹1500

Performing Withdrawal:

Withdrawn: ₹300

New Balance: ₹1200

Attempting to overdraw:

Error: Insufficient funds. Current balance: ₹1200

Final Account Details:

Account Number: ACC001

Holder Name: John Doe

Current Balance: ₹1200

Explanation:

The BankAccount class implements essential banking operations with proper

validation and error handling. The deposit() and withdraw() methods include

guard clauses to prevent invalid operations like negative amounts or

overdrawing. The class maintains balance integrity by updating it only after

validation passes. Return values (True/False) allow calling code to verify

transaction success. This implementation demonstrates defensive
programming

practices and state management.

Question 7: Temperature Converter

Task:

Create Python code that defines a class named `Temperature` with an attribute

`celsius`. Implement methods to:

- Convert to Fahrenheit: F = (C × 9/5) + 32

- Convert to Kelvin: K = C + 273.15

- `display_all()` to show temperature in all three units

Create a temperature object and display conversions.

Prompt:

Create a Python class Temperature to convert Celsius to Fahrenheit and Kelvin.

Include methods for each conversion and display all units.

Code:

```python
class Temperature:
    def __init__(self, celsius):
        """

        Initialize Temperature object with Celsius value

        Args:

            celsius: Temperature in Celsius
        """

        self.celsius = celsius


    def to_fahrenheit(self):
        """

        Convert Celsius to Fahrenheit

        Formula: F = (C × 9/5) + 32

        Returns:

            Temperature in Fahrenheit
        """

        return (self.celsius * 9/5) + 32
```

```python
    def to_kelvin(self):
        """

        Convert Celsius to Kelvin

        Formula: K = C + 273.15

        Returns:

            Temperature in Kelvin

        """

        return self.celsius + 273.15


    def display_all(self):
        """Display temperature in all three units"""
        print(f"Temperature Conversions:")
        print(f"Celsius: {self.celsius}°C")
        print(f"Fahrenheit: {self.to_fahrenheit():.2f}°F")
        print(f"Kelvin: {self.to_kelvin():.2f}K")


    def set_celsius(self, new_celsius):
        """Update the Celsius temperature"""
        self.celsius = new_celsius


# Example usage
if __name__ == "__main__":
    # Water freezing point
    temp1 = Temperature(0)
    print("Water Freezing Point:")
```

```python
    temp1.display_all()

    print()


    # Room temperature

    temp2 = Temperature(25)

    print("Room Temperature:")

    temp2.display_all()

    print()


    # Water boiling point

    temp3 = Temperature(100)

    print("Water Boiling Point:")

    temp3.display_all()

    print()


    # Demonstrate temperature update

    print("Updating temperature:")

    temp1.set_celsius(37)

    print("Human Body Temperature:")

    temp1.display_all()
```

Output(O/P):

Water Freezing Point:

Temperature Conversions:

Celsius: 0°C

Fahrenheit: 32.00°F

Kelvin: 273.15K

Room Temperature:

Temperature Conversions:

Celsius: 25°C

Fahrenheit: 77.00°F

Kelvin: 298.15K

Water Boiling Point:

Temperature Conversions:

Celsius: 100°C

Fahrenheit: 212.00°F

Kelvin: 373.15K

Updating temperature:

Human Body Temperature:

Temperature Conversions:

Celsius: 37°C

Fahrenheit: 98.60°F

Kelvin: 310.15K

Explanation:

The Temperature class encapsulates temperature conversion logic using standard

scientific formulas. The class stores temperature in Celsius as the base unit

and provides methods to convert to other scales on demand. This approach
maintains a single source of truth while allowing flexible representation. The
display_all() method provides a convenient interface for viewing all
conversions simultaneously. This implementation demonstrates encapsulation and
method-based calculations.

Question 8: Transparency in Algorithm Complexity

Task:

Use AI to generate two versions of a prime-checking algorithm: naive and
optimized. Instructions:

1. Ask AI to explain the time complexity of each.

2. Ask AI to justify why the optimized version is better.

3. Verify explanations with actual runtime comparison.

Expected Output:

• Two versions of prime-checking code.

• Detailed explanation of time complexity (O(n) vs O(√n)).

• Runtime verification showing performance difference.

Prompt:

Generate two prime-checking algorithms (naive O(n) and optimized O(√n)) with

clear time complexity explanations. Include documentation showing why the

optimized version is more efficient.

Code:

```
def is_prime_naive(n):
    """
    Naive prime checking algorithm.
```

Time Complexity: O(n)

This method checks all numbers from 2 to n-1 to see if any divide n.

For large numbers, this becomes very slow as it tests every number.
"""

```python
if n <= 1:
    return False
for i in range(2, n):
    if n % i == 0:
        return False
return True
```

```python
def is_prime_optimized(n):
    """
    Optimized prime checking algorithm.

    Time Complexity: O(sqrt(n))

    This method checks for factors only up to the square root of n.
    It significantly reduces the number of checks, especially for large numbers.
    """
    if n <= 1:
        return False
    if n <= 3:
        return True
```

```python
    if n % 2 == 0 or n % 3 == 0:

        return False

    i = 5

    while i * i <= n:

        if n % i == 0 or n % (i + 2) == 0:

            return False

        i += 6

    return True


# Example usage

number = 29

print(f"Naive method: {number} is prime? {is_prime_naive(number)}")

print(f"Optimized method: {number} is prime? {is_prime_optimized(number)}")
```

Output(O/P):

Naive method: 29 is prime? True

Optimized method: 29 is prime? True

Explanation:

The naive prime check tests all numbers from 2 to n minus one to see if any

divides the given number, which makes it slow with time complexity O(n). The

optimized method improves this by checking divisibility only up to the square

root of the number, skipping even numbers and multiples of three, which greatly

reduces the number of checks and runs in O(square root of n) time, making it

much faster for larger values.

Question 9: Transparency in Recursive Algorithms

Task:

Use AI to generate a recursive function to calculate Fibonacci numbers.

Instructions:

1. Ask AI to add clear comments explaining recursion.

2. Ask AI to explain base cases and recursive calls.

Expected Output:

• Well-commented recursive code.

• Clear explanation of how recursion works.

• Verification that explanation matches actual execution.

Prompt:

Generate a well-commented recursive Fibonacci function in Python. Explain base

cases, recursive calls, and provide visualization of how recursion works with

clear documentation.

Code:

```
def fibonacci(n):
    """
    Calculate the nth Fibonacci number using recursion.

    Base cases:
    - fibonacci(0) = 0
    - fibonacci(1) = 1

    Recursive case:
    - fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)
```

```python
    Args:
        n: Non-negative integer representing the position in Fibonacci sequence

    Returns:
        The nth Fibonacci number

    Example:
        fibonacci(5) = 5
        Sequence: 0, 1, 1, 2, 3, 5, 8, 13...
    """
    # Base cases: stop recursion
    if n == 0:
        return 0
    if n == 1:
        return 1

    # Recursive case: break problem into smaller subproblems
    return fibonacci(n - 1) + fibonacci(n - 2)


def fibonacci_with_trace(n, depth=0):
    """
    Fibonacci with recursion tree visualization.
    Shows how function calls branch during recursion.
    """
    indent = "  " * depth
    print(f"{indent} fibonacci({n})")
```

```python
    if n == 0:
        print(f"{indent} returns 0 (base case)")
        return 0
    if n == 1:
        print(f"{indent} returns 1 (base case)")
        return 1

    # Recursive calls shown with indentation
    left = fibonacci_with_trace(n - 1, depth + 1)
    right = fibonacci_with_trace(n - 2, depth + 1)
    result = left + right

    print(f"{indent} returns {result}")
    return result

# Test the functions
if __name__ == "__main__":
    print("Fibonacci(5):", fibonacci(5))
    print("\nRecursion tree for fibonacci(4):")
    fibonacci_with_trace(4)
```

Output(O/P):

Fibonacci(5): 5

Recursion tree for fibonacci(4):

 fibonacci(4)

```
fibonacci(3)
  fibonacci(2)
    fibonacci(1)
    returns 1 (base case)
    fibonacci(0)
    returns 0 (base case)
  returns 1
  fibonacci(1)
  returns 1 (base case)
returns 2
fibonacci(2)
  fibonacci(1)
  returns 1 (base case)
  fibonacci(0)
  returns 0 (base case)
returns 1
returns 3
```

Explanation:

This program calculates Fibonacci numbers using recursion, where the function

returns zero for input zero and one for input one, and for any larger value it

calls itself to add the two previous Fibonacci numbers. The tracing version

prints each function call with indentation to visually show how recursion

branches into smaller subproblems, helping to understand the recursion tree and

how results are combined step by step, though this approach is inefficient due

to repeated calculations.

Question 10: Transparency in Error Handling

Task:

Use AI to generate a Python program that reads a file and processes data.

Prompt: "Generate code with proper error handling and clear explanations for each exception."

Expected Output:

• Code with meaningful exception handling.

• Clear comments explaining each error scenario.

• Validation that explanations align with runtime behavior.

Prompt:

Generate Python code with comprehensive error handling using custom exceptions.

Include clear explanations for each error scenario: invalid input, calculation errors, and missing data. Demonstrate transparent exception handling with descriptive messages.

Code:

```
"""
Comprehensive error handling with custom exceptions.
Demonstrates transparent exception handling with descriptive messages.
"""
# Custom Exception Classes
class InvalidInputError(Exception):
    """Raised when input validation fails."""
    pass
```

```python
class CalculationError(Exception):
    """Raised when a calculation operation fails."""
    pass


class MissingDataError(Exception):
    """Raised when required data is missing."""
    pass


# Example functions demonstrating error handling
def divide_numbers(a, b):
    """
    Divides two numbers with error handling.

    Args:
        a: Numerator
        b: Denominator

    Raises:
        InvalidInputError: If inputs are not numeric
        CalculationError: If division by zero is attempted
    """
    try:
        if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
            raise InvalidInputError(f"Expected numeric values, got "
                        f"{type(a).__name__} and {type(b).__name__}")
```

```python
        if b == 0:
            raise CalculationError("Cannot divide by zero")

        return a / b

    except (InvalidInputError, CalculationError) as e:
        print(f"Error: {e}")
        raise

def process_user_data(user_dict, required_fields):
    """
    Validates and processes user data.

    Args:
        user_dict: Dictionary containing user information
        required_fields: List of required field names

    Raises:
        MissingDataError: If required fields are missing
    """
    missing_fields = [field for field in required_fields if field not in
            user_dict]

    if missing_fields:
        raise MissingDataError(f"Missing required fields: "
                f"{', '.join(missing_fields)}")
```

```python
        return user_dict


# Main execution with comprehensive error handling
if __name__ == "__main__":
    # Test 1: Valid calculation
    try:
        result = divide_numbers(10, 2)
        print(f" Division result: {result}")
    except Exception as e:
        print(f" Unexpected error: {e}")


    # Test 2: Invalid input (type error)
    try:
        result = divide_numbers("10", 2)
    except InvalidInputError as e:
        print(f" Caught invalid input: {e}")


    # Test 3: Calculation error (division by zero)
    try:
        result = divide_numbers(10, 0)
    except CalculationError as e:
        print(f" Caught calculation error: {e}")


    # Test 4: Valid user data
    try:
```

```python
    user = {"name": "Alice", "email": "alice@example.com"}

    process_user_data(user, ["name", "email"])

    print(" User data validated successfully")

except MissingDataError as e:

    print(f" Data validation failed: {e}")


# Test 5: Missing data

try:

    user = {"name": "Bob"}

    process_user_data(user, ["name", "email", "phone"])

except MissingDataError as e:

    print(f" Caught missing data: {e}")
```

Output(O/P):

Division result: 5.0

Error: Expected numeric values, got str and int

 Caught invalid input: Expected numeric values, got str and int

Error: Cannot divide by zero

 Caught calculation error: Cannot divide by zero

 User data validated successfully

 Caught missing data: Missing required fields: email, phone

Explanation:

This program demonstrates robust error handling in Python by defining custom

exceptions for invalid input, calculation failures, and missing data, making

errors clear and meaningful. The divide function validates input types and

prevents division by zero, while the user data function ensures all required

fields are present, and the main block shows how different exceptions are raised, caught, and handled cleanly to keep the program reliable and easy to debug.