Python Programming Guide - UNIT V: Object-Oriented Programming and Exception Handling

1. Object-Oriented Programming Concepts

Object-Oriented Programming (OOP) is a paradigm that structures code around objects and classes, promoting modular, reusable, and maintainable code.

Core OOP Concepts

Encapsulation: Bundling data and methods together, hiding internal details.

Inheritance: Creating new classes based on existing classes.

Polymorphism: Using the same interface for objects of different types.

Abstraction: Hiding complex implementation details, showing only essential features.

---

2. Class and Object

A class is a blueprint for creating objects. An object is an instance of a class.

```
        # Basic class definition
        class Person:
            """A class representing a person."""

            def __init__(self, name, age):
                """Constructor - initializes object attributes."""
                self.name = name
                self.age = age
```

```
            def greet(self):
                """Instance method - operates on object data."""
                return f"Hello, I'm {self.name} and I'm {self.age} years
old."

            def have_birthday(self):
                """Modify object state."""
                self.age += 1
                return f"{self.name} is now {self.age} years old."

        # Creating objects (instances)
        person1 = Person("Alice", 25)
        person2 = Person("Bob", 30)

        # Accessing attributes
        print(person1.name)              # Output: Alice
        print(person2.age)               # Output: 30

        # Calling methods
        print(person1.greet())           # Output: Hello, I'm Alice...
        print(person1.have_birthday())   # Output: Alice is now 26...

        # Objects are independent
        print(person1.age)               # Output: 26
        print(person2.age)               # Output: 30 (unchanged)

        # Class attributes (shared across all instances)
        class Counter:
            count = 0  # Class attribute

            def __init__(self, name):
                self.name = name  # Instance attribute
                Counter.count += 1

            def get_total_count(self):
                return Counter.count

        c1 = Counter("obj1")
        c2 = Counter("obj2")
        c3 = Counter("obj3")

        print(f"Total objects created: {c1.get_total_count()}")  # Output: 3
        print(f"Class attribute: {Counter.count}")               # Output: 3
```

Real-World Application: Employee Management System

```
        class Employee:
            """Represents an employee in a company."""

            # Class attributes
            company_name = "TechCorp"
            total_employees = 0

            def __init__(self, emp_id, name, department, salary):
                """Initialize employee with basic information."""
                self.emp_id = emp_id
                self.name = name
                self.department = department
                self.salary = salary
                self.hired_date = "2026-02-03"
```

```
                Employee.total_employees += 1

            def display_info(self):
                """Display employee information."""
                print(f"\n{'='*50}")
                print(f"EMPLOYEE INFORMATION")
                print(f"{'='*50}")
                print(f"ID: {self.emp_id}")
                print(f"Name: {self.name}")
                print(f"Department: {self.department}")
                print(f"Salary: ${self.salary:,.2f}")
                print(f"Company: {Employee.company_name}")
                print(f"{'='*50}")

            def give_raise(self, percentage):
                """Give salary raise to employee."""
                raise_amount = self.salary * (percentage / 100)
                self.salary += raise_amount
                return f"{self.name} received a {percentage}% raise:
${self.salary:,.2f}"

            @classmethod
            def get_total_employees(cls):
                """Class method - operates on class, not instance."""
                return cls.total_employees

            @staticmethod
            def calculate_annual_tax(salary, tax_rate=0.15):
                """Static method - doesn't use instance or class data."""
                return salary * tax_rate

        # Usage
        emp1 = Employee("E001", "Alice", "Engineering", 75000)
        emp2 = Employee("E002", "Bob", "Sales", 60000)

        emp1.display_info()
        print(emp1.give_raise(10))

        print(f"\nTotal Employees: {Employee.get_total_employees()}")
        print(f"Annual Tax (15%):
${Employee.calculate_annual_tax(emp1.salary):,.2f}")
```

---

3. Constructor (`__init__`)

The constructor is a special method called when an object is created. It initializes the object's attributes.

```
        # Constructor examples
        class BankAccount:
            """Example of constructor usage."""

            def __init__(self, account_holder, initial_balance=0):
                """Initialize bank account."""
                self.account_holder = account_holder
                self.balance = initial_balance
                self.transaction_history = []
                self.account_number = self._generate_account_number()
```

```
            def _generate_account_number(self):
                """Private method to generate account number."""
                import random
                return f"ACC{random.randint(100000, 999999)}"

            def deposit(self, amount):
                """Add money to account."""
                if amount > 0:
                    self.balance += amount
                    self.transaction_history.append(f"Deposit: +${amount}")
                    return True
                return False

            def withdraw(self, amount):
                """Remove money from account."""
                if 0 < amount <= self.balance:
                    self.balance -= amount
                    self.transaction_history.append(f"Withdrawal: -
${amount}")
                    return True
                return False

            def __str__(self):
                """String representation of object."""
                return f"Account({self.account_number},
{self.account_holder}, ${self.balance})"

            def __repr__(self):
                """Developer-friendly representation."""
                return f"BankAccount('{self.account_holder}',
{self.balance})"

        # Using constructor
        account1 = BankAccount("Alice", 1000)
        account2 = BankAccount("Bob", 500)

        print(account1)                 # Uses __str__()
        account1.deposit(500)
        account1.withdraw(200)

        print(f"\nAccount Info:")
        print(f"Balance: ${account1.balance}")
        print(f"Transactions: {account1.transaction_history}")
```

---

4. Inheritance

Inheritance allows a class to inherit attributes and methods from another class.

```
        # Base class (Parent)
        class Animal:
            """Base class for all animals."""

            def __init__(self, name, age):
                self.name = name
                self.age = age

            def speak(self):
```

```
            return f"{self.name} makes a sound"

        def eat(self):
            return f"{self.name} is eating"

    # Derived class (Child)
    class Dog(Animal):
        """Dog class inheriting from Animal."""

        def __init__(self, name, age, breed):
            super().__init__(name, age)  # Call parent constructor
            self.breed = breed

        def speak(self):
            """Override parent method."""
            return f"{self.name} barks: Woof!"

        def fetch(self):
            """New method specific to Dog."""
            return f"{self.name} is fetching the ball"

    class Cat(Animal):
        """Cat class inheriting from Animal."""

        def speak(self):
            return f"{self.name} meows: Meow!"

        def scratch(self):
            return f"{self.name} is scratching the sofa"

    # Usage
    dog = Dog("Buddy", 3, "Golden Retriever")
    cat = Cat("Whiskers", 2)

    print(dog.speak())              # Output: Buddy barks: Woof!
    print(dog.fetch())              # Output: Buddy is fetching the
ball
    print(cat.speak())              # Output: Whiskers meows: Meow!
    print(dog.eat())                # Inherited method

    # Multiple inheritance
    class Swimmer:
        def swim(self):
            return "Swimming"

    class Flyer:
        def fly(self):
            return "Flying"

    class Duck(Animal, Swimmer, Flyer):
        """Duck inherits from multiple classes."""

        def speak(self):
            return f"{self.name} quacks: Quack!"

    duck = Duck("Donald", 2)
    print(duck.speak())             # Output: Donald quacks: Quack!
    print(duck.swim())              # Output: Swimming
    print(duck.fly())               # Output: Flying
```

Real-World Application: Vehicle Management System

```
        class Vehicle:
            """Base class for all vehicles."""

            def __init__(self, brand, model, year, price):
                self.brand = brand
                self.model = model
                self.year = year
                self.price = price
                self.is_running = False

            def start_engine(self):
                self.is_running = True
                return f"{self.brand} {self.model} engine started"

            def stop_engine(self):
                self.is_running = False
                return f"{self.brand} {self.model} engine stopped"

            def display_info(self):
                return f"{self.year} {self.brand} {self.model}
($${self.price})"

        class Car(Vehicle):
            """Car class inheriting from Vehicle."""

            def __init__(self, brand, model, year, price, doors,
trunk_size):
                super().__init__(brand, model, year, price)
                self.doors = doors
                self.trunk_size = trunk_size

            def open_trunk(self):
                return f"{self.brand} trunk opened (size:
{self.trunk_size}L)"

        class Truck(Vehicle):
            """Truck class inheriting from Vehicle."""

            def __init__(self, brand, model, year, price, cargo_capacity,
towing_capacity):
                super().__init__(brand, model, year, price)
                self.cargo_capacity = cargo_capacity
                self.towing_capacity = towing_capacity

            def load_cargo(self, weight):
                if weight <= self.cargo_capacity:
                    return f"Loaded {weight}kg of cargo (Capacity:
{self.cargo_capacity}kg)"
                return f"Cargo too heavy! Max capacity:
{self.cargo_capacity}kg"

        class Motorcycle(Vehicle):
            """Motorcycle class inheriting from Vehicle."""

            def __init__(self, brand, model, year, price, engine_cc):
                super().__init__(brand, model, year, price)
                self.engine_cc = engine_cc
```

```
            def wheelie(self):
                return f"Performing wheelie on {self.brand} {self.model}!"

        # Usage
        car = Car("Toyota", "Camry", 2023, 25000, 4, 400)
        truck = Truck("Ford", "F-150", 2023, 35000, 1000, 5000)
        motorcycle = Motorcycle("Harley-Davidson", "Street 750", 2023, 7500,
750)

        print(car.display_info())
        print(car.start_engine())
        print(car.open_trunk())

        print(f"\n{truck.display_info()}")
        print(truck.start_engine())
        print(truck.load_cargo(800))

        print(f"\n{motorcycle.display_info()}")
        print(motorcycle.wheelie())
```

---

5. Polymorphism

Polymorphism allows different classes to be used interchangeably through the same interface.

```
        # Polymorphic behavior
        class Shape:
            """Base class for shapes."""

            def area(self):
                raise NotImplementedError("Subclasses must implement
area()")

            def perimeter(self):
                raise NotImplementedError("Subclasses must implement
perimeter()")

        class Rectangle(Shape):
            """Rectangle inheriting from Shape."""

            def __init__(self, width, height):
                self.width = width
                self.height = height

            def area(self):
                return self.width * self.height

            def perimeter(self):
                return 2 * (self.width + self.height)

        class Circle(Shape):
            """Circle inheriting from Shape."""

            def __init__(self, radius):
                self.radius = radius

            def area(self):
                import math
```

```
                return math.pi * self.radius ** 2

            def perimeter(self):
                import math
                return 2 * math.pi * self.radius

        class Triangle(Shape):
            """Triangle inheriting from Shape."""

            def __init__(self, a, b, c):
                self.a = a
                self.b = b
                self.c = c

            def area(self):
                # Using Heron's formula
                s = (self.a + self.b + self.c) / 2
                return (s * (s - self.a) * (s - self.b) * (s - self.c)) **
0.5

            def perimeter(self):
                return self.a + self.b + self.c

        # Polymorphic usage - same method works for different shapes
        shapes = [
            Rectangle(5, 10),
            Circle(7),
            Triangle(3, 4, 5)
        ]

        print("Shape Analysis:")
        print("=" * 50)

        total_area = 0
        for shape in shapes:
            area = shape.area()
            perimeter = shape.perimeter()
            total_area += area
            print(f"{shape.__class__.__name__:10} Area: {area:8.2f}
Perimeter: {perimeter:8.2f}")

        print("=" * 50)
        print(f"Total Area: {total_area:.2f}")
```

---

6. Encapsulation

Encapsulation is hiding internal details and providing a controlled interface.

```
        # Encapsulation using private attributes
        class BankAccount:
            """Example of encapsulation."""

            def __init__(self, account_holder, initial_balance):
                self.__account_holder = account_holder  # Private attribute
                self.__balance = initial_balance         # Private attribute
                self.__pin = "1234"                      # Private attribute
```

```
            # Public method - controlled access
            def deposit(self, amount):
                """Public method to deposit money."""
                if amount > 0:
                    self.__balance += amount
                    return f"Deposited ${amount}. New balance:
${self.__balance}"
                return "Invalid amount"

            def withdraw(self, amount, pin):
                """Public method to withdraw with validation."""
                if not self.__verify_pin(pin):
                    return "Incorrect PIN"

                if amount > 0 and amount <= self.__balance:
                    self.__balance -= amount
                    return f"Withdrawn ${amount}. New balance:
${self.__balance}"
                return "Insufficient funds or invalid amount"

            def get_balance(self, pin):
                """Public method to check balance."""
                if self.__verify_pin(pin):
                    return self.__balance
                return None

            def __verify_pin(self):
                """Private method - cannot be called directly."""
                # PIN verification logic
                pass

            @property
            def account_holder(self):
                """Property - controlled read access."""
                return self.__account_holder

            @account_holder.setter
            def account_holder(self, name):
                """Property - controlled write access."""
                if isinstance(name, str) and len(name) > 0:
                    self.__account_holder = name
                else:
                    raise ValueError("Invalid name")

        # Usage
        account = BankAccount("Alice", 1000)

        print(account.account_holder)            # Alice (using property)
        account.account_holder = "Alice Johnson" # Using property setter

        print(account.deposit(500))              # Public method
        print(account.withdraw(200, "1234"))     # Public method

        # Private attributes cannot be accessed directly
        # print(account.__balance)  # AttributeError
```

---

7. Abstraction

Abstraction hides complex implementation details and shows only essential features.

```
        from abc import ABC, abstractmethod

        # Abstract base class
        class DataProcessor(ABC):
            """Abstract class defining interface for data processors."""

            @abstractmethod
            def process(self, data):
                """Process data - must be implemented by subclasses."""
                pass

            @abstractmethod
            def validate(self, data):
                """Validate data - must be implemented by subclasses."""
                pass

            # Concrete method in abstract class
            def log_process(self, operation):
                """Concrete method available to all subclasses."""
                print(f"Processing: {operation}")

        class JSONProcessor(DataProcessor):
            """Concrete implementation for JSON processing."""

            def validate(self, data):
                import json
                try:
                    json.loads(data)
                    return True
                except:
                    return False

            def process(self, data):
                if self.validate(data):
                    import json
                    self.log_process("JSON data")
                    parsed = json.loads(data)
                    return parsed
                return None

        class CSVProcessor(DataProcessor):
            """Concrete implementation for CSV processing."""

            def validate(self, data):
                lines = data.strip().split('\n')
                return len(lines) > 0

            def process(self, data):
                if self.validate(data):
                    self.log_process("CSV data")
                    lines = data.strip().split('\n')
                    headers = lines[0].split(',')
                    rows = [dict(zip(headers, line.split(','))) for line in
lines[1:]]
                    return rows
                return None

        # Usage
```

```
json_processor = JSONProcessor()
csv_processor = CSVProcessor()

json_data = '{"name": "Alice", "age": 25}'
csv_data = "name,age\nAlice,25\nBob,30"

json_result = json_processor.process(json_data)
csv_result = csv_processor.process(csv_data)

print(f"JSON Result: {json_result}")
print(f"CSV Result: {csv_result}")
```

---

8. Exception Handling

Exception handling allows graceful error management without crashing the program.

Types of Errors

```
# 1. Syntax Errors - caught at parse time
# if True  # Missing colon - SyntaxError

# 2. Runtime Errors - occur during execution
# ZeroDivisionError
result = 10 / 0

# 3. Logical Errors - program runs but produces wrong results
x = 5
y = 10
print(x > y)  # Might be logical error if we expected True

# 4. Exception Types
try:
    # ValueError
    int("abc")
except ValueError:
    print("Cannot convert to integer")

try:
    # TypeError
    x = "hello" + 5
except TypeError:
    print("Cannot add string and integer")

try:
    # IndexError
    my_list = [1, 2, 3]
    print(my_list[10])
except IndexError:
    print("Index out of range")

try:
    # KeyError
    my_dict = {'name': 'Alice'}
    print(my_dict['age'])
except KeyError:
    print("Key not found in dictionary")
```

```
try:
    # FileNotFoundError
    with open('nonexistent.txt', 'r') as f:
        content = f.read()
except FileNotFoundError:
    print("File not found")

try:
    # AttributeError
    my_list = [1, 2, 3]
    my_list.nonexistent_method()
except AttributeError:
    print("Attribute or method not found")
```

try, except, else, finally

```
# Basic exception handling
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ValueError:
    print("Invalid input - not a number")
except ZeroDivisionError:
    print("Cannot divide by zero")
except Exception as e:
    print(f"An error occurred: {e}")

# try-except-else
try:
    file = open('data.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print("File not found")
else:
    # Executes if no exception occurs
    print(f"Successfully read {len(content)} characters")
    file.close()

# try-except-finally
try:
    file = open('data.txt', 'r')
    content = file.read()
except FileNotFoundError:
    print("File not found")
finally:
    # Always executes, even if exception occurs
    print("Cleanup completed")
    # file.close()  would be called here if file was opened

# try-except-else-finally
try:
    num = int(input("Enter a number: "))
    result = 100 / num
except ValueError:
    print("Invalid input")
except ZeroDivisionError:
    print("Cannot divide by zero")
else:
    print(f"Result: {result}")
finally:
```

```
            print("Operation completed")
```

Real-World Application: File Processing with Error Handling

```
        import os
        import json

        class FileProcessor:
            """Process files with comprehensive error handling."""

            @staticmethod
            def read_json_file(filename):
                """Read and parse JSON file with error handling."""
                try:
                    if not os.path.exists(filename):
                        raise FileNotFoundError(f"File '{filename}' not
found")

                    with open(filename, 'r') as f:
                        data = json.load(f)

                    return data

                except FileNotFoundError as e:
                    print(f"Error: {e}")
                    return None

                except json.JSONDecodeError:
                    print(f"Error: File '{filename}' is not valid JSON")
                    return None

                except Exception as e:
                    print(f"Unexpected error: {e}")
                    return None

            @staticmethod
            def write_json_file(filename, data):
                """Write data to JSON file with error handling."""
                try:
                    with open(filename, 'w') as f:
                        json.dump(data, f, indent=2)
                    return True

                except TypeError:
                    print("Error: Data is not JSON serializable")
                    return False

                except IOError:
                    print(f"Error: Cannot write to file '{filename}'")
                    return False

                except Exception as e:
                    print(f"Unexpected error: {e}")
                    return False

            @staticmethod
            def process_user_data(filename):
                """Process user data with comprehensive error handling."""
                try:
                    data = FileProcessor.read_json_file(filename)
```

```
                    if data is None:
                        return None

                    # Validate data
                    if not isinstance(data, list):
                        raise ValueError("Data should be a list of users")

                    processed = []
                    for user in data:
                        try:
                            if 'name' not in user or 'age' not in user:
                                raise KeyError("Missing 'name' or 'age'
field")

                            age = int(user['age'])
                            if age < 0 or age > 150:
                                raise ValueError(f"Invalid age: {age}")

                            processed.append({
                                'name': user['name'],
                                'age': age,
                                'is_adult': age >= 18
                            })

                        except KeyError as e:
                            print(f"Warning: Skipping user - {e}")
                        except ValueError as e:
                            print(f"Warning: Invalid data - {e}")

                    return processed

                except Exception as e:
                    print(f"Error processing file: {e}")
                    return None

        # Usage
        # processor = FileProcessor()
        # users = processor.process_user_data('users.json')
        # if users:
        #     print(f"Processed {len(users)} users successfully")
```

User-Defined Exceptions

```
        # Creating custom exceptions
        class InsufficientFundsException(Exception):
            """Raised when account has insufficient funds."""
            pass

        class InvalidAmountException(Exception):
            """Raised when amount is invalid."""
            pass

        class BankAccountAdvanced:
            """Advanced bank account with custom exceptions."""

            def __init__(self, holder, balance):
                self.holder = holder
                self.balance = balance

            def withdraw(self, amount):
```

```
                """Withdraw with custom exception handling."""
                try:
                    if amount <= 0:
                        raise InvalidAmountException("Amount must be
positive")

                    if amount > self.balance:
                        raise InsufficientFundsException(
                            f"Insufficient funds. Balance: ${self.balance},
"
                            f"Requested: ${amount}"
                        )

                    self.balance -= amount
                    return f"Withdrawal successful. New balance:
${self.balance}"

                except InvalidAmountException as e:
                    return f"Invalid amount error: {e}"
                except InsufficientFundsException as e:
                    return f"Insufficient funds error: {e}"

            def deposit(self, amount):
                """Deposit with custom exception handling."""
                try:
                    if amount <= 0:
                        raise InvalidAmountException("Deposit amount must be
positive")

                    self.balance += amount
                    return f"Deposit successful. New balance:
${self.balance}"

                except InvalidAmountException as e:
                    return f"Invalid amount error: {e}"

        # Usage
        account = BankAccountAdvanced("Alice", 1000)

        print(account.deposit(500))      # Success
        print(account.withdraw(1500))    # InsufficientFundsException
        print(account.withdraw(-100))    # InvalidAmountException
        print(account.withdraw(200))     # Success
```

Complete Real-World Application: Library Management System

```
        from datetime import datetime, timedelta
        from abc import ABC, abstractmethod

        class Item(ABC):
            """Abstract base class for library items."""

            def __init__(self, item_id, title, author):
                self.item_id = item_id
                self.title = title
                self.author = author
                self.available = True
```

```
            @abstractmethod
            def get_details(self):
                pass

        class Book(Item):
            """Concrete implementation for books."""

            def __init__(self, item_id, title, author, isbn, pages):
                super().__init__(item_id, title, author)
                self.isbn = isbn
                self.pages = pages

            def get_details(self):
                status = "Available" if self.available else "Checked Out"
                return f"Book: {self.title} by {self.author} ({self.pages}
pages) - {status}"

        class Magazine(Item):
            """Concrete implementation for magazines."""

            def __init__(self, item_id, title, author, issue,
publication_date):
                super().__init__(item_id, title, author)
                self.issue = issue
                self.publication_date = publication_date

            def get_details(self):
                status = "Available" if self.available else "Checked Out"
                return f"Magazine: {self.title} - Issue {self.issue}
({status})"

        class Member:
            """Represents a library member."""

            def __init__(self, member_id, name, email):
                self.member_id = member_id
                self.name = name
                self.email = email
                self.borrowed_items = []
                self.total_fines = 0

            def borrow_item(self, item):
                """Borrow an item from library."""
                if not item.available:
                    raise Exception(f"{item.title} is not available")

                item.available = False
                due_date = datetime.now() + timedelta(days=14)
                self.borrowed_items.append({
                    'item': item,
                    'borrow_date': datetime.now(),
                    'due_date': due_date
                })
                return f"{item.title} borrowed successfully. Due:
{due_date.strftime('%Y-%m-%d')}"

            def return_item(self, item_id):
                """Return a borrowed item."""
                for borrowed in self.borrowed_items:
```

```
                    if borrowed['item'].item_id == item_id:
                        item = borrowed['item']
                        item.available = True

                        # Calculate fines for overdue
                        if datetime.now() > borrowed['due_date']:
                            days_overdue = (datetime.now() -
borrowed['due_date']).days
                            fine = days_overdue * 1.0  # $1 per day
                            self.total_fines += fine
                            self.borrowed_items.remove(borrowed)
                            return f"{item.title} returned with ${fine:.2f}
fine (overdue by {days_overdue} days)"
                        else:
                            self.borrowed_items.remove(borrowed)
                            return f"{item.title} returned successfully"

                raise Exception("Item not found in borrowed items")

        class Library:
            """Main library system."""

            def __init__(self, name):
                self.name = name
                self.items = []
                self.members = []

            def add_item(self, item):
                """Add an item to library."""
                self.items.append(item)

            def register_member(self, member):
                """Register a new member."""
                self.members.append(member)

            def search_item(self, title):
                """Search for item by title."""
                for item in self.items:
                    if item.title.lower() == title.lower():
                        return item
                return None

            def get_member(self, member_id):
                """Get member by ID."""
                for member in self.members:
                    if member.member_id == member_id:
                        return member
                return None

            def display_inventory(self):
                """Display all items in library."""
                print(f"\n{self.name} - Inventory")
                print("=" * 60)
                for item in self.items:
                    print(item.get_details())
                print("=" * 60)

        # Usage
        try:
            # Create library
```

```
            library = Library("City Central Library")

            # Add items
            library.add_item(Book("B001", "Python Programming", "John Doe",
"123-456", 400))
            library.add_item(Book("B002", "Data Science Basics", "Jane
Smith", "123-457", 350))
            library.add_item(Magazine("M001", "Tech Monthly", "Various", 5,
"2026-01-01"))

            # Register members
            member1 = Member("MEM001", "Alice", "alice@example.com")
            member2 = Member("MEM002", "Bob", "bob@example.com")
            library.register_member(member1)
            library.register_member(member2)

            # Display inventory
            library.display_inventory()

            # Member borrows items
            print("\n" + "=" * 60)
            print("BORROWING OPERATIONS")
            print("=" * 60)

            book = library.search_item("Python Programming")
            if book:
                print(member1.borrow_item(book))

            magazine = library.search_item("Tech Monthly")
            if magazine:
                print(member2.borrow_item(magazine))

            # Display inventory again
            library.display_inventory()

            # Return items
            print("\n" + "=" * 60)
            print("RETURN OPERATIONS")
            print("=" * 60)
            print(member1.return_item("B001"))
            print(member2.return_item("M001"))

            # Final inventory
            library.display_inventory()

        except Exception as e:
            print(f"Error: {e}")
```

Summary of Unit V

This unit covers advanced Python concepts:

- **OOP Concepts**: Encapsulation, inheritance, polymorphism, abstraction
- **Classes and Objects**: Creating and using classes
- **Constructor**: Initializing objects with `__init__`
- **Inheritance**: Creating class hierarchies
- **Polymorphism**: Using objects of different types uniformly

- **Encapsulation**: Controlling access to attributes
- **Abstraction**: Hiding implementation details
- **Exception Handling**: try, except, else, finally
- **Custom Exceptions**: Creating application-specific exceptions

---

Practice Exercises for Unit V

1. Create a complete OOP system (e.g., banking, e-commerce, hospital management)
2. Implement inheritance hierarchies with multiple levels
3. Build polymorphic systems with abstract base classes
4. Create custom exceptions for your application
5. Write comprehensive error handling for file I/O operations
6. Combine all concepts in a real-world application project

Final Project: Complete Student Management System

```python “”” Complete Student Management System Demonstrates all OOP and exception handling concepts “””

from abc import ABC, abstractmethod from datetime import datetime import json

class CustomException(Exception): “””Base custom exception.””” pass

class InvalidStudentDataException(CustomException): “””Raised for invalid student data.””” pass

class StudentNotFoundException(CustomException): “””Raised when student is not found.””” pass

class Person: “””Base class for persons.”””

def __init__(self, person_id, name, email):
    self.person_id = person_id
    self.name = name
    self.email = email
    self.created_date = datetime.now()

def __str__(self):
    return f"{self.name} ({self.person_id})"
```

class Student(Person): “””Student class with academic features.”””

```
def __init__(self, student_id, name, email, major):
    super().__init__(student_id, name, email)
    self.major = major
    self.courses = []
    self.grades = {}
    self.gpa = 0.0

def enroll_course(self, course):
    """Enroll in a course."""
    if course not in self.courses:
        self.courses.append(course)
        return True
```

```
    return False

def add_grade(self, course, grade):
    """Add grade for a course."""
    try:
        if not (0 <= grade <= 100):
            raise InvalidStudentDataException("Grade must be between
0 and 100")
        self.grades[course] = grade
        self.calculate_gpa()
    except InvalidStudentDataException as e:
        print(f"Error: {e}")

def calculate_gpa(self):
    """Calculate GPA based on grades."""
    if not self.grades:
        self.gpa = 0.0
        return

    self.gpa = sum(self.grades.values()) / len(self.grades)

def get_summary(self):
    """Get student summary."""
    return {
        'id': self.person_id,
        'name': self.name,
        'email': self.email,
        'major': self.major,
        'courses': self.courses,
        'gpa': round(self.gpa, 2)
    }
```

class Teacher(Person): “””Teacher class.”””

```
def __init__(self, teacher_id, name, email, department):
    super().__init__(teacher_id, name, email)
    self.department = department
    self.courses_teaching = []
```

class Course: “””Course class.”””

```
def __init__(self, course_id, name, teacher, capacity):
    self.course_id = course_id
    self.name = name
    self.teacher = teacher
    self.capacity = capacity
    self.enrolled_students = []

def enroll_student(self, student):
    """Enroll student in course."""
    if len(self.enrolled_students) >= self.capacity:
        raise Exception("Course is full")
    self.enrolled_students.append(student)
```

class Academy: “””Main academy system.”””

```
def __init__(self, name):
    self.name = name
    self.students = {}
    self.teachers = {}
    self.courses = {}
```

```
def add_student(self, student):
    """Add student to academy."""
    try:
        if not isinstance(student, Student):
            raise InvalidStudentDataException("Invalid student
object")
        self.students[student.person_id] = student
        return True
    except InvalidStudentDataException as e:
        print(f"Error: {e}")
        return False

def get_student(self, student_id):
    """Get student by ID."""
    if student_id not in self.students:
        raise StudentNotFoundException(f"Student {student_id} not
found")
    return self.students[student_id]

def display_student_summary(self, student_id):
    """Display student summary."""
    try:
        student = self.get_student(student_id)
        print(f"\n{'='*50}")
        print(f"STUDENT SUMMARY - {student.name}")
        print(f"{'='*50}")
        summary = student.get_summary()
        for key, value in summary.items():
            print(f"{key.capitalize()}: {value}")
        print(f"{'='*50}")
    except StudentNotFoundException as e:
        print(f"Error: {e}")
```

Usage demonstration

if **name** == “**main**”: # Create academy academy = Academy(“Tech Academy”)

```
# Create students
try:
    student1 = Student("S001", "Alice Johnson", "alice@academy.edu",
"Computer Science")
    student2 = Student("S002", "Bob Smith", "bob@academy.edu",
"Information Technology")

    academy.add_student(student1)
    academy.add_student(student2)

    # Add grades
    student1.add_grade("Python Programming", 92)
    student1.add_grade("Data Structures", 88)

    student2.add_grade("Python Programming", 85)
    student2.add_grade("Data Structures", 90)

    # Display summaries
    academy.display_student_summary("S001")
```

```
    academy.display_student_summary("S002")

    # Attempt to get non-existent student
    academy.display_student_summary("S003")

except Exception as e:
    print(f"Error: {e}")
```

print(“” + “=”70) print(“COMPREHENSIVE PYTHON PROGRAMMING GUIDE - ALL UNITS COMPLETED”) print(“=”70) print(“covered:”) print(” Unit I: Introduction to Python”) print(” Unit II: Data Types and Operators”) print(” Unit III: Control Flow and Functions”) print(” Unit IV: Data Structures and File Handling”) print(” Unit V: Object-Oriented Programming and Exception Handling”) print(“coverage: All fundamental and advanced Python concepts”) print(“=”*70)