Complete Object-Oriented Programming (OOP) Guide in Python

Table of Contents

1. Introduction to OOP
2. The Four Pillars of OOP
    - Encapsulation
    - Inheritance
    - Polymorphism
    - Abstraction
3. Complete Library Management System
4. Advanced OOP Concepts

Introduction to OOP {#introduction}

Object-Oriented Programming (OOP) is a programming paradigm that organizes code into objects, which are instances of classes. OOP helps create modular, reusable, and maintainable code by modeling real-world entities and their relationships.

Key Benefits of OOP:

- **Modularity**: Code is organized into discrete units (objects)
- **Reusability**: Objects and classes can be reused across projects
- **Scalability**: Easy to extend and modify existing code
- **Security**: Data hiding and controlled access through encapsulation
- **Real-world modeling**: Natural way to represent real-world problems

The Four Pillars of OOP {#four-pillars}

1. Encapsulation {#encapsulation}

Definition: Encapsulation is the bundling of data (attributes) and methods that operate on that data within a single unit (class), while restricting direct access to some components.

Key Concepts:

- **Data Hiding**: Internal details are hidden from outside access
- **Access Modifiers**: Control visibility of attributes and methods
- **Getters and Setters**: Controlled access to private data

Python Implementation:

```
python
```

```python
class BankAccount:
    def __init__(self, account_number, initial_balance=0):
        # Private attributes (convention: prefix with underscore)
        self._account_number = account_number
        self._balance = initial_balance
        self._transaction_history = []

    # Getter method
    @property
    def balance(self):
        """Get the current balance"""
        return self._balance

    # Setter method with validation
    @balance.setter
    def balance(self, amount):
        """Set balance with validation"""
        if amount < 0:
            raise ValueError("Balance cannot be negative")
        self._balance = amount

    # Public method
    def deposit(self, amount):
        """Deposit money to account"""
        if amount <= 0:
            raise ValueError("Deposit amount must be positive")

        self._balance += amount
        self._record_transaction(f"Deposited ${amount}")
        return self._balance

    # Private method (convention: prefix with underscore)
    def _record_transaction(self, transaction):
        """Internal method to record transactions"""
        from datetime import datetime
        self._transaction_history.append({
            'timestamp': datetime.now(),
            'description': transaction,
            'balance': self._balance
        })

    def get_transaction_history(self):
        """Public method to access transaction history"""
        # Return a copy to prevent external modification
        return self._transaction_history.copy()
```

```python
# Usage example
account = BankAccount("12345", 1000)
print(f"Initial balance: ${account.balance}")  # Using getter

account.deposit(500)
print(f"After deposit: ${account.balance}")

# This would raise an error:
# account.balance = -100  # Setter validation prevents negative balance
```

2. Inheritance {#inheritance}

Definition: Inheritance allows a class (child/derived) to inherit attributes and methods from another class (parent/base), promoting code reuse and establishing a hierarchical relationship.

Types of Inheritance:

- **Single Inheritance**: Child inherits from one parent
- **Multiple Inheritance**: Child inherits from multiple parents
- **Multilevel Inheritance**: Chain of inheritance
- **Hierarchical Inheritance**: Multiple children from one parent

Python Implementation:

python

```python
# Base class (Parent)
class Vehicle:
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
        self._mileage = 0

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

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

    def get_info(self):
        return f"{self.year} {self.brand} {self.model}"

# Single Inheritance
class Car(Vehicle):
    def __init__(self, brand, model, year, num_doors):
        super().__init__(brand, model, year)  # Call parent constructor
        self.num_doors = num_doors

    def honk(self):
        return "Beep beep!"

    # Method overriding
    def get_info(self):
        parent_info = super().get_info()
        return f"{parent_info} - {self.num_doors} door car"

# Multiple Inheritance
class Electric:
    def __init__(self, battery_capacity):
        self.battery_capacity = battery_capacity
        self.battery_level = 100

    def charge(self):
        self.battery_level = 100
        return "Battery charged to 100%"

    def get_range(self):
        return f"Range: {self.battery_capacity * 3} miles"

class ElectricCar(Car, Electric):
    def __init__(self, brand, model, year, num_doors, battery_capacity):
        Car.__init__(self, brand, model, year, num_doors)
        Electric.__init__(self, battery_capacity)

    # Override parent method
    def start_engine(self):
        if self.battery_level > 10:
            return f"{self.brand} {self.model} electric motor activated silently"
        return "Battery too low to start"

# Usage example
regular_car = Car("Toyota", "Camry", 2023, 4)
print(regular_car.get_info())
print(regular_car.honk())

electric_car = ElectricCar("Tesla", "Model 3", 2023, 4, 75)
print(electric_car.start_engine())
print(electric_car.get_range())
print(electric_car.charge())
```

3. Polymorphism {#polymorphism}

Definition: Polymorphism allows objects of different types to be treated uniformly through a common interface, enabling the same method name to behave differently for different classes.

Types of Polymorphism:

- **Method Overriding**: Redefining parent methods in child classes
- **Method Overloading**: Multiple methods with same name (limited in Python)
- **Duck Typing**: "If it walks like a duck and quacks like a duck..."
- **Operator Overloading**: Customizing behavior of operators

Python Implementation:

```
python
```

```python
from abc import ABC, abstractmethod
import math

# Polymorphism through inheritance
class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

    def describe(self):
        return f"This shape has area: {self.area():.2f} and perimeter: {self.perimeter():.2f}"

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

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

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

class Triangle(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 math.sqrt(s * (s - self.a) * (s - self.b) * (s - self.c))

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

# Duck Typing Example
class Dog:
    def make_sound(self):
        return "Woof!"

class Cat:
    def make_sound(self):
        return "Meow!"

class Cow:
    def make_sound(self):
        return "Moo!"

# Operator Overloading Example
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):
        """Overload + operator"""
        return Vector(self.x + other.x, self.y + other.y)

    def __mul__(self, scalar):
        """Overload * operator"""
        return Vector(self.x * scalar, self.y * scalar)

    def __str__(self):
        """String representation"""
        return f"Vector({self.x}, {self.y})"

    def __eq__(self, other):
        """Overload == operator"""
        return self.x == other.x and self.y == other.y

# Usage examples
# Polymorphism with shapes
shapes = [
    Rectangle(5, 3),
    Circle(4),
    Triangle(3, 4, 5)
]
```

```
for shape in shapes:
    print(f"{shape.__class__.__name__}: {shape.describe()}")

# Duck typing
animals = [Dog(), Cat(), Cow()]
for animal in animals:
    print(f"{animal.__class__.__name__} says: {animal.make_sound()}")

# Operator overloading
v1 = Vector(2, 3)
v2 = Vector(4, 5)
v3 = v1 + v2
print(f"v1 + v2 = {v3}")
print(f"v1 * 3 = {v1 * 3}")
```

4. Abstraction {#abstraction}

Definition: Abstraction hides complex implementation details and shows only essential features of an object, defining a contract that concrete classes must follow.

Key Concepts:

- **Abstract Classes**: Cannot be instantiated, serve as templates
- **Abstract Methods**: Must be implemented by concrete subclasses
- **Interfaces**: Define method signatures without implementation

Python Implementation:

python

```python
from abc import ABC, abstractmethod
from datetime import datetime
from typing import List, Optional

# Abstract base class for payment processing
class PaymentProcessor(ABC):
    """Abstract base class for payment processors"""

    def __init__(self):
        self.transactions = []

    @abstractmethod
    def validate_payment_details(self, **kwargs) -> bool:
        """Validate payment details specific to payment method"""
        pass

    @abstractmethod
    def process_payment(self, amount: float, **kwargs) -> dict:
        """Process the actual payment"""
        pass

    @abstractmethod
    def refund_payment(self, transaction_id: str, amount: float) -> dict:
        """Process refund for a payment"""
        pass

    def record_transaction(self, transaction: dict) -> None:
        """Common method to record transactions"""
        transaction['timestamp'] = datetime.now()
        self.transactions.append(transaction)

    def get_transaction_history(self) -> List[dict]:
        """Get all transactions"""
        return self.transactions.copy()

# Concrete implementation for Credit Card
class CreditCardProcessor(PaymentProcessor):
    def __init__(self, merchant_id: str):
        super().__init__()
        self.merchant_id = merchant_id

    def validate_payment_details(self, card_number: str, cvv: str, expiry: str) -> bool:
        """Validate credit card details"""
        # Simplified validation
        if len(card_number) != 16 or not card_number.isdigit():
            return False
        if len(cvv) != 3 or not cvv.isdigit():
            return False
        # Add more validation logic
        return True

    def process_payment(self, amount: float, card_number: str, cvv: str, expiry: str) -> dict:
        """Process credit card payment"""
        if not self.validate_payment_details(card_number, cvv, expiry):
            return {'status': 'failed', 'error': 'Invalid card details'}

        # Simulate payment processing
        transaction = {
            'id': f"CC_{datetime.now().timestamp()}",
            'type': 'credit_card',
            'amount': amount,
            'card_last_four': card_number[-4:],
            'status': 'success'
        }

        self.record_transaction(transaction)
        return transaction

    def refund_payment(self, transaction_id: str, amount: float) -> dict:
        """Process credit card refund"""
        # Find original transaction
        original = next((t for t in self.transactions if t['id'] == transaction_id), None)

        if not original:
            return {'status': 'failed', 'error': 'Transaction not found'}

        if amount > original['amount']:
            return {'status': 'failed', 'error': 'Refund exceeds original amount'}

        refund = {
            'id': f"REF_{transaction_id}",
            'type': 'refund',
            'original_transaction': transaction_id,
            'amount': amount,
            'status': 'success'
        }

        self.record_transaction(refund)
        return refund

# Concrete implementation for PayPal
class PayPalProcessor(PaymentProcessor):
    def __init__(self, api_key: str):
        super().__init__()
        self.api_key = api_key

    def validate_payment_details(self, email: str, password: str) -> bool:
        """Validate PayPal credentials"""
        # Simplified validation
        return '@' in email and len(password) >= 8

    def process_payment(self, amount: float, email: str, password: str) -> dict:
        """Process PayPal payment"""
        if not self.validate_payment_details(email, password):
            return {'status': 'failed', 'error': 'Invalid PayPal credentials'}

        transaction = {
            'id': f"PP_{datetime.now().timestamp()}",
            'type': 'paypal',
            'amount': amount,
            'email': email,
            'status': 'success'
        }

        self.record_transaction(transaction)
        return transaction

    def refund_payment(self, transaction_id: str, amount: float) -> dict:
        """Process PayPal refund"""
        # Similar implementation to credit card
        original = next((t for t in self.transactions if t['id'] == transaction_id), None)

        if not original:
            return {'status': 'failed', 'error': 'Transaction not found'}

        refund = {
            'id': f"REF_{transaction_id}",
            'type': 'paypal_refund',
            'original_transaction': transaction_id,
            'amount': amount,
            'status': 'success'
        }

        self.record_transaction(refund)
        return refund

# Usage example
def process_customer_payment(processor: PaymentProcessor, amount: float, **payment_details) ->
    """Function that works with any payment processor"""
    return processor.process_payment(amount, **payment_details)
```

```
# Create processors
cc_processor = CreditCardProcessor("MERCHANT123")
pp_processor = PayPalProcessor("API_KEY_456")

# Process payments using abstraction
cc_payment = process_customer_payment(
    cc_processor,
    100.50,
    card_number="1234567812345678",
    cvv="123",
    expiry="12/25"
)
print(f"Credit Card Payment: {cc_payment}")

pp_payment = process_customer_payment(
    pp_processor,
    75.00,
    email="user@example.com",
    password="securepass123"
)
print(f"PayPal Payment: {pp_payment}")
```

Complete Library Management System {#library-system}

Now let's implement a comprehensive Library Management System that demonstrates all four OOP principles:

python

```python
from abc import ABC, abstractmethod
from datetime import datetime, timedelta
from typing import List, Optional, Dict
from enum import Enum
import json

# Enums for better type safety
class BookStatus(Enum):
    AVAILABLE = "available"
    BORROWED = "borrowed"
    RESERVED = "reserved"
    MAINTENANCE = "maintenance"

class MembershipType(Enum):
    BASIC = "basic"
    PREMIUM = "premium"
    STUDENT = "student"

# Abstract base class demonstrating Abstraction
class LibraryItem(ABC):
    """Abstract base class for all library items"""

    def __init__(self, item_id: str, title: str, author: str, publication_year: int):
        self._item_id = item_id
        self._title = title
        self._author = author
        self._publication_year = publication_year
        self._status = BookStatus.AVAILABLE
        self._location = None
        self._borrowing_history = []

    @property
    def item_id(self):
        return self._item_id

    @property
    def title(self):
        return self._title

    @property
    def status(self):
        return self._status

    @status.setter
    def status(self, new_status: BookStatus):
        self._status = new_status
    @abstractmethod
    def get_borrowing_period(self) -> int:
        """Return the number of days this item can be borrowed"""
        pass

    @abstractmethod
    def calculate_late_fee(self, days_late: int) -> float:
        """Calculate late fee based on item type"""
        pass

    def add_to_history(self, member_id: str, action: str):
        """Record borrowing history"""
        self._borrowing_history.append({
            'member_id': member_id,
            'action': action,
            'timestamp': datetime.now()
        })

# Concrete implementations demonstrating Inheritance
class Book(LibraryItem):
    """Regular book implementation"""

    def __init__(self, item_id: str, title: str, author: str,
                 publication_year: int, isbn: str, genre: str):
        super().__init__(item_id, title, author, publication_year)
        self.isbn = isbn
        self.genre = genre
        self._pages = 0

    def get_borrowing_period(self) -> int:
        return 14  # 2 weeks for regular books

    def calculate_late_fee(self, days_late: int) -> float:
        return days_late * 0.50  # $0.50 per day

class ReferenceBook(Book):
    """Reference books - special type that cannot be borrowed"""

    def __init__(self, item_id: str, title: str, author: str,
                 publication_year: int, isbn: str, subject: str):
        super().__init__(item_id, title, author, publication_year, isbn, "Reference")
        self.subject = subject
        self._status = BookStatus.AVAILABLE  # Always available for in-library use

    def get_borrowing_period(self) -> int:
        return 0  # Cannot be borrowed
    def calculate_late_fee(self, days_late: int) -> float:
        return 0  # No late fees as they can't be borrowed

class DigitalMedia(LibraryItem):
    """Digital media items like DVDs, CDs"""

    def __init__(self, item_id: str, title: str, author: str,
                 publication_year: int, media_type: str, duration_minutes: int):
        super().__init__(item_id, title, author, publication_year)
        self.media_type = media_type
        self.duration_minutes = duration_minutes

    def get_borrowing_period(self) -> int:
        return 7  # 1 week for digital media

    def calculate_late_fee(self, days_late: int) -> float:
        return days_late * 1.00  # $1.00 per day

# Member classes demonstrating Encapsulation and Polymorphism
class Member:
    """Library member with encapsulated data"""

    def __init__(self, member_id: str, name: str, email: str,
                 membership_type: MembershipType):
        self._member_id = member_id
        self._name = name
        self._email = email
        self._membership_type = membership_type
        self._borrowed_items: List[str] = []
        self._fines = 0.0
        self._reservation_list: List[str] = []
        self._join_date = datetime.now()

    @property
    def member_id(self):
        return self._member_id

    @property
    def name(self):
        return self._name

    @property
    def borrowed_items(self):
        return self._borrowed_items.copy()

    @property
    def fines(self):
        return self._fines

    def add_fine(self, amount: float):
        """Add fine to member account"""
        self._fines += amount

    def pay_fine(self, amount: float):
        """Pay off fines"""
        if amount > self._fines:
            raise ValueError("Payment exceeds fine amount")
        self._fines -= amount

    def can_borrow(self) -> bool:
        """Check if member can borrow items"""
        max_items = self._get_borrowing_limit()
        return len(self._borrowed_items) < max_items and self._fines == 0

    def _get_borrowing_limit(self) -> int:
        """Private method to get borrowing limit based on membership"""
        limits = {
            MembershipType.BASIC: 3,
            MembershipType.PREMIUM: 10,
            MembershipType.STUDENT: 5
        }
        return limits.get(self._membership_type, 3)

    def borrow_item(self, item_id: str):
        """Add item to borrowed list"""
        if not self.can_borrow():
            raise Exception("Cannot borrow more items or has outstanding fines")
        self._borrowed_items.append(item_id)

    def return_item(self, item_id: str):
        """Remove item from borrowed list"""
        if item_id in self._borrowed_items:
            self._borrowed_items.remove(item_id)

# Transaction class for record keeping
class Transaction:
    """Record of borrowing/returning transactions"""

    def __init__(self, transaction_id: str, member_id: str,
                 item_id: str, transaction_type: str):
        self.transaction_id = transaction_id
        self.member_id = member_id
        self.item_id = item_id
        self.transaction_type = transaction_type
        self.transaction_date = datetime.now()
        self.due_date = None
        self.return_date = None

    def set_due_date(self, days: int):
        """Set due date based on borrowing period"""
        self.due_date = self.transaction_date + timedelta(days=days)

    def complete_return(self):
        """Mark transaction as returned"""
        self.return_date = datetime.now()

    def calculate_days_late(self) -> int:
        """Calculate how many days late the return is"""
        if not self.return_date or not self.due_date:
            return 0

        if self.return_date > self.due_date:
            return (self.return_date - self.due_date).days
        return 0

# Main Library System demonstrating all OOP principles
class LibraryManagementSystem:
    """Main library system coordinating all operations"""

    def __init__(self, library_name: str):
        self.library_name = library_name
        self._catalog: Dict[str, LibraryItem] = {}
        self._members: Dict[str, Member] = {}
        self._transactions: Dict[str, Transaction] = {}
        self._next_transaction_id = 1

    def add_item(self, item: LibraryItem):
        """Add item to library catalog"""
        self._catalog[item.item_id] = item

    def add_member(self, member: Member):
        """Register new member"""
        self._members[member.member_id] = member

    def search_items(self, query: str) -> List[LibraryItem]:
        """Search for items by title or author"""
        results = []
        query_lower = query.lower()

        for item in self._catalog.values():
            if (query_lower in item.title.lower() or
                query_lower in item._author.lower()):
                results.append(item)

        return results

    def borrow_item(self, member_id: str, item_id: str) -> Transaction:
        """Process item borrowing"""
        # Validate member and item
        if member_id not in self._members:
            raise ValueError("Member not found")

        if item_id not in self._catalog:
            raise ValueError("Item not found")

        member = self._members[member_id]
        item = self._catalog[item_id]

        # Check if item is available
        if item.status != BookStatus.AVAILABLE:
            raise Exception(f"Item is {item.status.value}")

        # Check if member can borrow
        if not member.can_borrow():
            raise Exception("Member cannot borrow items")

        # Check if it's a reference book
        if isinstance(item, ReferenceBook):
            raise Exception("Reference books cannot be borrowed")

        # Create transaction
        transaction_id = f"TRX{self._next_transaction_id:06d}"
        self._next_transaction_id += 1

        transaction = Transaction(transaction_id, member_id, item_id, "BORROW")
        transaction.set_due_date(item.get_borrowing_period())

        # Update records
        item.status = BookStatus.BORROWED
        item.add_to_history(member_id, "BORROWED")
        member.borrow_item(item_id)
        self._transactions[transaction_id] = transaction

        return transaction

    def return_item(self, member_id: str, item_id: str) -> Dict[str, any]:
        """Process item return"""
```

```python
        # Find active transaction
        active_transaction = None
        for transaction in self._transactions.values():
            if (transaction.member_id == member_id and
                transaction.item_id == item_id and
                transaction.return_date is None):
                active_transaction = transaction
                break

        if not active_transaction:
            raise ValueError("No active borrowing found")

        member = self._members[member_id]
        item = self._catalog[item_id]

        # Complete return
        active_transaction.complete_return()
        item.status = BookStatus.AVAILABLE
        item.add_to_history(member_id, "RETURNED")
        member.return_item(item_id)

        # Calculate late fees
        days_late = active_transaction.calculate_days_late()
        late_fee = 0

        if days_late > 0:
            late_fee = item.calculate_late_fee(days_late)
            member.add_fine(late_fee)

        return {
            'transaction_id': active_transaction.transaction_id,
            'days_late': days_late,
            'late_fee': late_fee
        }

    def get_member_history(self, member_id: str) -> List[Transaction]:
        """Get borrowing history for a member"""
        return [t for t in self._transactions.values()
                if t.member_id == member_id]

    def get_available_items(self) -> List[LibraryItem]:
        """Get all available items"""
        return [item for item in self._catalog.values()
                if item.status == BookStatus.AVAILABLE]

    def generate_overdue_report(self) -> List[Dict]:
        """Generate report of overdue items"""
        overdue_items = []
        current_date = datetime.now()

        for transaction in self._transactions.values():
            if (transaction.return_date is None and
                transaction.due_date and
                current_date > transaction.due_date):

                days_overdue = (current_date - transaction.due_date).days
                item = self._catalog[transaction.item_id]
                member = self._members[transaction.member_id]

                overdue_items.append({
                    'member_name': member.name,
                    'item_title': item.title,
                    'days_overdue': days_overdue,
                    'estimated_fine': item.calculate_late_fee(days_overdue)
                })

        return overdue_items

# Example usage of the Library Management System
def demo_library_system():
    # Create library
    library = LibraryManagementSystem("City Central Library")

    # Add some books
    book1 = Book("B001", "Python Programming", "John Doe", 2023, "978-1234567890", "Technology"
    book2 = Book("B002", "Data Structures", "Jane Smith", 2022, "978-0987654321", "Computer Sci
    ref_book = ReferenceBook("R001", "Encyclopedia Britannica", "Various", 2023, "978-111111111
    dvd = DigitalMedia("D001", "Learn Python", "Tech Academy", 2023, "DVD", 120)

    library.add_item(book1)
    library.add_item(book2)
    library.add_item(ref_book)
    library.add_item(dvd)

    # Add members
    member1 = Member("M001", "Alice Johnson", "alice@email.com", MembershipType.PREMIUM)
    member2 = Member("M002", "Bob Smith", "bob@email.com", MembershipType.STUDENT)

    library.add_member(member1)
    library.add_member(member2)

    # Demonstrate borrowing
    try:
        # Successful borrowing
```

```python
        transaction1 = library.borrow_item("M001", "B001")
        print(f"Book borrowed successfully. Due date: {transaction1.due_date}")

        # Try to borrow reference book (should fail)
        try:
            library.borrow_item("M001", "R001")
        except Exception as e:
            print(f"Cannot borrow reference book: {e}")

        # Return book
        return_info = library.return_item("M001", "B001")
        print(f"Book returned. Late fee: ${return_info['late_fee']}")

        # Search for items
        search_results = library.search_items("Python")
        print(f"\nSearch results for 'Python':")
        for item in search_results:
            print(f"- {item.title} by {item._author} (Status: {item.status.value})")

        # Get available items
        available = library.get_available_items()
        print(f"\nAvailable items: {len(available)}")

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

# Run the demo
if __name__ == "__main__":
    demo_library_system()
```

Advanced OOP Concepts {#advanced-concepts}

Design Patterns

Here are some common design patterns implemented in Python:

python

```python
# Singleton Pattern
class DatabaseConnection:
    """Singleton pattern ensures only one instance exists"""
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(DatabaseConnection, cls).__new__(cls)
            cls._instance.connection = None
        return cls._instance

    def connect(self, connection_string):
        if self.connection is None:
            self.connection = f"Connected to: {connection_string}"
        return self.connection

# Factory Pattern
class NotificationFactory:
    """Factory pattern for creating different notification types"""

    @staticmethod
    def create_notification(notification_type: str, message: str):
        if notification_type == "email":
            return EmailNotification(message)
        elif notification_type == "sms":
            return SMSNotification(message)
        elif notification_type == "push":
            return PushNotification(message)
        else:
            raise ValueError(f"Unknown notification type: {notification_type}")

class EmailNotification:
    def __init__(self, message):
        self.message = message

    def send(self):
        return f"Email sent: {self.message}"

class SMSNotification:
    def __init__(self, message):
        self.message = message

    def send(self):
        return f"SMS sent: {self.message}"

class PushNotification:
    def __init__(self, message):
        self.message = message

    def send(self):
        return f"Push notification sent: {self.message}"

# Observer Pattern
class Subject:
    """Observable subject"""
    def __init__(self):
        self._observers = []
        self._state = None

    def attach(self, observer):
        self._observers.append(observer)

    def detach(self, observer):
        self._observers.remove(observer)

    def notify(self):
        for observer in self._observers:
            observer.update(self._state)

    def set_state(self, state):
        self._state = state
        self.notify()

class Observer(ABC):
    @abstractmethod
    def update(self, state):
        pass

class ConcreteObserver(Observer):
    def __init__(self, name):
        self.name = name

    def update(self, state):
        print(f"{self.name} received update: {state}")
```

Method Resolution Order (MRO)

Python uses C3 linearization for method resolution in multiple inheritance:

python

```python
class A:
    def method(self):
        print("Method from A")

class B(A):
    def method(self):
        print("Method from B")
        super().method()

class C(A):
    def method(self):
        print("Method from C")
        super().method()

class D(B, C):
    def method(self):
        print("Method from D")
        super().method()

# Check MRO
print(D.__mro__)
# Output: (<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A

# Call method
d = D()
d.method()
# Output:
# Method from D
# Method from B
# Method from C
# Method from A
```

Property Decorators and Descriptors

```
python
```

```python
class Temperature:
    """Example of property decorators"""
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        if value < -273.15:
            raise ValueError("Temperature below absolute zero is not possible")
        self._celsius = value

    @property
    def fahrenheit(self):
        return self._celsius * 9/5 + 32

    @fahrenheit.setter
    def fahrenheit(self, value):
        self.celsius = (value - 32) * 5/9

# Descriptor example
class ValidatedAttribute:
    """A descriptor that validates values"""
    def __init__(self, name, validator):
        self.name = name
        self.validator = validator

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        return obj.__dict__.get(self.name)

    def __set__(self, obj, value):
        if not self.validator(value):
            raise ValueError(f"Invalid value for {self.name}: {value}")
        obj.__dict__[self.name] = value

class Person:
    age = ValidatedAttribute("age", lambda x: 0 <= x <= 150)
    name = ValidatedAttribute("name", lambda x: isinstance(x, str) and len(x) > 0)

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

Best Practices

1. **Follow SOLID Principles**:
    - Single Responsibility Principle
    - Open/Closed Principle
    - Liskov Substitution Principle
    - Interface Segregation Principle
    - Dependency Inversion Principle
2. **Use meaningful names** for classes, methods, and variables
3. **Keep inheritance hierarchies shallow** (prefer composition over inheritance when appropriate)
4. **Document your code** with docstrings and type hints
5. **Use abstract base classes** to define interfaces
6. **Encapsulate what varies** and keep it separate from what stays the same
7. **Program to interfaces**, not implementations
8. **Favor object composition** over class inheritance when it makes sense

This comprehensive guide covers all major OOP concepts in Python with practical examples. The Library Management System demonstrates how these concepts work together in a real-world application.