Object-Oriented Programming

1. Classes and Objects

Object-oriented programming organizes code using classes and objects. A class is a blueprint, and an object is an instance of a class.

```
class Student:
    """Simple student class"""

    # Class variable (shared by all instances)
    school = "IUGB"

    def __init__(self, name, age):
        """Constructor - initializes instance"""
        self.name = name      # Instance variables
        self.age = age
        self.courses = []

    def add_course(self, course):
        """Add a course"""
        self.courses.append(course)

    def get_info(self):
        """Return student info"""
        return f"{self.name} ({self.age}) - Courses: {len(self.courses)}"

# Create objects
alice = Student("Alice", 20)
bob = Student("Bob", 21)
print(f"Created: {alice.get_info()}")
print(f"Created: {bob.get_info()}")
```

```
Created: Alice (20) - Courses: 0
Created: Bob (21) - Courses: 0
```

2. Instance vs Class Variables

Instance variables are unique to each object, while class variables are shared across all instances.

```
alice = Student("Alice", 20)
bob = Student("Bob", 21)

# Instance variables - unique to each object
alice.add_course("AI")
alice.add_course("ML")
bob.add_course("DL")

print(f"{alice.get_info()}")
print(f"{bob.get_info()}")

# Class variable - shared by all
print(f"Alice's school: {alice.school}")
print(f"Bob's school: {bob.school}")
print(f"Class variable: {Student.school}")
```

```
Alice (20) - Courses: 2
Bob (21) - Courses: 1
Alice's school: IUGB
Bob's school: IUGB
Class variable: IUGB
```

3. Special Methods

Special methods (dunder methods) provide special functionality. __init__ is the constructor, __str__ defines string representation.

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

    def __str__(self):
        """String representation"""
        return f"Student: {self.name}, Age: {self.age}"

    def __repr__(self):
        """Developer-friendly representation"""
        return f"Student('{self.name}', {self.age})"

alice = Student("Alice", 20)
print(str(alice))   # Uses __str__
print(repr(alice))  # Uses __repr__
```

```
Student: Alice, Age: 20
Student('Alice', 20)
```

4. Inheritance

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

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
        self.courses = []

    def add_course(self, course):
        self.courses.append(course)

    def get_info(self):
        return f"{self.name} ({self.age}) - Courses: {len(self.courses)}"

class GraduateStudent(Student):
    """Graduate student inherits from Student"""

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

    def get_info(self):
        """Override parent method"""
        base_info = super().get_info()
        return f"{base_info}, Thesis: {self.thesis_topic}"

# Create graduate student
```

```
charlie = GraduateStudent("Charlie", 24, "Deep Learning")
charlie.add_course("Advanced AI")
print(charlie.get_info())
```

```
Charlie (24) - Courses: 1, Thesis: Deep Learning
```

5. Encapsulation

Encapsulation restricts direct access to some attributes. Use underscore prefix for "private" attributes.

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self._balance = balance  # "Private" attribute

    def deposit(self, amount):
        if amount > 0:
            self._balance += amount
            return True
        return False

    def withdraw(self, amount):
        if 0 < amount <= self._balance:
            self._balance -= amount
            return True
        return False

    def get_balance(self):
        return self._balance

account = BankAccount("Alice", 1000)
account.deposit(500)
account.withdraw(200)
print(f"Balance: ${account.get_balance()}")
```

```
Balance: $1300
```

6. Properties

Properties provide controlled access to attributes using getter and setter methods.

```
class Temperature:
    def __init__(self, celsius=0):
        self._celsius = celsius

    @property
    def celsius(self):
        """Getter for celsius"""
        return self._celsius

    @celsius.setter
    def celsius(self, value):
        """Setter for celsius"""
        if value < -273.15:
            raise ValueError("Temperature below absolute zero!")
        self._celsius = value

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

```
temp = Temperature(25)
print(f"Celsius: {temp.celsius}°C")
print(f"Fahrenheit: {temp.fahrenheit}°F")

temp.celsius = 30
print(f"New temp: {temp.celsius}°C = {temp.fahrenheit}°F")
```

```
Celsius: 25°C
Fahrenheit: 77.0°F
New temp: 30°C = 86.0°F
```

Key Takeaways

- Classes are blueprints, objects are instances
- __init__ is the constructor
- Instance variables vs class variables
- Inheritance allows code reuse
- super() calls parent class methods
- Encapsulation with underscore prefix
- Properties for controlled attribute access