Python Programming Bootcamp
Week 6: Advanced Object-Oriented Programming

Hilal Khan

Center for Artificial Intelligence & Emerging Technologies

Learning Objectives

By the end of this week, you will be able to:

- Master the four pillars of OOP in depth
- Implement encapsulation with private and protected members
- Create class hierarchies using inheritance
- Apply polymorphism through method overriding
- Use abstraction with abstract base classes
- Implement magic methods for operator overloading
- Design complex class relationships
- Create advanced UML diagrams
- Build production-ready OOP applications

Week 6 Overview

What We'll Cover Today:

- Encapsulation: Data Hiding and Access Control
- Inheritance: Building Class Hierarchies
- Polymorphism: Flexible Code Design
- Abstraction: Hiding Complexity
- Magic Methods: Python's Special Methods
- Advanced UML Diagrams
- Aggregation and Composition
- Practical Design Patterns

Quick Recap: Previous Weeks

Week 1:

- Python basics, variables, data types, operators
- Input/output, conditionals, loops

Week 2:

- Data structures: Lists, strings, tuples, dictionaries, sets

Week 3:

- Functions, parameters, return values
- Modules and imports

Week 4:

- Error handling with try-except
- File I/O operations
- Working with CSV and JSON files

Quick Recap: Week 5

Last Week - OOP Fundamentals:

- Classes and Objects
- The `self` keyword
- Instance vs Class vs Static methods
- Instance vs Class attributes
- Basic Association
- UML class diagrams
- Overview of four OOP properties

This week: Deep dive into the four OOP properties and advanced concepts!

Encapsulation

Encapsulation: The First Pillar

What is Encapsulation?

- **Bundling** data and methods together in a class
- **Restricting** direct access to some components
- **Protecting** internal state from outside interference
- **Providing** controlled access through public methods

Why Encapsulation?

- Data protection and validation
- Flexibility to change internal implementation
- Reduced complexity for users
- Better maintenance and debugging

Access Modifiers in Python i

Python's Naming Conventions

- **Public**: `name` - accessible from anywhere
- **Protected**: `_name` - internal use (convention only)
- **Private**: `__name` - name mangling applied

```
class BankAccount:
    def __init__(self, owner, balance):
        self.owner = owner                  # Public attribute
        self._account_number = "12345"  # Protected (
    convention)
        self.__balance = balance    # Private
    attribute

    def deposit(self, amount):
```

Access Modifiers in Python ii

```
        if amount > 0:
            self.__balance += amount
            return True
        return False

    def get_balance(self):              # Public getter
    method
        return self.__balance

# Usage
account = BankAccount("Alice", 1000)
print(account.owner)                    # Works - public
print(account._account_number)          # Works but
    discouraged
# print(account.__balance)              # AttributeError -
    private
```

Access Modifiers in Python iii

```
print(account.get_balance())          # 1000 - proper
    access
```

Access Modifiers in Python iv

Name Mangling for Private Attributes:

```
class Example:
    def __init__(self):
        self.__private = "I'm private"

obj = Example()
# print(obj.__private)  # AttributeError

# Python mangles the name to _ClassName__attribute
print(obj._Example__private)  # Works but don't do
    this!

# This is how Python enforces privacy
print(dir(obj))
# Shows: ['_Example__private', ...]
```

Access Modifiers in Python v

Important

Python's privacy is more about convention than enforcement. Double underscore is for name mangling to avoid conflicts in inheritance, not true privacy.

Getters and Setters i

Controlling access to private attributes:

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

    # Getter methods
    def get_name(self):
        return self.__name

    def get_age(self):
        return self.__age

    # Setter methods with validation
    def set_name(self, name):
```

Getters and Setters ii

```
        if isinstance(name, str) and len(name) > 0:
            self.__name = name
        else:
            raise ValueError("Name must be a non-empty
    string")

    def set_age(self, age):
        if isinstance(age, int) and 0 <= age <= 150:
            self.__age = age
        else:
            raise ValueError("Age must be between 0
    and 150")

# Usage
student = Student("Alice", 20)
print(student.get_name())   # "Alice"
```

Getters and Setters iii

```
student.set_age(21)           # Valid
# student.set_age(-5)         # ValueError
# student.set_age("twenty")   # ValueError
```

Getters and Setters iv

Property Decorator (Pythonic Way):

```
class Employee:
    def __init__(self, name, salary):
        self.__name = name
        self.__salary = salary

    @property
    def name(self):
        """Getter for name"""
        return self.__name

    @property
    def salary(self):
        """Getter for salary"""
        return self.__salary
```

Getters and Setters v

```
    @salary.setter
    def salary(self, value):
        """Setter with validation"""
        if value < 0:
            raise ValueError("Salary cannot be
    negative")
        self.__salary = value

# Usage - looks like attribute access!
emp = Employee("Bob", 50000)
print(emp.name)          # "Bob"
print(emp.salary)        # 50000

emp.salary = 55000       # Uses setter
# emp.salary = -1000     # ValueError
# emp.name = "Alice"     # AttributeError - no setter
```

Getters and Setters vi

Encapsulation Example: Smart Temperature i

```
class Temperature:
    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!")
        self.__celsius = value

    @property
```

Encapsulation Example: Smart Temperature ii

```
    def fahrenheit(self):
        return (self.__celsius * 9/5) + 32

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

    @property
    def kelvin(self):
        return self.__celsius + 273.15

    @kelvin.setter
    def kelvin(self, value):
        self.celsius = value - 273.15  # Uses celsius
    setter
```

Encapsulation Example: Smart Temperature iii

```
# Usage
temp = Temperature(25)
print(f"Celsius: {temp.celsius}")        # 25
print(f"Fahrenheit: {temp.fahrenheit}") # 77.0
print(f"Kelvin: {temp.kelvin}")          # 298.15

temp.fahrenheit = 100
print(f"Celsius: {temp.celsius}")        # 37.77...

temp.kelvin = 0
print(f"Celsius: {temp.celsius}")        # -273.15

# temp.celsius = -300  # ValueError - below absolute
    zero!
```

Encapsulation Example: Smart Temperature iv

Benefits Demonstrated

- Private data (`__celsius`)
- Validation in setters
- Computed properties (fahrenheit, kelvin)
- Clean, attribute-like syntax

Inheritance

Inheritance: The Second Pillar

What is Inheritance?

- Creating new classes from existing ones
- Child class **inherits** attributes and methods from parent
- Represents "is-a" relationship
- Enables code reuse and hierarchical organization

Key Terminology

- **Parent/Base/Super class**: Class being inherited from
- **Child/Derived/Sub class**: Class doing the inheriting
- **Method overriding**: Child redefines parent's method
- **super()**: Access parent class methods

Basic Inheritance Syntax i

```
# Parent class
class Animal:
    def __init__(self, name, species):
        self.name = name
        self.species = species

    def make_sound(self):
        return "Some generic sound"

    def info(self):
        return f"{self.name} is a {self.species}"

# Child class inherits from Animal
class Dog(Animal):
    def __init__(self, name, breed):
        # Call parent constructor
```

Basic Inheritance Syntax ii

```
        super().__init__(name, "Dog")
        self.breed = breed

    # Override parent method
    def make_sound(self):
        return "Woof! Woof!"

    # Add new method
    def fetch(self):
        return f"{self.name} is fetching the ball!"
```

Basic Inheritance Syntax iii

```
# Usage
dog = Dog("Buddy", "Golden Retriever")

# Inherited methods
print(dog.info())          # "Buddy is a Dog"

# Overridden method
print(dog.make_sound())    # "Woof! Woof!"

# New method
print(dog.fetch())         # "Buddy is fetching the ball
    !"

# Inherited attributes
print(dog.name)            # "Buddy"
```

Basic Inheritance Syntax iv

```
print(dog.species)          # "Dog"

# New attributes
print(dog.breed)            # "Golden Retriever"
```

What Happened?

- Dog **inherited** `info()` from Animal
- Dog **overrode** `make_sound()`
- Dog **added** `fetch()` and `breed`

Using super() Properly i

Calling parent methods:

```
class Vehicle:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        self.mileage = 0

    def drive(self, distance):
        self.mileage += distance
        return f"Drove {distance} km"

class ElectricCar(Vehicle):
    def __init__(self, brand, model, battery_capacity)
    :
        # Call parent constructor
```

Using super() Properly ii

```
        super().__init__(brand, model)
        self.battery_capacity = battery_capacity
        self.battery_level = 100

    def drive(self, distance):
        # Use parent's drive method
        result = super().drive(distance)
        # Add child-specific behavior
        battery_used = distance * 0.2
        self.battery_level -= battery_used
        return f"{result}, Battery: {self.
    battery_level:.1f}%"

    def charge(self):
        self.battery_level = 100
        return "Battery fully charged"
```

Using super() Properly iii

```
# Usage
tesla = ElectricCar("Tesla", "Model 3", 75)

print(tesla.drive(50))
# "Drove 50 km, Battery: 90.0%"

print(f"Total mileage: {tesla.mileage}")  # 50

print(tesla.charge())
# "Battery fully charged"

print(tesla.drive(100))
# "Drove 100 km, Battery: 80.0%"
```

Using super() Properly iv

super() Benefits

- Reuses parent code (DRY principle)
- Maintains consistency with parent class
- Easier to maintain and extend
- Works with multiple inheritance

Multi-level Inheritance i

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

    def breathe(self):
        return f"{self.name} is breathing"

class Animal(LivingBeing):
    def __init__(self, name, species):
        super().__init__(name)
        self.species = species

    def move(self):
        return f"{self.name} is moving"

class Mammal(Animal):
```

Multi-level Inheritance ii

```
    def __init__(self, name, species, fur_color):
        super().__init__(name, species)
        self.fur_color = fur_color

    def feed_young(self):
        return f"{self.name} is feeding its young with
     milk"

# Usage
lion = Mammal("Simba", "Lion", "golden")
print(lion.breathe())       # From LivingBeing
print(lion.move())          # From Animal
print(lion.feed_young())    # From Mammal
```

Method Resolution Order (MRO) i

Understanding inheritance hierarchy:

```
class A:
    def method(self):
        return "A's method"

class B(A):
    def method(self):
        return "B's method"

class C(A):
    def method(self):
        return "C's method"

class D(B, C):  # Multiple inheritance
    pass
```

Method Resolution Order (MRO) ii

```
# Check Method Resolution Order
print(D.mro())
# [<class 'D'>, <class 'B'>, <class 'C'>, <class 'A'>,
    <class 'object'>]

obj = D()
print(obj.method())  # "B's method"
# Searches: D -> B -> C -> A -> object
```

Important
Python uses C3 linearization for MRO. Methods are searched left-to-right in parent list, depth-first.

Polymorphism

Polymorphism: The Third Pillar

What is Polymorphism?

- **"Many forms"** - same interface, different implementations
- Objects of different types respond to same method call
- Enables writing flexible, reusable code
- Supports abstraction and code organization

Types of Polymorphism in Python

- **Method Overriding**: Child redefines parent method
- **Duck Typing**: "If it walks like a duck..."
- **Operator Overloading**: Magic methods (next section)

Method Overriding Polymorphism i

```
class Shape:
    def area(self):
        raise NotImplementedError("Subclass must
    implement")

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

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

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

Method Overriding Polymorphism ii

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

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius
```

Method Overriding Polymorphism iii

```
# Polymorphism in action
shapes = [
    Rectangle(5, 10),
    Circle(7),
    Rectangle(3, 4)
]

# Same method call, different behavior
for shape in shapes:
    print(f"Area: {shape.area():.2f}")
    print(f"Perimeter: {shape.perimeter():.2f}")
    print("---")

# Output:
# Area: 50.00
```

Method Overriding Polymorphism iv

```
# Perimeter: 30.00
# ---
# Area: 153.94
# Perimeter: 43.98
# ---
# Area: 12.00
# Perimeter: 14.00
```

Duck Typing i

Python's Philosophy
"If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck."

Python doesn't check types, it checks **behavior**.

```
class Dog:
    def speak(self):
        return "Woof!"

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

class Robot:
```

Duck Typing ii

```
    def speak(self):
        return "Beep boop!"

# Function works with any object that has speak()
def make_it_speak(thing):
    return thing.speak()

# No inheritance needed!
animals = [Dog(), Cat(), Robot()]
for animal in animals:
    print(make_it_speak(animal))
```

Duck Typing iii

Duck typing with file-like objects:

```
class StringBuffer:
    def __init__(self):
        self.buffer = []

    def write(self, text):
        self.buffer.append(text)

    def getvalue(self):
        return ''.join(self.buffer)

# Function expects file-like object (needs write
    method)
def save_report(file_obj, data):
    file_obj.write("Report Header\n")
    file_obj.write(f"Data: {data}\n")
```

Duck Typing iv

```
    file_obj.write("Report Footer\n")

# Works with actual file
with open('report.txt', 'w') as f:
    save_report(f, "Sales Data")

# Also works with StringBuffer (duck typing!)
buffer = StringBuffer()
save_report(buffer, "Memory Data")
print(buffer.getvalue())
```

Abstraction

Abstraction: The Fourth Pillar

What is Abstraction?

- **Hiding** complex implementation details
- **Showing** only essential features
- Defining interfaces without implementation
- Forcing child classes to implement specific methods

Abstract Base Classes (ABC)

- Cannot be instantiated directly
- Define abstract methods (no implementation)
- Child classes must implement abstract methods
- Python's `abc` module provides this

Creating Abstract Classes i

```
from abc import ABC, abstractmethod

class PaymentProcessor(ABC):
    @abstractmethod
    def process_payment(self, amount):
        """Process a payment - must be implemented by
    subclass"""
        pass

    @abstractmethod
    def refund_payment(self, transaction_id):
        """Refund a payment - must be implemented by
    subclass"""
        pass

    # Concrete method (has implementation)
```

Creating Abstract Classes ii

```
    def validate_amount(self, amount):
        if amount <= 0:
            raise ValueError("Amount must be positive"
    )
        return True

# This will raise TypeError - cannot instantiate
    abstract class
# processor = PaymentProcessor()  # Error!
```

Creating Abstract Classes iii

```
class CreditCardProcessor(PaymentProcessor):
    def process_payment(self, amount):
        self.validate_amount(amount)  # Use inherited
    method
        return f"Processing ${amount} via Credit Card"

    def refund_payment(self, transaction_id):
        return f"Refunding transaction {transaction_id
    }"

class PayPalProcessor(PaymentProcessor):
    def process_payment(self, amount):
        self.validate_amount(amount)
        return f"Processing ${amount} via PayPal"
```

Creating Abstract Classes iv

```
    def refund_payment(self, transaction_id):
        return f"Refunding PayPal transaction {
    transaction_id}"

# Now we can create instances
cc = CreditCardProcessor()
print(cc.process_payment(100))  # Works!

paypal = PayPalProcessor()
print(paypal.process_payment(50))  # Works!
```

Creating Abstract Classes v

Abstract class with partial implementation:

```
from abc import ABC, abstractmethod

class Database(ABC):
    def __init__(self, connection_string):
        self.connection_string = connection_string
        self.connected = False

    def connect(self):
        """Concrete method"""
        self.connected = True
        print(f"Connected to {self.connection_string}"
    )

    @abstractmethod
    def execute_query(self, query):
```

Creating Abstract Classes vi

```
        """Abstract - must be implemented"""
        pass

    @abstractmethod
    def close(self):
        """Abstract - must be implemented"""
        pass

class MySQLDatabase(Database):
    def execute_query(self, query):
        if not self.connected:
            return "Not connected!"
        return f"Executing MySQL query: {query}"

    def close(self):
        self.connected = False
```

```
        return "MySQL connection closed"
```

Magic Methods

Magic Methods (Dunder Methods)

What are Magic Methods?

- Special methods with `__double_underscores__`
- Python calls them automatically in certain situations
- Enable operator overloading and special behaviors
- Make custom objects behave like built-in types

Common Magic Methods

- `__init__`: Constructor
- `__str__`: String representation (for users)
- `__repr__`: Official representation (for developers)
- `__len__`: Length
- `__eq__`: Equality comparison
- `__add__`: Addition operator
- And many more...

String Representation Methods i

```
class Book:
    def __init__(self, title, author, pages):
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        """User-friendly string representation"""
        return f"{self.title} by {self.author}"

    def __repr__(self):
        """Developer-friendly representation"""
        return f"Book('{self.title}', '{self.author}',
    {self.pages})"

# Usage
```

String Representation Methods ii

```
book = Book("Python Crash Course", "Eric Matthes",
    544)

print(str(book))    # "Python Crash Course by Eric
    Matthes"
print(repr(book))   # Book('Python Crash Course', 'Eric
     Matthes', 544)

# In interactive shell
# >>> book
# Book('Python Crash Course', 'Eric Matthes', 544)

# String formatting
print(f"Reading: {book}")  # Uses __str__
```

Comparison Magic Methods i

```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __eq__(self, other):
        """Equal to: =="""
        if not isinstance(other, Point):
            return False
        return self.x == other.x and self.y == other.y

    def __lt__(self, other):
        """Less than: <"""
        # Compare by distance from origin
        self_dist = (self.x**2 + self.y**2) ** 0.5
        other_dist = (other.x**2 + other.y**2) ** 0.5
```

Comparison Magic Methods ii

```
        return self_dist < other_dist

    def __le__(self, other):
        """Less than or equal: <="""
        return self < other or self == other

    def __str__(self):
        return f"Point({self.x}, {self.y})"

# Usage
p1 = Point(1, 2)
p2 = Point(1, 2)
p3 = Point(3, 4)

print(p1 == p2)  # True (uses __eq__)
print(p1 == p3)  # False
```

Comparison Magic Methods iii

```
print(p1 < p3)    # True (uses __lt__)
print(p1 <= p2)   # True (uses __le__)
```

Arithmetic Magic Methods i

```
class Vector:
    def __init__(self, x, y):
        self.x = x
        self.y = y

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

    def __sub__(self, other):
        """Subtraction: -"""
        return Vector(self.x - other.x, self.y - other
    .y)

    def __mul__(self, scalar):
```

Arithmetic Magic Methods ii

```
        """Multiplication by scalar: *"""
        return Vector(self.x * scalar, self.y * scalar
    )

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

# Usage
v1 = Vector(2, 3)
v2 = Vector(1, 4)

v3 = v1 + v2  # Calls __add__
print(v3)     # Vector(3, 7)

v4 = v2 - v1  # Calls __sub__
print(v4)     # Vector(-1, 1)
```

Arithmetic Magic Methods iii

```
v5 = v1 * 3    # Calls __mul__
print(v5)      # Vector(6, 9)
```

Container Magic Methods i

```
class Playlist:
    def __init__(self, name):
        self.name = name
        self.songs = []

    def __len__(self):
        """Length: len()"""
        return len(self.songs)

    def __getitem__(self, index):
        """Indexing: playlist[i]"""
        return self.songs[index]

    def __setitem__(self, index, value):
        """Assignment: playlist[i] = value"""
        self.songs[index] = value
```

Container Magic Methods ii

```
    def __contains__(self, item):
        """Membership: in"""
        return item in self.songs

    def add_song(self, song):
        self.songs.append(song)

# Usage
playlist = Playlist("My Favorites")
playlist.add_song("Song A")
playlist.add_song("Song B")
playlist.add_song("Song C")

print(len(playlist))            # 3 (uses __len__)
```

Container Magic Methods iii

```
print(playlist[0])                  # "Song A" (uses
    __getitem__)
playlist[1] = "Song B Updated" # Uses __setitem__
print("Song A" in playlist)     # True (uses
    __contains__)

# Can iterate because of __getitem__
for song in playlist:
    print(song)
```

Context Manager Magic Methods i

```
class FileManager:
    def __init__(self, filename, mode):
        self.filename = filename
        self.mode = mode
        self.file = None

    def __enter__(self):
        """Called when entering 'with' block"""
        print(f"Opening {self.filename}")
        self.file = open(self.filename, self.mode)
        return self.file

    def __exit__(self, exc_type, exc_val, exc_tb):
        """Called when exiting 'with' block"""
        print(f"Closing {self.filename}")
        if self.file:
```

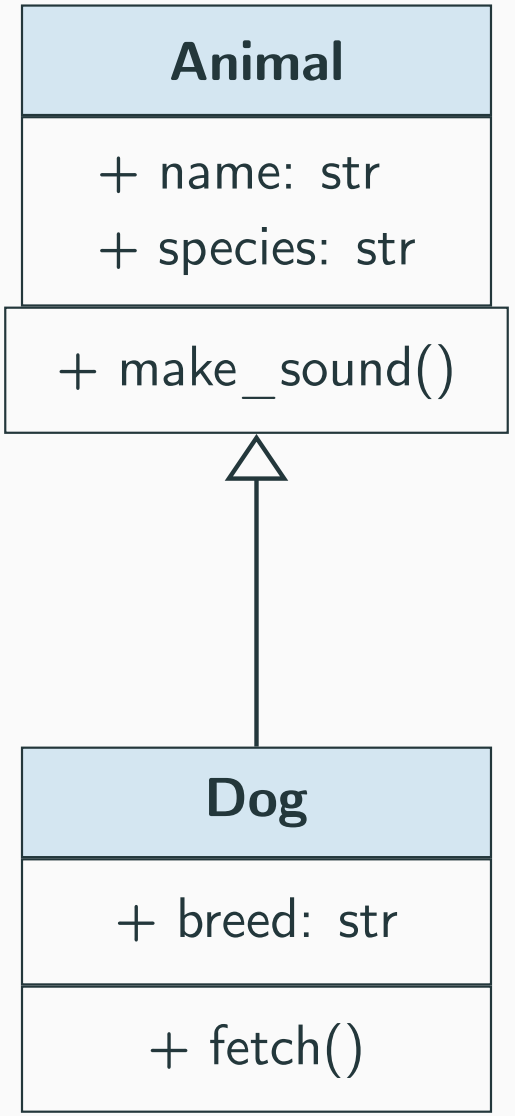
Context Manager Magic Methods ii

```
            self.file.close()
        # Return False to propagate exceptions
        return False

# Usage
with FileManager('test.txt', 'w') as f:
    f.write("Hello, World!")
# File automatically closed after block

# Equivalent to:
# manager = FileManager('test.txt', 'w')
# f = manager.__enter__()
# try:
#     f.write("Hello, World!")
# finally:
#     manager.__exit__(None, None, None)
```

Context Manager Magic Methods iii

Advanced UML Diagrams

UML: Inheritance Relationships



Inheritance Arrow

UML: Aggregation vs Composition

Aggregation (weak "has-a"):

Books can exist without Library

Composition (strong "part-of"):

Engine cannot exist without Car

Visual Difference

- **Aggregation**: Hollow diamond
- **Composition**: Filled diamond
- Both are "has-a" relationships
- Composition = stronger dependency

Aggregation Example i

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

class Library:
    def __init__(self, name):
        self.name = name
        self.books = []  # Aggregation

    def add_book(self, book):
        self.books.append(book)

    def remove_book(self, book):
        if book in self.books:
            self.books.remove(book)
```

Aggregation Example ii

```
# Books exist independently
book1 = Book("Python 101", "Alice")
book2 = Book("OOP Mastery", "Bob")

# Library aggregates books
library = Library("City Library")
library.add_book(book1)
library.add_book(book2)

# If library is destroyed, books still exist
del library
print(book1.title)  # Still works! "Python 101"
```

Composition Example i

```
class Engine:
    def __init__(self, horsepower):
        self.horsepower = horsepower

    def start(self):
        return f"Engine with {self.horsepower}hp
    starting..."

class Car:
    def __init__(self, brand, model, horsepower):
        self.brand = brand
        self.model = model
        # Engine is created inside Car (composition)
        self.engine = Engine(horsepower)

    def start(self):
```

Composition Example ii

```
        return f"{self.brand} {self.model}: {self.
    engine.start()}"

    def __del__(self):
        print(f"{self.brand} {self.model} destroyed")
        # Engine is destroyed with Car

# Engine only exists as part of Car
car = Car("Toyota", "Camry", 200)
print(car.start())

# When car is destroyed, engine is also destroyed
del car  # Both Car and Engine are destroyed
```

Practical Example

Complete Example: Employee Management System i

```
from abc import ABC, abstractmethod

class Employee(ABC):
    """Abstract base class for all employees"""
    total_employees = 0

    def __init__(self, name, employee_id, base_salary)
    :
        self.__name = name
        self.__employee_id = employee_id
        self.__base_salary = base_salary
        Employee.total_employees += 1

    @property
    def name(self):
        return self.__name
```

Complete Example: Employee Management System ii

```
    @property
    def employee_id(self):
        return self.__employee_id

    @property
    def base_salary(self):
        return self.__base_salary

    @base_salary.setter
    def base_salary(self, value):
        if value < 0:
            raise ValueError("Salary cannot be
negative")
        self.__base_salary = value
```

Complete Example: Employee Management System iii

```
    @abstractmethod
    def calculate_salary(self):
        """Must be implemented by subclasses"""
        pass

    @abstractmethod
    def get_role(self):
        """Must be implemented by subclasses"""
        pass

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

    def __repr__(self):
```

Complete Example: Employee Management System iv

```
        return f"{self.__class__.__name__}('{self.name
    }', '{self.employee_id}')"

    def __eq__(self, other):
        if not isinstance(other, Employee):
            return False
        return self.employee_id == other.employee_id

    @classmethod
    def get_total_employees(cls):
        return cls.total_employees
```

Complete Example: Employee Management System v

```
class Developer(Employee):
    def __init__(self, name, employee_id, base_salary,
     programming_languages):
        super().__init__(name, employee_id,
    base_salary)
        self.programming_languages =
    programming_languages

    def calculate_salary(self):
        # Bonus for each programming language
        bonus = len(self.programming_languages) * 1000
        return self.base_salary + bonus

    def get_role(self):
        return "Software Developer"
```

Complete Example: Employee Management System vi

```
    def add_language(self, language):
        if language not in self.programming_languages:
            self.programming_languages.append(language
    )

class Manager(Employee):
    def __init__(self, name, employee_id, base_salary,
     team_size):
        super().__init__(name, employee_id,
    base_salary)
        self.team_size = team_size
        self.team_members = []

    def calculate_salary(self):
        # Bonus based on team size
```

Complete Example: Employee Management System vii

```
        bonus = self.team_size * 2000
        return self.base_salary + bonus

    def get_role(self):
        return "Manager"

    def add_team_member(self, employee):
        self.team_members.append(employee)
```

Complete Example: Employee Management System viii

```
class Company:
    def __init__(self, name):
        self.name = name
        self.employees = []  # Aggregation

    def hire(self, employee):
        self.employees.append(employee)
        print(f"Hired: {employee}")

    def total_payroll(self):
        return sum(emp.calculate_salary() for emp in
    self.employees)

    def list_employees(self):
        for emp in self.employees:
```

Complete Example: Employee Management System ix

```
            print(f"{emp} - Salary: ${emp.
    calculate_salary():,}")

# Usage
company = Company("Tech Corp")

dev1 = Developer("Alice", "E001", 60000, ["Python", "
    JavaScript"])
dev2 = Developer("Bob", "E002", 65000, ["Java", "C++",
     "Python"])
mgr = Manager("Charlie", "M001", 80000, 5)

company.hire(dev1)
company.hire(dev2)
company.hire(mgr)
```

Complete Example: Employee Management System x

```
print(f"\nTotal Employees: {Employee.
    get_total_employees()}")
print(f"Total Payroll: ${company.total_payroll():,}")
print("\nEmployee List:")
company.list_employees()
```

Best Practices

OOP Best Practices

Design Principles

- **SOLID Principles**:
  - Single Responsibility
  - Open/Closed
  - Liskov Substitution
  - Interface Segregation
  - Dependency Inversion
- **DRY**: Don't Repeat Yourself
- **KISS**: Keep It Simple, Stupid
- **Favor composition over inheritance**
- **Program to interfaces, not implementations**

When to Use Each Pillar

Encapsulation

- Always! Hide implementation details
- Use properties for controlled access
- Validate data in setters

Inheritance

- For clear "is-a" relationships
- When code reuse makes sense
- Keep hierarchies shallow (max 2-3 levels)

Polymorphism

- Write flexible, extensible code
- Work with interfaces, not concrete types
- Leverage duck typing

Abstraction

Practice Exercises

Exercise 1: Bank Account Hierarchy

Create a banking system with inheritance:

Task
Create an abstract `BankAccount` class with:

- Abstract method: `calculate_interest()`
- Concrete methods: `deposit()`, `withdraw()`
- Properties for balance (with validation)

Then create:

- `SavingsAccount`: 5% interest
- `CheckingAccount`: No interest, has overdraft limit

Add magic methods for string representation and comparison.

Exercise 2: Shape Calculator with Magic Methods

Implement shapes with operator overloading:

Task
Create shape classes that support:

- Addition: Combine areas (`shape1 + shape2`)
- Comparison: Compare by area (`shape1 > shape2`)
- String representation: Show dimensions and area
- Container protocol: Store shapes in list and sort

Implement: `Rectangle`, `Circle`, `Triangle`

Exercise 3: Complete Library System

Design a comprehensive library management system:

Task
Implement:

- `Book` class with properties
- `Member` class with borrowing history
- `Library` class using aggregation
- Abstract `LibraryItem` base class
- Derived classes: `Book`, `Magazine`, `DVD`

Requirements:

- Encapsulation (private attributes)
- Inheritance (item hierarchy)
- Polymorphism (different checkout rules)
- Magic methods (string representation, comparison)

Summary

Week 6 Summary

What We Learned:

Four Pillars of OOP

- **Encapsulation**: Data hiding with properties
- **Inheritance**: Building class hierarchies with super()
- **Polymorphism**: Method overriding and duck typing
- **Abstraction**: Abstract base classes with ABC

Advanced Concepts

- Magic methods for operator overloading
- Advanced UML diagrams
- Aggregation vs Composition
- MRO and multiple inheritance
- Properties and descriptors

Skills Checklist

You should now be able to:

- Implement encapsulation with private attributes
- Use @property decorator for getters/setters
- Create inheritance hierarchies with super()
- Override methods for polymorphic behavior
- Design abstract base classes
- Implement magic methods
- Use composition and aggregation appropriately
- Create advanced UML diagrams
- Design production-ready OOP systems

Next Steps

Continue Learning

- Practice implementing all four pillars
- Study design patterns (Singleton, Factory, Observer, etc.)
- Build real-world projects using OOP
- Explore advanced topics:
  - Metaclasses
  - Descriptors
  - Decorators with classes
  - Context managers

Homework

- Complete all practice exercises
- Design and implement a complete OOP project
- Draw UML diagrams for your design
- Review all OOP concepts from Weeks 5-6

Thank You!

Thank you for your attention!

Keep Practicing!

Access Course Materials:

Download Course Materials

Center for Artificial Intelligence & Emerging Technologies