Welcome to Lecture 19: Intro to OOP II

1) Open a Code Editor
2) Use Iclicker for attendance

Announcements

- Victoria’s OH today will be online
- OH queue is still down - University-level issue

Topics

- Encapsulation
- Inheritance
- Polymorphism

REVIEW: OBJECT-ORIENTED PROGRAMMING (OOP)

- OOP is a programming paradigm with its own vocabulary:
    - Class: A template for defining entities (called objects.)
    - Object: An entity defined by (an instance of) a particular class.
        - Every object has a type, called its class.
    - To create new types of data, we implement new classes.

- Classes are an essential part organizing code in Object Oriented Programming (OOP)

```
name = input("What is your name: ")

print(name)

print(type(name))
```

```
Name:
```

```
Name: Lisa
Lisa
<class 'str'>
```

- Every object has a type, called its class.

```
>>> some_list = ["Eggsalad", "Alonzo"]
>>> type(some_list)
<class 'list'>
>>> some_dict = {"Eggsalad":"Alonzo", "Malhotra":"Vedansh"}
>>> type(some_dict)
<class 'dict'>
```

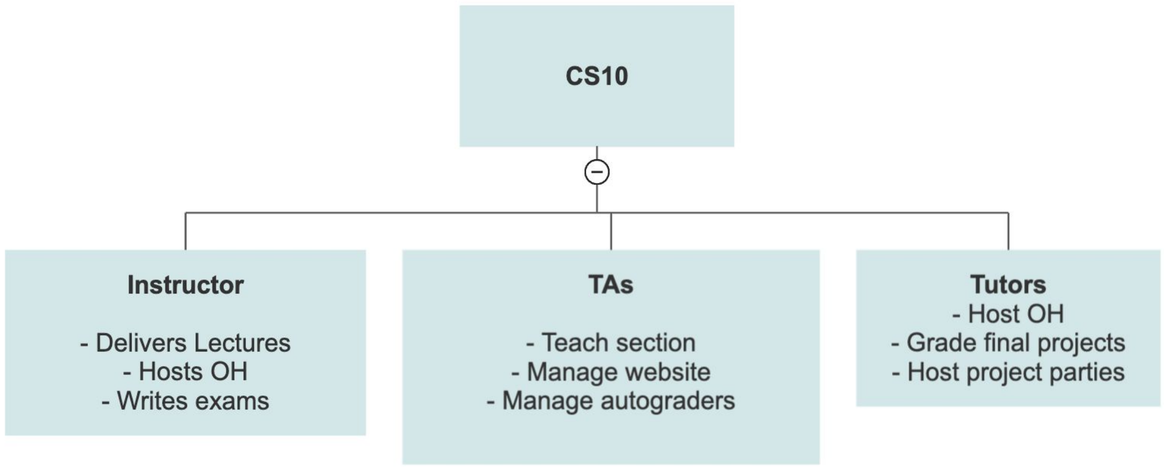
- These are built-in classes, we'll make our own!

OBJECT-ORIENTED PROGRAMMING (OOP)

- **Modular Programming:** Separating the functionality of a program into independent chunks (modules.)
- **What is it?**
  - It's a way of writing computer programs by breaking them into smaller, separate parts.
- **Why do it?**
  - It makes the program easier to understand, manage, and fix.
  - Each part (or module) can be worked on independently.

OBJECT-ORIENTED PROGRAMMING (OOP)

- Class → CS10 (Template)
- Object→ CS10 Summer 2024



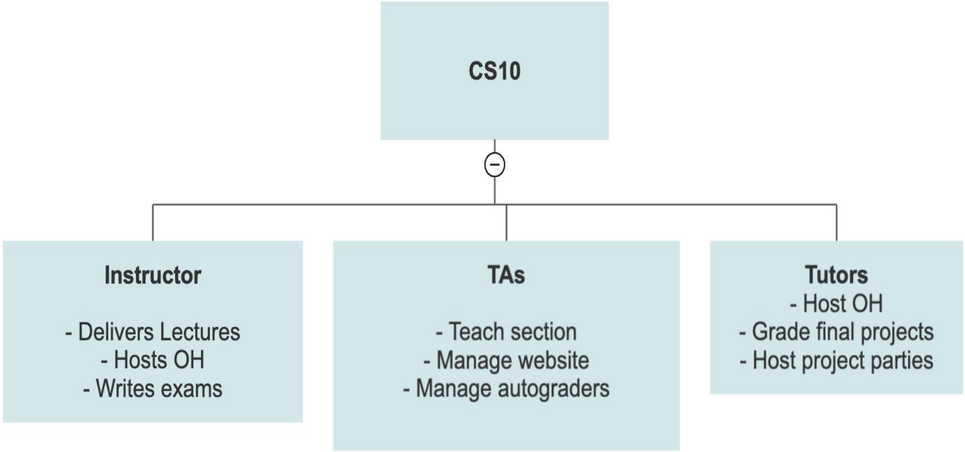
OBJECT-ORIENTED PROGRAMMING (OOP)

- Modular Programming: Separating the functionality of a program into independent chunks (modules.)

- Example of a modular procedure:

    - Modules communicate

    - Abstraction barriers!



Constructors and Instance Attributes

- The "dunder init" (double-under) method is the constructor of the class `Dog`.

- When we call `dog1 = dog("Costa")`, the parameter `self` is bound to the newly created `dog` object.

- The constructor binds the value `"Costa"` to the object's `name` attribute.

```python
class dog:
  def __init__(self, my_name):
    self.name = my_name

dog1 = dog("Costa")
print(f"the dog is named {dog1.name}")
```

DOT NOTATION

- We could also rename Costa using Dot notation

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

dog1 = dog("Costa")

dog1.name = "Wonder Dog"
print(dog1.name)
#output: Wonder Dog
```

CLASS ATTRIBUTES

- Assigned in the suite of the class, outside any method definitions.

```
class dog:

  species = "canine"

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

dog1 = dog("Wonder Dog")
print(dog1.name)
print(dog1.species)

dog2 = dog("Glen")
print(dog1.name)
print(dog1.species)
```

vars() function to print all the attributes of an Object

```
class dog:

  species = "canine"

  def __init__(self, my_name, breed):
    self.name = my_name
    self.breed = breed

dog1 = dog("Wonder Dog", "Springer Spaniel")

print(vars(dog1))
#output: {'name': 'Wonder Dog', 'breed': 'Springer Spaniel'}
```

INSTANCE METHODS

- Include a special first parameter `self`,
- implicitly bound to the object on which the method is invoked, thanks to dot notation.

```python
def __init__(self, my_name):
    self.name = my_name

def bark(self, greeting):
    print(f"Woof Woof, {self.name} says {greeting}")

dog1 = dog("Wonder Dog")
dog1.bark("Give me a treat")
#output: Woof Woof, Wonder Dog says Give me a treat

dog2 = dog("Glen")
dog2.bark("Get off my lawn!")
#output: Woof Woof, Glen says Get off my lawn!
```

Advanced OOP concepts (you will recognize some of this…)

Encapsulation: Bundling data and methods into a single unit (class) and restricting access to certain parts of the object.

Inheritance: Creating new classes from existing classes, inheriting attributes and methods.

Polymorphism:

- A base class (parent class) defines a common interface (methods) which can be overridden by its subclasses.
- Subclasses modify methods
- But we can call methods on subclasses with the same name as defined in the Parent class

Encapsulation

- Encapsulation introduces the idea of bundling data and methods together into classes (from OOP Day 1)
- Restricting access to protect the integrity of the data
- Using getter and setter methods,, you can control how an attribute is accessed and modified.
    - This helps maintain the integrity and consistency of the data.

```python
class Player:
    def __init__(self, name, team):
        self.__name = name
        self.__team = team

    def get_name(self):
        return self.__name

    def set_name(self, name):
        if isinstance(name, str):
            self.__name = name
        else:
            print("Invalid Name")

    def get_team(self):
        return self.__team

    def set_team(self, team):
        if isinstance(team, str):
            self.__team = team
        else:
            print("Invalid Team")
```

Encapsulation

- The double underscore makes an attribute private
- We can no longer access/modify via dot notation
    - ~~Return {self.name}~~
    - ~~self.name = “Rothman”~~
- We create get and set methods to access and modify data
    - “getters” and “setters”
- We can also have private methods (not covered today)

```python
    def __init__(self, name, team):
        self.__name = name
        self.__team = team

    def get_name(self):
        return self.__name

    def set_name(self, name):
        if isinstance(name, str):
            self.__name = name
        else:
            print("Invalid Name")

    def get_team(self):
        return self.__team

    def set_team(self, team):
        if isinstance(team, str):
            self.__team = team
        else:
```

- Using getters and setters, you can control how an attribute is accessed and modified.
    - This helps maintain the integrity and consistency of the data.
- You change access and modify data through methods,
    - you cannot directly access/modify data directly through the attributes
- ***(Demo this)***

```
player1 = Player("Lebron", "Lakers")
print(f"Player's Name and Team: {player1.get_na
and {player1.get_team()}")
#Output: Player's Name and Team: Lebron and Lak

#Change team to the Warriors
player1.set_team("Warriors")

print(f"Player's Team: {player1.get_team()}")
#output: Player's Team: Warriors

#try to change team via attribute
player1.__team = "Lakers"
print(f"Player's Team: {player1.get_team()}")
#output: Player's Team: Warriors
#still on the Warriors, sorry Lakers fans
```

Encapsulation Benefits

Data Integrity: Prevents external code from directly altering an object's state in a way that could leave it inconsistent or invalid.

Security: Sensitive data is protected from unauthorized access.

Abstraction: Users of the object need not worry about the internal implementation details. They interact with the object through its public interface (methods), promoting a clean separation between how an object is used and how it is implemented.

Question: When might this apply?

Task 1: Code a Student Class

- Code a student class with:
  - 2 private attributes:
    - Name
    - GPA
  - Getters and Setters
    - get_name, set_name
    - get_GPA, set_GPA

Inheritance: allows a new class to inherit attributes and methods from an existing class.

Parent Class

```
class Player:
  def __init__(self, name, team):
    self.__name = name
    self.__team = team

  def get_name(self):
    return self.__name

  def set_name(self, name):
    if isinstance(name, str):
      self.__name = name
    else:
        print("Invalid Name")
```

Child Classes

```
class Basketball(Player):
  def __int__(self, name, team, points):
    super().__init__(name, team)
    self.points = points

    def get_points(self):
      return self.points

    def set_points(self):
      if isinstance(points, int):
        self.points = points
      else:
        print("Need points as int")
```

```
class Football(Player):
  def __int__(self, name, team, touchdowns):
    super().__init__(name, team)
    self.touchdowns = touchdowns

    def get_touchdowns(self):
      return self.touchdowns

    def set_points(self):
      if isinstance(touchdowns, int):
        self.touchdowns = touchdowns
      else:
        print("Need touchdowns as int")
```

Inheritance: **Parent Class (Base Class or Superclass)**:

Definition: The class whose attributes and methods are inherited.

Purpose: Encapsulates common attributes and methods that can be shared by multiple child classes.

```
class Player:
  def __init__(self, name, team):
    self.__name = name
    self.__team = team

  def get_name(self):
    return self.__name

  def set_name(self, name):
    if isinstance(name, str):
      self.__name = name
    else:
        print("Invalid Name")
```

Inheritance: Child Class and super() function

- **Child Class** inherits from the parent class and can have additional attributes and methods or override the parent class's methods.
- **super()** function used to call the parent class's methods and constructors from the child class.

```
class Basketball(Player):
  def __int__(self, name, team, points):
    super().__init__(name, team)
    self.points = points

    def get_points(self):
      return self.points

    def set_points(self):
      if isinstance(points, int):
        self.points = points
      else:
        print("Need points as int")
```

```
class Football(Player):
  def __int__(self, name, team, touchdowns):
    super().__init__(name, team)
    self.touchdowns = touchdowns

    def get_touchdowns(self):
      return self.touchdowns

    def set_points(self):
      if isinstance(touchdowns, int):
        self.touchdowns = touchdowns
      else:
        print("Need touchdowns as int")
```

Why not just use Classes and Objects?

With Inheritance

```
class Vehicle:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start_engine(self):
        print("Engine started")

class Car(Vehicle):
    def __init__(self, make, model, num_doors):
        super().__init__(make, model)
        self.num_doors = num_doors

class Truck(Vehicle):
    def __init__(self, make, model, cargo_capacity):
        super().__init__(make, model)
        self.cargo_capacity = cargo_capacity
```

Benefit: Common functionality is centralized in the `Vehicle` class, and specific attributes or methods can be added in the `Car` and `Truck` subclasses without duplicating code.

Without Inheritance

```
class Car:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start_engine(self):
        print("Engine started")

class Truck:
    def __init__(self, make, model):
        self.make = make
        self.model = model

    def start_engine(self):
        print("Engine started")
```

Issue: Code duplication and difficulty in managing changes to common functionality.

Inheritance: Why Use?

- **Code Reuse**: Common functionality defined in the parent class can be reused in multiple child classes, reducing redundancy and promoting DRY (Don't Repeat Yourself) principles.

- **Extensibility**: Existing classes can be extended to add new features without modifying the original class, allowing for more flexible and scalable code.

- **Polymorphism**: Enables objects of different classes to be treated as objects of a common superclass, allowing for dynamic method calls and more flexible code.

When might this apply?

Hierarchical Relationships:

- **Example**: Animal kingdom, company organizational structure.
- **Why**: Models natural hierarchies where subtypes share common behavior but also have specific behaviors.

Extending Functionality:

- **Example**: Creating specialized versions of general-purpose classes.
- **Why**: Allows extension of existing classes without modifying them, preserving the original functionality.

Promoting Reusability:

- **Example**: GUI components like buttons, text fields, etc., which share common behaviors.
- **Why**: Reduces redundancy by allowing shared behavior to be defined once and reused.

Task 2: Code 2 Child Classes for Student Class

Child Classes should:

- Inherent attributes and methods from Parent class
  - Pay attention to super() function
- Child class for 2 seperate levels of School (your choice)
  - Elementary School
  - High School
  - College
  - Grad School
- Have unique attributes:
  - age
  - school

Polymorphism

Definition: Polymorphism means "many forms"

- allows objects of different classes to be treated as objects of a common superclass.
- It is implemented through method overriding and interfaces.

```python
class Player:
  def __init__(self, name, team):
    self.__name = name
    self.__team = team

  def get_name(self):
    return self.__name

  def get_team(self):
     return self.__team

  def player_introduction(self):
    raise NotImplementedError("subclass must implement abstract method")

class Basketball(Player):
  def __init__(self, name, team, points):
      super().__init__(name, team)
      self.points = points

  def player_introduction(self):
      print(f"{self.get_name()} plays for {self.get_team()} in the NBA")

class Football(Player):
  def __init__(self, name, team, touchdowns):
      super().__init__(name, team)
      self.touchdowns = touchdowns

  def player_introduction(self):
      print(f"{self.get_name()} plays for {self.get_team()} in the NFL")

f1 = Football("Brock", "49ers", 7)
b1 = Basketball("Steph", "Warriors", 1000)
f1.player_introduction()  # Add parentheses to call the method
b1.player_introduction()  # Add parentheses to call the method
```

Polymorphism

Definition: Polymorphism means "many forms"

- A base class (parent class) defines a common interface (methods) which can be overridden by its subclasses.
- Subclasses modify methods
- But we can call methods on subclasses with the same name as defined in the Parent class

```python
class Player:
  def __init__(self, name, team):
    self.__name = name
    self.__team = team

  def get_name(self):
    return self.__name

  def get_team(self):
     return self.__team

  def player_introduction(self):
    raise NotImplementedError("subclass must implement abstract method")

class Basketball(Player):
  def __init__(self, name, team, points):
      super().__init__(name, team)
      self.points = points

  def player_introduction(self):
      print(f"{self.get_name()} plays for {self.get_team()} in the NBA")

class Football(Player):
  def __init__(self, name, team, touchdowns):
      super().__init__(name, team)
      self.touchdowns = touchdowns

  def player_introduction(self):
      print(f"{self.get_name()} plays for {self.get_team()} in the NFL")

f1 = Football("Brock", "49ers", 7)
b1 = Basketball("Steph", "Warriors", 1000)
f1.player_introduction()  # Add parentheses to call the method
b1.player_introduction()  # Add parentheses to call the method
```

Polymorphism: Method Overriding

Definition: Allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

Purpose: Enables the subclass to tailor the inherited method to fit its needs.

```
class Media:
    def __init__(self, title, creator):
        self.title = title
        self.creator = creator

    def play(self):
        raise NotImplementedError("Subclass must implement abstrac
```

```
class Song(Media):
    def __init__(self, title, creator, duration):
        super().__init__(title, creator)
        self.duration = duration

    def play(self):
        return f"Playing song '{self.title}' by {self.cr
```

Polymorphism: Method Overriding

Common Interface: Media class with an abstract play method.

Child Classes: Song and Podcast implement the play method.

Polymorphic Behavior: Treating different media items uniformly through the play method.

```
class Media:
    def __init__(self, title, creator):
        self.title = title
        self.creator = creator

    def play(self):
        raise NotImplementedError("Subclass must implement abstract 
```

```
class Song(Media):
    def __init__(self, title, creator, duration):
        super().__init__(title, creator)
        self.duration = duration

    def play(self):
        return f"Playing song '{self.title}' by {self.crea
```

Task 3 Add a Polymorphic Method: Share_info

- Add an abstract method “Share_info” in the Parent Class
- In each Child Class, implement the method that prints the name of the student and what level of school they are in

Why Polymorphism?

Flexibility and Reusability:

- **Benefit**: Promotes code reusability by allowing the same interface to interact with objects of different types.

Maintainability:

- **Benefit**: Enhances maintainability by centralizing method interfaces while allowing specialized implementations.

Extensibility:

- **Benefit**: Simplifies the extension of code by enabling new classes to integrate seamlessly with existing code.

Putting It All Together

Here's a combined example demonstrating encapsulation, inheritance, and polymorphism:

```python
class Vehicle:
    def __init__(self, make, model):
        self.make = make  # public attribute
        self._model = model  # protected attribute

    def start_engine(self):
        raise NotImplementedError("Subclass must implement abstract method")

    def get_model(self):
        return self._model

class Car(Vehicle):
    def __init__(self, make, model):
        super().__init__(make, model)

    def start_engine(self):
        return "Car engine started."

class Motorcycle(Vehicle):
    def __init__(self, make, model):
        super().__init__(make, model)

    def start_engine(self):
        return "Motorcycle engine started."

# Polymorphism in action
vehicles = [Car("Toyota", "Corolla"), Motorcycle("Honda", "CBR")]

for vehicle in vehicles:
    print(f"{vehicle.make} {vehicle.get_model()}: {vehicle.start_engine()}")
```

Summary

- **Encapsulation**: Bundling data and methods into a single unit (class) and restricting access to certain parts of the object.
- **Inheritance**: Creating new classes from existing classes, inheriting attributes and methods.
- **Polymorphism**: Treating objects of different classes that share a common superclass in a uniform way.

These concepts help in organizing code more efficiently, promoting reuse, and enhancing flexibility and maintainability.