Course Title: Object Oriented Programming

Mid Review

Introduction to OOP and Python Basics

What is OOP?

- **Definition:** A programming paradigm that organizes data and behavior into objects.

Introduction to OOP:

- **Class:** Blueprint for objects (e.g., Student class).
- **Object:** Instance of a class (e.g., swakkhar = Student()).
- **Constructor:** __init__ method for initializing attributes.
- **Code:**

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

Why OOP?:

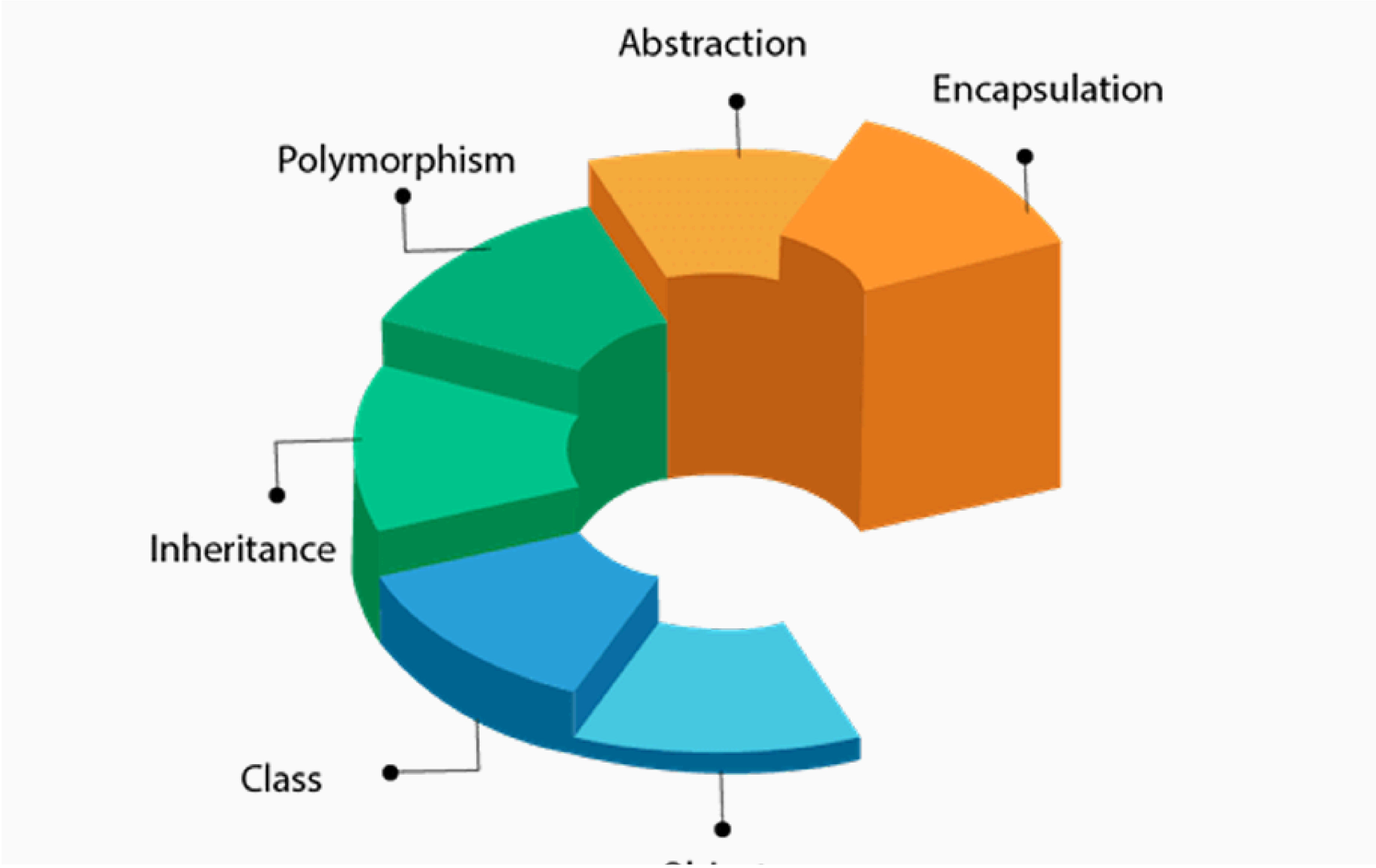
- **Modularity:** Break complex systems into reusable components.
- **Maintainability:** Easier to debug and update.
- **Scalability:** Extend functionality without rewriting code.

Attributes and Methods:

- **Constructor:** __init__ initializes object attributes.
- **Methods vs. Functions:** Methods belong to classes; functions are standalone.
- **Instance Variables:** Unique to each object (e.g., self.cgpa).
- **Class Variables:** Shared across all instances (e.g., university = "UIU").
- **Docstrings:** Document classes/methods for readability.

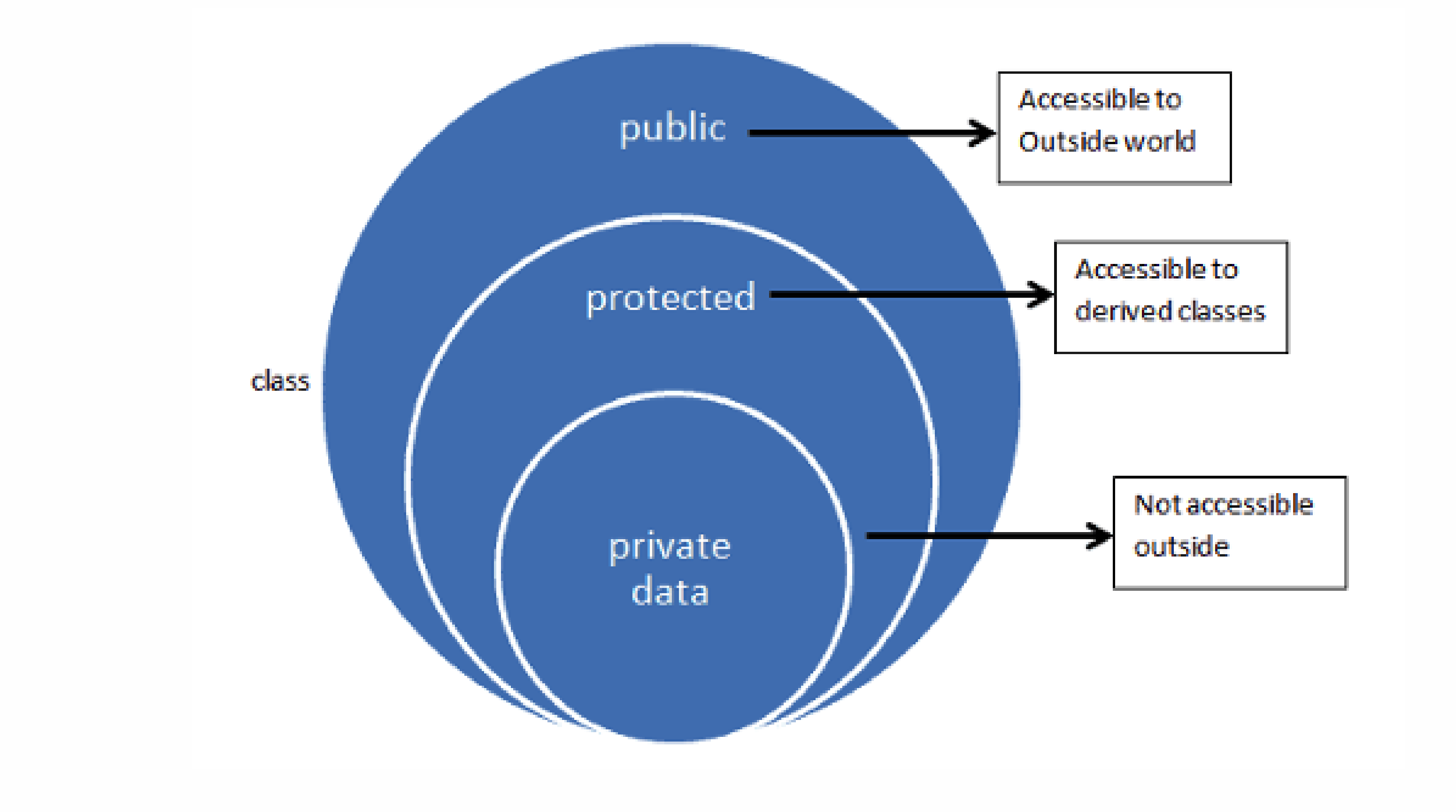
```
class Date:
    """Represents a date with day, month, and year"""
    def __init__(self, day, month):
        """Initialize day and month attributes"""
        self.day = day
        self.month = month
```

Core OOP Principles:



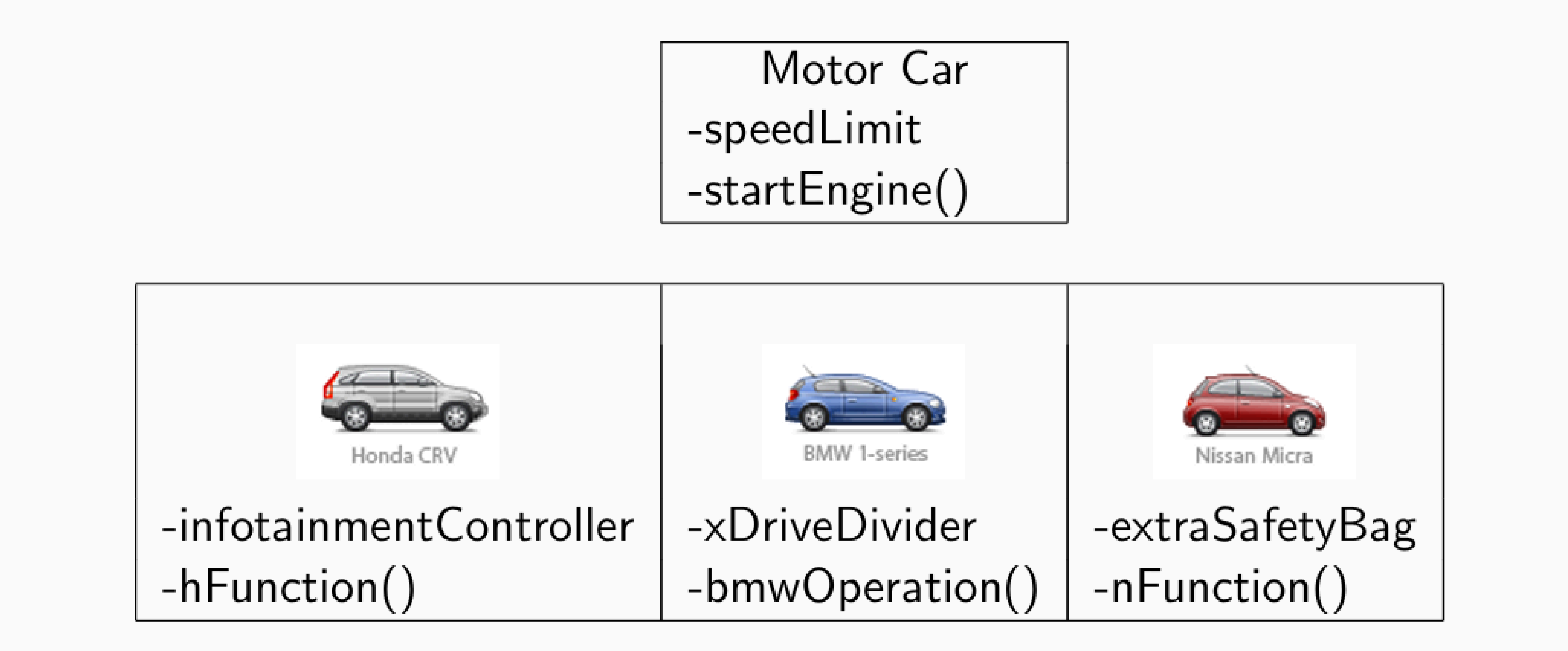
- **Abstraction:** Simplify real-world entities into classes.
    - **Example:** **A Car class models start_engine(), not individual engine parts.**

- **Encapsulation:** Protect data using private attributes. Real Life Example: Car



```
class Student:
    def __init__(self, name, sid):
            self.name = name  # Public
            self.__sid = sid  # Private (name mangling: _Student__sid)
```

- **Inheritance:** Reuse code via parent-child relationships.



Motor Car
-speedLimit
-startEngine()

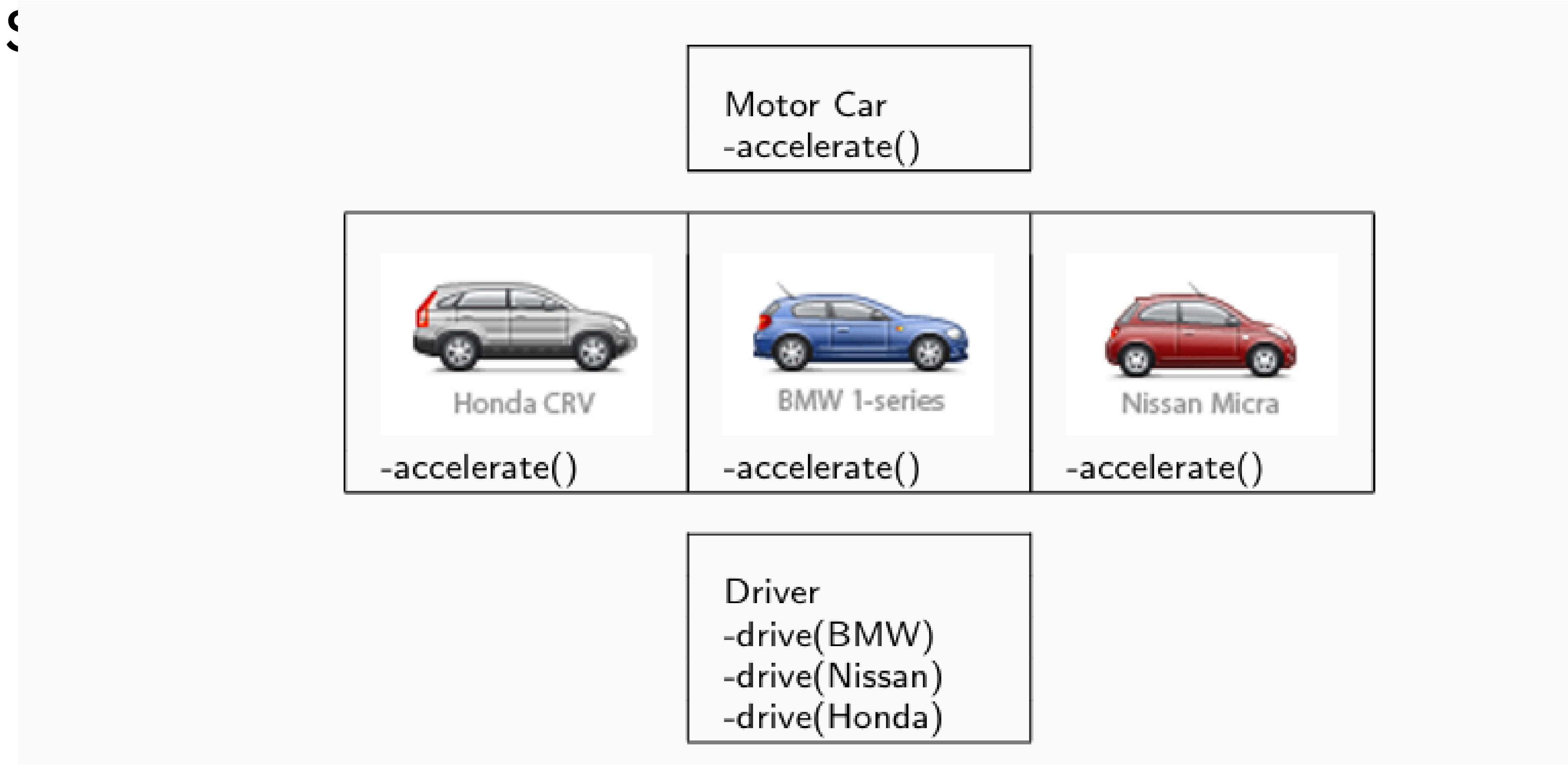
-infotainmentController
-hFunction()

-xDriveDivider
-bmwOperation()

-extraSafetyBag
-nFunction()

```
class Employee:
    def calculate_salary(self): pass
class SalariedEmployee(Employee):
    def calculate_salary(self): return self.monthly_salary
```

- **Polymorphism (from Greek, meaning “many forms”):** One interface, multiple implementations



```
class Shape:
    def area(self):  pass
class Circle(Shape):
    def area(self): return 3.14 * self.radius ** 2
```

- **Exercise:**
  - Create a Vehicle class with n_wheels and start_engine() method.
  - Add docstrings to explain functionality.

Class Design and Advanced Attributes

What are Classes and Objects?

- **Class:** A blueprint for creating objects (e.g., Student class).
- **Object:** An instance of a class (e.g., swakkhar = Student()).
- **Instantiation:** The process of creating an object from a class.

Built-in Python Objects

- **Example:** Strings are objects of the str class.

```
s = "I am a string"
print(type(s))  # Output: <class 'str'>
```

Class Structure

- Use the class keyword.
- **Naming convention: CamelCase (**e.g., BankAccount, StudentDetails**).**

```
class Student:
    pass  # Empty class
print(Student)  # Output: <class '__main__.Student'>
```

Adding Attributes

- Attributes define the state of an object.
- Added dynamically or via a constructor.

```
swakkhar = Student()
swakkhar.name = "Swakkhar"  # Dynamic attribute
```

Constructor (__init__)

- Special method to initialize object attributes.
- Syntax:

```
classs student:
 def __init__(self,name):
     self.name = name
```

Code Example:

```
class Student:
    def __init__(self, name, age):
        self.name = name
        self.age = age
student1 = Student("Alice", 20)
print(student1.name)   # Output: Alice
```

Constructors with Default Parameters

- **Initialize attributes with optional parameters.**

```
class Rectangle:
    def __init__(self, width=0, height=0):
        self.width = width
        self.height = height
rect = Rectangle()  # width=0, height=0
```

Class vs. Instance Variables

- **Instance Variables:** Unique to each object (self.cgpa).
- **Class Variables:** Shared across all instances (university = "UIU").

```
class Student:
    university = "UIU"  # Class variable
    def __init__(self, name):
        self.name = name  # Instance variable
```

Deep vs. Shallow Copy

- **Shallow Copy:** Copies references (changes affect original).
    - **a = [1, 2, 3]; b = a; b[0] = 99** # a becomes [99, 2, 3]

- **Deep Copy:** Creates independent copies.

    -
      ```
      import copy
      b = copy.deepcopy(a)  # b is a new list
      ```

Code Example:

```
class Period:
        def __init__(self, start, end):
                self.start = copy.deepcopy(start)  # Avoid shared references
                self.end = copy.deepcopy(end)
 d1 = Date(31, 1, 2024)
 p = Period(d1, Date(31, 5, 2024))
 d1.year = 2025  # p.start.year remains 2024 (deep copy)
```

Methods in Classes

- **Functions defined within a class.**

```
class Student:
    def display_info(self):
        print(f"Name: {self.name}, CGPA: {self.cgpa}")
```

Dunder Methods

- Special methods like __str__ for string representation.
- **__del__ (Destructor):** Called when an object is destroyed.
-

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

***Return Values:** Methods can return results based on object state.

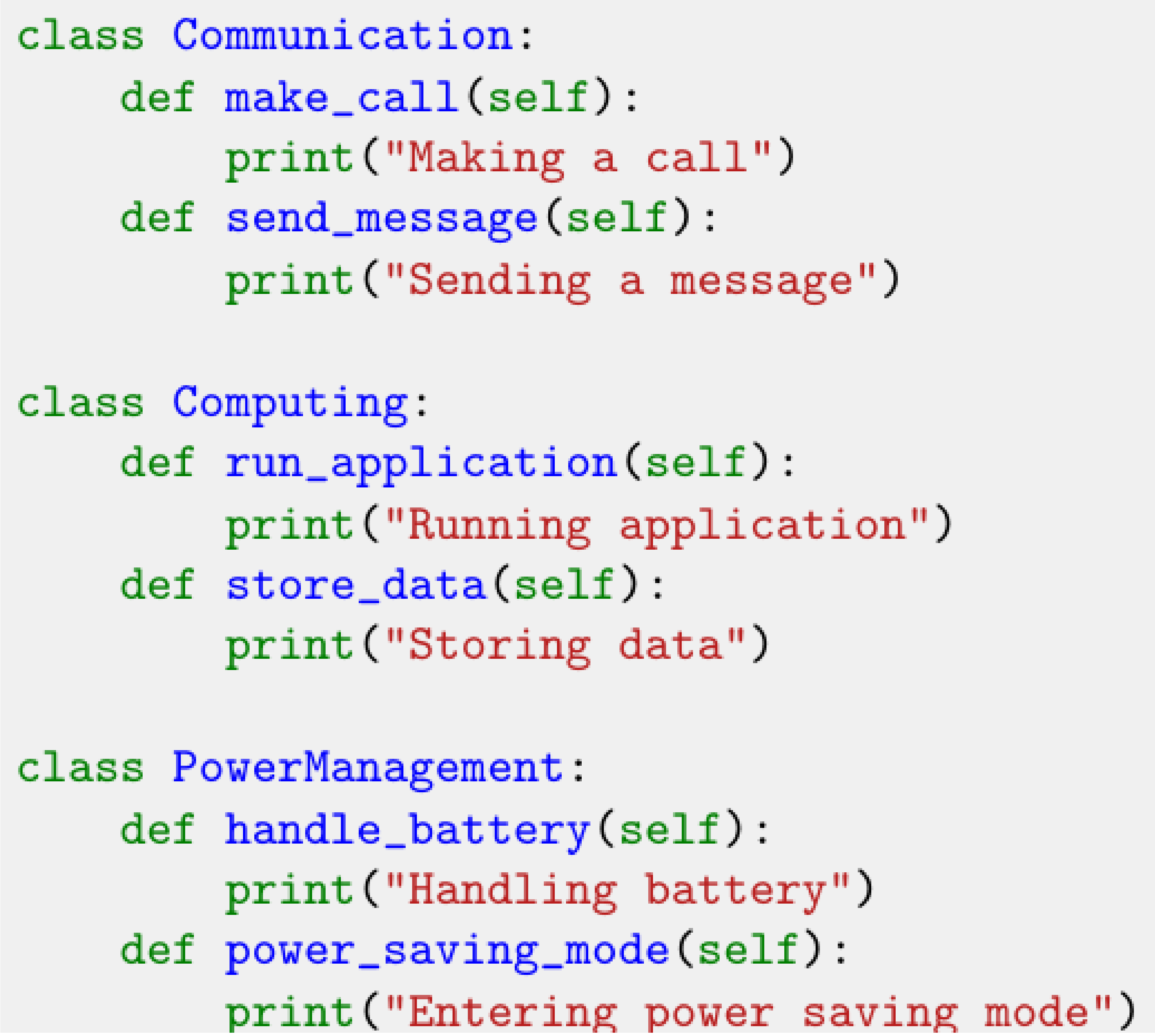
Inheritance and Abstract Classes

- ## Basic Inheritance
  - **Parent Class:** Base functionality.
  - **Child Class:** Extends/overrides parent methods.

```
class Employee:
    def __init__(self, name):
        self.name = name
    def calculate_salary(self):
        pass  # Abstract method
class HourlyEmployee(Employee):
    def __init__(self, name, hourly_rate):
        super().__init__(name)  # Initialize parent's attributes
        self.hourly_rate = hourly_rate
    def calculate_salary(self, hours):
        return self.hourly_rate * hours
```

- **Multiple Inheritance and MRO**
  - **Diamond Problem:** Ambiguity in method resolution.
  - **MRO:** Determines the order of method lookup.

```
class A: pass
class B(A): pass
class C(A): pass
class D(B, C): pass
print(D.__mro__)  # Output: (D, B, C, A, object)
```

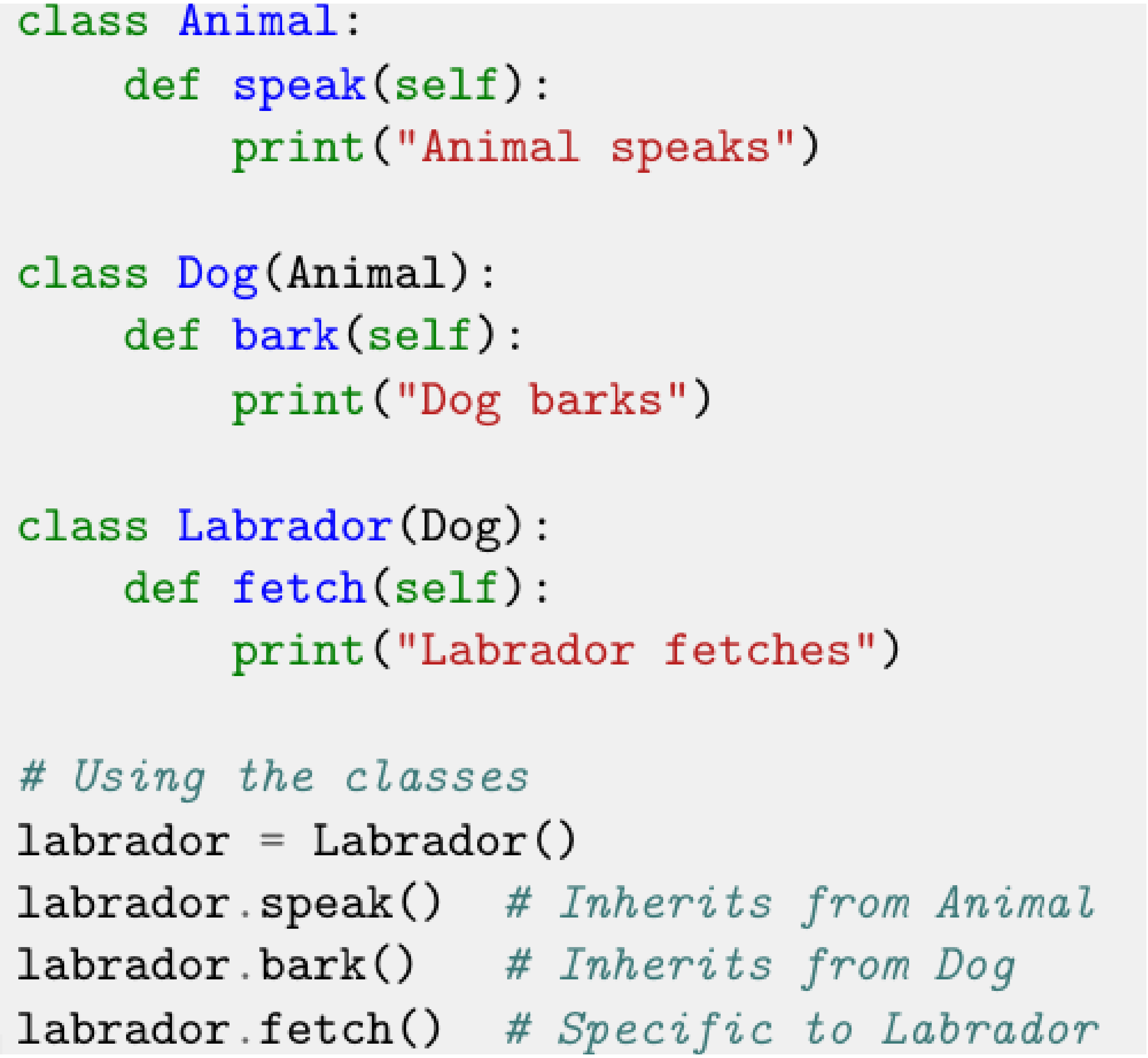


```
class Communication:
    def make_call(self):
        print("Making a call")
    def send_message(self):
        print("Sending a message")

class Computing:
    def run_application(self):
        print("Running application")
    def store_data(self):
        print("Storing data")

class PowerManagement:
    def handle_battery(self):
        print("Handling battery")
    def power_saving_mode(self):
        print("Entering power saving mode")
```

- **Exercise:**
  - Design a VIPCustomer class inheriting from Customer and VIPStatus.
  - Override calculate_discount() to include loyalty points.

- ## Multilevel Inheritance

Multilevel Inheritance refers to a scenario in object-oriented programming **where a derived class inherits from another derived class**, creating a hierarchy of three or more levels. It forms a "parent → child → grandchild" relationship, where each subsequent class adds new features while inheriting properties from its ancestors.



```python
class Animal:
    def speak(self):
        print("Animal speaks")

class Dog(Animal):
    def bark(self):
        print("Dog barks")

class Labrador(Dog):
    def fetch(self):
        print("Labrador fetches")

# Using the classes
labrador = Labrador()
labrador.speak()  # Inherits from Animal
labrador.bark()   # Inherits from Dog
labrador.fetch()  # Specific to Labrador
```

Control Execution with __name__:

```python
if __name__ == "__main__":
    # Code runs only when the file is executed directly
    print("Testing module.")
```

- **Abstract Base Classes (ABC)**
    - **Purpose:** Enforce method implementation in subclasses.
    - **Implementation:** Use abc module.

```
from abc import ABC, abstractmethod
class SmartDevice(ABC):
    @abstractmethod
    def calculate_energy(self, hours):
        pass
class SmartLight(SmartDevice):
    def calculate_energy(self, hours):
        return hours * 0.5  # 0.5 kWh per hour
```

Advanced OOP Techniques

Operator Overloading

- Customize behavior for operators like +, -, ==.

```
class Date:
    def __init__(self, day, month):
        self.day= day
        self.month = month
    def __add__(self,days):
        new_day = self.day + days
        return Date(new_day, self.month)
    def __eq__(self, other):
        return self.day == other.day and self.month == other.month
d1 = Date(9,11)
d2 = d1 + 4  # d2.day = 13
print(d1 == d2)  # False
```

Static and Class Methods

- **Static Methods:** Utility functions not tied to instances.

```
class Date:
    @staticmethod
    def is_valid(day, month):
        return 1 <= day <= 31 and 1 <= month <= 12
```

- **Class Methods:** Alternate constructors.

```
class Person:
    @classmethod
    def from_string(cls, data):
        name, age = data.split(",")
        return cls(name, int(age))
```

Lists of Objects

- **__repr__**: For readable object representation in lists.
- **__eq__**: To compare objects (e.g., for list.remove()).

```
class Age:
    def __eq__(self, other):
        return self.year == other.year and self.month == other.month

# Without __eq__:
listAges.remove(Age(12, 3))  # ValueError
 # With __eq__: Works!
```

Dictionaries of Objects

- **__hash__:** For objects to be used as keys.
- **__eq__:** For key comparison.

```
class Age:
    def __hash__ (self):
        return hash((self.year, self.month))

dictAges = {Age(12, 3): "Eligible"}
print(dictAges[Age(12, 3)]) # Output: "Eligible"
```

Polymorphism in Action

- **Override parent methods in subclasses.**

```
class Animal:
    def speak(self): print(“Generic sound”)
class Dog(Animal):
    def speak(self): print(“Bark!”)
```

- **Exercise:**
    - Overload > to compare Student objects by CGPA.
    - Implement __str__ for the Date class.