ENG 346 Data Structures and Algorithms for Artificial Intelligence

Object-Oriented Programming

Dr. Mehmet PEKMEZCİ

mpekmezci@gtu.edu.tr

https://github.com/mehmetpekmezci/GTU-ENG-346

What is a Class?

- A blueprint or template in object-oriented programming that defines the *structure* and *behavior* of objects.

- An abstraction to a key concept through its structure and behavior.

- Attributes: Data the class will be holding.
- Methods: Operations on the data

- Class encapsulates both data (attributes/properties) and behavior (methods/messages) making it easier to model and manipulate complex systems.

Example: Car Class

Class: Car

Attributes:
- Color
- Production Year
- Maximum Speed
- …

Methods:
- Accelerate()
- Break()
- Turn_on_the_lights()
- …

What is an Object?

- An object is an *instance* of a class.
  - Realization of a Class

- Objects represent real-world entities, concepts, or things in your program.

Example: My Car

Class: Car

Attributes:
- Color
- Production Year
- Maximum Speed
- …

Methods:
- Accelerate()
- Break()
- Turn_on_the_lights()
- …

Object: My Car

Attributes:
- Color: Black
- Production Year: 2010
- Maximum Speed: 180
- …

Methods:
- Accelerate()
- Break()
- Turn_on_the_lights()
- …

Object: Wife's Car

Attributes:
- Color: White
- Production Year: 2015
- Maximum Speed: 170
- …

Methods:
- Accelerate()
- Break()
- Turn_on_the_lights()
- …

Benefits of OOP

- Modularity: Helps to organize the code.
- Reusability: Encourages code reuse at the Class level.
- Abstraction: Abstract complex systems into simpler, high-level classes/objects.
- Encapsulation: Bundling data (attributes) and the methods (functions) that operate on that data into a single unit.
- Inheritance: Inherit attributes and methods from ancestors (parent classes).
- Polymorphism: Treat subclasses as if they are of one type, the Superclass.
- Scalability: Perform coding on large scale.
- Ease of Maintenance: Code organization and modularity makes it easy to debug, localize bugs, and perform maintenance.
- Team Collaboration: Work on individual classes without disturbing one another.
- Real-World Modeling: Natural way of thinking, design, and programming.

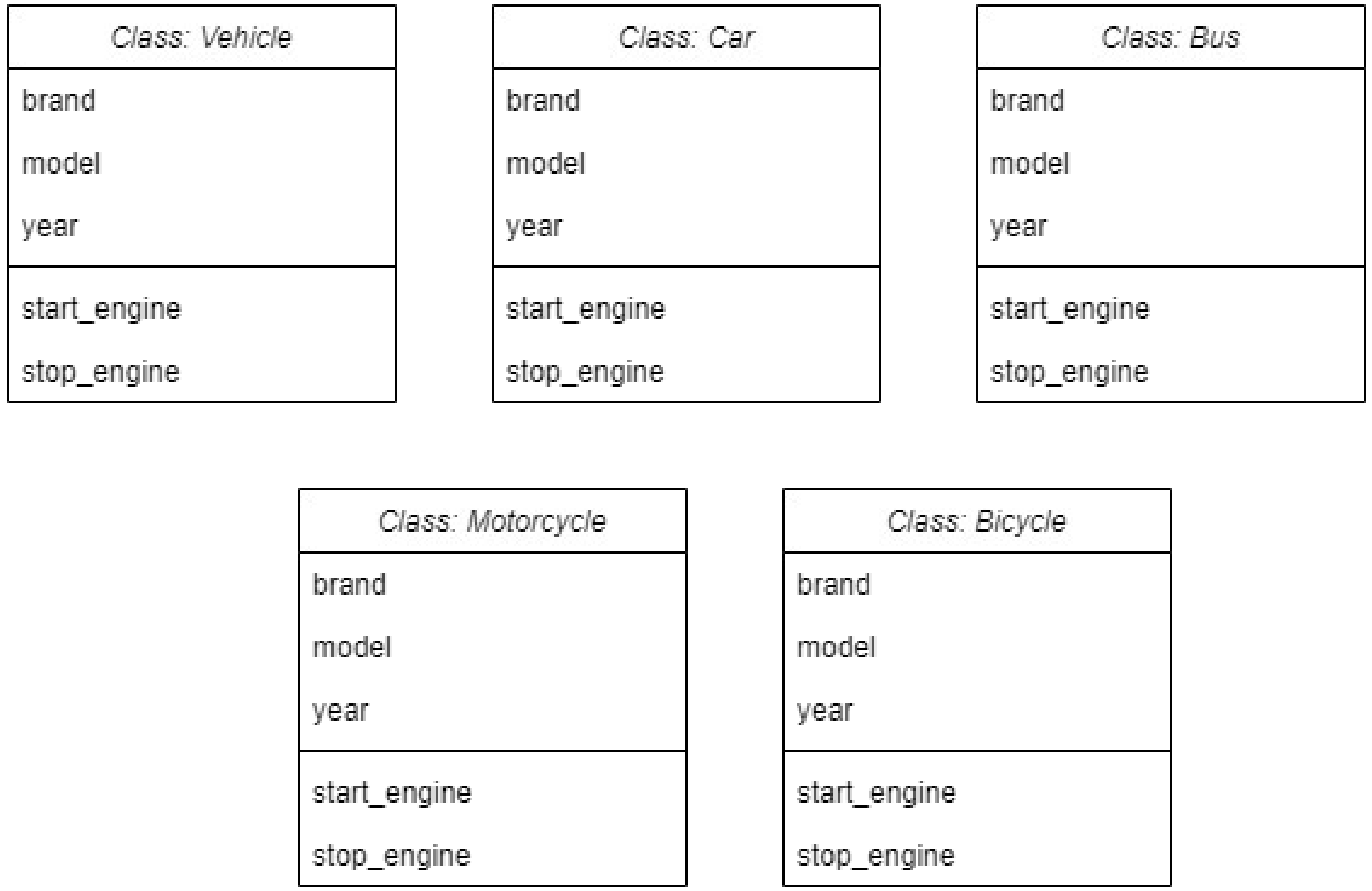
Abstraction

- Conceptualize real-world entities, their relevant attributes and behaviors.
- Focus is on essential characteristics of the while ignoring non-essential details.

- Example: in a car simulation, "Vehicle" class with attributes like speed and capacity, and methods like "accelerate" and "brake."

- Importance: Abstraction helps in managing complexity by allowing you to break down a system into smaller, more manageable parts. It also promotes code reusability and makes it easier to understand and maintain the software.

Abstraction – Example



Encapsulation

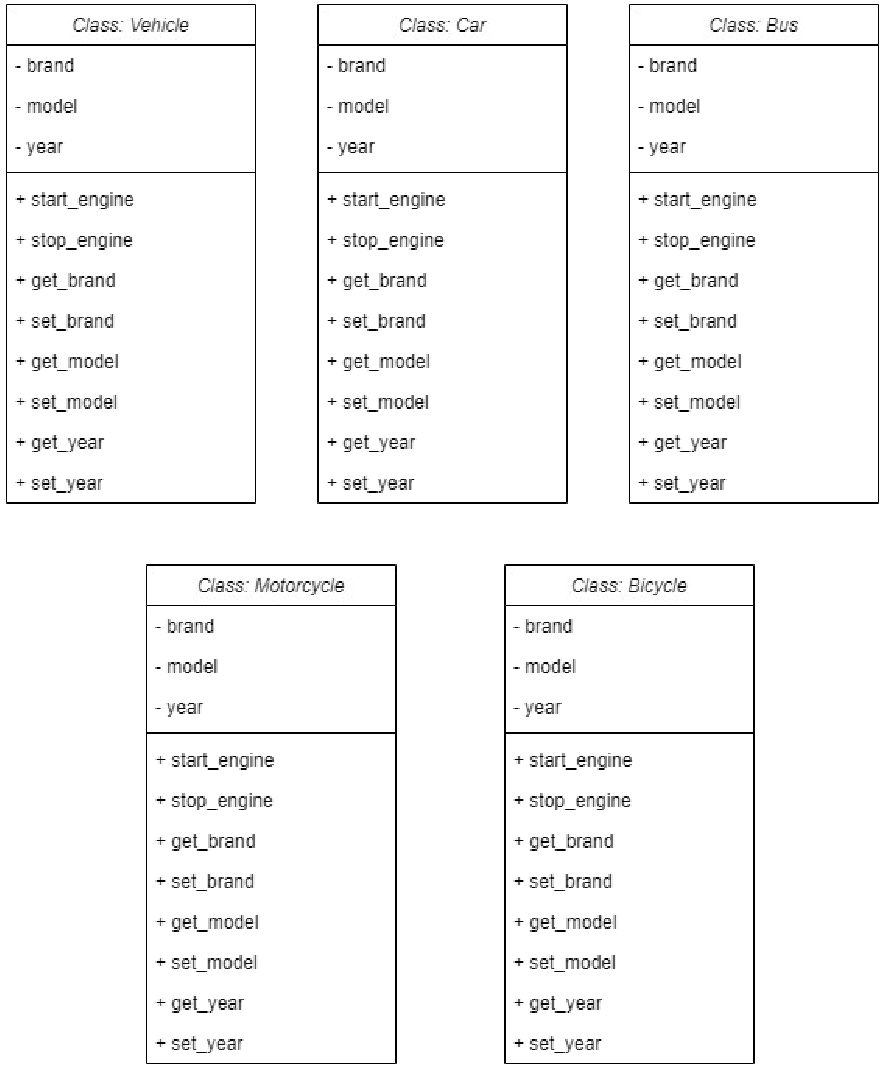
- Concept of bundling data (attributes/properties) and methods (functions/behaviors) that operate on that data into a Class.
- Encapsulation enforces data hiding, ensuring that data integrity is maintained and preventing unauthorized modifications.
  - Access to data and methods are restricted to the Class.
  - Any external access should be done through getters and setters methods.

- Access types:
  - Public: No access restrictions (default)
  - Protected: Can access from Class and its subclasses.
  - Private: Can access from Class only.

Encapsulation – continued

- Importance: Encapsulation enhances security and maintainability by preventing direct access to internal data, allowing for controlled and consistent interactions with objects. It also supports the principle of information hiding, making it easier to change the internal implementation of a class without affecting its users.

Encapsulation – Example

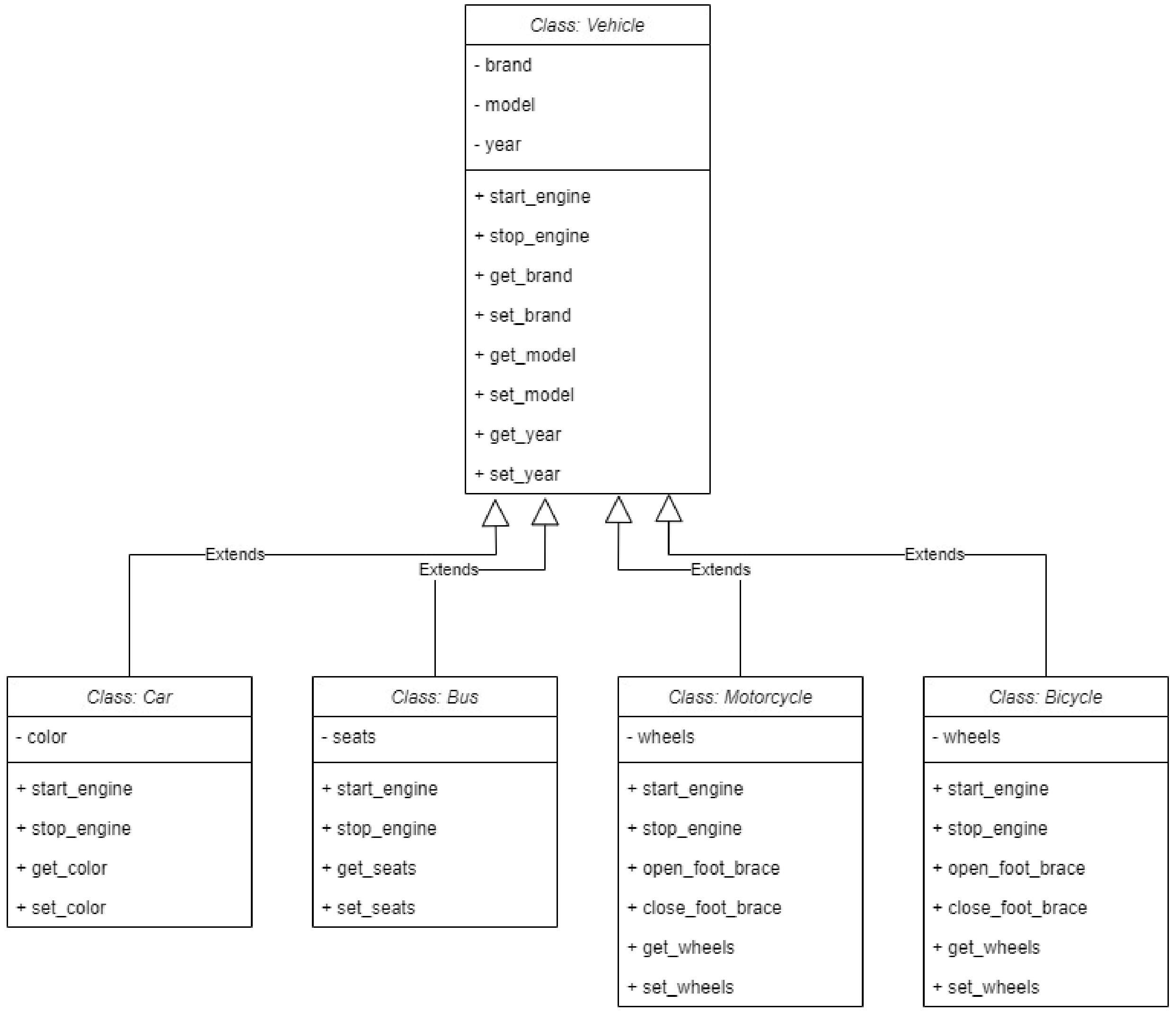


Inheritance

- Hierarchical extension mechanism in OOP.
- The subclass inherits attributes and behaviors from the superclass and can extend or override them.
- Inheritance facilitates code reuse and allows you to model hierarchical relationships between classes.

- Importance: Inheritance promotes code reusability, as you can create new classes that inherit features from existing classes, reducing code duplication. It also simplifies the organization of related classes in a hierarchy, making it easier to understand and maintain the code.

Inheritance – Example



Polymorphism

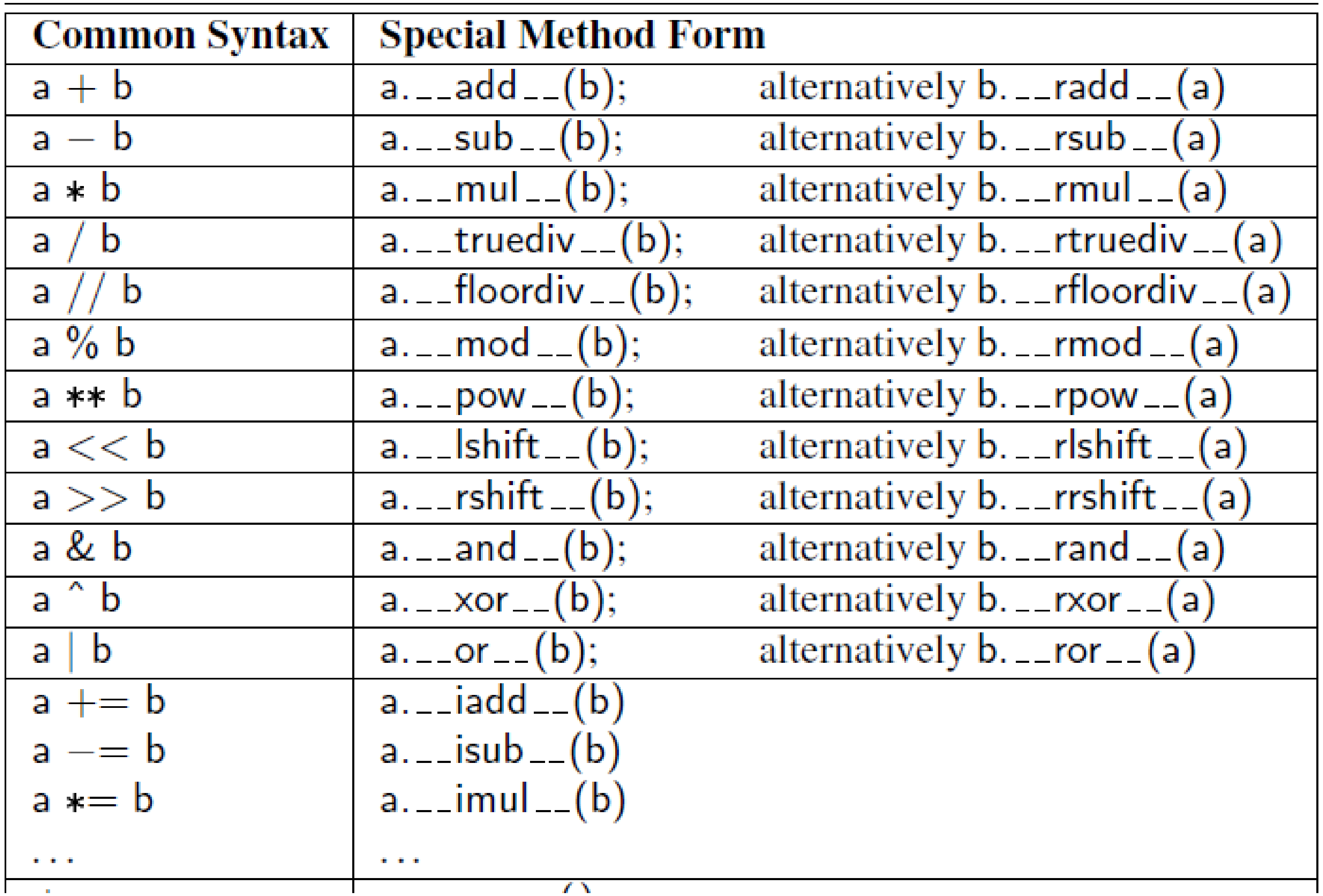
- Subclasses are be treated as *instances* of a common superclass.
- It allows objects of different classes to be accessed and manipulated through a common (Class) interface, often through inheritance and method overriding.

- Importance: Polymorphism enhances flexibility and extensibility in OOP. It simplifies code by allowing you to write generic algorithms that can work with various objects without knowing their specific types. This makes it easier to adapt and extend software as new classes are added, leading to more maintainable and scalable code.

Operator Overloading

- Overloading common operators (such as + operator) to extend the functionality of a (new) class.
- Example: Assume we are defining Complex Class and we want arithmetic operators to function on this new class.

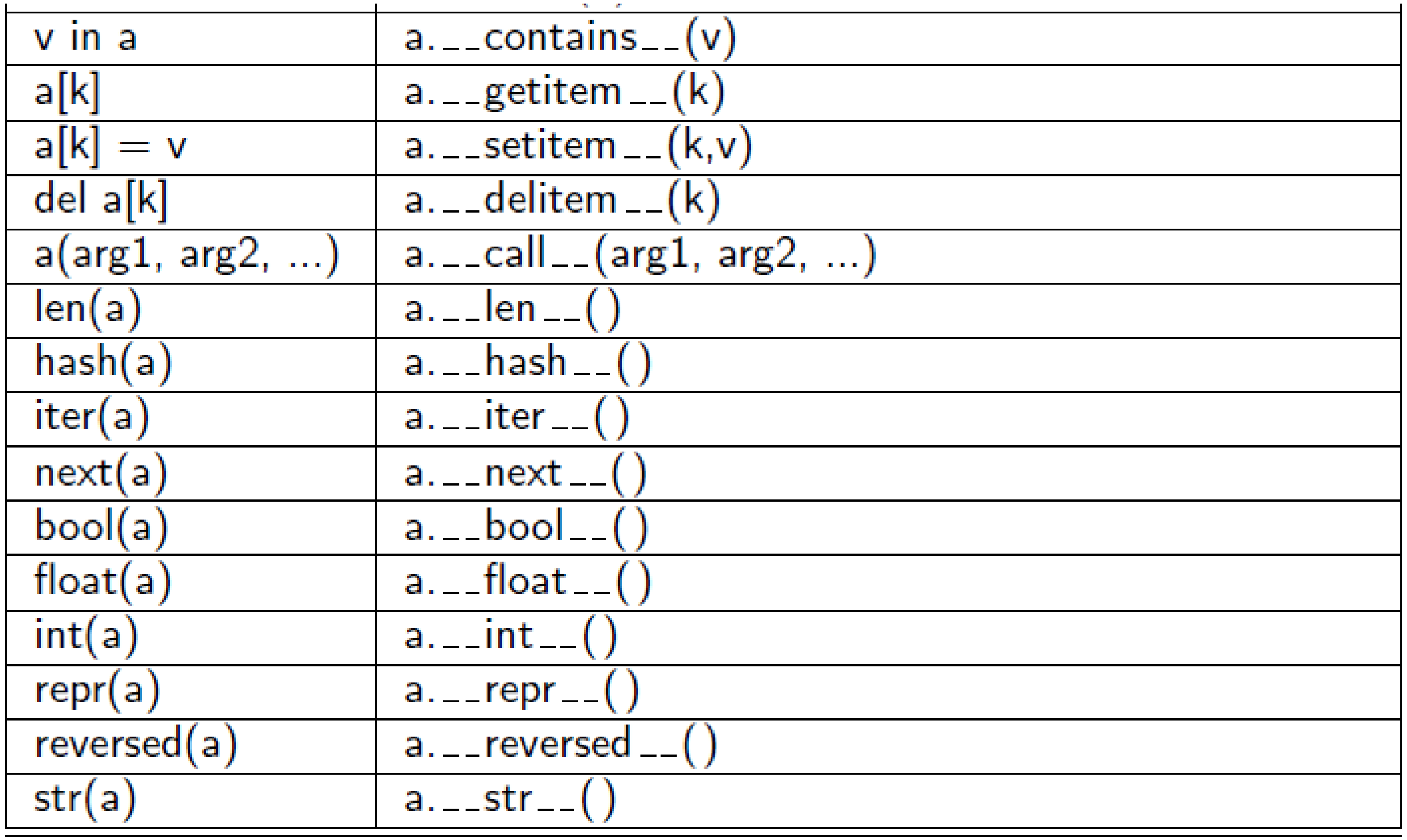
Operator Overloading – continued



Common Syntax	Special Method Form	
a + b	a.__add__(b);	alternatively b.__radd__(a)
a − b	a.__sub__(b);	alternatively b.__rsub__(a)
a ∗ b	a.__mul__(b);	alternatively b.__rmul__(a)
a / b	a.__truediv__(b);	alternatively b.__rtruediv__(a)
a // b	a.__floordiv__(b);	alternatively b.__rfloordiv__(a)
a % b	a.__mod__(b);	alternatively b.__rmod__(a)
a ∗∗ b	a.__pow__(b);	alternatively b.__rpow__(a)
a << b	a.__lshift__(b);	alternatively b.__rlshift__(a)
a >> b	a.__rshift__(b);	alternatively b.__rrshift__(a)
a & b	a.__and__(b);	alternatively b.__rand__(a)
a ^ b	a.__xor__(b);	alternatively b.__rxor__(a)
a | b	a.__or__(b);	alternatively b.__ror__(a)
a += b	a.__iadd__(b)	
a −= b	a.__isub__(b)	
a ∗= b	a.__imul__(b)	
...	...	

Non-Operator Overloads

- Overloading commonly used Python functions, such as str(), len(), etc.



v in a	a.__contains__(v)
a[k]	a.__getitem__(k)
a[k] = v	a.__setitem__(k,v)
del a[k]	a.__delitem__(k)
a(arg1, arg2, ...)	a.__call__(arg1, arg2, ...)
len(a)	a.__len__()
hash(a)	a.__hash__()
iter(a)	a.__iter__()
next(a)	a.__next__()
bool(a)	a.__bool__()
float(a)	a.__float__()
int(a)	a.__int__()
repr(a)	a.__repr__()
reversed(a)	a.__reversed__()
str(a)	a.__str__()

Named Variables in Function Definitions

- Example: Range Class

```
class RangeClass:
    def __init__ (self, start, stop=None, step=1):
```

Exercises

- Flower
- MessagingApp