Object Oriented Programming

Introduction

- A programmer can model real-world entities as objects for better program design and organization.
- A **class** defines a type of object with **attributes** and **methods**.
- Many instances of a class type can be created to represent multiple objects in a program.
- **Object-oriented programming** (OOP) is a style of programming that groups related fields, or data members, and procedures into objects.
- Real-world entities are modeled as individual objects that interact with each other.
  - Ex: A social media account can follow other accounts, and accounts can send messages to each other. An account can be modeled as an object in a program.

Introduction

- Object-Oriented Programming organizes code around objects (data) and the methods that operate on that data.
  - **Class:** a blueprint (e.g., Car)
  - **Object / Instance:** a concrete thing built from the blueprint (e.g., my_car)
  - **Attributes:** data stored on the object (e.g., brand, speed)
  - **Methods:** functions that belong to the class and use/modify its state (e.g., accelerate())

Introduction

- **Encapsulation:**
  - It is a key concept in OOP that involves wrapping data and procedures that operate on that data into a single unit.
  - Access to the unit's data is restricted to prevent other units from directly modifying the data.
- **Abstraction:**
  - It is a key concept in OOP in which a unit's inner workings are hidden from users and other units that don't need to know the inner workings.

Classes

- A class is an object that describes how its instances should look and behave.
- Technically, a class is itself an object (an instance of the metaclass type), with its own namespace
- Everything inside runs once at class creation time; the results are collected into the class's namespace.

Classes: Example

```python
class Car:
    def __init__(self, brand, model):
        self.brand = brand
        self.model = model
        self.speed = 0

    def accelerate(self, delta):
        self.speed += delta

    def info(self):
        return f"{self.brand} {self.model} @ {self.speed} km/h"

my_car = Car("Toyota", "Camry")
my_car.accelerate(30)
print(my_car.info())
```

Classes: Example

```
class Student:
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no
        self.attendance = 0

    def mark_present(self):
        self.attendance += 1

s1 = Student("Asha", 101)
s2 = Student("Vikram", 102)
s1.mark_present()
print(s1.attendance, s2.attendance)  # 1 0  (independent state)
```

Object / Instance

- You create an instance by calling the class like a function:

my_car = Car("Toyota", "Camry")

- What happens:
  - Python calls **Car.__new__(Car, ...)** to allocate the object.
  - Then calls **Car.__init__(self, ...)** to initialize it.
- Most of the time you only implement __init__.

Attributes

- An **instance attribute** is a variable that is unique to each instance of a class and is accessed using the format **instance_name.attribute_name.**
- Another type of attribute, a **class attribute**, belongs to the class and is shared by all class instances. Class attributes are accessed using the format **class_name.attribute_name.**

```python
class Car:
    wheels = 4                      # class attribute (shared)
    def __init__(self, brand):
        self.brand = brand          # instance attribute (per object)
```

```python
c1 = Car("Toyota")
c2 = Car("Honda")
print(c1.wheels, c2.wheels, Car.wheels)  # 4 4 4
c1.wheels = 3
print(c1.wheels, c2.wheels, Car.wheels)
```

```
4 4 4
3 4 4
```

Methods

- A **method** is just a function stored on a class.
- When accessed through an instance, Python’s **descriptor protocol** binds that function and supplies the instance as the first argument

```
class Student:
    def __init__(self, name, roll_no):
        self.name = name
        self.roll_no = roll_no
        self.attendance = 0

    def mark_present(self):
        self.attendance += 1

s1 = Student("Asha", 101)
s2 = Student("Vikram", 102)
s1.mark_present()
print(s1.attendance, s2.attendance)  # 1 0  (independent state)
```

Custom Classes

- Classes are created using the following syntax:

```
class className:
    suite

class className(base_classes):
    suite
```

- A class's methods are created using def statements in the class's suite.
- Class instances are created by calling the class with any necessary arguments

Custom Classes: Attributes and Methods

- Lets create a simple class point:
  - Has two attributes x and y coordinates
  - And a method to calculate the distance of the point from origin
- Assume that the class is defined in a separate python file Shape.py

```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return math.hypot(self.x, self.y)

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __repr__(self):
        return "Point({0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return "({0.x!r}, {0.y!r})".format(self)
```

Custom Classes: Attributes and Methods

```
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

def distance_from_origin(self):
    return math.hypot(self.x, self.y)

def __eq__(self, other):
    return self.x == other.x and self.y == other.y

def __repr__(self):
    return "Point({0.x!r}, {0.y!r})".format(self)

def __str__(self):
    return "({0.x!r}, {0.y!r})".format(self)
```

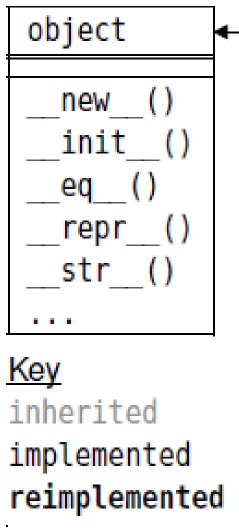
```
import Shape
a = Shape.Point()
repr(a)                      # returns: 'Point(0, 0)'
b = Shape.Point(3, 4)
str(b)                       # returns: '(3, 4)'
b.distance_from_origin()     # returns: 5.0
b.x = -19
str(b)                       # returns: '(-19, 4)'
a == b, a != b               # returns: (False, True)
```

Custom Classes: Attributes and Methods

- The Point class has two data attributes, **self.x** and **self.y**, and five methods
- Once the Shape module is imported, the Point class can be used like any other. The data attributes can be accessed directly (e.g., y = a.y)
- Python automatically supplies the first argument in method calls—it is an object reference to the object itself
- We must include this argument in the parameter list, and by convention the parameter is called **self**.
- *All object attributes (data and method attributes) must be qualified by self.*

Custom Classes: Attributes and Methods

- To create an object, two steps are necessary. First a raw or uninitialized object must be created, and then the object must be initialized, ready for use.
- When an object is created (e.g., p = Shape.Point()), first the special method __new__() is called to create the object, and then the special method __init__() is called to initialize it.



Custom Classes: Attributes and Methods

```
def __init__(self, x=0, y=0):
    self.x = x
    self.y = y

def distance_from_origin(self):
    return math.hypot(self.x, self.y)

def __eq__(self, other):
    return self.x == other.x and self.y == other.y

def __repr__(self):
    return "Point({0.x!r}, {0.y!r})".format(self)

def __str__(self):
    return "({0.x!r}, {0.y!r})".format(self)
```

```
import Shape
a = Shape.Point()
repr(a)                      # returns: 'Point(0, 0)'
b = Shape.Point(3, 4)
str(b)                       # returns: '(3, 4)'
b.distance_from_origin()     # returns: 5.0
b.x = -19
str(b)                       # returns: '(-19, 4)'
a == b, a != b               # returns: (False, True)
```

Custom Classes: Attributes and Methods

Special Method	Usage	Description
`__lt__(self, other)`	`x < y`	Returns `True` if `x` is less than `y`
`__le__(self, other)`	`x <= y`	Returns `True` if `x` is less than or equal to `y`
`__eq__(self, other)`	`x == y`	Returns `True` if `x` is equal to `y`
`__ne__(self, other)`	`x != y`	Returns `True` if `x` is not equal to `y`
`__ge__(self, other)`	`x >= y`	Returns `True` if `x` is greater than or equal to `y`
`__gt__(self, other)`	`x > y`	Returns `True` if `x` is greater than `y`

Inheritance

- **Inheritance:** It allows a **class** (called the **child class** or **derived class**) to **acquire the properties and behaviors** (attributes and methods) of another **class** (called the **parent class** or **base class**).

- *"Inheritance is the process of creating a new class (child) from an existing class (parent) so that the child class **inherits** the attributes and methods of the parent while allowing customization or extension of behavior."*

- **Why Use Inheritance?**
  - **Code Reusability** → No need to rewrite existing logic.
  - **Extensibility** → Extend or modify parent class functionality.
  - **Organized Code** → Creates a clear hierarchy between classes.
  - **Polymorphism Support** → Enables the same interface for different objects.

Inheritance

```python
class Animal:
    def eat(self):
        print("This animal eats food.")

# Child class inherits from Animal
class Dog(Animal):
    def bark(self):
        print("The dog barks.")

# Create objects
d = Dog()
d.eat()   # Inherited method from Animal
d.bark()  # Dog's own method
```

Inheritance

- **Single Inheritance** *(One parent → One child)*

```
class Parent:
    def greet(self):
        print("Hello from Parent!")

class Child(Parent):
    def display(self):
        print("Hello from Child!")

c = Child()
c.greet()   # Inherited
c.display() # Own method
```

Inheritance

- **Multiple Inheritance** *(One child → Multiple parents)*

```python
class Father:
    def skills(self):
        print("Father: Gardening")

class Mother:
    def skills(self):
        print("Mother: Cooking")

class Child(Father, Mother):
    def play(self):
        print("Child loves playing.")

c = Child()
c.skills()   # Follows Method Resolution Order (MRO)
c.play()
```

Inheritance

- **Multilevel Inheritance** *(Chain of inheritance)*

```python
class Grandparent:
    def feature(self):
        print("Grandparent's feature.")

class Parent(Grandparent):
    def ability(self):
        print("Parent's ability.")

class Child(Parent):
    def talent(self):
        print("Child's talent.")

c = Child()
c.feature()  # From Grandparent
c.ability()  # From Parent
c.talent()   # From Child
```

Inheritance

- **Hierarchical Inheritance** *(One parent → Multiple children)*

```python
class Parent:
    def display(self):
        print("This is the parent class.")

class Child1(Parent):
    def feature(self):
        print("Child1 feature.")

class Child2(Parent):
    def skill(self):
        print("Child2 skill.")

c1 = Child1()
c1.display()
c2 = Child2()
c2.display()
```

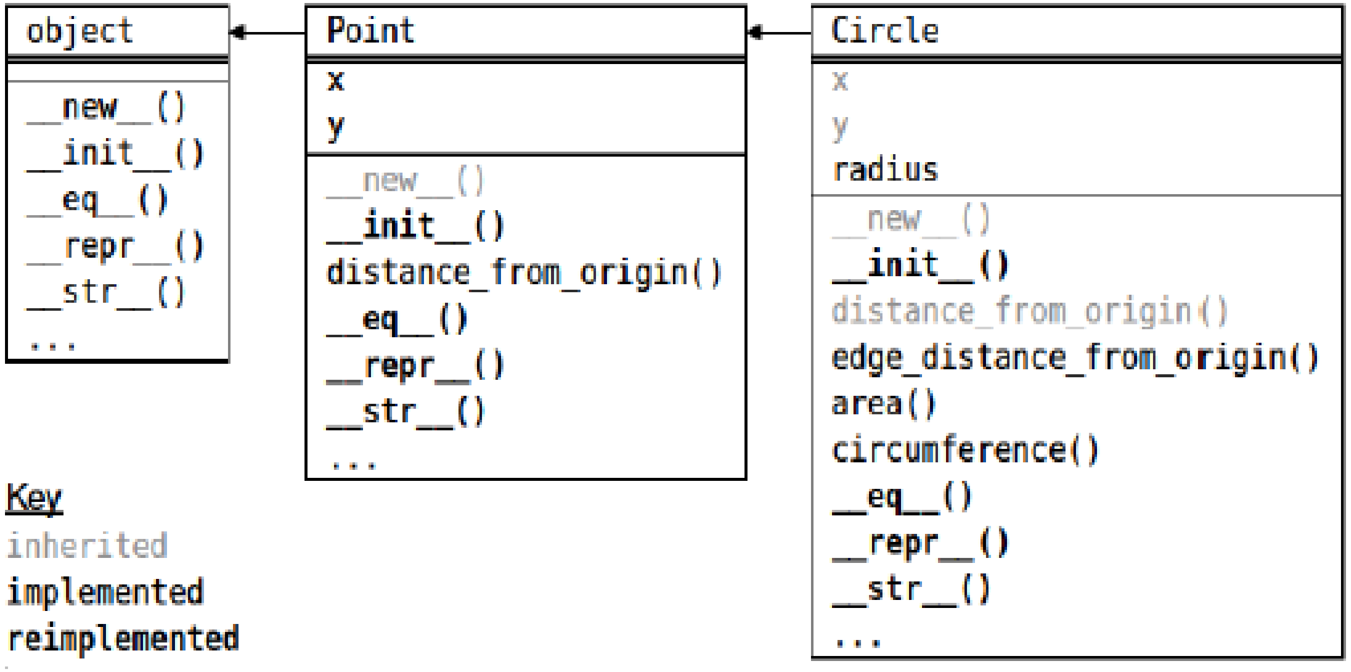
Inheritance

- **Hybrid Inheritance** *(Combination of multiple types)*

```python
class A:
    def featureA(self):
        print("Feature from A")

class B(A):
    def featureB(self):
        print("Feature from B")

class C(A):
    def featureC(self):
        print("Feature from C")

class D(B, C):  # Inherits from B and C
    def featureD(self):
        print("Feature from D")

d = D()
d.featureA()  # From A
d.featureB()  # From B
d.featureC()  # From C
```

Inheritance

- The Circle class builds on the Point class using inheritance. The Circle class adds one additional data attribute (radius), and three new methods.



```python
class Circle(Point):

    def __init__(self, radius, x=0, y=0):
        super().__init__(x, y)
        self.radius = radius

    def edge_distance_from_origin(self):
        return abs(self.distance_from_origin() - self.radius)

    def area(self):
        return math.pi * (self.radius ** 2)

    def circumference(self):
        return 2 * math.pi * self.radius

    def __eq__(self, other):
        return self.radius == other.radius and super().__eq__(other)

    def __repr__(self):
        return "Circle({0.radius!r}, {0.x!r}, {0.y!r})".format(self)

    def __str__(self):
        return repr(self)
```

Polymorphism

- Polymorphism in Python is an **object-oriented programming (OOP)** concept where the **same function or method behaves differently depending on the object or data type it is operating on**.
- "Polymorphism in Python is the ability of a single function, method, or operator to work in different ways depending on the context, such as the object type or data it operates on."
- Different types:
  - Method Overriding
  - Method Overloading
  - Operator Overloading
  - Duck Typing

Polymorphism: Method Overriding

- Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
- The method in the subclass **"overrides"** the method in the superclass.
- This is often referred to as **runtime polymorphism** because the decision of which method to call (the parent's or the child's) is made at runtime based on the object's type.
- In this example, both Dog and Cat objects have a method named speak(), but they exhibit different behaviors.
- The dog object calls the **speak()** method of the Dog class, while the cat object calls the **speak()** method of the Cat class. The same method name, different behavior.

```python
# Superclass
class Animal:
    def speak(self):
        raise NotImplementedError("Subclass must implement abstract method")

# Subclass 1
class Dog(Animal):
    def speak(self):
        return "Woof!"

# Subclass 2
class Cat(Animal):
    def speak(self):
        return "Meow!"

# Create objects
dog = Dog()
cat = Cat()

# Call the speak() method on each object
print(dog.speak())  # Output: Woof!
print(cat.speak())  # Output: Meow!
```

Polymorphism: Method Overloading

- Method overloading is the ability to define multiple methods within the same class that have the same name but differ in the number or type of their parameters.

- ***Python does not support true method overloading*** *in the same way as languages like Java or C++.*

- If you define two methods with the same name, the second definition will simply override the first one.

- However, Python achieves similar functionality using **default arguments** and **variable-length arguments**.

```python
class Calculator:
    def add(self, a, b, c=None):
        if c is not None:
            return a + b + c
        else:
            return a + b

# Create an object
calc = Calculator()

# Call the add() method with two arguments
print(calc.add(10, 20))    # Output: 30

# Call the add() method with three arguments
print(calc.add(10, 20, 30)) # Output: 60
```

Polymorphism: Operator Overloading

- Operator overloading allows you to redefine how standard operators (like +, -, *, ==) work for your custom objects.
- This is done by implementing special methods known as ***"dunder"*** (double underscore) methods.
- **__add__(self, other):** This method is called when you use the + operator.
  - It takes another object (other) and returns a new Vector object representing the sum.
- **__eq__(self, other):** This method is called when you use the == operator.
  - It defines the logic for checking if two Vector objects are equal.

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

    # Overload the addition operator (+)
    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    # Overload the equality operator (==)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    # A friendly string representation for printing
    def __repr__(self):
        return f"Vector({self.x}, {self.y})"

# Create two Vector objects
v1 = Vector(1, 2)
v2 = Vector(3, 4)

# Use the overloaded '+' operator
v3 = v1 + v2
print(v3)  # Output: Vector(4, 6)

# Use the overloaded '==' operator
print(v1 == v2)      # Output: False
v4 = Vector(1, 2)
print(v1 == v4)      # Output: True
```

Polymorphism: Duck Typing

- Duck typing is a concept in which the "type" or "class" of an object is less important than the methods it defines.
- The name comes from the saying: ***"If it walks like a duck and it quacks like a duck, then it must be a duck."***
- In Python, we don't care if an object is an instance of a specific class. Instead, we only care if it has the methods or attributes we need to call. This allows for extremely flexible and decoupled code.

```python
class Car:
    def move(self):
        print("The car is driving on the road.")

class Boat:
    def move(self):
        print("The boat is sailing on the water.")

class Plane:
    def fly(self):
        print("The plane is flying in the sky.")

def make_it_move(vehicle):
    """This function moves any object that has a 'move' method."""
    vehicle.move()
```

```python
# Create objects
my_car = Car()
my_boat = Boat()
my_plane = Plane()

# The function works with both Car and Boat objects
make_it_move(my_car)  # Output: The car is driving on the road.
make_it_move(my_boat) # Output: The boat is sailing on the water.

# This will cause an AttributeError because Plane has no 'move' method
try:
    make_it_move(my_plane)
except AttributeError as e:
    print(f"Error: {e}")
    # Output: Error: 'Plane' object has no attribute 'move'
```

Instance variables vs Class variables

- **Instance variables:** belong to a particular instance (self.attr). Different objects can have different values.
- **Class variables:** belong to the class itself and are shared across all instances unless overridden on the instance.
- school defined on the class is accessible via class and instances.
- Assigning **s1.school =** ... creates an instance attribute that shadows the class attribute for that instance only.

```python
class Student:
    school = "MIT"   # class variable (shared)

    def __init__(self, name):
        self.name = name  # instance variable (unique)


s1 = Student("Alice")
s2 = Student("Bob")

print(Student.school)  # MIT
print(s1.school)       # MIT (falls back to class variable)
print(s1.name)         # Alice

# modify class variable (affects all instances that haven't overridden it)
Student.school = "MAHE"
print(s2.school)       # MAHE

# override on instance only
s1.school = "LocalSchool"
print(s1.school)       # LocalSchool (instance attribute hides class attr)
print(s2.school)       # MAHE (still class-level)
```

Instance methods, class methods and static methods

- Defined normally inside a class.
- First parameter: self
- Operates on object instances.
- Can access and modify instance variables and class variables.
- Instance methods **need an object** to work because they depend on instance-specific data.

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

    # Instance method
    def display_info(self):
        print(f"Name: {self.name}, Marks: {self.marks}")

    def update_marks(self, new_marks):
        self.marks = new_marks

# Create object
s1 = Student("Alice", 85)
s1.display_info()

# Update marks using instance method
s1.update_marks(92)
s1.display_info()
```

Instance methods, class methods and static methods

- Defined with the decorator @classmethod.
- First parameter: cls (refers to the class itself, not the object).
- Works on the class level.
- Can access and modify class variables, but not instance variables directly.

```python
class Employee:
    company = "Google"  # Class variable

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

    @classmethod
    def change_company(cls, new_company):
        cls.company = new_company  # Modifies class variable for all objects

    def display(self):
        print(f"Employee: {self.name}, Company: {self.company}")

# Create objects
e1 = Employee("John")
e2 = Employee("David")

e1.display()
e2.display()

# Change company name for ALL employees
Employee.change_company("OpenAI")

e1.display()
e2.display()
```

Instance methods, class methods and static methods

- Static Methods (@staticmethod)
- Defined using the decorator @staticmethod.
- Does NOT take self or cls.
- Behaves like a normal function, but it belongs to a class. Cannot access or modify instance variables or class variables directly.
- When you want a **utility function** that **logically belongs to the class**, but does not depend on object or class state.

```python
class MathOperations:
    @staticmethod
    def add(a, b):
        return a + b

    @staticmethod
    def multiply(a, b):
        return a * b

# No object needed to call static methods
print(MathOperations.add(5, 3))       # 8
print(MathOperations.multiply(4, 6))  # 24
```

Instance methods, class methods and static methods

- **Instance method:** usual methods that take self and can access instance and class data.
- **Class method:** decorated with @classmethod and receives the class (cls) as first argument — useful for factory methods or modifying class state.
- **Static method:** decorated with @staticmethod; a plain function placed inside the class namespace — it neither receives self nor cls.

```python
class Employee:
    raise_percent = 1.05  # class variable

    def __init__(self, name, salary):
        self.name = name         # instance variable
        self.salary = salary

    def apply_raise(self):     # instance method
        self.salary *= Employee.raise_percent

    @classmethod
    def set_raise_percent(cls, amount):  # class method
        cls.raise_percent = amount

    @staticmethod
    def is_valid_name(name):   # static method (utility)
        return isinstance(name, str) and len(name) > 0

# using them
e = Employee("Gita", 50000)
print(Employee.is_valid_name("Gita"))  # True

Employee.set_raise_percent(1.10)  # update class-level raise percent
e.apply_raise()
print(e.salary)  # 55000.0
```

Instance methods, class methods and static methods

- apply_raise uses instance data and the class variable.
- set_raise_percent updates class-level data for all employees (unless overridden per instance).
- is_valid_name is utility logic that logically belongs to the class domain but does not need instance/class state.

Feature	Instance Method	Class Method	Static Method
Decorator	None	@classmethod	@staticmethod
First Parameter	self	cls	None
Access Instance Variables	Yes	No	No
Access Class Variables	Yes	Yes	No
Called Using	Object	Class or Object	Class or Object
Use Case	Object-specific operations	Modify class-level data	Utility/helper functions

Encapsulation (Data Hiding & Access Control)

- Encapsulation groups data (attributes) and behavior (methods) inside a class and *controls* how external code interacts with that data.
- Public attributes/methods:
  - normal names with no leading underscore.
  - Meant for public use
  - Example: obj.name, obj.do_something()
- Protected (convention):
  - single leading underscore _attr.
  - Indicates “internal use” (not enforced).
  - Subclasses and module code can access it, but you should treat it as non-public.
  - Example: _cache, _load_from_db()
- Private (name mangling):
  - double leading underscore __attr (but not double trailing underscores).
  - Python rewrites the name to include the class name — this prevents accidental overrides and name collisions in subclasses.
  - It’s still accessible (via the mangled name) if needed.
  - Example: __balance becomes _BankAccount__balance internally.

Encapsulation (Data Hiding & Access Control)

- Encapsulation is one of the core principles of object-oriented programming (OOP).
- It is the mechanism of bundling data (attributes) and the methods that operate on that data into a single unit (a class).
- It also involves controlling the visibility of that data, protecting it from accidental modification.
- It keeps the data safe inside the class and only allows access through specific, controlled methods. This concept is often called **data hiding**.
- Python handles encapsulation differently from languages like Java or C++ which use keywords like public, protected, and private.

Encapsulation (Data Hiding & Access Control)

- In Python, the level of data protection is indicated by naming conventions using underscores.
- **Public Attributes (Normal Variables)**
  - **Convention:** No leading underscore.
  - **Access:** Can be accessed and modified from anywhere, both inside and outside the class.
  - **Behavior:** Public attributes are the default in Python. They are meant to be a part of the class's public interface.

```python
class Student:
    def __init__(self, name, age):
        self.name = name   # Public attribute
        self.age = age     # Public attribute

# Create an object
s1 = Student("Alice", 20)

# Accessing and modifying public attributes
print(s1.name)  # Output: Alice
print(s1.age)   # Output: 20

# We can directly modify them, which can be a risk
s1.age = 21
print(s1.age)   # Output: 21
```

Encapsulation (Data Hiding & Access Control)

- **Protected Attributes**
  - **Convention:** A single leading underscore (e.g., _protected_var).
  - **Access:** Can be accessed from inside the class and its subclasses, but should not be accessed directly from outside.
  - **Behavior:** This is a convention for developers. It signals that a variable is intended for internal use and should be treated as non-public.
  - Python does not strictly enforce this; you can still access it from outside, but it is considered bad practice.

```python
class Car:
    def __init__(self, make, model):
        self.make = make          # Public
        self._model = model       # Protected (convention)

    def display_info(self):
        print(f"Make: {self.make}, Model: {self._model}")

# Create an object
my_car = Car("Toyota", "Corolla")

# Accessing via a method (good practice)
my_car.display_info()  # Output: Make: Toyota, Model: Corolla

# Accessing directly from outside (possible, but not recommended)
print(my_car._model)   # Output: Corolla
```

Encapsulation (Data Hiding & Access Control)

- **Private Attributes**
  - **Convention:** Two leading underscores (e.g., __private_var).
  - **Access:** Strictly private. Cannot be accessed directly from outside the class or its subclasses.
  - **Behavior:** Python implements a feature called name mangling to make these attributes inaccessible from the outside.
  - When the interpreter sees a name with two leading underscores, it automatically renames it to _ClassName__varname.

```python
class BankAccount:
    def __init__(self, account_holder, balance):
        self.account_holder = account_holder
        self.__balance = balance  # Private attribute

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited {amount}. New balance: {self.__balance}")
        else:
            print("Deposit amount must be positive.")
```

```python
# Create an object
account = BankAccount("John Doe", 1000)

# Accessing a public attribute is fine
print(account.account_holder)  # Output: John Doe

# Trying to access the private attribute directly will fail
try:
    print(account.__balance)
except AttributeError as e:
    print(f"Error: {e}")
    # Output: Error: 'BankAccount' object has no attribute '__balance'

# Accessing via a public method (the correct way)
print(account.get_balance()) # Output: 1000

# Using a public method to modify the private attribute
account.deposit(500)
# Output: Deposited 500. New balance: 1500
```

Encapsulation (Data Hiding & Access Control)

- **Name Mangling (_ClassName__var)**
  - As mentioned, Python's "private" attributes are not truly private;
  - they are just renamed by the interpreter to prevent direct access.
  - This renaming process is called name mangling.
  - The name __balance in BankAccount is automatically changed to _BankAccount__balance.
- While you *can* access the private attribute using its mangled name, this is a clear violation of the encapsulation principle.
- It's a hack and should be avoided. The purpose of name mangling is to make accidental access very difficult, not impossible.

```python
class MyClass:
    def __init__(self):
        self.public_var = "I'm public"
        self._protected_var = "I'm protected"
        self.__private_var = "I'm private"

# Create an object
obj = MyClass()

# Accessing the mangled name (possible, but bypasses encapsulation)
print(obj._MyClass__private_var)  # Output: I'm private
```

```python
class Person:
    public_name = "public"        # public class attribute
    _protected_info = "hidden"    # protected-by-convention
    __private_note = "secret"     # private (name-mangled)

    def __init__(self, name):
        self.name = name          # public instance attribute
        self._nickname = name[:3]  # protected-by-convention
        self.__ssn = "000-00-0000" # private instance attribute

p = Person("David")

# Public access
print(p.name)            # David

# Protected – allowed but indicates internal use
print(p._nickname)       # Dav

# Private – direct attribute name raises AttributeError
try:
    print(p.__ssn)
except AttributeError as e:
    print("AttributeError:", e)

# But name mangling reveals it:
print("Private via mangled name:", p._Person__ssn)

# Class-level private
try:
    print(Person.__private_note)
except AttributeError as e:
    print("AttributeError:", e)

print("Class private via mangled name:", Person._Person__private_note)
```

```python
class BankAccount:
    def __init__(self, initial_balance=0):
        # private attribute (name-mangled)
        self.__balance = float(initial_balance)

    def deposit(self, amount):
        if amount <= 0:
            raise ValueError("Deposit must be positive")
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= 0:
            raise ValueError("Withdraw must be positive")
        if amount > self.__balance:
            raise ValueError("Insufficient funds")
        self.__balance -= amount

    def get_balance(self):      # public accessor
        return self.__balance
```

```python
# Usage
acct = BankAccount(100)
acct.deposit(50)
try:
    acct.withdraw(200)      # Insufficient funds
except ValueError as e:
    print("Error:", e)

print("Balance:", acct.get_balance())

# Attempt to cheat by writing acct.__balance = 1
#(creates new attribute and doesn't change private one)
acct.__balance = 1
print("acct.__balance (shadow):", acct.__balance)
print("Actual (private) balance:", acct.get_balance())
print("Private via mangled:", acct._BankAccount__balance)
```

Concept	Definition	When It Happens	Example
Method Overriding	Same method name, different behavior in child class	Runtime	Parent/Child sound()
Method Overloading	Same method name, different params (**not natively supported**)	Compile-time (simulated)	add(a=0, b=0, c=0)
Operator Overloading	Customizing operators via dunder methods	Runtime	__add__, __eq__
Duck Typing	Focus on **behavior**, not object type	Runtime	Any object with fly()

Abstraction (Hiding Implementation Details)

- Abstraction is a core principle of object-oriented programming (OOP) that focuses on hiding complex implementation details and showing only the essential features of an object.
- It provides a blueprint for classes, defining what methods a class *must* have without specifying how those methods should be implemented.
- **Abstraction** is the process of defining the required functionality of a class without worrying about the specifics of its implementation. This is achieved by creating an **abstract class**, which cannot be instantiated on its own and serves as a template for other classes.
- **Why is it useful?**
  - **Enforces a Standard:** Abstraction ensures that all subclasses conform to a required structure. If a subclass is created from an abstract base class, it *must* implement all the abstract methods defined in the base class, or it will raise an error. This guarantees a consistent API.
  - **Reduces Complexity:** It simplifies the user's interaction with the program. By exposing only the necessary functionality, it prevents the user from being overwhelmed by intricate details.
  - **Encourages Modularity:** It allows you to design your code in a highly modular way, where different components can be swapped out easily as long as they adhere to the same abstract interface.

Abstraction (Hiding Implementation Details)

- **abc module (Abstract Base Classes)**
  - In Python, you create abstract classes using the built-in abc (Abstract Base Classes) module.
  - An abstract class is a class that inherits from ABC from the abc module.
  - It can contain one or more abstract methods.
  - You cannot create an object directly from an abstract class
  - @abstractmethod: This is a decorator used to declare a method as abstract.
  - Any class inheriting from an abstract base class must provide an implementation for all methods decorated with @abstractmethod.

```python
from abc import ABC, abstractmethod

# Define the abstract base class 'Shape'
class Shape(ABC):
    @abstractmethod
    def area(self):
        """Calculates the area of the shape."""
        pass  # Abstract method has no implementation

    @abstractmethod
    def perimeter(self):
        """Calculates the perimeter of the shape."""
        pass  # Abstract method has no implementation

# --- Trying to instantiate the abstract class (This will fail) ---
try:
    s = Shape()
except TypeError as e:
    print(f"Error: {e}")
    # Output: Error: Can't instantiate abstract class
    #Shape with abstract methods area, perimeter
```

Abstraction (Hiding Implementation Details)

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

# Concrete class inheriting from Shape
class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    # Must implement the 'area' method
    def area(self):
        return self.length * self.width

    # Must implement the 'perimeter' method
    def perimeter(self):
        return 2 * (self.length + self.width)
```

```python
# Another concrete class
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

    def perimeter(self):
        return 2 * 3.14 * self.radius

# --- Usage of the concrete classes ---
rect = Rectangle(10, 5)
print(f"Rectangle Area: {rect.area()}")           # Output: Rectangle Area: 50
print(f"Rectangle Perimeter: {rect.perimeter()}")  # Output: Rectangle Perimeter: 30

circle = Circle(7)
print(f"Circle Area: {circle.area()}")           # Output: Circle Area: 153.86
print(f"Circle Perimeter: {circle.perimeter()}")  # Output: Circle Perimeter: 43.96
```

Abstraction (Hiding Implementation Details)

```python
class Square(Shape):
    def __init__(self, side):
        self.side = side

    # We forgot to implement 'perimeter'!
    def area(self):
        return self.side * self.side

# --- This will fail because 'perimeter' is not implemented ---
try:
    sq = Square(4)
except TypeError as e:
    print(f"Error: {e}")
    # Output: Error: Can't instantiate abstract class Square with abstract method perimeter
```

Abstraction (Hiding Implementation Details)

- Abstraction provides the external blueprint.
- It says, "Any Shape will have an area() method.“
- Encapsulation provides the internal protection.
- A Rectangle's area() method might use private attributes like __length and __width to perform its calculation.
- The user of the Rectangle object doesn't need to know about these private attributes; they only interact with the area() method.