Object Oriented Programming

Object-oriented programming (referred to as OOP) is another popular paradigm as it is applicable to certain types of problem with lots of reusable components which have similar characteristics. OOP is built on **entities called objects formed from classes which have certain attributes and methods.** OOP focuses on making programs that are reusable and easy to update and maintain.

It relies on the concept of classes and objects.

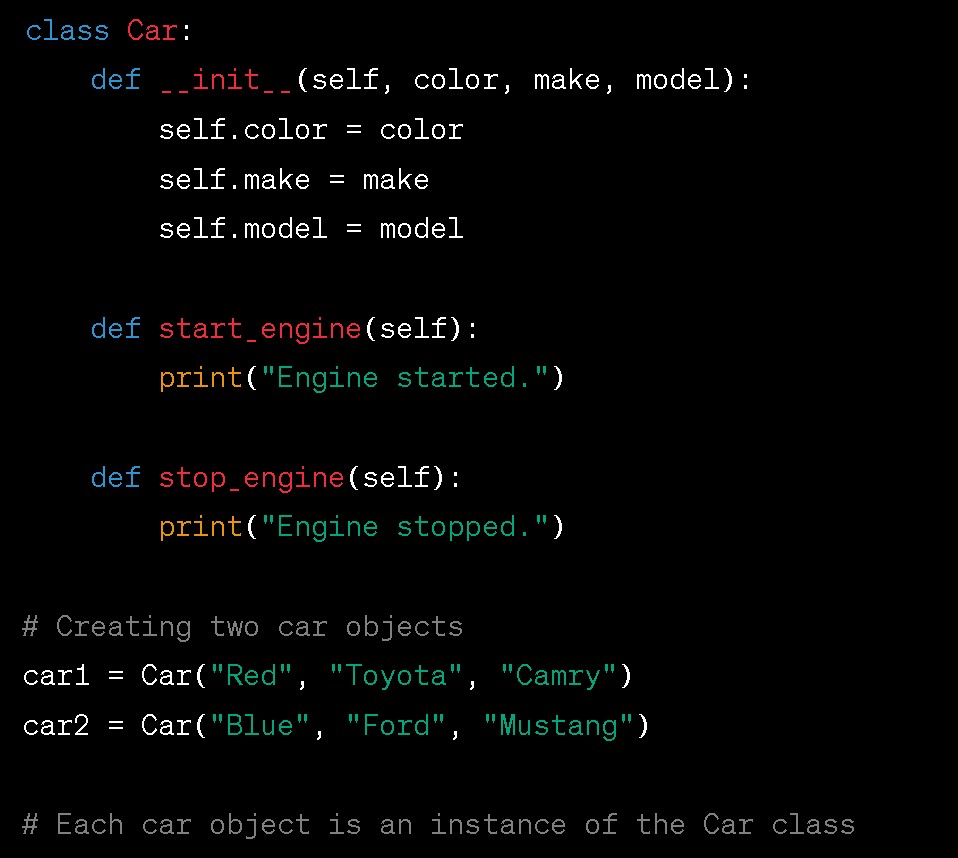
Diagram

Description automatically generatedIt is used to structure a software program into simple, reusable pieces of code blueprints (usually called classes), which are used to create individual instances of objects.

The features of OOP

Object

An object in Object-Oriented Programming (OOP) is like a thing or an item that represents something specific. It could be an object from the real world, like a car or a person, or it could represent something more abstract, like an idea or a concept. It is an instance of a class and represents a specific entity or concept in the system. For example, if you have a class called "Car," an object of that class could be a specific car with its own unique.

In this example, *car1* and *car2* are two different objects created from the "Car" class. They each have their own set of attributes (color, make, model) and can perform actions like starting and stopping the engine independently

Think of an object as a container that holds two important things:

1. Data: An object has some data associated with it, which represents its characteristics or properties. For example, if we have an object representing a car, the data **(attributes)** associated with it and could include the car's color, make, model, and year.
2. Behavior: An object also has some behavior or actions it can perform. These behaviors are defined as **methods** in OOP. For example, a car object might have methods like "start," "accelerate," and "brake" to represent the actions that a car can perform

Properties/Attributes: Properties, also known as attributes or fields, are the characteristics or data associated with an object. They represent the state or the data that an object holds. For example, a car object may have properties such as color, make, model, and year.

Methods: Methods are the functions or behaviors associated with an object. They define the operations that an object can perform. For example, a car object may have methods such as start(), accelerate(), and brake().

**Classes**

A class is like a blueprint or a template that describes how to create objects. It defines the structure and behavior that objects of that class will have. A class represents a **real life concept** such as a person, an animal, a car, a house or a football.

For example, let's say you have a class called "Car". The Car class would define what a car should have, such as properties like color, make, model, and year, and methods like start, accelerate, and brake.

Once you have defined the Car class, you can create **individual car objects**, also **known as instances**, based on that class. Each car object will have its own unique data for the properties (e.g., a red Ford Mustang from 2020) and can perform the actions defined by the methods (e.g., starting the engine, accelerating, and braking).

An instance of the Class “’car would be an object’.

Creating two instances of the Class car

car1 = Car("Red", "Toyota", "Camry")

car2 = Car("Blue", "Ford", "Mustang")

A class in OOP is like a blueprint or a recipe that describes the structure and behaviour of objects. It defines the properties/data (attributes/characteristics) and methods (actions) that objects of that class will have. When you create an object from a class, you are following the blueprint and building a specific instance of that object.

Encapsulation

Encapsulation is the concept of bundling data (attributes/properties) and the operations (methods) that manipulate that data within a class. It allows for data hiding and this protects the data from being directly accessed or modified from outside the class, ensuring that it remains consistent and valid. Access to the data is controlled through methods, which are like the buttons on the toy, providing a controlled way to interact with the object's internal state.

class Toy:  
 def\_ \_init\_ \_(self):

self.\_button\_hidden\_inside = "Press me!"

def play(self):

print("You can press the hidden button to play:", self.\_button\_hidden\_inside)

my\_toy = Toy() **# Creating a toy object**

my\_toy.play() **# You can play with the toy by pressing its hidden button**

**# You cannot access the hidden button directly**

**# This demonstrates encapsulation - the button is hidden and protected from direct access**

**# You can only interact with it through the "play" method.**

print(my\_toy.\_button\_hidden\_inside) **# This is not a recommended practice in Python**

In the example above, the **\_button\_hidden\_inside** attribute is encapsulated within the Toy class. You can only interact with it through the play method, ensuring that the internal state is protected and accessed in a controlled manner.

Abstraction

Abstraction is the process of simplifying complex systems by focusing on the essential features while hiding unnecessary details. In OOP, abstraction is achieved through abstract classes and interfaces. It lets you create classes and objects that provide a clear and simplified interface for users, while the underlying complexities are hidden from view. This simplification makes it easier to work with objects and reduces the complexity of your code.

They provide a way to interact with objects without worrying about their internal complexities.

class Car:  
 def \_\_init\_\_(self, make, model):

self.make = make

self.model = model

def start(self):

print(f"{self.make} {self.model} is starting.")

def drive(self):

print(f"{self.make} {self.model} is now driving.")

# Creating a car object

my\_car = Car("Toyota", "Camry")

# Using abstraction to interact with the car without knowing its internal details

my\_car.start()

my\_car.drive()

In this example, you don't need to know how the start and drive methods of the Car class are implemented or what's happening under the hood. You only need to know that you can call these methods to start and drive the car. This is abstraction in action, simplifying your interaction with objects by hiding the underlying complexity.

Creating

# Abstract base class for all vehicles

class Vehicle:

def \_\_init\_\_(self, make, model):

self.make = make

self.model = model

def start(self):

pass # This method will be implemented in subclasses

def stop(self):

pass # This method will be implemented in subclasses

# Concrete class for Cars

class Car(Vehicle):

def start(self):

print(f"{self.make} {self.model} is starting with a key.")

def stop(self):

print(f"{self.make} {self.model} is stopping and turning off the engine.")

# Concrete class for Bicycles

class Bicycle(Vehicle):

def start(self):

print(f"{self.make} {self.model} is starting by pedaling.")

def stop(self):

print(f"{self.make} {self.model} is stopping by applying brakes.")

# Concrete class for Motorcycles

class Motorcycle(Vehicle):

def start(self):

print(f"{self.make} {self.model} is starting with a kick-start.")

def stop(self):

print(f"{self.make} {self.model} is stopping and turning off the engine.")

# Creating instances of different vehicles

my\_car = Car("Toyota", "Camry")

my\_bicycle = Bicycle("Schwinn", "Mountain Bike")  
my\_motorcycle = Motorcycle("Harley-Davidson", "Sportster")

# Using abstraction to start and stop the vehicles without knowing their details

my\_car.start()

my\_car.stop()

my\_bicycle.start()

my\_bicycle.stop()

my\_motorcycle.start()

my\_motorcycle.stop()

In the example above, we have an abstract base class Vehicle with common methods start and stop. Each concrete subclass (Car, Bicycle, and Motorcycle) inherits from Vehicle and provides its own implementation for these methods. When we create instances of these classes, we can use the same interface (start and stop) to interact with them, regardless of the specific vehicle type. This demonstrates abstraction, as we're focusing on the common functionality while hiding the differences in implementation between the vehicle types.

Inheritance

Inheritance allows a new class (called a subclass or derived class) to inherit properties and behaviors (attributes and methods) from an existing class (called a superclass or base class).

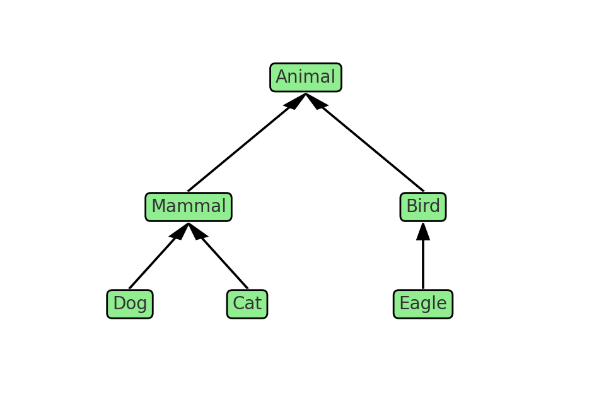
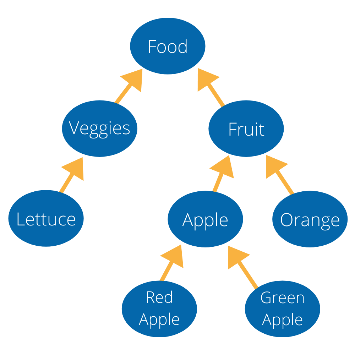
It is like passing down traits or characteristics from one thing to another. It allows you to create new classes based on existing classes, inheriting their properties and behaviors. just like a parent who has specific traits like eye color, hair type, and height. The children would inherit some of these traits from the parent.

Similarly, in OOP, you can create a new class (child class or subclass) that inherits attributes and behaviors from an existing class (parent class or superclass).

Inheritance enables the creation of a hierarchical relationship between classes, where the subclass inherits the characteristics of the superclass. This means that the subclass automatically has access to all the attributes and methods defined in the superclass, without the need to redefine them.

The primary benefits of inheritance include

* code reuse (since common functionality can be defined in a superclass and reused in multiple subclasses),
* organization of code into logical hierarchies,
* the ability to model real-world relationships and specialization.
* allows developers to build upon existing classes and extend their functionality, making it a powerful tool for structuring and designing object-oriented software systems.





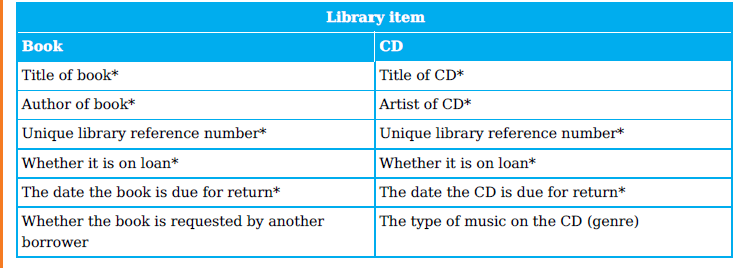
Example of inheritance

**Implementing a library system**

Consider the following problem.

A college library has items for loan.

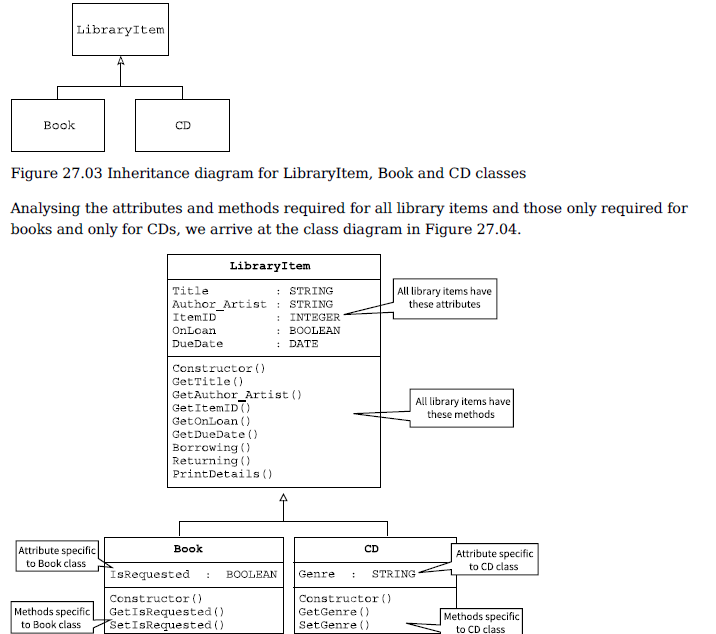
* The items are currently books and CDs.
* Items can be borrowed for three weeks.
* If a book is on loan, it can be requested by another borrower.



The information to be stored about books and CDs needs further analysis. Note that we could have a variable Title, which stores the book title or the CD title, depending on which type of library item we are working with.

There are some items of data that are different for books and CDs. Books can be requested by a borrower. For CDs, the genre is to be stored.

We can define a class LibraryItem and derive a Book class and a CD class from it. We can draw the inheritance diagrams for the LibraryItem, Book and CD classes



Polymorphism

Polymorphism in simple terms means "many shapes" or "many forms." In programming, it refers to the ability of different objects to respond to the same function or method call in their own unique way.

his means you can write code that works with objects of different classes as if they were objects of a single class. These different objects can all be treated as instances of the same class, and they can use the same interface (methods or functions) to perform their actions, but each class will provide its own specific behavior for those interfaces. This makes it easier to write general code for a wide range of scenarios without knowing the exact details of every class involved.

Imagine you have a remote control (the function) that can control different devices like a TV, a radio, and a fan (the objects). Even though you press the same button (invoke the same method), each device responds differently: the TV changes channels, the radio adjusts its volume, and the fan changes speed.

Example of polymorphism in Python using classes and methods:

class Animal:

def speak(self):

raise NotImplementedError("Subclass must implement abstract method")

class Dog(Animal):

def speak(self):

return "Woof!"

class Cat(Animal):

def speak(self):

return "Meow!"

class Cow(Animal):

def speak(self):

return "Moo!"

def animal\_speak(animal):

print(animal.speak())

# Creating instances of the classes

dog = Dog()

cat = Cat()

cow = Cow()

# Demonstrating polymorphism

for animal in [dog, cat, cow]:

animal\_speak(animal)

* We have a base class Animal with a method speak that is not implemented (it raises a NotImplementedError).
* We have three subclasses: Dog, Cat, and Cow, each of which overrides the speak method to return their respective sounds.
* The function animal\_speak takes an Animal object and prints the result of its speak method.
* When we create instances of Dog, Cat, and Cow and pass them to the animal\_speak function, each object invokes its own version of the speak method, demonstrating polymorphism.

Despite the different implementations of speak in each subclass, the interface (i.e., the method name speak) is consistent across all classes. This is the essence of polymorphism in OOP, allowing different objects to be used interchangeably while still behaving appropriately for their respective types.

OOP groups together data structure and subroutines that operate on the data items in the data structure. Such a group is called an **object**. The data of an object are called **attributes** and the subroutines acting on the attributes are called **methods**. The idea behind OOP is that attributes can only be accessed through methods. The direct path to the data is unavailable.

Attributes are referred to as ‘private’. The methods to access the data are made available to programmers, so these are ‘public’. The feature of data being combined with the subroutines acting on this data is known as **encapsulation**. To use an object, we first define an object type. An object type is called a **class**. Classes are templates for objects. When a class type has been defined it can be used to create one or more objects of this class type. Therefore, an object is an instance of a class.

The first stage of writing an object-oriented program to solve a problem is to design the classes. This is part of object-oriented design. From this design, a program can be written using an object-oriented programming (OOP) language.

The programming languages the syllabus prescribes can be used for OOP: Python 3, VB.NET and Java.

**Designing classes and objects**

When designing a class, we need to think about the attributes we want to store. We also need to think about the methods we need to access the data and assign values to the data of an object. A data type is a blueprint when declaring a variable of that data type. A class definition is a blueprint when declaring an object of that class. Creating a new object is known as ‘instantiation’.

Any data that is held about an object must be accessible, otherwise there is no point in storing it. We therefore need methods to access each one of these attributes. These methods are usually referred to as **getters**. They get an attribute of the object.

When we first set up an object of a particular class, we use a constructor. A **constructor** instantiates the object and assigns initial values to the attributes.

Any attributes that might be updated after instantiation will need subroutines to update their values.

These are referred to as **setters**. Some attributes get set only at instantiation. These don’t need setters.

This makes an object more robust, because you cannot change attributes that were not designed to be changed.

Python and VB.NET allow properties to be declared. A **property** combines the attribute with its associated setter and/or getter

**Creating a class**

When a car is manufactured it is given a unique vehicle ID that will remain the same throughout the car’s existence. The engine size of the car is fixed at the time of manufacture. The registration ID will be given to the car when the car is sold.

In our program, when a car is manufactured, we want to create a new car object. We need to instantiate it using the constructor. Any attributes that are already known at the time of instantiation can be set with the constructor. In our example, VehicleID and EngineSize can be set by the constructor. The other attributes are assigned values at the time of purchase and registration.

So, we need setters for them. The identifier table for the Car class is shown below

**Identifier Data**

**T**

|  |  |  |
| --- | --- | --- |
| **Identifier** | **Data type** | **Description** |
| Car | Class | Class identifier |
| VehicleID | STRING | Unique ID assigned at time of manufacture |
| Registration | String | Unique ID assigned after time of purchase |
| DateOfRegistration | Date | Date of registration |
| EngineSize | INTERGER | Engine size assigned at the time of registration |
| PurchasePrice | CURRENCY | Purchase price assigned at a time of purchase |
| Constructor ( ) |  | Method to create a car object and set properties assigned at manufacture |
| SetPurchasePrice ( ) |  | Method to assign purchase price at time of purchase |
| SetRegistration ( ) |  | Method to assign registration ID |
| SetDateOfRegistration |  | Method to assign date of registration |
| GetVehicleID ( ) |  | Method to access Vehicle ID |
| GetRegistration ( ) |  | Method to access registration ID |
| GetDateOfRegistration |  | Method to access date of registration |
| GetEngineSize |  | Method to access engine Size |
| GetPurchasePrice |  | Method to access purchase price |

This information can be represented using a class diagram.

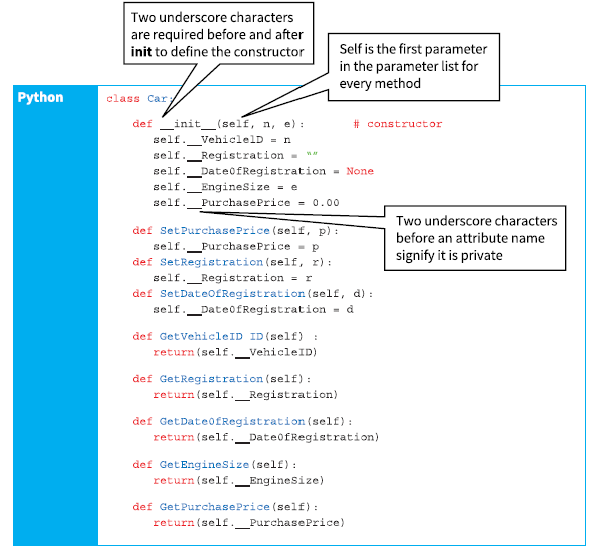
A screenshot of a computer

Description automatically generated

**Writing object-oriented code**

**Declaring a class**

Attributes should always be declared as ‘Private’. This means they can only be accessed through the class methods. So that the methods can be called from the main program, they have to be declared as ‘Public’



**Declaring a class in Python**

The code below shows how a constructor, getters and setters can be declared in Python.

Python also supports properties:



**Instantiating a class**

To use an object of a class type in a program the object must first be instantiated. This means the memory space must be reserved to store the attributes.

*ThisCar = Car("ABC1234", 2500)*

**Using a method**

To call a method in program code, the object identifier is followed by the method identifier and the parameter list.

ThisCar.SetPurchasePrice(12000)

ThisCar.PurchasePrice = 12000 # using properties

The following code gets and prints the vehicle ID for an object ThisCar of class Car.

print(ThisCar.GetVehicleID())

print(ThisCar.VehicleID) # using properties

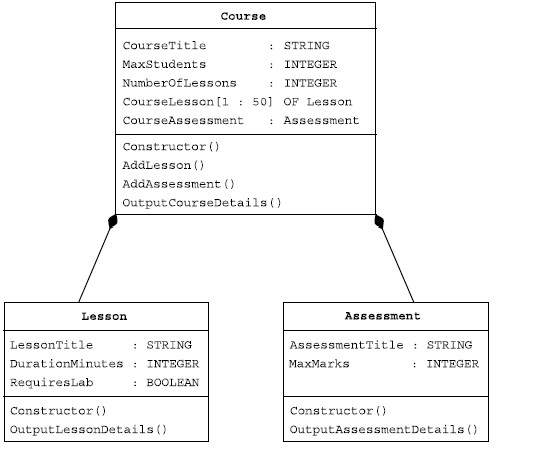
Containment

In object-oriented programming (OOP), containment, also often referred to as "composition," is a fundamental design principle where one class (the container or composite class) includes instances of another class (the contained or component class) as part of its state. It is a way to establish a relationship between objects where one object is composed of or includes another object. This relationship signifies a strong form of association, indicating that the contained objects are part of the larger whole represented by the container object

For example, consider a class called "Car" and another class called "Engine." The Car class can have a member variable of type Engine, which represents the engine component of the car. The Car object contains an instance of the Engine object, and through this containment relationship, the Car object can access and utilize the functionalities provided by the Engine object.

Example

A college runs courses of up to 50 lessons. A course may end with an assessment. Object-oriented programming is to be used to set up courses. The classes required are shown below.



**Garbage collection**

When objects are created, they occupy memory. When they are no longer needed, they should be made to release that memory, so it can be re-used. If objects do not let go of memory, we eventually end up with no free memory when we try and run a program. This is known as ‘memory leakage’.

How do our programming languages handle this?

In Python Memory management involves a private heap containing all Python objects and data structures. The management of the Python heap is performed by the interpreter itself. The programmer does not need to do any housekeeping.

What is containment?

In object-oriented programming (OOP), containment, also often referred to as **"composition,"** is a fundamental design principle where one class (the container or composite class) includes instances of another class (the contained or component class) as part of its state. This relationship signifies a strong form of association, indicating that the contained objects are part of the larger whole represented by the container object.

Analogue

*Some objects (like the crayons/books) are kept inside other objects (like the box/library), and the bigger object takes care of the smaller ones, just like you take care of your crayons by keeping them in the box.*

An example in Python.

Using a Library class that contains instances of a Book class. This example showcases how the Library class (the container) can include multiple Book instances (the contained objects), managing their lifecycle and providing an interface to interact with them

class Book:

def \_\_init\_\_(self, title, author):

self.title = title

self.author = author

def display\_info(self):

print(f"Book: {self.title} by {self.author}")

class Library:

def \_\_init\_\_(self):

self.books = [ ] # Initialization of the containment relationship

def add\_book(self, title, author):

new\_book = Book(title, author)

self.books.append(new\_book)

def display\_books(self):

if not self.books:

print("The library has no books.")

for book in self.books:

book.display\_info()

# Usage

library = Library()

library.add\_book("The Hitchhiker's Guide to the Galaxy", "Douglas Adams")

library.add\_book("1984", "George Orwell")

library. display\_books()

* The Book class defines a book with a title and an author.
* The Library class contains a list of Book objects. This demonstrates the containment relationship, as the library "contains" books.
* The Library class has methods to add books to its collection (add\_book) and to display information about all its books (display\_books), showcasing how interaction with contained objects (books) is managed through the container object (the library).

This example highlights key aspects of containment:

* Ownership: The Library owns the Book instances. If the Library object is destroyed, its Book instances would logically cease to be relevant.
* Lifecycle Management: The Library creates Book instances when a new book is added, managing the lifecycle of these contained objects.
* Encapsulation: The Library provides an interface to interact with its books, encapsulating the complexity of book management.

Setters in OOP

In object-oriented programming (OOP), setters, also known as mutator methods, are a type of method used to set the value of an attribute or property of an object. They are typically used to control the write access to the private or protected data members of a class.

Setters provide a mechanism for encapsulation and data hiding, allowing the class to maintain control over how its attributes are modified. By using a setter method, you can enforce any necessary validation or business rules before assigning a new value to an attribute.

class Person:

def \_\_init\_\_(self, name):

self.\_name = name

def set\_name(self, name):

self.\_name = name

def get\_name(self):

return self.\_name

In the example above, the set\_name method is a setter that allows you to set the value of the \_name attribute. It takes a name parameter and assigns it to the \_name attribute. The corresponding getter method, get\_name, allows you to retrieve the value of the \_name attribute.

By using setters, you can ensure that any necessary checks or modifications are performed when assigning new values to attributes, providing more control and maintaining the integrity of the object's state.

class Person:

def \_\_init\_\_(self, name, age):

self.name = name

self.\_age = age # Note the use of underscore to indicate a "protected" attribute

@property

def age(self):

return self.\_age

@age.setter

def age(self, value):

if value < 0:

raise ValueError("Age cannot be negative")

self.\_age = value

# Example usage

person = Person("John", 30)

print(person.age) # This will print 30

person.age = 25 # This updates the age using the setter

print(person.age) # This will print 25

# Trying to set a negative age will raise an error

# person.age = -5 # This will raise ValueError: Age cannot be negative

In the example above, the Person class has a private \_age attribute and a public age property with a setter. The setter method checks if the new value is negative and raises an error if it is, ensuring the Person object always has a valid age.

Getters

Getters, also known as accessor methods, are **used to retrieve or access the value of an attribute or property of an object**. Getters provide a way to access the private or protected data members of a class from outside the class, while still maintaining control over how the data is accessed.

The primary purpose of using getters is to encapsulate the internal state of an object and provide controlled access to it. By using getters, you can enforce read-only or read-write access to the attributes based on your design requirements, allowing you to implement additional logic or validation before returning the attribute value.

class Person:

def \_\_init\_\_(self, name):

self.\_name = name

def get\_name(self):

return self.\_name

In the example above, the get\_name method is a getter that retrieves the value of the \_name attribute. It simply returns the value of the \_name attribute when called.

By using getters, you can provide controlled access to the internal state of an object and maintain the principle of encapsulation. They allow you to hide the implementation details of a class and provide a consistent interface for accessing the object's attributes, which can be helpful for maintaining code flexibility, security, and abstraction

Drawing a class diagram

Constructing a class diagram involves visually representing the structure of classes within a system, including their attributes, methods, and relationships to other classes. Class diagrams are a part of UML (Unified Modeling Language), which is a standardized modeling language used in object-oriented software development.

Step-by-step guide on how to construct a class diagram:

1. Identify the Classes  
   Start by identifying the classes you need to model. Classes represent entities with common properties and behaviours. For example, in a library management system, you might have classes like Book, Library, and Patrons.
2. Define Class Attributes  
   For each class, list its attributes. Attributes are the data points the class will hold. For example, a Book class might have attributes like title, author, and ISBN.
3. Define Class Methods  
   Define the methods (operations) that each class can perform. For example, the Book class might have methods like checkOut() and returnBook().
4. Determine Relationships  
   Identify the relationships between classes. Common types of relationships include:
   * Association: A general link between two classes indicating a relationship or interaction.
   * Aggregation: A specialized form of association that represents a "whole-part" relationship but without strong ownership.
   * Composition: A strong "whole-part" relationship, where the "part" cannot exist independently of the "whole".
   * Inheritance: Indicates that one class (the child class) inherits attributes and methods from another (the parent class).
5. Draw the Diagram  
   Once you have the information outlined above, you can start drawing the diagram. Use a UML tool or even simple drawing tools. Here’s what to include:
   * Classes: Represented by rectangles divided into three parts: the top part for the class name, the middle part for attributes, and the bottom part for methods.
   * Relationships: Illustrated with different types of lines. For example, solid lines with open arrowheads for associations, lines with a diamond end for composition, etc.

Exercise

A business wants to store data about companies they supply. The data to be stored includes: company name, email address, date of last contact.

* Design a class Company and draw a class diagram.
* Write program code to declare the class. Company name and email address are to be set by the constructor and will never be changed.
* Instantiate one object of this class and test your class code works.