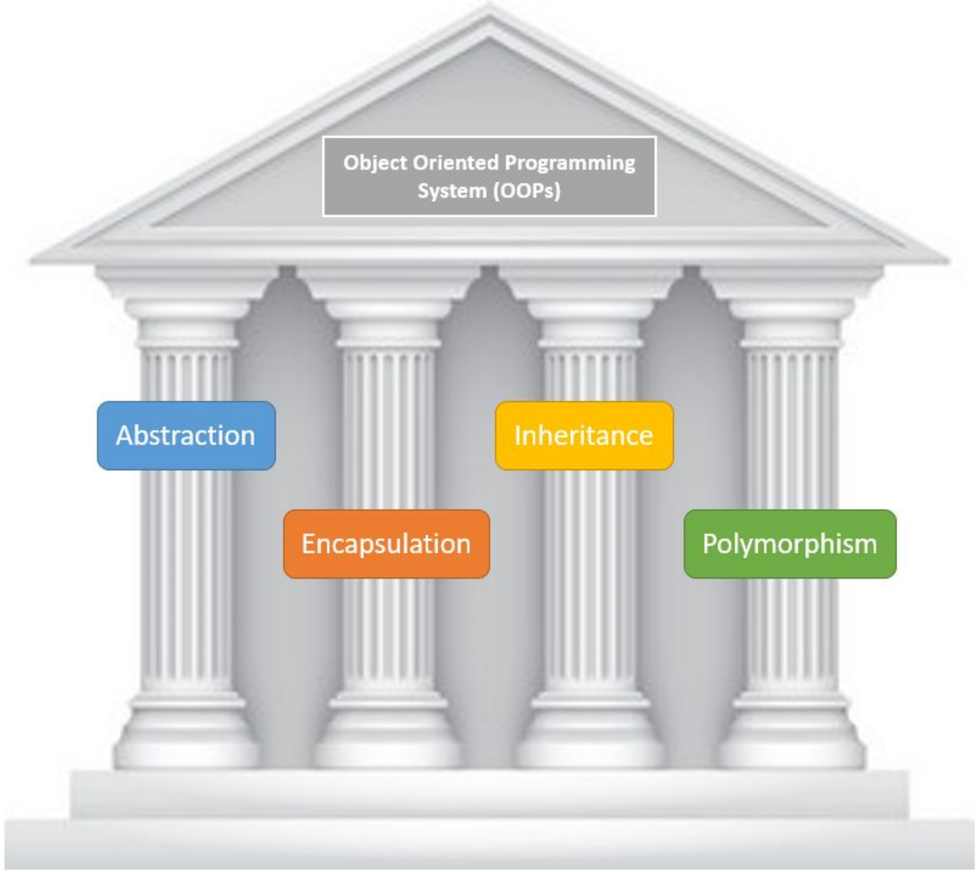
Object Oriented Programming

with python

By Andrei Noguera Gil

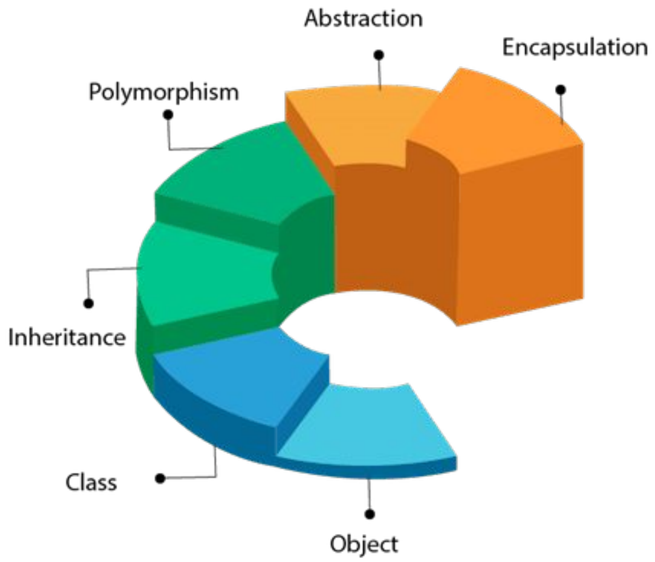
Table content:



What is OOP?

Object-Oriented Programming (OOP) is a programming paradigm that **organizes code around objects**, which are instances of classes. It is based on the concept of **encapsulating data (attributes) and behavior (methods)** into objects, allowing for better organization, reusability, and modularity in code.

OOP promotes the use of objects, classes, inheritance, polymorphism, and other concepts to create modular, reusable, and maintainable code. It is widely used in various programming languages such as Python, Java, C++, and many others.

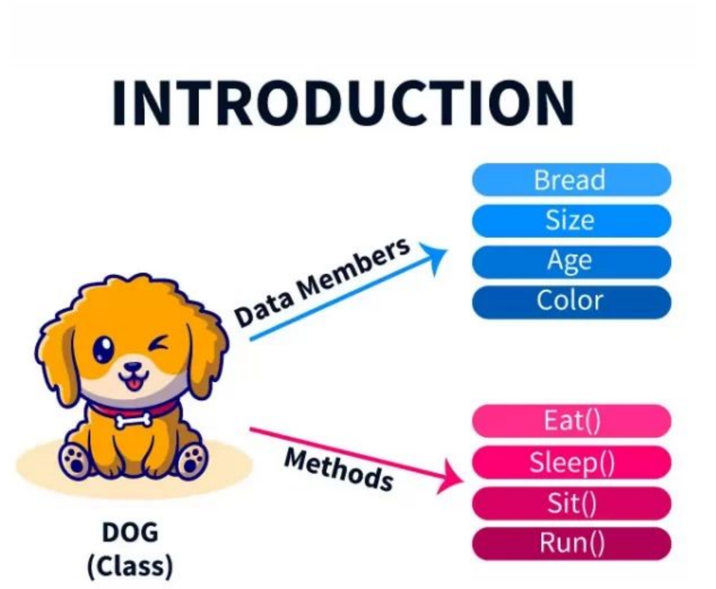


Classes and Objects

What are classes and objects?

A **class is a blueprint or template** that defines the structure and behavior of objects. It serves as a blueprint for creating multiple objects of the same type. A class encapsulates data (attributes) and behavior (methods) related to a specific concept or entity.

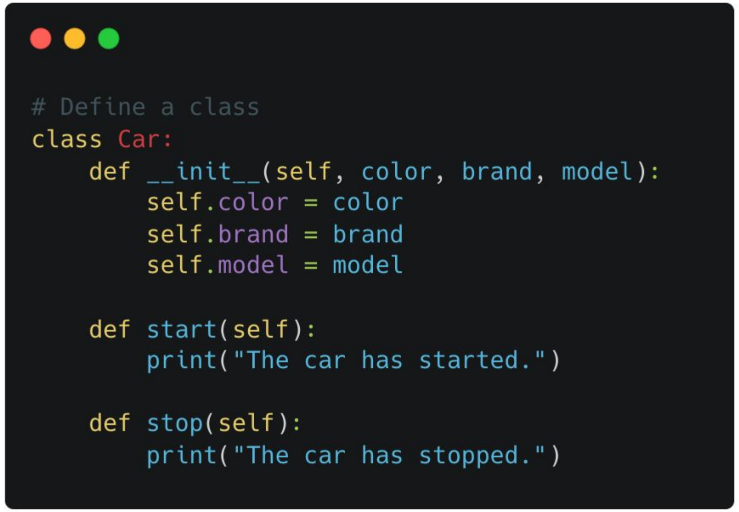
An **object**, on the other hand, is an **instance of a class**. It is a concrete entity that exists in memory and can be manipulated. An object is created based on the definition provided by a class.



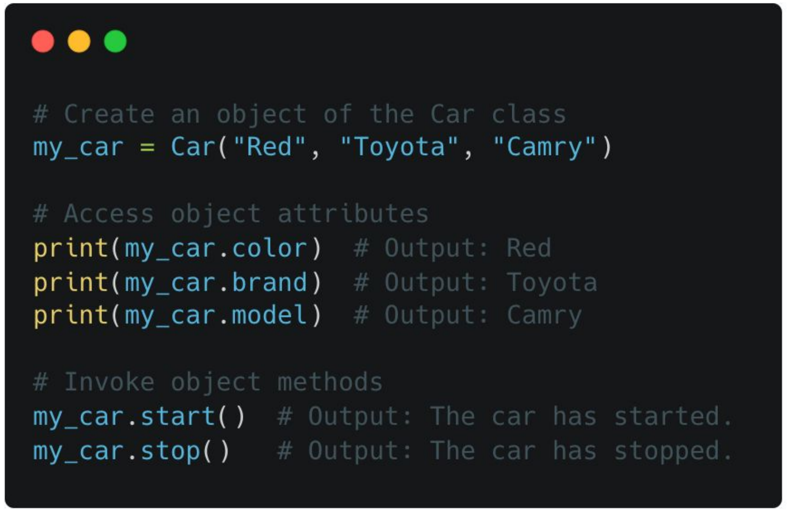
Classes and Objects

Creating classes and instances (objects)

To create an object, you instantiate a class by calling its constructor, which is a special method defined in the class. The constructor initializes the object and sets its initial state. Once an object is created, you can access its attributes and invoke its methods.



```python
# Define a class
class Car:
    def __init__(self, color, brand, model):
        self.color = color
        self.brand = brand
        self.model = model

    def start(self):
        print("The car has started.")

    def stop(self):
        print("The car has stopped.")
```



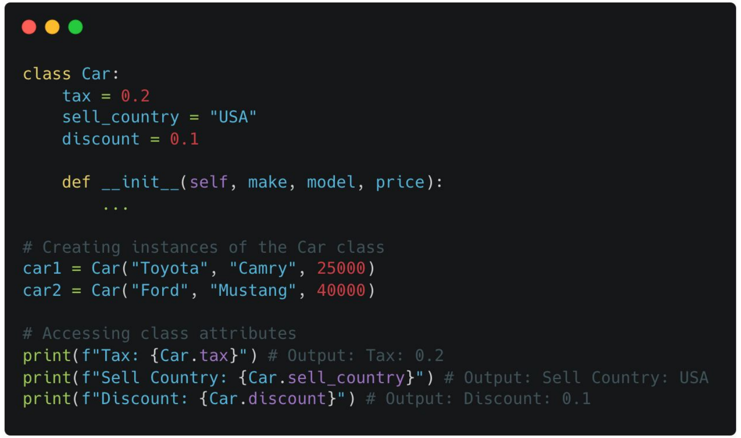
```python
# Create an object of the Car class
my_car = Car("Red", "Toyota", "Camry")

# Access object attributes
print(my_car.color)  # Output: Red
print(my_car.brand)  # Output: Toyota
print(my_car.model)  # Output: Camry

# Invoke object methods
my_car.start()  # Output: The car has started.
my_car.stop()   # Output: The car has stopped.
```

Classes and Objects

Class attributes and instance attributes

Class Attributes:

- Class attributes are attributes that are shared by all instances of a class.
- They are defined within the class but outside any class methods.
- Class attributes have the same value for all instances of the class.
- They are accessed using the class name.
- Are typically used to define properties or characteristics that are common to all objects of the class.



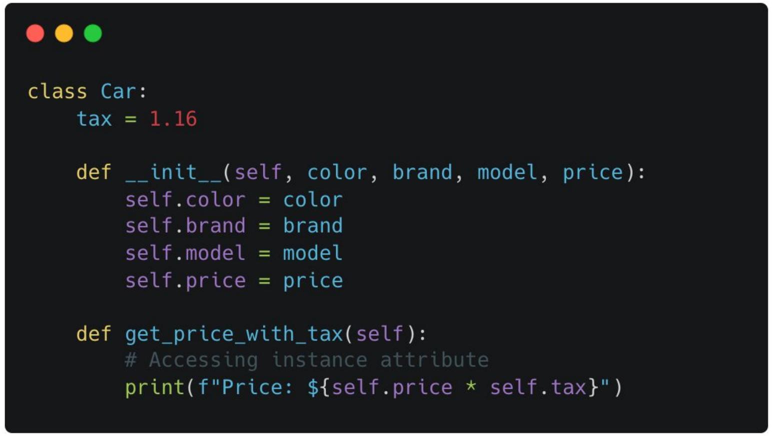
```python
class Car:
    tax = 0.2
    sell_country = "USA"
    discount = 0.1

    def __init__(self, make, model, price):
        ...

# Creating instances of the Car class
car1 = Car("Toyota", "Camry", 25000)
car2 = Car("Ford", "Mustang", 40000)

# Accessing class attributes
print(f"Tax: {Car.tax}") # Output: Tax: 0.2
print(f"Sell Country: {Car.sell_country}") # Output: Sell Country: USA
print(f"Discount: {Car.discount}") # Output: Discount: 0.1
```

Classes and Objects

Class attributes and instance attributes

Instance Attributes:

- Instance attributes are specific to each instance of a class.
- They are defined within the class methods, usually within the `__init__` method.
- Each instance of the class can have its own unique set of instance attributes.
- Instance attributes are accessed using the instance name.
- Instance attributes are used to store data that is unique to each object or instance of the class.



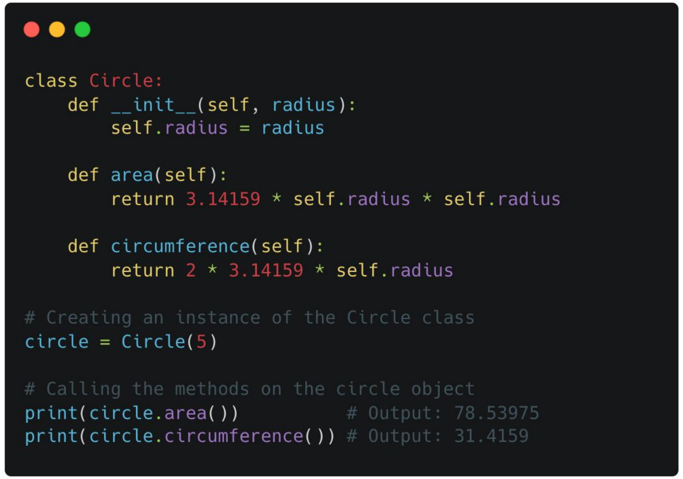
```python
class Car:
    tax = 1.16

    def __init__(self, color, brand, model, price):
        self.color = color
        self.brand = brand
        self.model = model
        self.price = price

    def get_price_with_tax(self):
        # Accessing instance attribute
        print(f"Price: ${self.price * self.tax}")
```

Methods and Attributes

Definition and use of methods

A **method is a function that is defined within a class and is associated with objects (instances) of that class**. Methods define the behavior or actions that objects of a class can perform. They **encapsulate the functionality** related to the class and allow objects to interact with each other and manipulate their internal state.

Methods are defined within a class using the def keyword, similar to how functions are defined. However, **methods have an additional parameter called self**, which refers to the instance of the class on which the method is being called. **This self parameter allows the method to access and modify the instance attributes and perform actions specific to that particular object.**



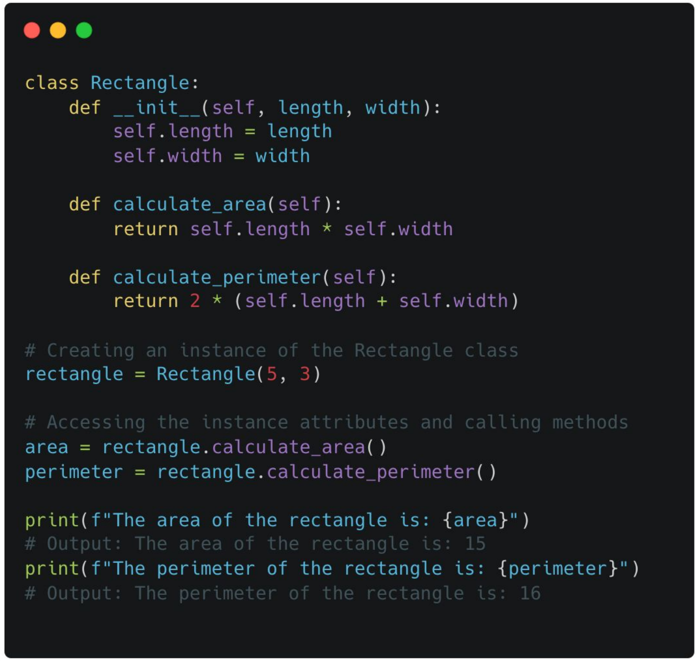
```python
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius * self.radius

    def circumference(self):
        return 2 * 3.14159 * self.radius

# Creating an instance of the Circle class
circle = Circle(5)

# Calling the methods on the circle object
print(circle.area())          # Output: 78.53975
print(circle.circumference()) # Output: 31.4159
```

Methods and Attributes

Why is named “self”?

The parameter name **self is a convention in Python for the first parameter of a method within a class**. It is not a reserved keyword, but you can technically use any valid variable name. However, it **is considered best practice** to stick with self to maintain consistency and improve code readability allowing other developers to easily understand and follow the code.

The purpose of self is to refer to the instance of the class on which the method is being called. It serves as a reference to the specific object that is invoking the method, **allowing the method to access and manipulate the object's attributes and perform actions specific to that particular instance.**



```python
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        return self.length * self.width

    def calculate_perimeter(self):
        return 2 * (self.length + self.width)

# Creating an instance of the Rectangle class
rectangle = Rectangle(5, 3)

# Accessing the instance attributes and calling methods
area = rectangle.calculate_area()
perimeter = rectangle.calculate_perimeter()

print(f"The area of the rectangle is: {area}")
# Output: The area of the rectangle is: 15
print(f"The perimeter of the rectangle is: {perimeter}")
# Output: The perimeter of the rectangle is: 16
```

Methods and Attributes

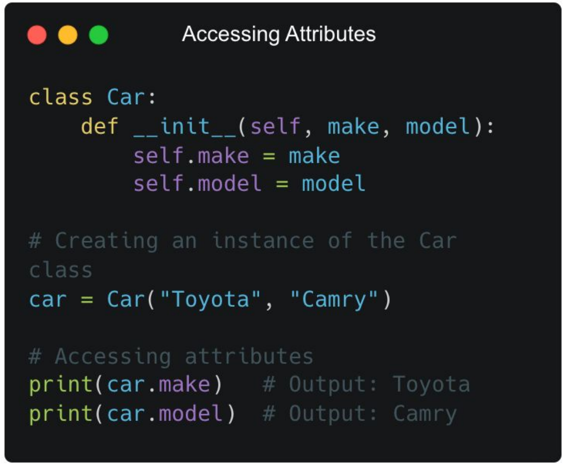
Accessing and modifying attributes

Accessing Attributes:

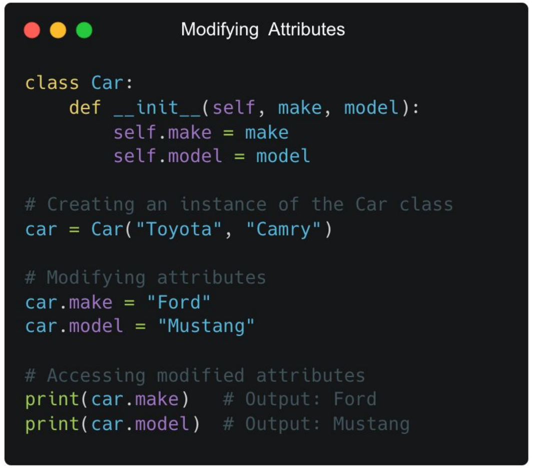
- To access an attribute of an object, you can use the dot notation: object_name.attribute_name.

Modifying Attributes:

- You can modify the value of an attribute by assigning a new value to it using the assignment operator (=).



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

# Creating an instance of the Car
class
car = Car("Toyota", "Camry")

# Accessing attributes
print(car.make)   # Output: Toyota
print(car.model)  # Output: Camry
```



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

# Creating an instance of the Car class
car = Car("Toyota", "Camry")

# Modifying attributes
car.make = "Ford"
car.model = "Mustang"

# Accessing modified attributes
print(car.make)   # Output: Ford
print(car.model)  # Output: Mustang
```

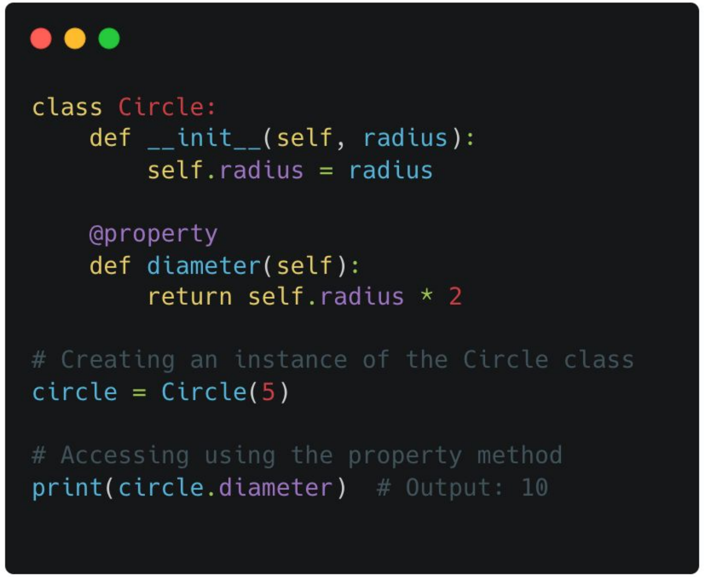
Methods and Attributes

Property methods

In Python, property methods and attribute getters/setters **provide a way to control access to class attributes and add additional logic when getting or setting their values**. They help in encapsulating attribute access and can be useful for maintaining data integrity and implementing computed properties.

Property Methods:

- Property methods **allow you to define methods that can be accessed like attributes.** They provide a way to define custom behavior for getting, setting, and deleting attribute values. Property methods are defined using the @property decorator.



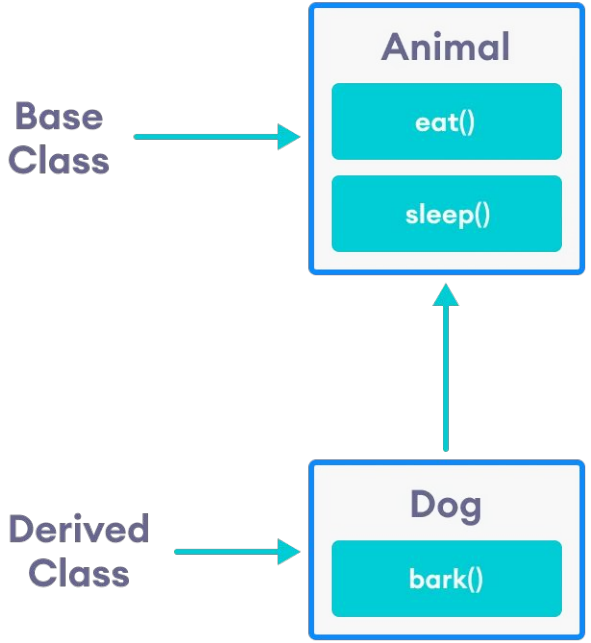
```python
class Circle:
    def __init__(self, radius):
        self.radius = radius

    @property
    def diameter(self):
        return self.radius * 2

# Creating an instance of the Circle class
circle = Circle(5)

# Accessing using the property method
print(circle.diameter)  # Output: 10
```

Inheritance

Understanding inheritance relationships

Inheritance is a fundamental concept in object-oriented programming that **allows classes to inherit attributes and methods from other classes**. It establishes a **hierarchical relationship between classes, where the child class (derived class) inherits properties and behaviors from its parent class (base class)**. This allows for code reuse, as common attributes and behaviors can be defined in the parent class and inherited by multiple child classes.

When a class inherits from another class, it automatically gains access to all the attributes and methods of the parent class. The child class can then extend or modify the inherited attributes and methods, or define its own unique attributes and methods.



Inheritance

Understanding inheritance relationships

Here are some key points to understand about inheritance relationships:

- **Parent Class:** Also known as the base class or superclass, it is the class from which other classes inherit. It provides a blueprint for common attributes and behaviors that can be shared among multiple child classes.

- **Child Class:** Also known as the derived class or subclass, it is the class that inherits attributes and methods from its parent class. It can add new attributes and methods, override inherited methods, or extend the functionality of the parent class.

- **Inherited Attributes and Methods:** When a child class inherits from a parent class, it automatically inherits all the attributes and methods defined in the parent class. These inherited attributes and methods can be accessed and used directly in the child class.

Inheritance

Understanding inheritance relationships

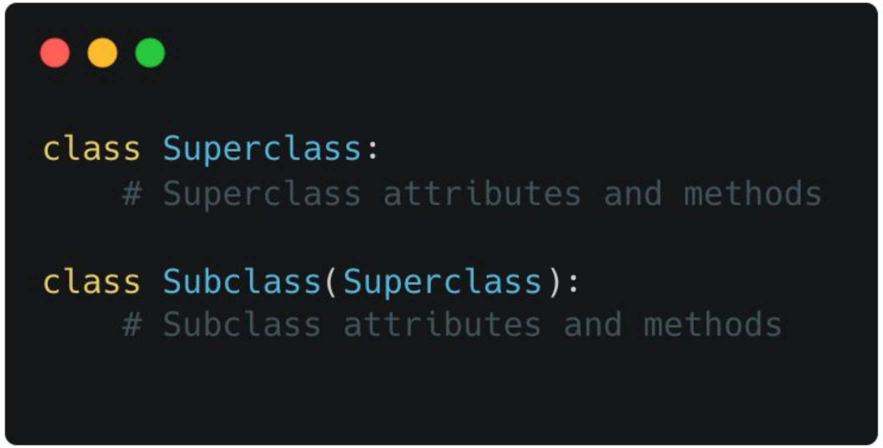
- **Overriding Methods:** Child classes have the ability to override (redefine) inherited methods from the parent class. This allows them to provide their own implementation of a method with the same name as the parent class. When the method is called on an instance of the child class, the overridden method in the child class will be executed instead of the parent class method.

- **Method Resolution Order (MRO):** In cases where multiple inheritance is involved (i.e., a child class inherits from multiple parent classes), the order in which the parent classes are specified matters. Python follows the C3 linearization algorithm to determine the order in which methods are resolved in the inheritance hierarchy.

Inheritance

Creating and using subclasses and superclasses

Creating subclasses and superclasses is a fundamental concept in object-oriented programming that **allows for specialization and hierarchical relationships between classes**. A superclass is a class from which other classes inherit, and a **subclass is a class that inherits from a superclass**. The subclass inherits all the attributes and methods of the superclass and can add its own unique attributes and methods or modify the inherited ones.



```python
class Superclass:
    # Superclass attributes and methods

class Subclass(Superclass):
    # Subclass attributes and methods
```

Inheritance

Method overriding and inheritance hierarchy

Method Overriding:
Refers to the **ability of a subclass to provide a different implementation of a method that is already defined in its superclass**. When a method is overridden in a subclass, the subclass provides its **own implementation of the method**, which is used instead of the implementation in the superclass when the method is called on an instance of the subclass.

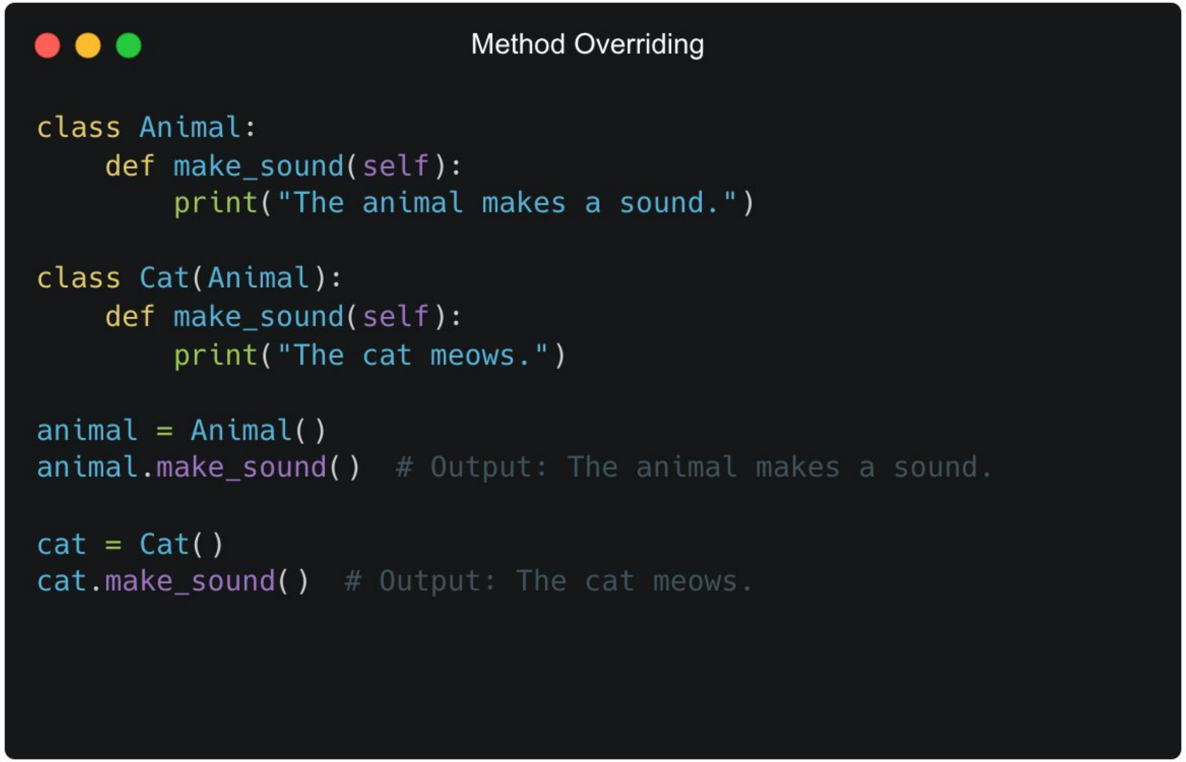
The process of method overriding involves the following:

1. The superclass defines a method.
2. The subclass declares a method with the same name and parameters as the superclass method.
3. When the method is called on an instance of the subclass, the subclass method is executed instead of the superclass method.

This is useful when you want to modify or extend the behavior of a method inherited from the superclass. It allows you to customize the behavior of the method to better suit the needs of the subclass while maintaining the common interface defined by the superclass.

Inheritance

Method overriding and inheritance hierarchy


Method Overriding

```
class Animal:
    def make_sound(self):
        print("The animal makes a sound.")

class Cat(Animal):
    def make_sound(self):
        print("The cat meows.")

animal = Animal()
animal.make_sound()  # Output: The animal makes a sound.

cat = Cat()
cat.make_sound()  # Output: The cat meows.
```

Inheritance

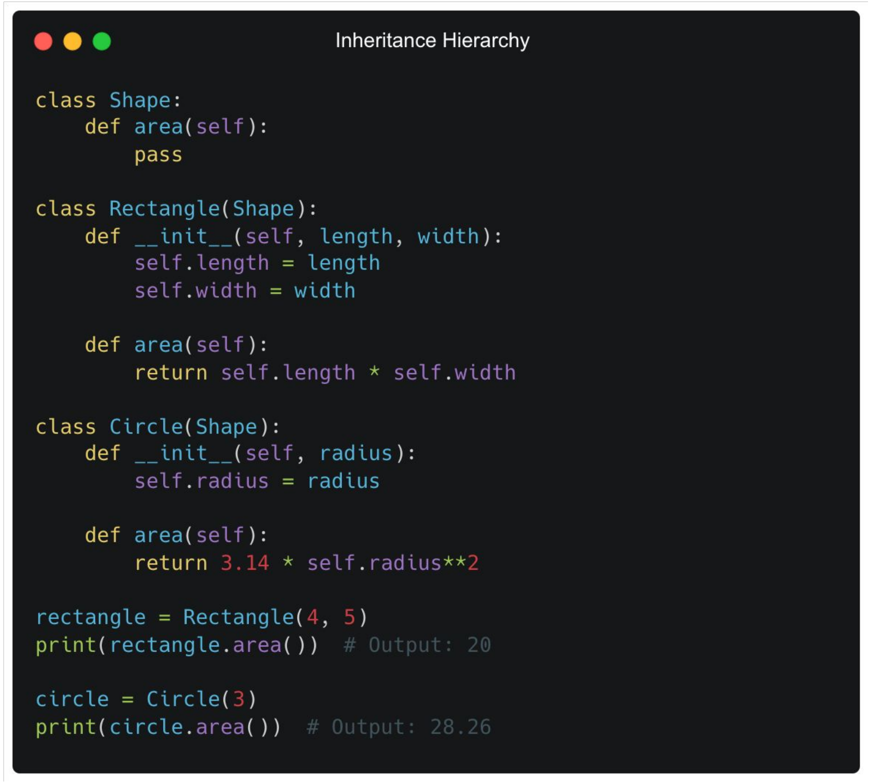
Method overriding and inheritance hierarchy

Inheritance Hierarchy:
Refers to the **hierarchical structure of classes formed by inheritance relationships**. In an inheritance hierarchy, **classes are organized in a tree-like structure**, where each class inherits attributes and methods from its superclass or superclasses. This creates a hierarchy of classes where more specific or specialized classes are derived from more general or abstract classes.

1. **Superclass:** A **superclass is a class that is higher in the hierarchy** and serves as a base for other classes. It defines common attributes and methods that are shared by its subclasses.
2. **Subclass:** A **subclass is a class that inherits attributes and methods from its superclass**. It can add its own unique attributes and methods or override inherited ones to provide specialized behavior.
3. **Single Inheritance: Single inheritance refers to the concept of a class having only one direct superclass**. Each subclass has a single superclass from which it inherits attributes and methods.
4. **Multiple Inheritance: Multiple inheritance refers to the concept of a class having multiple direct superclasses.** Each subclass can inherit attributes and methods from multiple superclasses, allowing for more complex class relationships.

Inheritance

Method overriding and inheritance hierarchy



```python
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

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

rectangle = Rectangle(4, 5)
print(rectangle.area())  # Output: 20

circle = Circle(3)
print(circle.area())  # Output: 28.26
```

Polymorphism

Polymorphism and its importance in OOP

Polymorphism is a fundamental concept in object-oriented programming (OOP) that allows objects of **different classes to be treated as objects of a common superclass**. It enables different objects to respond to the same method call in different ways, based on their specific implementation.

Polymorphism is important in OOP for the following reasons:

1. **Code Reusability:** Polymorphism promotes code reuse by allowing objects of different classes to be used interchangeably. This means that a method written to operate on a superclass can be applied to any of its subclasses without modifications, as long as they implement the required methods.

2. **Flexibility and Extensibility:** Polymorphism enables the addition of new subclasses without modifying existing code. This makes the code more flexible and extensible, as new classes can be seamlessly integrated into the existing codebase, leveraging the common interface provided by the superclass.

Polymorphism

Polymorphism and its importance in OOP

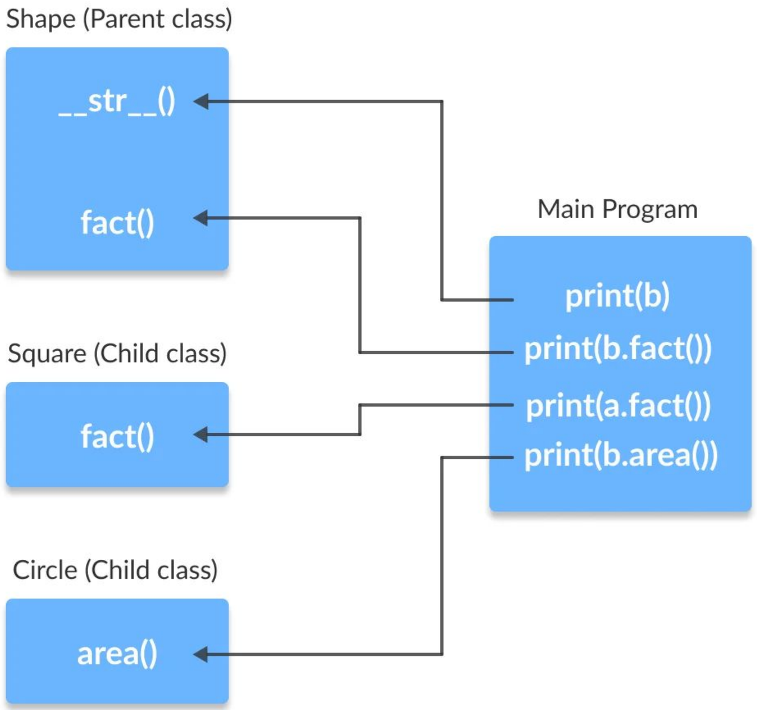
3. **Simplified Interface Design:** Polymorphism allows for the design of simplified and consistent interfaces. By defining a common set of methods in a superclass, client code can interact with objects based on their superclass interface, without needing to know the specific subclass implementation details. This promotes loose coupling and modularity.

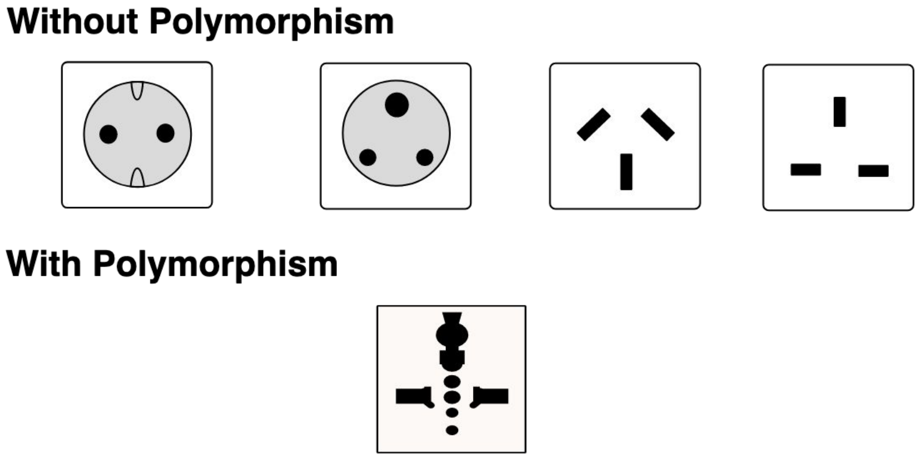
4. **Method Overriding:** Polymorphism facilitates method overriding, which allows a subclass to provide its own implementation of a method defined in its superclass. This enables customization and specialization of behavior, providing a way to adapt the behavior of a superclass method to suit the specific needs of each subclass.

5. **Polymorphic Functionality:** Polymorphism enables the creation of polymorphic functionality, where a single method can handle different types of objects based on their common interface. This allows for more concise and modular code, as the same method can be used with different object types.

Polymorphism

Concept



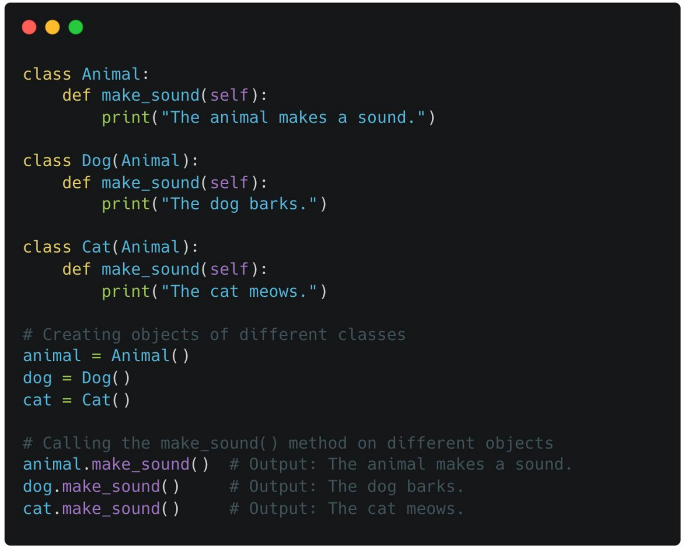


Polymorphism

Method overriding and dynamic method dispatch

Method overriding allows the subclass to override the behavior of the method inherited from the superclass and provide its own implementation.

Dynamic method dispatch, also known as runtime polymorphism, is the mechanism by which the appropriate method implementation is called at runtime based on the actual object type rather than the reference type. This enables the program to determine the specific implementation of a method based on the actual object being referenced.



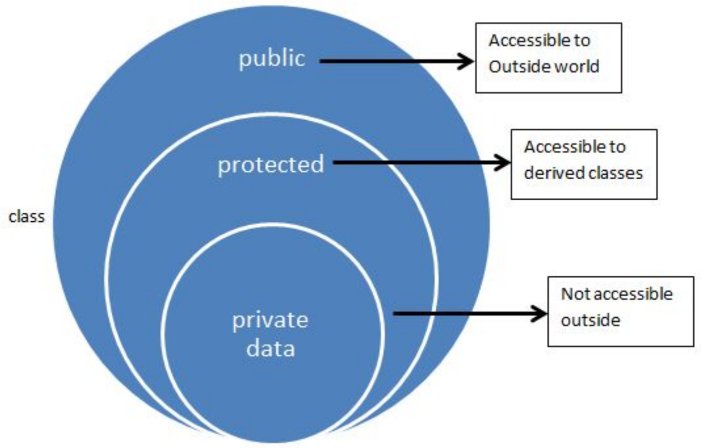
```python
class Animal:
    def make_sound(self):
        print("The animal makes a sound.")

class Dog(Animal):
    def make_sound(self):
        print("The dog barks.")

class Cat(Animal):
    def make_sound(self):
        print("The cat meows.")

# Creating objects of different classes
animal = Animal()
dog = Dog()
cat = Cat()

# Calling the make_sound() method on different objects
animal.make_sound()  # Output: The animal makes a sound.
dog.make_sound()     # Output: The dog barks.
cat.make_sound()     # Output: The cat meows.
```

Encapsulation

Encapsulation and data hiding

Encapsulation is an important concept in object-oriented programming (OOP) that combines data and methods into a single unit called a class. It **involves hiding the internal details of an object and providing access to the object's properties** and behavior through well-defined interfaces.

The main goal of encapsulation is to **ensure that the internal state of an object is protected from direct external access**. This is achieved by making the internal data of the object private, which means it can only be accessed and modified through specific methods or properties defined within the class.



Encapsulation

Encapsulation and data hiding

Encapsulation provides several benefits:

- **Data Protection:** By encapsulating data within a class, you can **control how it is accessed and modified**. This **prevents external code from directly manipulating** the internal state of an object, ensuring data integrity and consistency.

- **Code Organization:** Encapsulation helps **organize code by grouping related data and methods into a single class**. This makes the code more modular, easier to understand, and promotes code reusability.

- **Flexibility:** Encapsulation **allows you to change the internal implementation of a class without affecting the external code** that uses the class. This provides flexibility in maintaining and evolving your codebase.

Encapsulation

Access modifiers (public, private, protected)

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects, which are instances of classes. It is based on the concept of encapsulating data (attributes) and behavior (methods) into objects, allowing for better organization, reusability, and modularity in code.

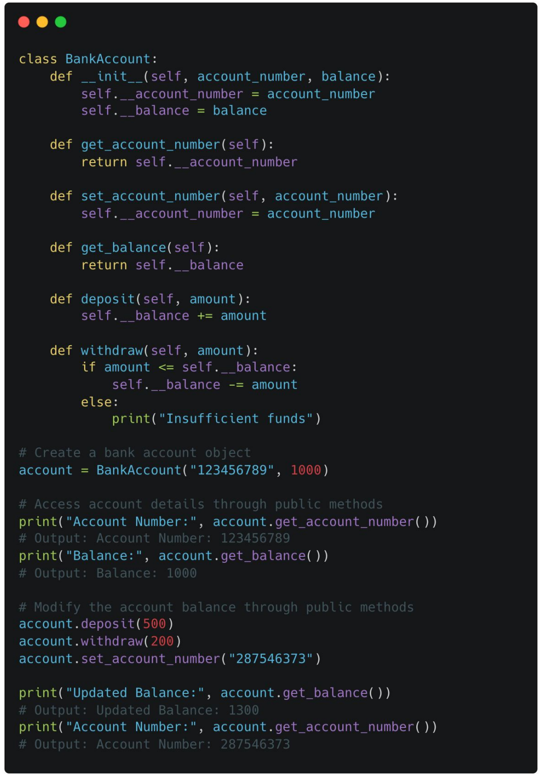
OOP promotes the use of objects, classes, inheritance, polymorphism, and other concepts to create modular, reusable, and maintainable code. It is widely used in various programming languages such as Python, Java, C++, and many others.

Encapsulation

Attribute getters/setters

Attribute Getters/Setters:

- Attribute getters and setters are methods that **allow you to control the access and modification of attributes**. They provide additional logic and validation when getting or setting attribute values. **Getters are used to retrieve the value of an attribute, while setters are used to set the value of an attribute.** Attribute getters and setters can be defined using decorators and can have the same name as the attribute.



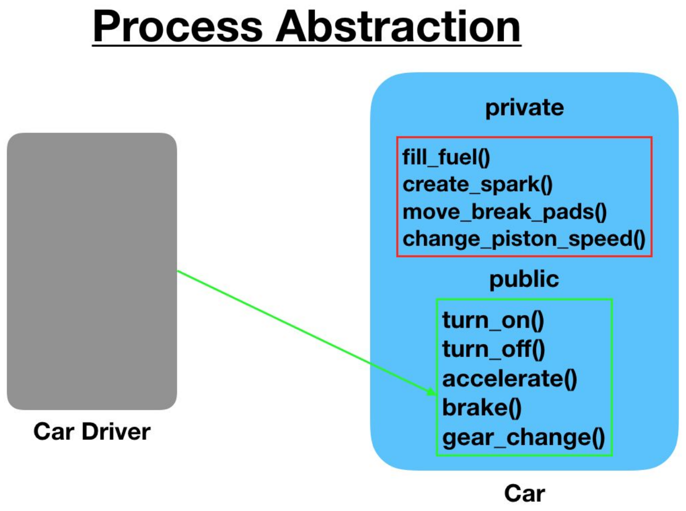
```python
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number
        self.__balance = balance

    def get_account_number(self):
        return self.__account_number

    def set_account_number(self, account_number):
        self.__account_number = account_number

    def get_balance(self):
        return self.__balance

    def deposit(self, amount):
        self.__balance += amount

    def withdraw(self, amount):
        if amount <= self.__balance:
            self.__balance -= amount
        else:
            print("Insufficient funds")

# Create a bank account object
account = BankAccount("123456789", 1000)

# Access account details through public methods
print("Account Number:", account.get_account_number())
# Output: Account Number: 123456789
print("Balance:", account.get_balance())
# Output: Balance: 1000

# Modify the account balance through public methods
account.deposit(500)
account.withdraw(200)
account.set_account_number("287546373")

print("Updated Balance:", account.get_balance())
# Output: Updated Balance: 1300
print("Account Number:", account.get_account_number())
# Output: Account Number: 287546373
```

Abstraction

Abstract classes

Abstract classes and interfaces are concepts used in object-oriented programming to define common behaviors and establish contracts for classes.

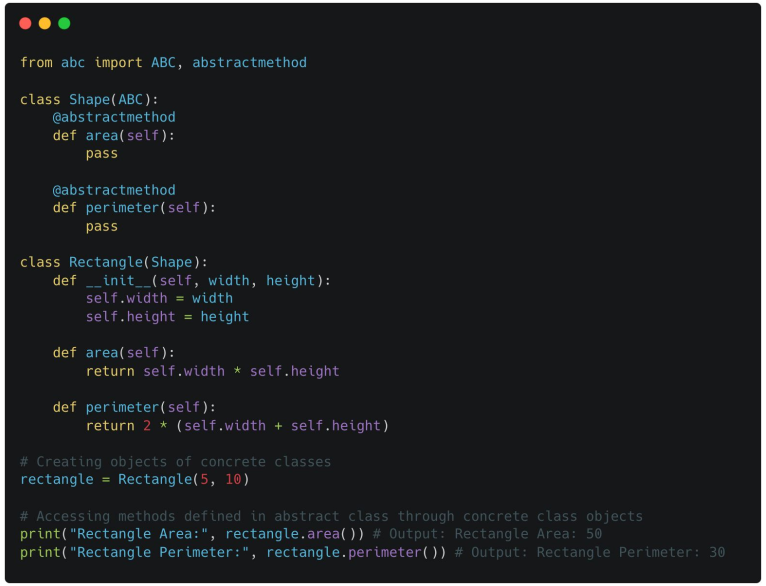
An abstract class is a class that cannot be instantiated and is meant to serve as a blueprint for its subclasses. It contains one or more abstract methods, which are methods without an implementation. Subclasses of an abstract class must provide an implementation for all the abstract methods defined in the abstract class. **Abstract classes can also have non-abstract (concrete) methods that are inherited by the subclasses.**



Abstraction

Abstract methods and their implementations

Abstract methods are methods defined in an abstract class that have no implementation. They serve as placeholders for the methods that must be implemented by the concrete subclasses. The abstract methods are defined using the @abstractmethod decorator.



```python
from abc import ABC, abstractmethod

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

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

# Creating objects of concrete classes
rectangle = Rectangle(5, 10)

# Accessing methods defined in abstract class through concrete class objects
print("Rectangle Area:", rectangle.area()) # Output: Rectangle Area: 50
print("Rectangle Perimeter:", rectangle.perimeter()) # Output: Rectangle Perimeter: 30
```

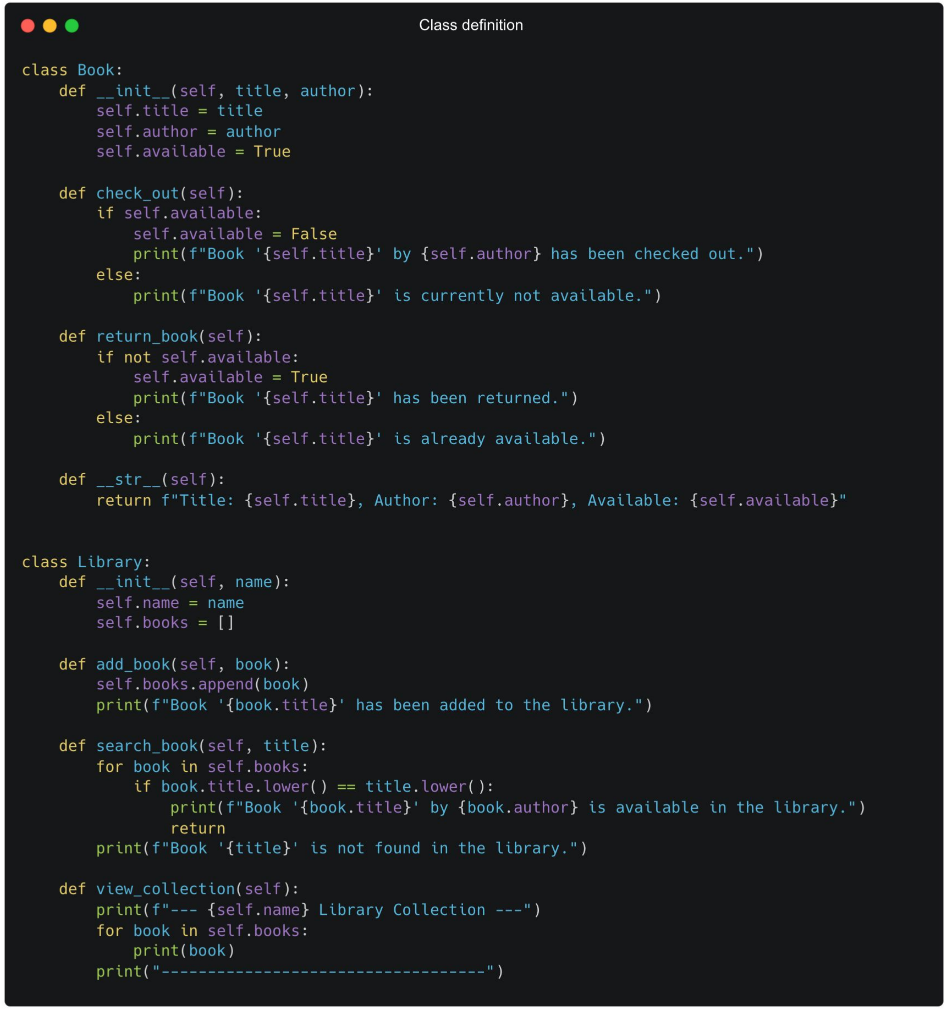
Extra

Example

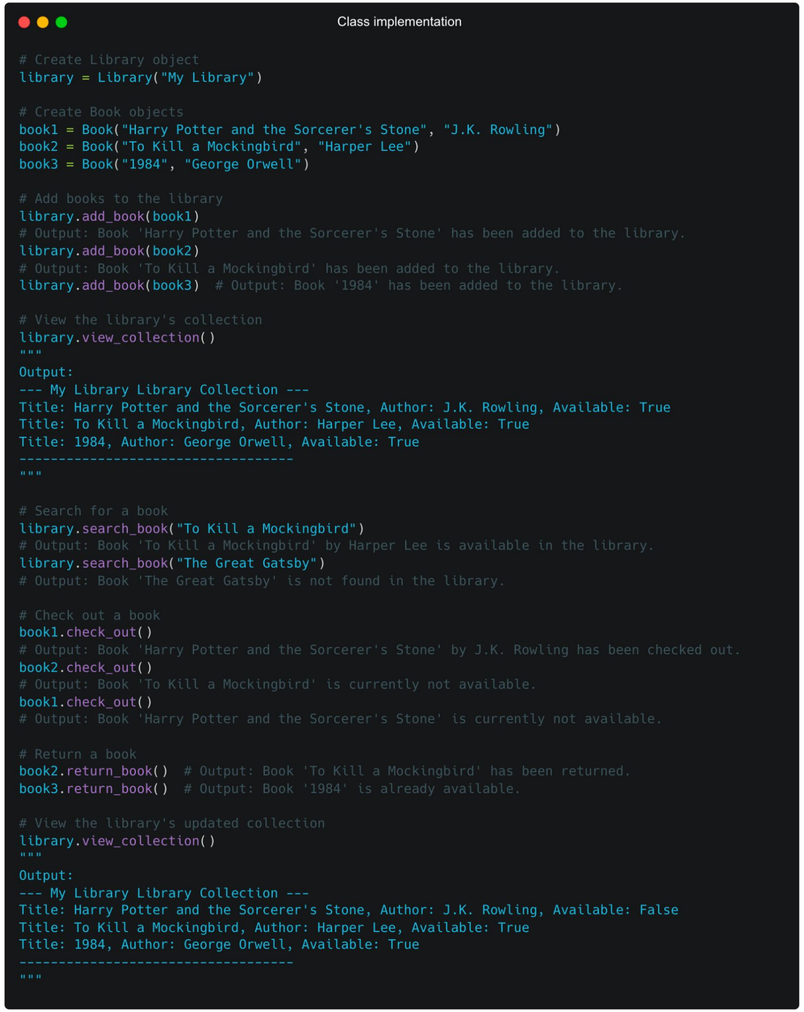
Library system

We will create a program that models a library system. The library contains books, and each book has a title, author, and availability status (whether it is currently checked out or available for borrowing). Users can search for books, check out a book, return a book, and view the library's collection. We will use OOP principles to represent the library, books, and users as classes with their respective attributes and methods.

Code solution in the next slide ↓


Class definition

```python
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author
        self.available = True

    def check_out(self):
        if self.available:
            self.available = False
            print(f"Book '{self.title}' by {self.author} has been checked out.")
        else:
            print(f"Book '{self.title}' is currently not available.")

    def return_book(self):
        if not self.available:
            self.available = True
            print(f"Book '{self.title}' has been returned.")
        else:
            print(f"Book '{self.title}' is already available.")

    def __str__(self):
        return f"Title: {self.title}, Author: {self.author}, Available: {self.available}"


class Library:
    def __init__(self, name):
        self.name = name
        self.books = []

    def add_book(self, book):
        self.books.append(book)
        print(f"Book '{book.title}' has been added to the library.")

    def search_book(self, title):
        for book in self.books:
            if book.title.lower() == title.lower():
                print(f"Book '{book.title}' by {book.author} is available in the library.")
                return
        print(f"Book '{title}' is not found in the library.")

    def view_collection(self):
        print(f"--- {self.name} Library Collection ---")
        for book in self.books:
            print(book)
        print("-----------------------------------")
```


Class implementation

```python
# Create Library object
library = Library("My Library")

# Create Book objects
book1 = Book("Harry Potter and the Sorcerer's Stone", "J.K. Rowling")
book2 = Book("To Kill a Mockingbird", "Harper Lee")
book3 = Book("1984", "George Orwell")

# Add books to the library
library.add_book(book1)
# Output: Book 'Harry Potter and the Sorcerer's Stone' has been added to the library.
library.add_book(book2)
# Output: Book 'To Kill a Mockingbird' has been added to the library.
library.add_book(book3)  # Output: Book '1984' has been added to the library.

# View the library's collection
library.view_collection()
"""
Output:
--- My Library Library Collection ---
Title: Harry Potter and the Sorcerer's Stone, Author: J.K. Rowling, Available: True
Title: To Kill a Mockingbird, Author: Harper Lee, Available: True
Title: 1984, Author: George Orwell, Available: True
-----------------------------------
"""

# Search for a book
library.search_book("To Kill a Mockingbird")
# Output: Book 'To Kill a Mockingbird' by Harper Lee is available in the library.
library.search_book("The Great Gatsby")
# Output: Book 'The Great Gatsby' is not found in the library.

# Check out a book
book1.check_out()
# Output: Book 'Harry Potter and the Sorcerer's Stone' by J.K. Rowling has been checked out.
book2.check_out()
# Output: Book 'To Kill a Mockingbird' is currently not available.
book1.check_out()
# Output: Book 'Harry Potter and the Sorcerer's Stone' is currently not available.

# Return a book
book2.return_book()  # Output: Book 'To Kill a Mockingbird' has been returned.
book3.return_book()  # Output: Book '1984' is already available.

# View the library's updated collection
library.view_collection()
"""
Output:
--- My Library Library Collection ---
Title: Harry Potter and the Sorcerer's Stone, Author: J.K. Rowling, Available: False
Title: To Kill a Mockingbird, Author: Harper Lee, Available: True
Title: 1984, Author: George Orwell, Available: True
-----------------------------------
"""
```