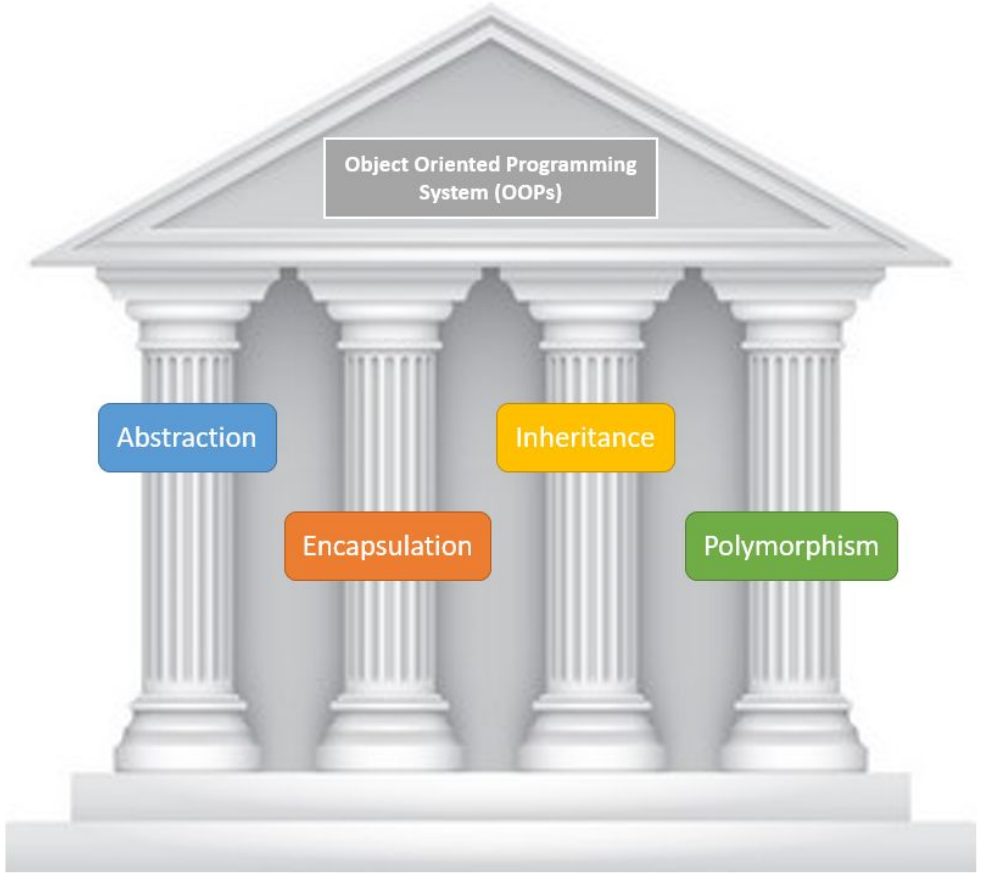
Programación orientada a objetos

con python

Por Andrei Noguera Gil

Contenido:

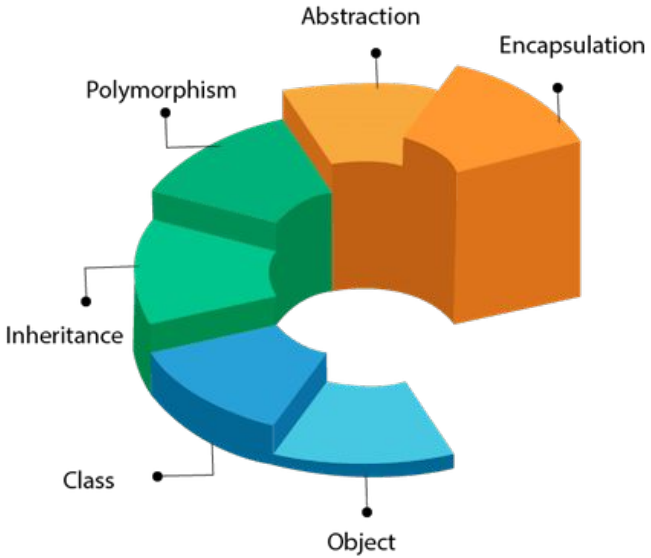
1. Clases y objetos
2. Métodos y atributos
3. Herencia
4. Polimorfismo
5. Encapsulación
6. Abstracción



¿Qué es POO?

La programación orientada a objetos (POO) es un paradigma de programación que **organiza el código en torno a objetos**, que son instancias de clases. Se basa en el concepto de **encapsular los datos (atributos) y el comportamiento (métodos)** en objetos, lo que permite una mejor organización, reutilización y modularidad del código.

La programación orientada a objetos promueve el uso de objetos, clases, herencia, polimorfismo y otros conceptos para crear código modular, reutilizable y fácil de mantener. Se utiliza ampliamente en varios lenguajes de programación como Python, Java, C++ y muchos otros.

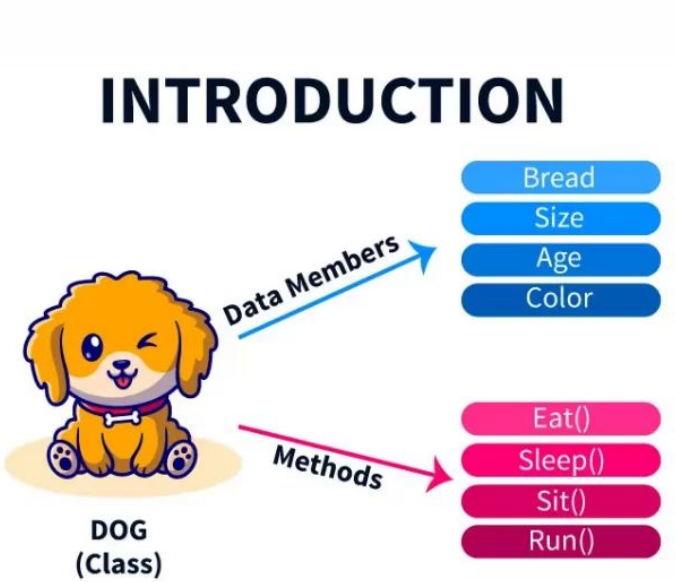


Clases y objetos

¿Qué son las clases y los objetos?

Una **clase es un plano o plantilla** que define la estructura y el comportamiento de los objetos. Sirve para crear varios objetos del mismo tipo. Una clase encapsula datos (atributos) y comportamientos (métodos) relacionados con un concepto o entidad específicos.

En cambio, un **objeto** es una **instancia de una clase**. Es una entidad concreta que existe en memoria y puede manipularse. Un objeto se crea a partir de la definición proporcionada por una clase.



Clases y objetos

Creación de clases e instancias (objetos)

Para crear un objeto, instancie una clase llamando a su constructor, que es un método especial definido en la clase. El constructor inicializa el objeto y establece su estado inicial. Una vez creado un objeto, puedes acceder a sus atributos e invocar sus métodos.

```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.
```

Clases y Objetos

Atributos de clase y atributos de instancia

Atributos de clase:

- Los atributos de clase son atributos que comparten todas las instancias de una clase.
- Se definen dentro de la clase pero fuera de cualquier método de la clase.
- Los atributos de clase tienen el mismo valor para todas las instancias de la clase.
- Se accede a ellos utilizando el nombre de la clase.
- Se utilizan normalmente para definir propiedades o características que son comunes a todos los objetos de la clase.

```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
```

Clases y Objetos

Atributos de clase y atributos de instancia

Atributos de Instancia:

- Los atributos de instancia son específicos de cada instancia de una clase.
- Se definen en los métodos de la clase, normalmente en el método `__init__`.
- Cada instancia de la clase puede tener su propio conjunto único de atributos de instancia.
- Se accede a los atributos de instancia utilizando el nombre de instancia.
- Los atributos de instancia se utilizan para almacenar datos que son únicos para cada objeto o instancia de la clase.

```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}")
```

Métodos y atributos

Definición y uso de métodos

Un **método es una función que se define dentro de una clase y se asocia con objetos (instancias) de esa clase**. Los métodos definen el comportamiento o las acciones que pueden realizar los objetos de una clase. **Encapsulan la funcionalidad** relacionada con la clase y permiten a los objetos interactuar entre sí y manipular su estado interno.

Los métodos se definen dentro de una clase utilizando la palabra clave def, de forma similar a como se definen las funciones. Sin embargo, **los métodos tienen un parámetro adicional llamado self**, que se refiere a la instancia de la clase sobre la que se está llamando al método. **Este parámetro self permite al método acceder y modificar los atributos de la instancia y realizar acciones específicas de ese objeto en particular.**

```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
```

Métodos y atributos

¿Por qué se llama «self»?

El nombre de parámetro **self es una convención en Python para el primer parámetro de un método dentro de una clase**. No es una palabra clave reservada, pero técnicamente se puede utilizar cualquier nombre de variable válido. Sin embargo, **se considera la mejor práctica** utilizar self para mantener la consistencia y mejorar la legibilidad del código permitiendo a otros desarrolladores entender y seguir el código fácilmente.

El propósito de self es referirse a la instancia de la clase en la que el método está siendo llamado. Sirve como referencia al objeto específico que está invocando el método, **permitiendo al método acceder y manipular los atributos del objeto y realizar acciones específicas a esa instancia en particular.**

```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
```

Métodos y atributos

Acceso y modificación de atributos

Acceso a atributos:

- Para acceder a un atributo de un objeto, puede utilizar la notación de puntos: nombre_objeto.nombre_atributo.

Modificación de atributos:

- Puedes modificar el valor de un atributo asignándole un nuevo valor mediante el operador de asignación (=).

Accessing Attributes

```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
```

Modifying Attributes

```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
```

Métodos y atributos

Métodos de propiedad

En Python, los métodos de propiedades y los getters/setters de atributos **proporcionan una forma de controlar el acceso a los atributos de la clase y añadir lógica adicional al obtener o establecer sus valores**. Ayudan a encapsular el acceso a atributos y pueden ser útiles para mantener la integridad de los datos e implementar propiedades computadas.

```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
```

Métodos de propiedad:

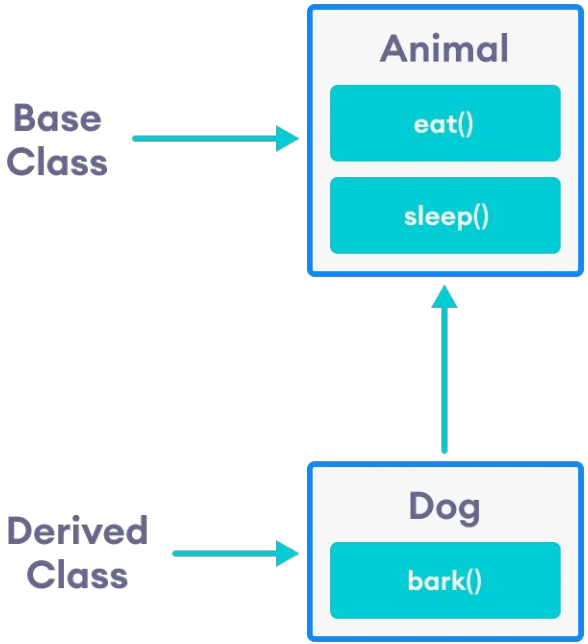
- Los métodos de propiedad **permiten definir métodos a los que se puede acceder como atributos.** Proporcionan una forma de definir un comportamiento personalizado para obtener, establecer y eliminar valores de atributos. Los métodos de propiedad se definen mediante el decorador @property.

Herencia

Comprender las relaciones de herencia

La herencia es un concepto fundamental en la programación orientada a objetos que **permite a las clases heredar atributos y métodos de otras clases**. Establece una **relación jerárquica entre clases, en la que la clase hija (clase derivada) hereda propiedades y comportamientos de su clase padre (clase base)**. Esto permite la reutilización de código, ya que los atributos y comportamientos comunes pueden definirse en la clase padre y ser heredados por múltiples clases hijas.

Cuando una clase hereda de otra, obtiene automáticamente acceso a todos los atributos y métodos de la clase padre. A continuación, la clase hija puede ampliar o modificar los atributos y métodos heredados, o definir sus propios atributos y métodos exclusivos.



Herencia

Comprender las relaciones de herencia

Aquí hay algunos puntos clave para entender acerca de las relaciones de herencia:

- **Clase padre:** También conocida como clase base o superclase, es la clase de la que heredan otras clases. Proporciona un modelo de atributos y comportamientos comunes que pueden ser compartidos entre múltiples clases hijas.

- **Clase** hija**:** También conocida como clase derivada o subclase, es la clase que hereda atributos y métodos de su clase padre. Puede agregar nuevos atributos y métodos, anular métodos heredados o extender la funcionalidad de la clase padre.

- **Atributos y métodos** heredados**:** Cuando una clase hija hereda de una clase padre, hereda automáticamente todos los atributos y métodos definidos en la clase padre. Estos atributos y métodos heredados se pueden acceder y utilizar directamente en la clase hija.

Herencia

Comprender las relaciones de herencia

Anulación de métodos: Las clases hijas tienen la capacidad de sobrescribir (redefinir) métodos heredados de la clase padre. Esto les permite proporcionar su propia implementación de un método con el mismo nombre que la clase padre. Cuando se llama al método en una instancia de la clase hija, el método anulado en la clase hija se ejecutará en lugar del método de la clase padre.

Orden de resolución de métodos(MRO): En los casos en los que hay herencia múltiple (es decir, una clase hija hereda de varias clases padre), el orden en el que se especifican las clases padre es importante. Python sigue el algoritmo de linealización C3 para determinar el orden en que se resuelven los métodos en la jerarquía de herencia.

Herencia

Creación y uso de subclases y superclases

La creación de subclases y superclases es un concepto fundamental en la programación orientada a objetos que **permite la especialización y las relaciones jerárquicas entre clases**. Una superclase es una clase de la que heredan otras clases, y una **subclase es una clase que hereda de una superclase**. La subclase hereda todos los atributos y métodos de la superclase y puede añadir sus propios atributos y métodos exclusivos o modificar los heredados.

```
class Superclass:
    # Superclass attributes and methods

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

Herencia

Sobreescritura de métodos y jerarquía de herencia

Sobreescritura de métodos:
Se refiere a la **capacidad de una subclase para proporcionar una implementación diferente de un método que ya está definido en su superclase**. Cuando un método se sobrescribe en una subclase, la subclase proporciona su **propia implementación del método**, que se utiliza en lugar de la implementación en la superclase cuando el método es llamado en una instancia de la subclase.

El proceso de sobreescritura de métodos implica lo siguiente:

1. La superclase define un método.
2. La subclase declara un método con el mismo nombre y parámetros que el método de la superclase.
3. Cuando se llama al método en una instancia de la subclase, se ejecuta el método de la subclase en lugar del método de la superclase.

Esto es útil cuando se desea modificar o ampliar el comportamiento de un método heredado de la superclase. Permite personalizar el comportamiento del método para adaptarlo mejor a las necesidades de la subclase, manteniendo al mismo tiempo la interfaz común definida por la superclase.

Herencia

Sobreescritura de métodos y jerarquía de herencia

Method Overriding

```python
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.
```

Herencia

Sobreescritura de métodos y jerarquía de herencia

Jerarquía de herencia:
Se refiere a la **estructura jerárquica de clases formada por relaciones de herencia**. En una jerarquía de herencia, **las clases se organizan en una estructura arborescente**, en la que cada clase hereda atributos y métodos de su superclase o superclases. Esto crea una jerarquía de clases donde las clases más específicas o especializadas derivan de clases más generales o abstractas.

1. **Superclase:** Una superclase **es una clase que está más arriba en la jerarquía** y sirve de base para otras clases. Define atributos y métodos comunes que comparten sus subclases.
2. **Subclase:** Una **subclase es una clase que hereda atributos y métodos de su superclase**. Puede añadir sus propios atributos y métodos únicos o anular los heredados para proporcionar un comportamiento especializado.
3. **Herencia única: La herencia única se refiere al concepto de una clase que sólo tiene una superclase directa**. Cada subclase tiene una única superclase de la que hereda atributos y métodos.
4. **Herencia múltiple: La herencia múltiple se refiere al concepto de una clase que tiene múltiples superclases directas.** Cada subclase puede heredar atributos y métodos de varias superclases, lo que permite relaciones de clase más complejas.

Herencia

Sobreescritura de métodos y jerarquía de herencia

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
```

Polimorfismo

El polimorfismo y su importancia en la programación orientada a objetos

El polimorfismo es un concepto fundamental de la programación orientada a objetos (POO) que permite **tratar** objetos de **clases diferentes como objetos de una superclase común**. Permite que diferentes objetos respondan a la misma llamada de método de diferentes maneras, basándose en su implementación específica.

El polimorfismo es importante en la programación orientada a objetos por las siguientes razones:

1. Reutilización del código: El polimorfismo promueve la reutilización de código al permitir que objetos de diferentes clases se utilicen indistintamente. Esto significa que un método escrito para operar sobre una superclase puede aplicarse a cualquiera de sus subclases sin modificaciones, siempre que implementen los métodos requeridos.

2. Flexibilidad y extensibilidad: El polimorfismo permite añadir nuevas subclases sin modificar el código existente. Esto hace que el código sea más flexible y extensible, ya que las nuevas clases pueden integrarse sin problemas en el código base existente, aprovechando la interfaz común proporcionada por la superclase.

Polimorfismo

El polimorfismo y su importancia en la programación orientada a objetos

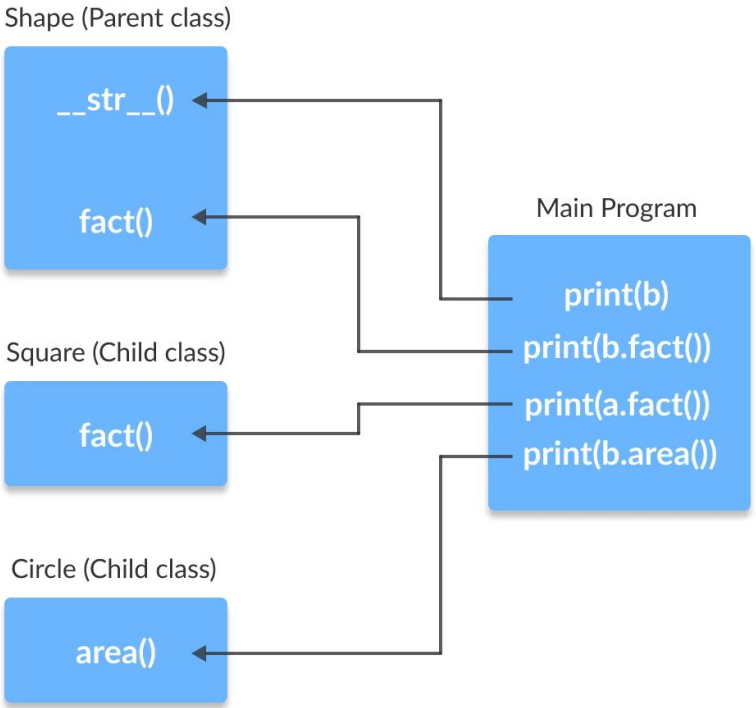
3. **Diseño simplificado de interfaces:** El polimorfismo permite el diseño de interfaces simplificadas y consistentes. Al definir un conjunto común de métodos en una superclase, el código cliente puede interactuar con los objetos basándose en la interfaz de su superclase, sin necesidad de conocer los detalles específicos de implementación de la subclase. Esto promueve el acoplamiento flexible y la modularidad.

4. **Anulación de métodos:** El polimorfismo facilita la sustitución de métodos, lo que permite a una subclase proporcionar su propia implementación de un método definido en su superclase. Esto permite la personalización y especialización del comportamiento, proporcionando una manera de adaptar el comportamiento de un método de la superclase para satisfacer las necesidades específicas de cada subclase.

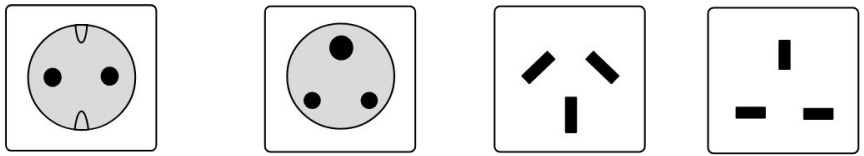
5. **Funcionalidad polimórfica:** El polimorfismo permite la creación de funcionalidad polimórfica, donde un único método puede manejar diferentes tipos de objetos basados en su interfaz común. Esto permite un código más conciso y modular, ya que el mismo método puede utilizarse con diferentes tipos de objetos.

Polimorfismo

Concepto



Without Polymorphism



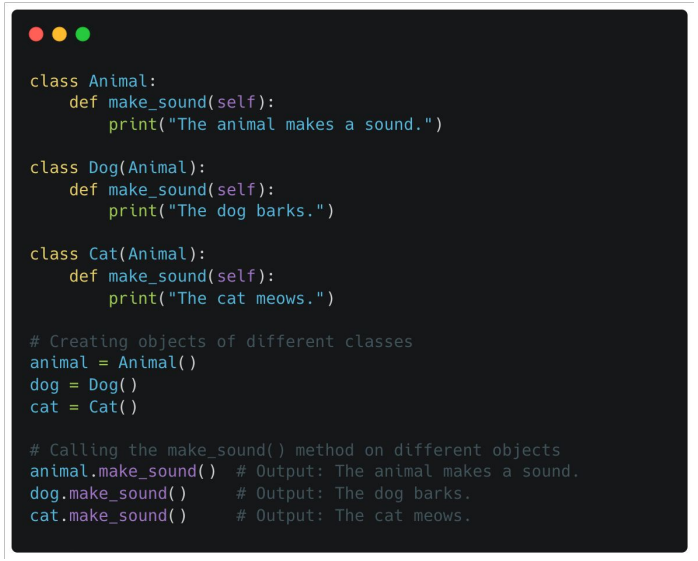
With Polymorphism

Polimorfismo

Sobreescritura de métodos y envío dinámico de métodos

La sobreescritura de métodos permite a la subclase redefinir el comportamiento del método heredado de la superclase y proporcionar su propia implementación.

El envío dinámico de métodos, también conocido como polimorfismo en tiempo de ejecución, es el mecanismo por el cual se llama a la implementación del método apropiado en tiempo de ejecución basándose en el tipo de objeto real en lugar del tipo de referencia. Esto permite al programa determinar la implementación específica de un método basándose en el objeto real al que se hace referencia.



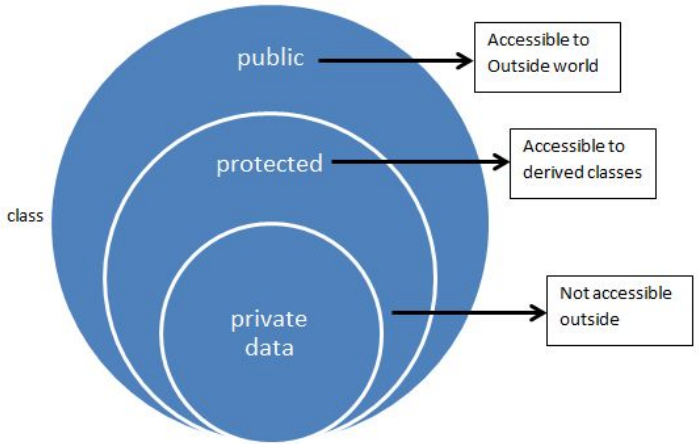
```python
class Animal:
    def make_sound(self):
        print("The animal makes a sound.")

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.
```

Encapsulamiento

Encapsulamiento y ocultación de datos

El encapsulamiento es un concepto importante de la programación orientada a objetos (POO) que combina datos y métodos en una sola unidad llamada clase. **Consiste en ocultar los detalles internos de un objeto y proporcionar acceso a sus propiedades** y comportamiento a través de interfaces bien definidas.

El objetivo principal de la encapsulación es **garantizar que el estado interno de un objeto esté protegido del acceso externo directo**. Esto se consigue haciendo que los datos internos del objeto sean privados, lo que significa que sólo se puede acceder a ellos y modificarlos a través de métodos o propiedades específicos definidos dentro de la clase.



Encapsulamiento

Encapsulamiento y ocultación de datos

La encapsulación proporciona varias ventajas:

- **Protección de** datos**:** Al encapsular los datos dentro de una clase, puedes **controlar cómo se accede a ellos y cómo se modifican**. Esto **evita que código externo manipule directamente** el estado interno de un objeto, asegurando la integridad y consistencia de los datos.

- **Organización** del código**:** La encapsulación ayuda a **organizar el código mediante la agrupación de datos y métodos relacionados en una sola clase**. Esto hace que el código sea más modular, más fácil de entender y promueve la reutilización del código.

- Flexibilidad**:** La encapsulación **le permite cambiar la implementación interna de una clase sin afectar el código externo** que utiliza la clase. Esto proporciona flexibilidad en el mantenimiento y la evolución de su código base.

Encapsulamiento

Modificadores de acceso (público, privado, protegido)

La programación orientada a objetos (POO) es un paradigma de programación que organiza el código en torno a objetos, que son instancias de clases. Se basa en el concepto de encapsular los datos (atributos) y el comportamiento (métodos) en objetos, lo que permite una mejor organización, reutilización y modularidad del código.

La programación orientada a objetos promueve el uso de objetos, clases, herencia, polimorfismo y otros conceptos para crear código modular, reutilizable y fácil de mantener. Se utiliza ampliamente en varios lenguajes de programación como Python, Java, C++ y muchos otros.

Encapsulación

Atributos getters/setters

Atributos getters/setters :

- Los getters y setters de atributos son métodos que **permiten controlar el acceso y la modificación de atributos**. Proporcionan lógica y validación adicionales al obtener o establecer valores de atributos. **Los getters se utilizan para recuperar el valor de un atributo, mientras que los setters se utilizan para establecer el valor de un atributo.** Los getters y setters de atributos pueden definirse utilizando decoradores y pueden tener el mismo nombre que el atributo.

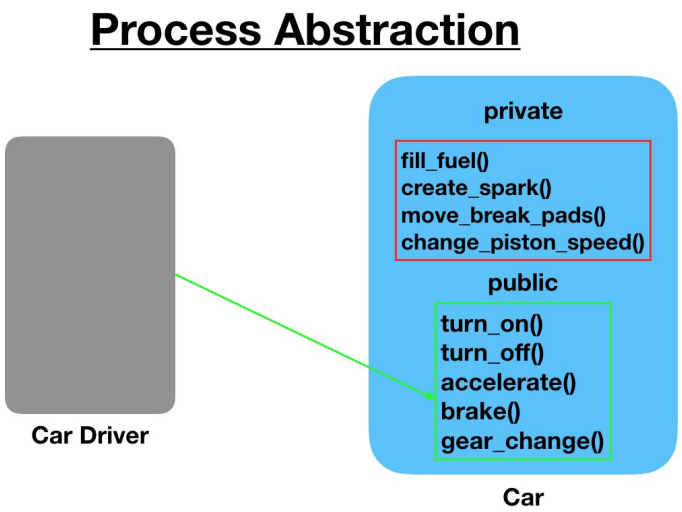
```python
class BankAccount:
    def __init__(self, account_number, balance):
        self.__account_number = account_number
        self.__balance = balance

    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
```

Abstracción

Clases abstractas

Las clases abstractas y las interfaces son conceptos utilizados en la programación orientada a objetos para definir comportamientos comunes y establecer contratos para las clases.

Una clase abstracta es una clase que no puede instanciarse y que sirve de modelo para sus subclases. Contiene uno o más métodos abstractos, que son métodos sin implementación. Las subclases de una clase abstracta deben proporcionar una implementación para todos los métodos abstractos definidos en la clase abstracta. **Las clases abstractas también pueden tener métodos no abstractos (concretos) que son heredados por las subclases.**



Abstracción

Métodos abstractos y sus implementaciones

Los métodos abstractos son métodos definidos en una clase abstracta que no tienen implementación. Sirven como marcadores de posición para los métodos que deben ser implementados por las subclases concretas. Los métodos abstractos se definen utilizando el decorador @abstractmethod.

```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

Ejemplo

Sistema de bibliotecas

Crearemos un programa que modele un sistema de biblioteca. La biblioteca contiene libros, y cada libro tiene un título, un autor y un estado de disponibilidad (si está actualmente sacado o disponible para préstamo). Los usuarios pueden buscar libros, sacar un libro, devolver un libro y ver la colección de la biblioteca. Utilizaremos principios de programación orientada a objetos para representar la biblioteca, los libros y los usuarios como clases con sus respectivos atributos y métodos.

Solución de código en la siguiente diapositiva ↓

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
-----------------------------------
"""
```