

# PROGRAMACIÓN ORIENTADA OBJETOS

CONCEPTOS

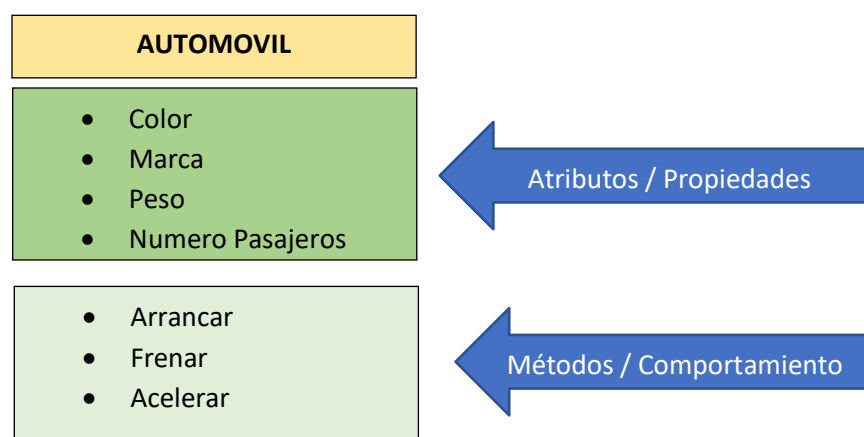




**CLASE**

Modelo donde se redactan las características comunes de un grupo de objetos. Se puede decir que una clase es una plantilla genérica de un objeto. La clase proporciona variables iniciales de estado (donde se guardan los atributos) e implementaciones de comportamiento (métodos) necesarias para crear nuevos objetos, son los modelos sobre los cuáles serán construidos.

Siguiendo con el ejemplo del Automóvil, definamos su clase:

**Sintaxis:**

```
Class NombreDeLaClase:  
    Atributo1  
    Atributo2  
    Atributo2  
    ....  
    Def nombreMetodo1 (self):  
        Líneas de código del metodo1  
    Def nombreMetodo2 (self):  
        Líneas de código del metodo2  
    ....
```

Codifiquemos la clase Automóvil

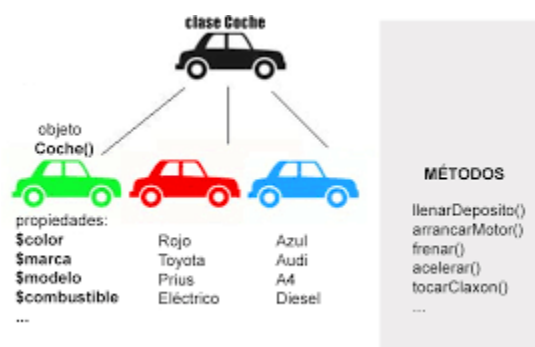
```
clases.py
class Automovil:
    color="Blanco"
    marca="Mazda"
    peso="800 kg"
    numPasajeros=5

    def arrancar(self):
        print("El automovil arranco")
    def frenar(self):
        print("El automovil freno")
    def acelerar(self):
        print("El automovil acelero")
```

## INSTANCIAR UNA CLASE

Una instancia es una copia específica de la clase con todo su contenido, al instanciar una clase obtendremos un objeto. En los lenguajes de programación orientada a objetos un objeto es una instancia de una clase

Instanciamos la clase *Automóvil* y crearemos un objeto llamado *miAuto*



```
1 ▼ class Automovil:  
2     color="Blanco"  
3     marca="Mazda"  
4     peso="800 kg"  
5     numPasajeros=5  
6  
7     def arrancar(self):  
8         print("El automovil arranco")  
9     def frenar(self):  
10        print("El automovil freno")  
11    def acelerar(self):  
12        print("El automovil acelero")  
13  
14    miAuto=Automovil|  
15
```

## OBJETO

Un objeto se crea a partir de instanciar una clase, como ya se instancio la clase *Automóvil*, ahora vamos a ver cómo podemos acceder a sus atributos y métodos.

```
clases.py x
1 class Automovil:
2     color="Blanco"
3     marca="Mazda"
4     peso="800 kg"
5     numPasajeros=5
6
7     def arrancar(self):
8         return "El automovil arranco"
9     def frenar(self):
10            return "El automovil freno"
11    def acelerar(self):
12        return "El automovil acelero"
13
14 miAuto=Automovil
15 print("----OBJETO miAuto---- ")
16 print("Atributos")
17 print("Su color es: ", miAuto.color)
18 print("Su marca es: ", miAuto.marca)
19 print("Su peso es: ", miAuto.peso)
20 print("Su capacidad de pasajeros es: ", miAuto.numPasajeros)
21 print("Metodos")
22 print(miAuto.arrancar(miAuto))
23 print(miAuto.frenar(miAuto))
24 print(miAuto.acelerar(miAuto))
25
```

\*REPL\* [python] - Sublime Text (UNREGISTERED)  
File Edit Selection Find View Goto Tools Proj

\*REPL\* [python] x

----OBJETO miAuto----

Atributos

Su color es: Blanco  
Su marca es: Mazda  
Su peso es: 800 kg  
Su capacidad de pasajeros es: 5

Metodos

El automovil arranco  
El automovil freno  
El automovil acelero

\*\*\*Repl Closed\*\*\*

Line 13, Column 1

Como ya tenemos una clase creada podriamos crear los objetos que queramos.

A continuacion creamos un segundo automovil a partir de la misma clase.

```
class Automovil:
    color="Blanco"
    marca="Mazda"
    peso="800 kg"
    numPasajeros=5

    def arrancar(self):
        print("El automovil arranco")
    def frenar(self):
        print("El automovil freno")
    def acelerar(self):
        print("El automovil acelero")

miAuto=Automovil
print("Su color es: ", miAuto.color)
print("Su marca es: ", miAuto.marca)
print("Su peso es: ", miAuto.peso)
print("Su capacidad de pasajeros es: ", miAuto.numPasajeros)
print("Metodos")
miAuto.arrancar(miAuto)
miAuto.frenar(miAuto)
miAuto.acelerar(miAuto)

print("SEGUNDO OBJETO AUTOMOVIL")
miAuto2=Automovil
print("Su color es: ", miAuto2.color)
print("Su marca es: ", miAuto2.marca)
print("Su peso es: ", miAuto2.peso)
print("Su capacidad de pasajeros es: ", miAuto2.numPasajeros)
print("Metodos")
miAuto2.arrancar(miAuto)
miAuto2.frenar(miAuto)
miAuto2.acelerar(miAuto)
```

\*REPL\* [python] x

Su color es: Blanco  
Su marca es: Mazda  
Su peso es: 800 kg  
Su capacidad de pasajeros es: 5  
Metodos  
El automovil arranco  
El automovil freno  
El automovil acelero  
**SEGUNDO OBJETO AUTOMOVIL**  
~~Su color es: Blanco~~  
Su marca es: Mazda  
Su peso es: 800 kg  
Su capacidad de pasajeros es: 5  
Metodos

Line 20, Column 1

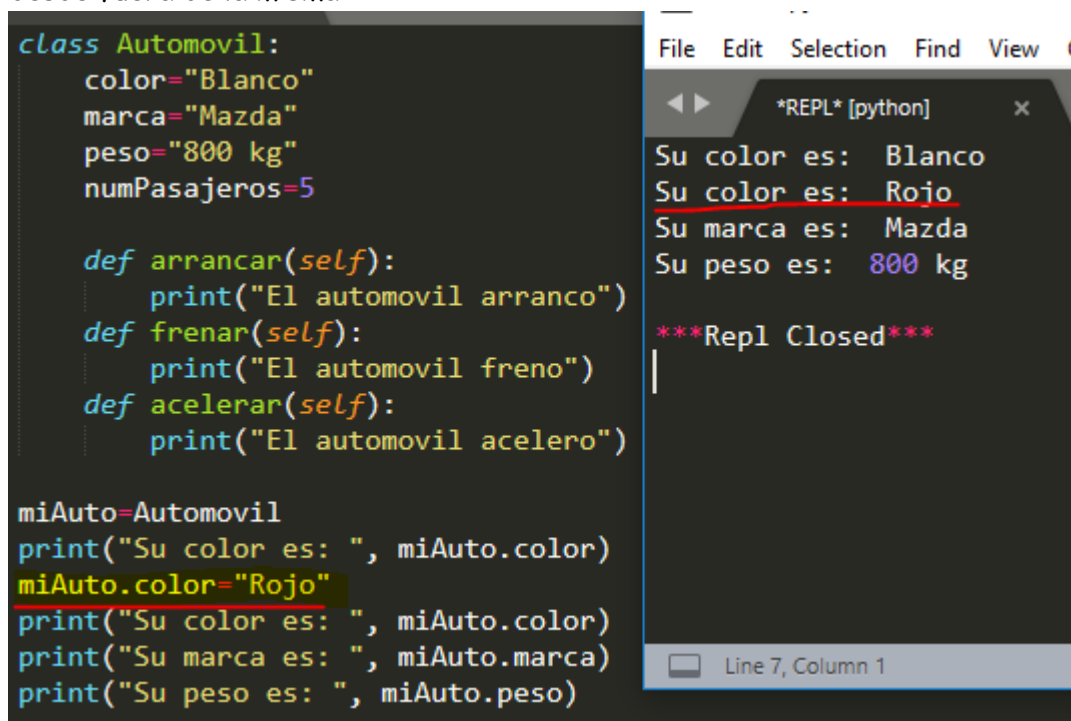


## ENCAPSULAMIENTO

La encapsulación es un mecanismo que consiste en proteger datos y métodos de una clase, evitando el acceso a datos por cualquier otro medio distinto a los especificados. Por lo tanto, la encapsulación garantiza la integridad de los datos (propiedades y métodos) que contiene un objeto

Es la propiedad que permite asegurar que la información de un objeto está oculta del mundo exterior. El encapsulamiento consiste en agrupar en una Clase las características (atributos) con un acceso privado y los comportamientos (métodos) con un acceso público. Acceder o modificar los miembros de una clase a través de sus métodos.

Antes del encapsulamiento, en el siguiente ejemplo podemos ver que la propiedad *color* está siendo modificada desde fuera de la clase, a esto es que le llamamos que NO hay encapsulamiento, ósea que está permitiendo cambiar las propiedades de una clase desde fuera de la misma



```
class Automovil:
    color="Blanco"
    marca="Mazda"
    peso="800 kg"
    numPasajeros=5

    def arrancar(self):
        print("El automovil arranco")
    def frenar(self):
        print("El automovil freno")
    def acelerar(self):
        print("El automovil acelero")

miAuto=Automovil
print("Su color es: ", miAuto.color)
miAuto.color="Rojo"
print("Su color es: ", miAuto.color)
print("Su marca es: ", miAuto.marca)
print("Su peso es: ", miAuto.peso)
```

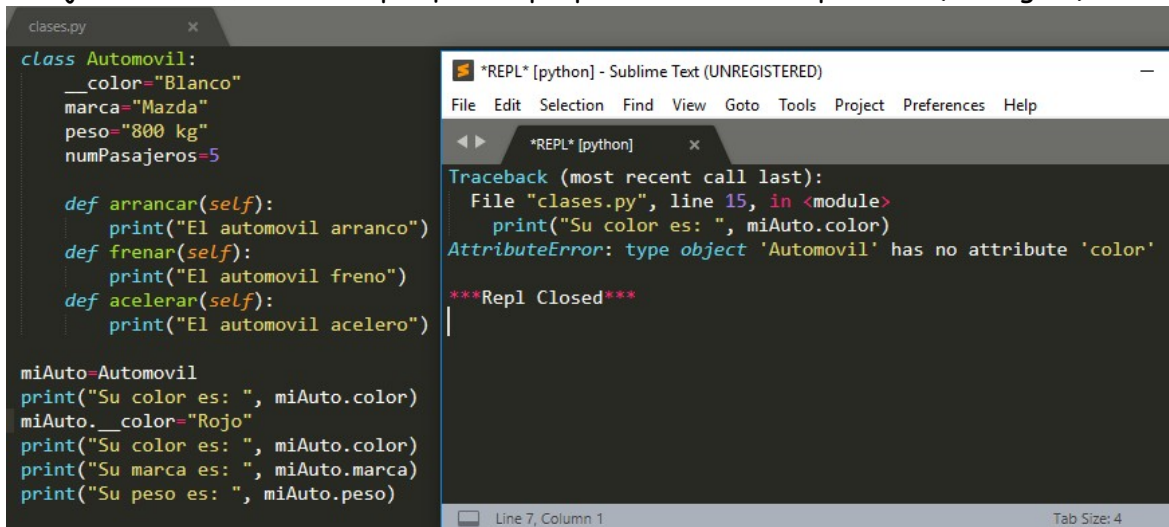
```
Su color es: Blanco
Su color es: Rojo
Su marca es: Mazda
Su peso es: 800 kg

***Repl Closed***
```

Para evitar este acceso a los datos de una clase debemos de encapsular la información, para ello utilizamos la siguiente sintaxis: le antepoñemos dos guiones bajos a la propiedad o método que deseamos encapsular (proteger) `"_propiedad"`; `"_método"`



Como podemos ver a la propiedad *color* la encapsulamos anteponiéndole dos guiones bajos, y tratamos de cambiarle el valor a la propiedad desde fuera de la clase y nos arroja un error, esto es porque la propiedad esta encapsulada (Protegida).



The screenshot shows a Python IDE with two panels. The left panel displays a Python class named `Automovil` with attributes `__color`, `marca`, `peso`, and `numPasajeros`. It includes methods `arrancar`, `frenar`, and `acelerar`. Below the class definition, an instance `miAuto` is created, and its `__color` attribute is attempted to be changed from "Blanco" to "Rojo". The right panel shows a traceback error: `AttributeError: type object 'Automovil' has no attribute 'color'`, indicating that the attribute is not accessible due to encapsulation. The error message also includes `***Repl Closed***`.

```
class Automovil:
    __color="Blanco"
    marca="Mazda"
    peso="800 kg"
    numPasajeros=5

    def arrancar(self):
        print("El automovil arranco")
    def frenar(self):
        print("El automovil freno")
    def acelerar(self):
        print("El automovil acelero")

miAuto=Automovil
print("Su color es: ", miAuto.color)
miAuto.__color="Rojo"
print("Su color es: ", miAuto.color)
print("Su marca es: ", miAuto.marca)
print("Su peso es: ", miAuto.peso)
```

Traceback (most recent call last):  
File "clases.py", line 15, in <module>  
 print("Su color es: ", miAuto.color)  
AttributeError: type object 'Automovil' has no attribute 'color'  
  
\*\*\*Repl Closed\*\*\*

## CONSTRUCTOR

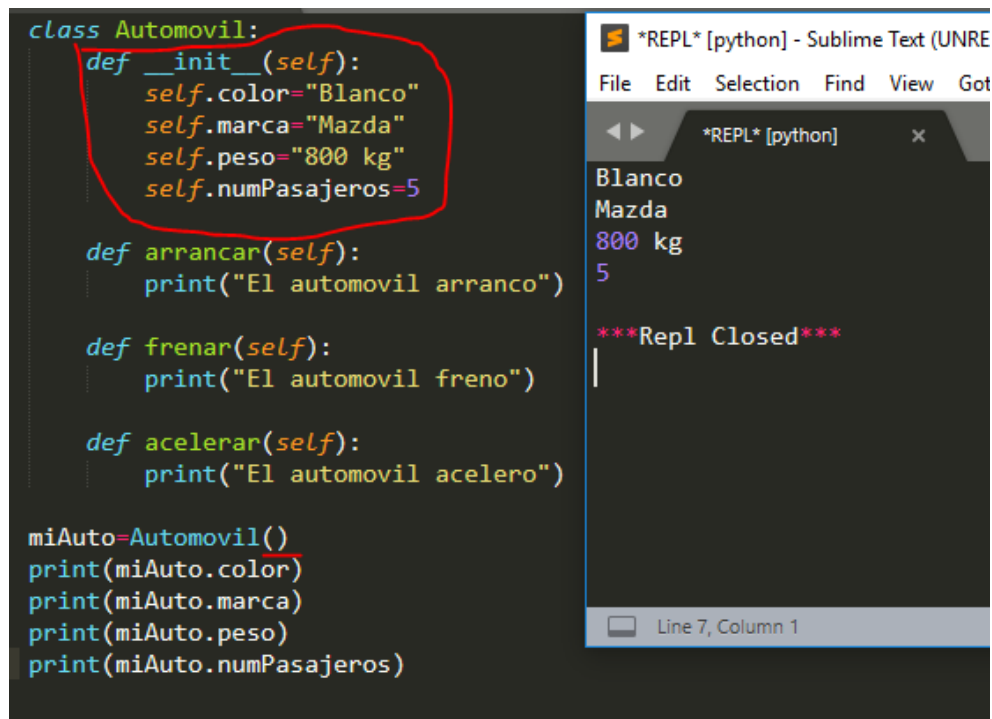
Como podemos observar los dos objetos tienen unas características en común, como son: color, marca, peso y número de pasajeros y unos métodos en común como son: arrancar, acelerar y frenar.

Pero los vehículos tienen otras características que los diferencia entre si como puede ser: Tipo (terrestre, acuático, aéreo), carga(si/no), combustible (gas, acpm, gasolina), etc.

Para estos casos se crea un método que se conoce como el **CONSTRUCTOR** que le permite crear un estado inicial a los objetos que se van a crear a partir de esa clase.

Sintaxis de un Constructor:

```
Def __init__(self)
    Self.__propiedad1
    Self.__propiedad1
    Self.__propiedad1
    ....
nomObleto = nomClase()
```



```
class Automovil:
    def __init__(self):
        self.color="Blanco"
        self.marca="Mazda"
        self.peso="800 kg"
        self.numPasajeros=5

    def arrancar(self):
        print("El automovil arranco")

    def frenar(self):
        print("El automovil freno")

    def acelerar(self):
        print("El automovil acelero")

miAuto=Automovil()
print(miAuto.color)
print(miAuto.marca)
print(miAuto.peso)
print(miAuto.numPasajeros)
```

\*REPL\* [python] - Sublime Text (UNRE)

File Edit Selection Find View Got

Blanco  
Mazda  
800 kg  
5

\*\*\*Repl Closed\*\*\*

Line 7, Column 1

Por lo general las propiedades que están en un constructor son comunes e inmodificables estos los debemos de encapsular. Como las propiedades color, marca, peso y numero de pasajeros están encapsuladas accedemos a ellas por medio de un método llamada "estado", el cual nos imprime los valores de las propiedades de la clase

```
class Automovil:
    def __init__(self):
        self.__color="Blanco"
        self.__marca="Mazda"
        self.__peso="800 kg"
        self.__numPasajeros=5

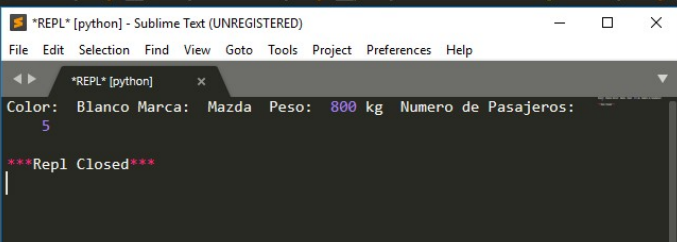
    def estado(self):
        print("Color: ",self.__color, "Marca: ",self.__marca, " Peso: ",self.__peso, " Numero de Pasajeros: ", self.__numPasajeros)

    def arrancar(self):
        print("El automovil arranco")

    def frenar(self):
        print("El automovil freno")

    def acelerar(self):
        print("El automovil acelero")

miAuto=Automovil()
miAuto.estado()
```

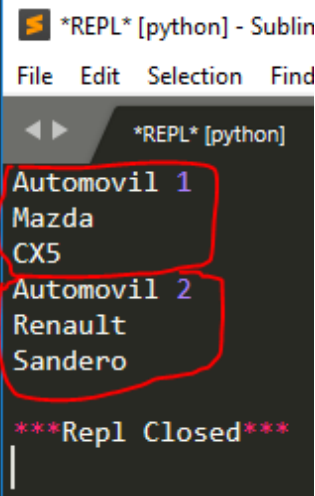


Podemos crear objetos pasándole parámetros al constructor de una clase, en este caso a la hora de instanciar la clase Automóvil ósea al crear el objeto *miAuto* le pasamos la marca y el modelo que le pertenece, y hacemos lo mismo para el objeto 2 *miAuto2*

```
class Automovil():
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

miAuto=Automovil("Mazda","CX5")
print("Automovil 1")
print(miAuto.marca)
print(miAuto.modelo)

print("Automovil 2")
miAuto2 = Automovil("Renault","Sandero")
print(miAuto2.marca)
print(miAuto2.modelo)
```

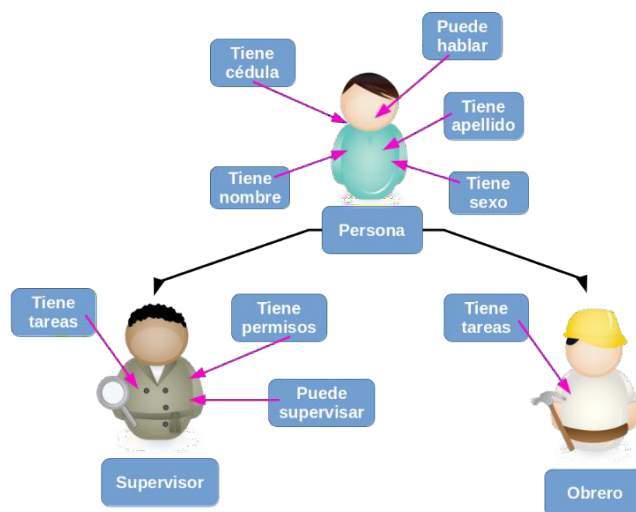
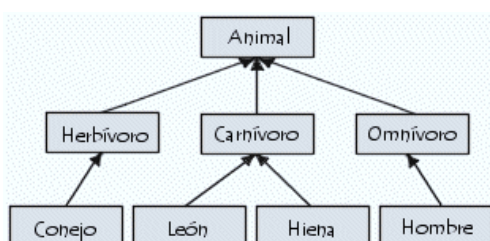


## HERENCIA

La herencia es específica de la programación orientada a objetos, donde una clase nueva se crea a partir de una clase existente. La herencia (a la que habitualmente se denomina subclase) proviene del hecho de que la subclase (la nueva clase creada) contiene los atributos y métodos de la clase primaria. La principal ventaja de la herencia es la capacidad para definir atributos y métodos nuevos para la subclase, que luego se aplican a los atributos y métodos heredados.

### JERARQUÍA DE CLASE

La relación padre-hijo entre clases puede representarse desde un punto de vista jerárquico, denominado vista de clases en árbol. La vista en árbol comienza con una clase general llamada superclase (a la que algunas veces se hace referencia como clase primaria, clase padre, clase principal, o clase madre; existen muchas metáforas genealógicas). Las clases derivadas (clase secundaria o subclase) se vuelven cada vez más especializadas a medida que van descendiendo en el árbol



Sintaxis:

```

Class NomClaseHija(nomClasePadre)
    Atributo1ClaseHija
    Atributo2 ClaseHija
    ....
    Def nombreMetodo1 ClaseHija (self):
        Líneas de código del metodo1
    Def nombreMetodo2 ClaseHija (self):
        Líneas de código del metodo2
  
```

En el siguiente ejemplo tenemos una clase padre llamada *Automóvil*, la cual tiene las propiedades: *marca*, *modelo*, *arrancar*, *acelerar* y un método *estado*.

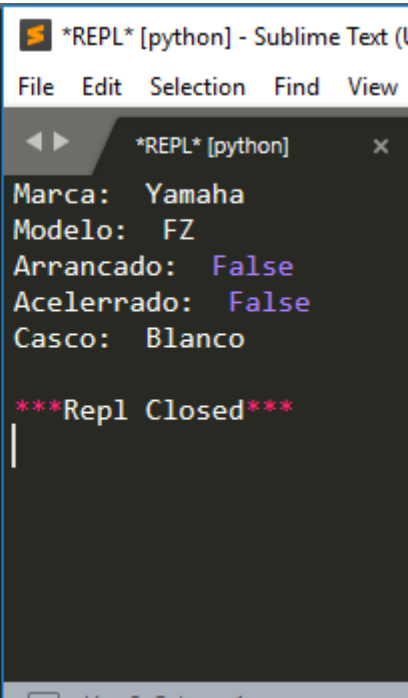
Y una clase hija *Moto* que es una herencia de *Automóvil*, la clase *Moto* tiene en particular una propiedad llamada *casco*

```
class Automovil():
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.arrancar = False
        self.acelerar = False

    def estado(self):
        print("Marca: ",self.marca)
        print("Modelo: ",self.modelo)
        print("Arrancado: ",self.arrancar)
        print("Acelerrado: ",self.acelerar)

class Moto(Automovil):
    casco="Blanco"

miMoto=Moto("Yamaha","FZ")
miMoto.estado()
print("Casco: ",miMoto.casco)
```



## **SOBREESCRITURA DE METODOS**

Otra característica asociada a la herencia es la sobreescritura de métodos. La sobreescritura de métodos nos permite redefinir un método que heredamos para que este funcione de acuerdo a nuestras necesidades y no a lo definido en la superclase. Cuando en un objeto llamamos a un método el compilador comprueba si el método existe en nuestro objeto, si existe lo usa y si no existe en nuestro objeto entonces lo busca en la superclase. Esto ocurre así hasta que el compilador encuentra el método definido. El compilador busca el método de abajo a arriba.

Miremos un ejemplo de sobre escritura de métodos, siguiendo con el mismo ejercicio, podemos ver que la clase padre (*Automovil*) y la clase hija (*Moto*) tiene un método llamado de igual forma *estado*, en este caso se esta sobre escribiendo el método *estado* de la clase hija. En la línea de código *miMoto.estado()*, se invoca el metodo que se encuentra dentro de la clase hija (*Moto*)

```
class Automovil():
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.arrancar = False
        self.acelerar = False

    def estado(self):
        print("Marca: ",self.marca)
        print("Modelo: ",self.modelo)
        print("Arrancado: ",self.arrancar)
        print("Acelerrado: ",self.acelerar)

class Moto(Automovil):
    casco="Blanco"
    canguro=True

    def estado(self):
        print("Marca: ",self.marca)
        print("Modelo: ",self.modelo)
        print("Arrancado: ",self.arrancar)
        print("Acelerrado: ",self.acelerar)
        print("Casco: ",self.casco)
        print("Hace Canguro",self.canguro)

miMoto=Moto("Yamaha","FZ")
miMoto.estado()
```

\*REPL\* [python] - Sublime 1

File Edit Selection Find \

\*REPL\* [python]

Marca: Yamaha  
Modelo: FZ  
Arrancado: False  
Acelerrado: False  
Casco: Blanco  
Hace Canguro True

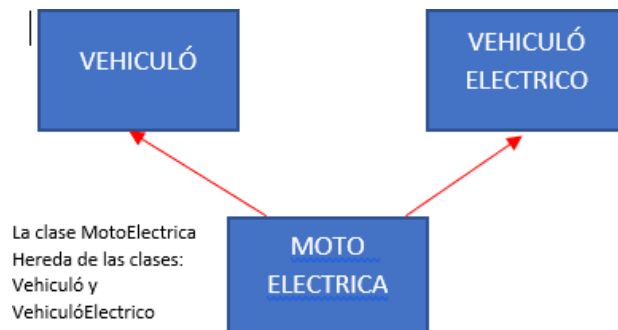
\*\*\*Repl Closed\*\*\*

Line 9, Column 1



## HERENCIA MULTIPLE

La herencia múltiple es la capacidad de una subclase de heredar de múltiples súper clases. Esto conlleva un problema, y es que, si varias súper clases tienen los mismos atributos o métodos, la subclase sólo podrá heredar de una de ellas. En estos casos Python dará prioridad a las clases más a la izquierda en el momento de la declaración de la subclase.



## FUNCION SUPER

Esta función nos permite invocar y conservar un método o atributo de una clase padre (primaria) desde una clase hija (secundaria) sin tener que nombrarla explícitamente. Esto nos brinda la ventaja de poder cambiar el nombre de la clase padre (base) o hija (secundaria) cuando queramos y aún así mantener un código funcional, sencillo y mantenible.

En el siguiente ejemplo podemos ver una clase padre llamada *Persona* y otra clase hija llamada *Alumno* que es herencia de la clase *Persona*, creamos un objeto llamado *per1* de tipo de *Alumno*, esta clase requiere dos parámetros para su construcción *colegio* y *grado* (Liceo,10), posteriormente imprimimos la descripción de la *Persona*, pero como podemos ver nos arroja un error porque aun no se le asigno un *nombre*, *apellido* y *edad* a la *Persona*.

```
class Persona():
    def __init__(self, nombre, apellido, edad):
        self.nombre=nombre
        self.apellido=apellido
        self.edad=edad
    def descripcion(self):
        print("Nombre: ",self.nombre, "\nApellido: ",self.apellido, "\nEdad: ",self.edad)

class Alumno(Persona):
    def __init__(self,colegio,grado):
        self.colegio=colegio
        self.grado=grado

per1=Alumno("Liceo",10)
per1.descripcion()
```

\*REPL\* [python] - Sublime Text (UNREGISTERED)

Traceback (most recent call last):  
File "HerenciaSuper.py", line 15, in <module>  
per1.descripcion()  
File "HerenciaSuper.py", line 7, in descripcion  
print("Nombre: ",self.nombre, "\nApellido: ",self.apellido, "\nEdad: ",self.edad)  
AttributeError: 'Alumno' object has no attribute 'nombre'



Para dar solución a este problema vamos a utilizar la función *super*, que nos permite acceder a las propiedades, métodos y constructores de la clase padre, en este caso accedemos al constructor de la clase *Persona* a través de la función *super*, en este caso la clase *Alumno* en su constructor queda con 5 parámetros: 2 de su propia clase y 3 de la clase *Padre (Persona)*, como se puede ver en la siguiente imagen.

```
class Persona():
    def __init__(self, nombre, apellido, edad):
        self.nombre=nombre
        self.apellido=apellido
        self.edad=edad
    def descripcion(self):
        print("Nombre: ",self.nombre, "\nApellido: ",self.apellido, "\nEdad: ",self.edad)

class Alumno(Persona):
    def __init__(self, colegio, grado, nombre, apellido, edad):
        super().__init__(nombre,apellido,edad)
        self.colegio=colegio
        self.grado=grado

per1=Alumno("Liceo",10,"Juan","Muñoz",11)
per1.descripcion()
```

## FUNCION ISINSTANCE

Esta función permite saber si un objeto es una instancia de una clase, nos retorna True (si es verdadero, ósea si, si es una instancia), False (Falso, sino no es una instancia).

```
class Persona():
    def __init__(self, nombre, apellido, edad):
        self.nombre=nombre
        self.apellido=apellido
        self.edad=edad
    def descripcion(self):
        print("Nombre: ",self.nombre, "\nApellido: ",self.apellido, "\nEdad: ",self.edad)

class Alumno(Persona):
    def __init__(self, colegio, grado, nombre, apellido, edad):
        super().__init__(nombre,apellido,edad)
        self.colegio=colegio
        self.grado=grado

per1=Alumno("Liceo",10,"Juan","Muñoz",11)
per1.descripcion()
print(isinstance(per1,Alumno))
print(isinstance(per1,Persona))
per2=Persona("Ana","Gomez",16)
print(isinstance(per2,Alumno))
```

## **POLIMORFISMO**

En programación orientada a objetos se denomina polimorfismo a la capacidad que tienen los objetos de una clase de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación. Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa.

En algunos lenguajes, el término polimorfismo es también conocido como 'Sobrecarga de parámetros' ya que las características de los objetos permiten aceptar distintos parámetros para un mismo método (diferentes implementaciones) generalmente con comportamientos distintos e independientes para cada una de ellas.

En el siguiente ejercicio entendernos este concepto, tenemos tres clases *Bicicleta*, *Carro* y *Moto*, las cuales tienen un método (comportamiento) llamado *desplazamiento*, el cual imprime un mensaje. En este caso para acceder a cada uno de los comportamientos debemos de crear un objeto para cada clase. Como se muestra la siguiente imagen.

```
class Bicicleta():
    def desplazamiento(self):
        print("Te estas desplazando en una BICICLETA")

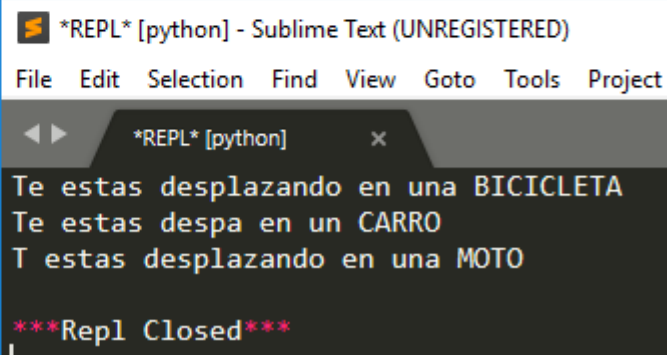
class Carro():
    def desplazamiento(self):
        print ("Te estas despa en un CARRO")

class Moto():
    def desplazamiento(self):
        print ("T estas desplazando en una MOTO")

vehiculo1 = Bicicleta()
vehiculo1.desplazamiento()

vehiculo2 = Carro()
vehiculo2.desplazamiento()

vehiculo3 = Moto()
vehiculo3.desplazamiento()
```



EL problema es cuando tengamos demasiados *vehículos* (Camión, triciclo, barco, avión, helicóptero, bicicleta, moto, carro, etc.), en este caso se debería crear un objeto para cada clase.

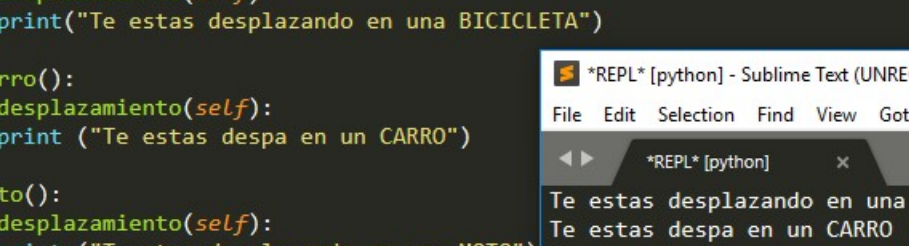
```
class Bicicleta():
    def desplazamiento(self):
        print("Te estas desplazando en una BICICLETA")

class Carro():
    def desplazamiento(self):
        print("Te estas despa en un CARRO")

class Moto():
    def desplazamiento(self):
        print("T estas desplazando en una MOTO")

def desplazamientoVehiculo(vehiculo):
    vehiculo.desplazamiento()

vehiculo=Bicicleta()
desplazamientoVehiculo(vehiculo)
vehiculo=Carro()
desplazamientoVehiculo(vehiculo)
vehiculo=Moto()
desplazamientoVehiculo(vehiculo)
```



\*REPL\* [python] - Sublime Text (UNREGISTERED)

File Edit Selection Find View Goto Tools Proj

\*REPL\* [python] x

Te estas desplazando en una BICICLETA  
Te estas despa en un CARRO  
T estas desplazando en una MOTO

\*\*\*Repl Closed\*\*\*



**INTEGRACIÓN**  
con la educación  
**MEDIA**