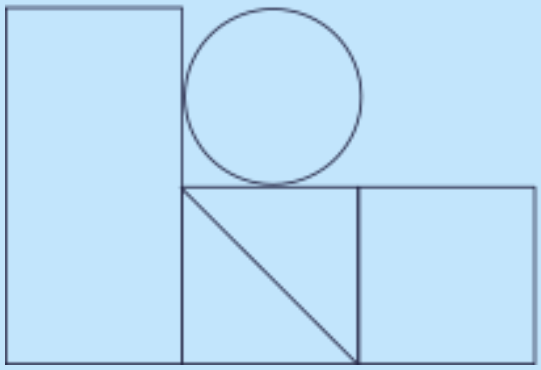
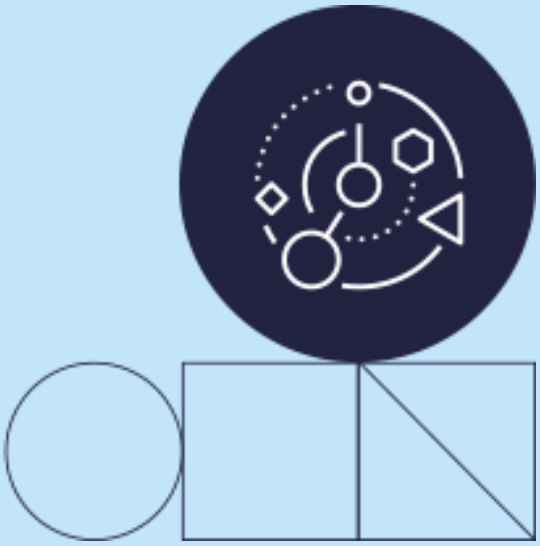
IBM SkillsBuild

IBM

POO con Python

Herencia, Encapsulación y Polimorfismo



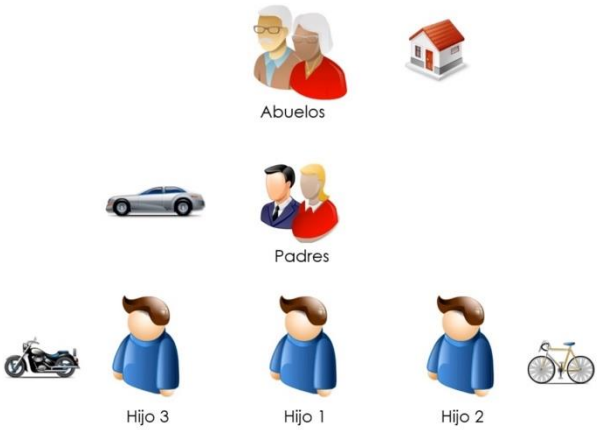


Índice

Introducción

La herencia en programación es un concepto que se asemeja perfectamente al de herencia en la vida real.

Supongamos que tenemos una familia compuesta por unos abuelos, que poseen una casa, unos hijos, que tienen un coche y unos nietos que poseen, por ejemplo, una moto y una bicicleta.



Cuando los abuelos fallezcan, los hijos pasarán a tener la casa de los abuelos como patrimonio junto al coche que ya poseían.

Y cuando los padres fallezcan, sus hijos heredarán tanto la casa como el coche de sus padres, de modo, que al final, cada hijo tendrá como patrimonio lo que ya poseía más el patrimonio heredado de sus padres que a su vez lo heredaron de sus abuelos.

Pues la herencia en Python se comporta igual que en la vida real. Podemos tener uno o varios objetos "hijo" que herede de uno o varios objetos "padre" de forma que los hijos heredan los atributos y los métodos de los padres y pueden usarlos como si fuesen suyos propios. A esto lo llamamos una jerarquía de herencia.

La herencia en programación nos proporciona dos ventajas muy claras:

- **Reutilización de código**: Si tenemos dos objetos que van a tener propiedades y atributos idénticos, podemos hacer que uno de ellos herede del otro y nos ahorramos tener que implementar todos los métodos y atributos en ambos objetos.
- **Simplificación**: Cuando utilizamos la herencia el código se simplifica sustancialmente.

A la hora de pasar todo esto a código, en Python, para indicar que una clase hereda de otra, la sintaxis sería la siguiente:

```
Class Coche(Vehiculo):
    pass
```

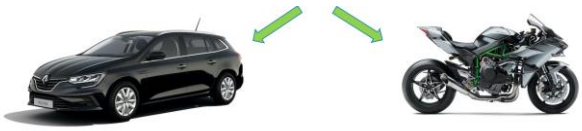
Nota: Usaremos la palabra clave **pass** cuando no deseemos agregar otras propiedades o métodos a la clase, es decir, cuando vayamos a dejar la clase vacía (de momento).

Aquí estamos indicando que la clase **Coche** hereda de **Vehículo**. Se heredan atributos y métodos, incluido el constructor.

Vamos a ver otro ejemplo de jerarquía de herencia. Supongamos que tenemos una clase padre vehículo con las siguientes propiedades y métodos:

Vehículo
Marca
Modelo
Color = “negro”
Arrancado = False
Parado = True
Arrancar ()
Parar ()
Resumen()

Ahora necesitamos crear dos clases hijas, **Coche** y **Moto** que heredarán de **Vehículo**.



Posteriormente crearemos una instancia de la clase **Coche** y otra de la clase **Moto**.

Vamos a pasar esta jerarquía de herencia a código:

```
class Vehiculo():
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.color = "Negro"
        self.arrancado = False
        self.parado = True

    def arrancar(self):
        self.arrancado = True
        self.parado = False

    def parar(self):
        self.parado = True
        self.arrancado = False

    def resumen(self):
        print("Marca:", self.marca, "\n",
              "Modelo:", self.modelo,
"\n",
              "Color:", self.color, "\n",
              "Está arrancado:",
self.arrancado,"\n",
              "Está parado:", self.parado
              )


miCoche = Vehiculo("Renault", "Megane")

miCoche.arrancar()

miCoche.resumen()

class Moto(Vehiculo):
    pass

miMoto = Moto("Kawasaki", "Ninja")

miMoto.resumen()
```

Veamos un segundo ejemplo en el que nuestra jerarquía de clases tenga una clase “abuelo” una clase “hija” y una clase “nieta”.

En esta ocasión vamos a crear una clase **Vehículo** de la que heredará una clase **Moto** y que a su vez será padre de una clase **Kwad**, es decir, gráficamente sería algo como:

Pasemos el ejemplo a código:

```
class Vehiculo():
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.color = "negro"
        self.arrancado = False
        self.parado = True
```

```
    def arrancar(self):
        self.arrancado = True
        self.parado = False

    def parar(self):
        self.parado = True
        self.arrancado = False

    def resumen(self):
        print("El modelo es un coche",
"\n",
            "Marca:", self.marca, "\n",
              "Modelo:", self.modelo,
"\n",
              "Color:", self.color, "\n",
              "Está arrancado:",
self.arrancado,"\n",
              "Está parado:", self.parado
              )


miCoche = Vehiculo("Renault", "Megane")

miCoche.arrancar()

miCoche.resumen()

class Moto(Vehiculo):
    is_carenado = False

      #Método propio de la clase Moto, no
heredado del padre.
    def poner_carenado(self):
        self.is_carenado = True

      #La clase Moto sobrescribe el
método resumen() heredado del padre
    def resumen(self):
        print("El modelo es una moto",
"\n",
            "Marca:", self.marca, "\n",
            "Modelo:", self.modelo, "\n",
            "Color:", self.color, "\n",
            "Está arrancado:",
self.arrancado,"\n",
            "Está parado:", self.parado,
"\n",
            "Tiene carenado:",
self.is_carenado
            )

miMoto = Moto("Kawasaki", "Ninja")

miMoto.resumen()

class kwad(Moto):
    pass

miKwad = kwad("Linhai", "LH 500")

miKwad.resumen()
```

Como podemos ver en este ejemplo, la clase **Moto**, a pesar de haber heredado el método **resumen()** lo ha reescrito con nuevas características. Es lo que se denomina sobreescritura de métodos. Es decir, que una clase hija herede un método de una clase padre no significa que esté obligada a usarlo siempre tal cual lo heredó, lo puede modificar según sus necesidades. Si no lo modifica, lo heredará tal cual está implementado en el padre, como es el caso de la clase **Kwad**, que ha heredado el método **resumen()** de la clase **Moto** tal y como está implementado, sin modificarlo.

Herencia múltiple

Como hemos comentado anteriormente, Python admite la herencia múltiple, es decir, admite que una clase herede de dos o más clases. O dicho de otro modo, que una clase “hija” pueda heredar de dos o más clases “padre”. No todos los lenguajes de programación orientada a objetos soportan esta característica.

La sintaxis a usar en este caso es simplemente poner entre paréntesis y separados por una coma las clases padre de las que heredará nuestra clase hija.

En el siguiente ejemplo la clase bicicleta eléctrica hereda de **vehículos** y de **vehículos eléctricos**:

```
class B_electrica(Vehículos,
V_electricos):
```

Simplemente poniendo entre paréntesis las clases de las que hereda, nuestra clase hija ya tiene todos los métodos y las propiedades de ambas clases.

Muy importante: Se da preferencia siempre a la primera clase que se indique entre paréntesis. Hereda el constructor de la primera clase que pusimos en el paréntesis, y en caso de que haya métodos comunes, también hereda el del primero.

Para sobrescribir un método heredado de la clase padre, simplemente volvemos a escribir el método con todos sus argumentos y añadimos el nuevo argumento.

Por ejemplo, suponiendo que tenemos un método **estado()** de la clase padre "**vehículo**":

```
def estado(self):
    print("Marca", self.marca,"Modelo",
self.modelo)
```

Si quisiéramos sobrescribir dicho método en una clase hija "**coche**" y añadir el método "**cilindrada**", que se supone que ya he instanciado:

```
def cilindrada(self):
    self.cilindrada=3000

def estado(self):
    print("Marca", self.marca,"Modelo",
self.modelo, "Cilindrada", self.cilindrada)
```

Podemos ver cómo hemos sobrescrito el método **estado()** del padre para añadir el atributo **cilindrada**, que es propio del hijo y no existe en el padre.

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 aun así mantener un código funcional, sencillo y mantenible.

Supongamos que queremos que una clase hija o secundaria herede los atributos de la clase padre. Si nosotros no recurrimos a la función **super()**, o llamamos al constructor **init** especificando los atributos, deberemos reescribirlos, lo cual en una clase por ejemplo con 20 atributos sería una pérdida de tiempo enorme.

Para personalizar el constructor del padre de acuerdo a las necesidades del hijo se usa **super().**

```
class Persona():
    def __init__(self, nombre, edad,
lugar):
        self.nombre=nombre
        self.edad=edad
        self.lugar=lugar


    def descripcion(self):
        print("El nombre es ",
self.nombre, ", tiene ", self.edad, "
anyos", " y es de ", self.lugar)

class Empleado(Persona):
    def __init__(self, salario,
antiguedad, nombre_emp, edad_emp,
lugar_epm):
```

```
        super().__init__(nombre_emp,
edad_emp, lugar_epm)
        self.salario=salario
        self.antiguedad=antiguedad

    def descripcion(self):
        super().descripcion()
        print("Salario: ", self.salario,
", antiguedad: ", self.antiguedad)

Angel=Persona("Angel", 43, "Malaga")
Angel.descripcion()

Empleado1=Empleado(2000, 2017, "Manolo",
33, "Madrid")
Empleado1.descripcion()
```

Veamos otro ejemplo de uso de **super()**:

Jerarquía de clases sin usar **super()**, sobrescribiendo los atributos del padre:

```
class Padre(): #Creamos la clase Padre
    def __init__(self, ojos, cejas):
#Definimos los Atributos en el
constructor de la clase
        self.ojos = ojos
        self.cejas = cejas

class Hijo(Padre): #Creamos clase hija
que hereda de Padre
    def __init__(self, ojos, cejas,
cara): #Definimos los atributos en el
constructor
        self.ojos = ojos #Sobreescribimos
cada atributo
        self.cejas = cejas
        self.cara = cara #Especificamos
el nuevo atributo para Hijo

Tomas = Hijo('Marrones', 'Negras',
'Larga') #Instanciamos
print (Tomas.ojos, Tomas.cejas,
Tomas.cara) #Imprimimos los atributos del
objeto
```

Mismo ejemplo llamando al constructor de la clase padre:

```
class Padre(object): #Creamos la clase
Padre
    def __init__(self, ojos, cejas):
#Definimos los Atributos
        self.ojos = ojos
        self.cejas = cejas


class Hijo(Padre): #Creamos clase hija
que hereda de Padre
    def __init__(self, ojos, cejas,
cara): #creamos el constructor de la
clase especificando atributos
        Padre.__init__(self, ojos, cejas)
#Especificamos la clase y llamamos a su
constructor + Atributos
        self.cara = cara #Especificamos
el nuevo atributo para Hijo

Tomas = Hijo('Marrones', 'Negras',
'Larga')
print (Tomas.ojos, Tomas.cejas,
Tomas.cara)
```

Utilizando **super()**. De esta forma es casi el mismo código, pero no necesitamos especificar la clase padre, por lo que podremos cambiarle el nombre en cualquier momento y nuestro código seguirá funcional.

```
class Padre(object): #Creamos la clase
Padre
    def __init__(self, ojos, cejas):
#Definimos los Atributos
        self.ojos = ojos
        self.cejas = cejas

class Hijo(Padre): #Creamos clase hija
que hereda de Padre
    def __init__(self, ojos, cejas,
cara): #creamos el constructor de la
clase especificando atributos
```

```
        super().__init__(ojos,
cejas)#Solicitamos a super llamar de la
clase padre esos atributos
        self.cara = cara #Especificamos
el nuevo atributo para Hijo

Tomas = Hijo('Marrones', 'Negras',
'Larga')
print (Tomas.ojos, Tomas.cejas,
Tomas.cara)
```

De todas estas opciones debemos quedarnos con el uso de **super()** como forma de programar más correcta.

Nota: En el caso de la Herencia Múltiple **super()** no nos sirve. Debemos llamar a los constructores de ambas clases especificándolas por su nombre y si cambiamos el nombre u orden de la clase deberemos especificarlo.

Encapsulación

En la mayoría de lenguajes que usan el paradigma de la POO nos encontramos con el concepto de encapsulación. Este consiste en poder ocultar los atributos o métodos de una clase para que no se puedan cambiar salvo mediante otros métodos que dispongamos nosotros. Es decir, la encapsulación consiste en denegar el acceso a los atributos y métodos internos de la clase desde el exterior.

Supongamos que tenemos una clase **Coche** con un atributo que sea **color = "negro"**. Cualquiera podría cambiar ese atributo simplemente poniendo **color = "rojo"**, por ejemplo. A veces no nos interesa que un atributo se pueda cambiar. Una forma de "blindarlo" para que no se pueda cambiar su valor es mediante la encapsulación.

Tendremos la posibilidad de especificar unos modificadores a nuestros atributos que permitirán o no que su valor se pueda cambiar.

Como norma general tendremos tres modificadores de acceso:

- **Public**: Los atributos public serán accesibles y modificables desde cualquier parte de nuestro código. Es el valor por defecto y sería el equivalente a no poner nada.
- **Protected**: Podemos acceder a él desde la misma clase y clases hijas.
- **Private**: Accesible únicamente desde su clase.

Pero todo esto que hemos explicado no se aplica a Python. En **Python** no se especifican métodos o atributos privados ni públicos. Esto es así porque en **Python** todos los atributos de una clase son públicos. Es decir, técnicamente en Python no existe la encapsulación.

No obstante, si queremos tener algún atributo o método "oculto" a la interfaz pública, es posible indicarlo con la siguiente convención:

1. Por defecto, todos los atributos son públicos. Python asume que "Aquí todos somos adultos" y sabemos leer la documentación de nuestro código y utilizarlo bien.

2. Si queremos indicar que algún atributo miembro de una clase es privado, lo haremos añadiendo el símbolo '_' delante del nombre del tributo. Por ejemplo:

'MiClase._atributoPrivado'. Esto no hace que el atributo sea privado como en otros lenguajes de programación, es sólo una indicación al programador, para que sepa que debe usarlo (aunque si quiere, puede hacerlo).

3. Si queremos una "protección extra" utilizaremos 2 guiones *'__'*, en lugar de uno: p. ej. ***'MiClase.__otroAtribPrivado'.*** Esto además hará que el intérprete no lo muestre cuando llamemos a la función ***help*** para obtener más información de la clase, etc.

Por lo tanto, Python proporciona una implementación conceptual de modificadores de acceso público, protegido y privado, pero no como otros lenguajes como C# , Java, C++, etc, es simplemente una encapsulación “simulada”.

Para acceder a esos datos encapsulados se deberían crear métodos públicos que hagan de interfaz. En otros lenguajes les llamaríamos **getters** y **setters** y es lo que da lugar a las propiedades, que no son más que atributos protegidos con interfaces de acceso:

```
class Ejemplo:
    __atributo_privado = "Soy un atributo
inalcanzable desde fuera."

    def __metodo_privado(self):
        print("Soy un método inalcanzable
desde fuera.")

    def atributo_publico(self):
        return self.__atributo_privado

    def metodo_publico(self):
        return self.__metodo_privado()

e = Ejemplo()
print(e.atributo_publico())
e.metodo_publico()
```

```
Soy un atributo inalcanzable desde fuera.
Soy un método inalcanzable desde fuera.
```

Pero esta simulación de **getter** y **setters** es simplemente eso, una simulación.

Veamos otro ejemplo de encapsulación en Python:

```
class Coche:
    # Método constructor
    def __init__(self):
        self.__largo = 250
        self.__ancho = 120
        self.__ruedas = 4
        self.__peso = 900
        self.__color = "rojo"
        self.__is_enMarcha = False

    # Declaración de métodos
    def arrancar(self):  # self hace
referencia a la instancia de clase.
        self.is_enMarcha = True  # Es
como si pusiésemos miCoche.is_enMarcha =
True

    def estado(self):
        if (self.is_enMarcha == True):
            return "El coche está
arrancado"
        else:
            return "El coche está parado"

        # Declaración de una instancia de
clase, objeto de clase o ejemplar de
clase.


miCoche = Coche()

miCoche.__ruedas = 9
print("Mi coche tiene", miCoche.__ruedas,
"ruedas.")
```

En este ejemplo hemos creado una clase Coche con una serie de atributos privados: **largo**, **ancho**, **ruedas**, **peso**, **color**, **is_enMarcha**. En otro lenguaje de programación con encapsulación real no podríamos cambiar los valores de dichos atributos.

En Python sí podríamos, pero como los hemos declarado como privados, vamos a entender que el programador no quiere que el valor de dichos métodos se cambie manualmente y para ello nos ha ofrecido un método, **arrancar()**, que cambia el valor del atributo **is_enMarcha**. Se supone que, si quisiésemos cambiar el valor de ese atributo, lo ideal es usar el método implementado para tal efecto y no hacerlo de forma manual.

Polimorfismo

Si hablamos de herencia, tenemos que hablar de polimorfismo. Este término se refiere a la capacidad de que todos los objetos pertenecientes a una misma familia de clases, es decir que heredan de la misma clase, pueden ser llamados con los métodos de la clase padre que han sobrecargado, pero se comportarán con las llamadas a sus propios métodos sobrecargados.

El polimorfismo es una propiedad de la herencia por la que objetos de distintas subclases pueden responder a una misma acción.

El polimorfismo está implícito en Python, ya que todas las clases son subclases de una superclase común llamada **Object**.

Por ejemplo, la siguiente función aplica una rebaja al precio de un producto:

```
def rebajar_producto(producto, rebaja):
    producto.pvp = producto.pvp -
(producto.pvp/100 * rebaja)
```

Gracias al polimorfismo no tenemos que comprobar si un objeto tiene o no el atributo **pvp**, simplemente intentamos acceder y si existe, premio:

```
print(alimento, "\n")
rebajar_producto(alimento, 10)
print(alimento)
```

```
REFERENCIA      2035
NOMBRE          Botella de Aceite de Oliva
PVP             5
DESCRIPCIÓN     250 ML
PRODUCTOR       La Aceitera
DISTRIBUIDOR    Distribuciones SA

REFERENCIA      2035
NOMBRE          Botella de Aceite de Oliva
PVP             4.5
DESCRIPCIÓN     250 ML
PRODUCTOR       La Aceitera
DISTRIBUIDOR    Distribuciones SA
```

Ejemplo 2:

```
class Cuadrado(Poligono):

        def __init__(self, lado,
color=None):
            Poligono.__init__(self,4,colo
r)
            self.lado = lado

        def show(self):
            super().show()
            print('lado:', self.lado)

c1 = Cuadrado(2, 'verde')   #Declaramos
un cuadrado

poligonos = t1, t2, c1      #Tupla con
dos Trinagulo y un Cuadrado

for poligono in poligonos:
    poligono.show()
    print()
```

```
Color: None
Lados: 3
Base: 3
Altura 4

Color: blanco
Lados: 3
Base: 10
Altura 1
```

```
Color: verde
Lados: 4
Lado: 2
```

En este ejemplo estamos, primero definiendo la clase **Cuadrado,** que hereda de **Poligono.** Creamos una tupla con dos **Triangulo** y un **Cuadrado** y recorriéndola con un bucle **for.** Como vemos, llamamos al método **show** de cada elemento de la tupla. Como es lógico, cada llamada produce el resultado específico para el tipo de polígono al que pertenece. Notad que esto en **Python** es trivial ya que siempre tratamos directamente con referencias a objetos, así que la llamada a **show** es siempre la correspondiente a la del objeto referenciado.

Ejemplo 3:

```
class Empleado:
    def __init__(self, nombre, sueldo):
        self.nombre = nombre
        self.sueldo = sueldo

    def __str__(self):
        return f'Empleado: [Nombre:
{self.nombre}, Sueldo: {self.sueldo}]'

    def mostrar_detalles(self):
        return self.__str__()


class Gerente(Empleado):
    def __init__(self, nombre, sueldo,
departamento):
        super().__init__(nombre, sueldo)
        self.departamento = departamento

    def __str__(self):
        return f'Gerente [Departamento:
{self.departamento}] {super().__str__()}'

    # def mostrar_detalles(self):
    #     return self.__str__()



def imprimir_detalles(objeto):
    # print(objeto)
    print(type(objeto))
    print(objeto.mostrar_detalles())


empleado = Empleado('Juan', 5000)
imprimir_detalles(empleado)

gerente = Gerente('Karla', 6000,
'Sistemas')
imprimir_detalles(gerente)
```