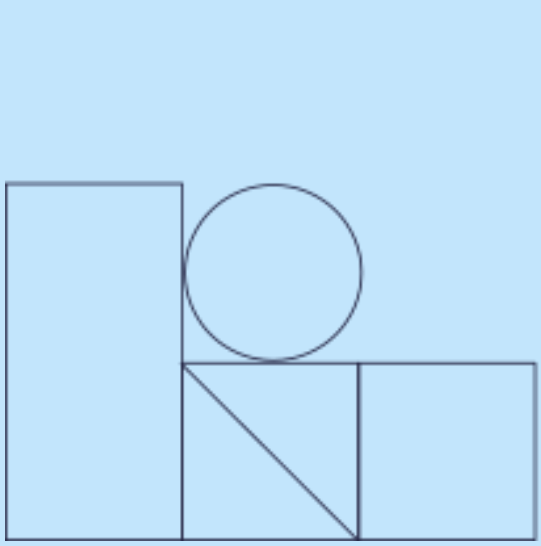
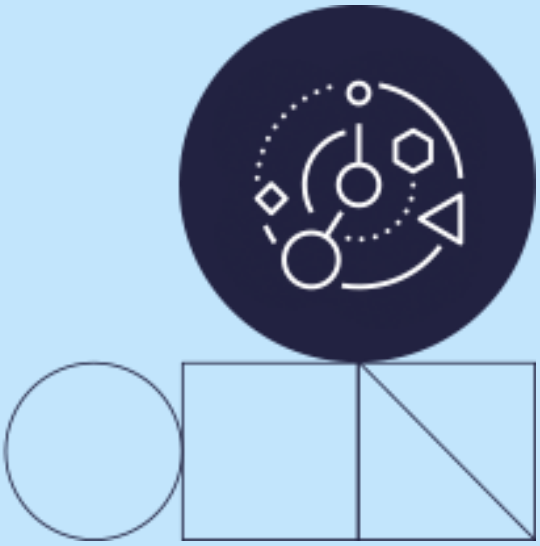
IBM SkillsBuild

IBM

OOP com Python

Herança, Encapsulamento e Polimorfismo





Índice

Introdução

Herança na programação é um conceito que se assemelha perfeitamente à herança na vida real.

Suponha que tenhamos uma família composta por avós, que possuem uma casa, filhos, que possuem um carro, e netos, que possuem, por exemplo, uma motocicleta e uma bicicleta.

Quando os avós falecerem, as crianças terão a casa dos avós como uma propriedade junto com o carro que já possuíam.

E quando os pais morrerem, seus filhos herdarão tanto a casa dos pais quanto o carro dos pais, de modo que, no final, cada criança herdará o que já possuía mais a riqueza herdada de seus pais, que por sua vez a herdaram de seus avós.

Bem, a herança em Python se comporta como na vida real. Podemos ter um ou mais objetos "filhos" que herdam de um ou mais objetos "pais" para que os filhos herdem os atributos e métodos dos pais e possam usá-los como se fossem seus próprios. Chamamos isto de uma hierarquia de herança.

A herança na programação nos dá duas vantagens muito claras:

- **Reutilização de código**: Se tivermos dois objetos que vão ter propriedades e atributos idênticos, podemos fazer um deles herdar do outro e economizar a necessidade de implementar todos os métodos e atributos em ambos os objetos.
- **Simplificação**: Quando utilizamos a herança, o código é substancialmente simplificado.

Ao passar tudo isso para o código, em Python, para indicar que uma classe herda de outra, a sintaxe seria a seguinte:

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

Nota: Vamos usar a palavra-chave **pass** quando não queremos acrescentar outras propriedades ou métodos à classe, ou seja, quando vamos deixar a classe vazia (por enquanto).

Aqui estamos indicando que a classe **O carro herda de Vehículo**. São herdados, incluindo o construtor.

Vejamos outro exemplo de hierarquia sucessória. Suponha que tenhamos uma classe de veículos com as seguintes propriedades e métodos:

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

Agora precisamos criar duas classes de filhas, **Coche** e **Moto** que herdarão de **Vehículo**.

Posteriormente, criaremos uma instância da classe **Coche** e outro da classe **Moto**.

Vamos passar esta hierarquia de herança para o 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()
```

Vejamos um segundo exemplo onde nossa hierarquia de classes tem uma classe "avô", uma classe " filha" e uma classe "neto".

Desta vez vamos criar uma classe **Vehículo** da qual herdará uma classe **Moto** e que, por sua vez, será o pai de uma classe **Kwad**, ou seja, graficamente, seria algo como:

Vamos transformar o exemplo em 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 neste exemplo, a classe **Moto**, apesar de ter herdado o **resumen()** reescreveu-o com novas características. Isto é chamado de sobregravação de método. Ou seja, que uma classe infantil herde um método de uma classe de pais não significa que ela seja obrigada a usá-lo sempre como o herdou, ela pode modificá-lo de acordo com suas necessidades. Se não o modificar, ele o herdará como é implementado no pai, como é o caso da classe **Kwad**, que herdou o método **resumen()** da classe **Moto** conforme implementado, sem modificá-lo.

Herança múltipla

Como mencionado acima, Python suporta herança múltipla, ou seja, permite que uma classe herde de duas ou mais classes. Em outras palavras, uma classe " filha " pode herdar de duas ou mais classes " pai ". Nem todas as linguagens de programação orientadas a objetos suportam esta característica.

A sintaxe a ser usada neste caso é simplesmente colocar entre parênteses e separar por vírgula as classes dos pais das quais nossa classe infantil herdará.

No exemplo a seguir, a classe de bicicletas elétricas herda de **vehículos** e **vehículos eléctricos**:

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

Simplesmente colocando as classes das quais herda entre parênteses, nossa classe infantil já possui todos os métodos e propriedades de ambas as classes.

Muito importante: A preferência é sempre dada à primeira classe indicada entre parênteses. Herda o construtor da primeira classe que colocamos entre parênteses, e caso haja métodos comuns, herda também o construtor da primeira classe.

Para sobregravar um método herdado da classe mãe, simplesmente reescrevemos o método com todos os seus argumentos e acrescentamos o novo argumento.

Por exemplo, supondo que temos um método **estado()** da classe pai "**vehículo**":

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

Se quiséssemos sobrescrever esse método em uma classe filha "**coche**" e acrescente o método "**cilindrada**", que eu já deveria ter instanciado:

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

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

Podemos ver como sobregravamos o método **estado()** do pai para adicionar o atributo **cilindrada**, que é próprio do filho e não existe no pai.

Super()

Esta função nos permite invocar e preservar um método ou atributo de uma classe pai (primário) de uma classe filha (secundário) sem ter que nomeá-lo explicitamente. Isto nos dá a vantagem de poder mudar o nome da classe dos pais (base) ou da criança (criança) sempre que quisermos e ainda manter o código funcional, simples e de fácil manutenção.

Suponhamos que queremos que uma classe infantil herde os atributos da classe dos pais. Se não utilizarmos o **super()**, ou chamamos o construtor **init** especificando os atributos, teremos que reescrevê-los, o que numa classe com 20 atributos, por exemplo, seria uma enorme perda de tempo.

Para personalizar o construtor dos pais de acordo com as necessidades da criança que utilizamos **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()
```

Vejamos outro exemplo do uso de **super()**:

Hierarquia de classes não utilizada **super()**, sobrescrevendo os atributos dos pais:

```
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
```

O mesmo exemplo chamando o construtor da classe pai:

```
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)
```

Usando **super()**. Assim, é quase o mesmo código, mas não precisamos especificar a classe mãe, então podemos renomeá-la a qualquer momento e nosso código ainda funcionará.

```
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 essas opções, devemos nos ater ao uso de **super()** como uma forma mais correta de programação.

Nota: No caso de Herança Múltipla **super()** não tem utilidade para nós. Devemos chamar os construtores de ambas as classes especificando-as pelo nome e se mudarmos o nome ou a ordem da classe, devemos especificá-la.

Encapsulamento

Na maioria das línguas que utilizam o paradigma OOP, encontramos o conceito de encapsulamento. Isto consiste em poder ocultar os atributos ou métodos de uma classe para que não possam ser alterados, exceto por meio de outros métodos que temos. Em outras palavras, o encapsulamento consiste em negar o acesso aos atributos e métodos internos da classe a partir do exterior.

Suponha que tenhamos uma classe **Coche** com um atributo que é **color = "negro"**. Qualquer pessoa poderia mudar esse atributo simplesmente colocando **color = "rojo"**, por exemplo. Algumas vezes não queremos que um atributo possa ser alterado. Uma maneira de "protegê-lo" para que seu valor não possa ser alterado é através do encapsulamento.

Teremos a possibilidade de especificar alguns modificadores para nossos atributos que permitirão ou não que seu valor seja alterado.

Como regra geral, teremos três modificadores de acesso:

- **Public**: Os atributos public serão acessíveis e modificáveis a partir de qualquer lugar em nosso código. É o valor padrão e seria o equivalente a não colocar nada.
- **Protected**: Pode ser acessado a partir da mesma classe e classes filhas.
- **Private**: Acessível apenas a partir de sua sala de aula.

Mas tudo isso não se aplica à Python. Em **Python** não são especificados métodos ou atributos privados ou públicos. Isto é porque em **Python** todos os atributos de uma classe são públicos. Ou seja, tecnicamente, não há nenhum encapsulamento em Python.

Entretanto, se quisermos ter algum atributo ou método "escondido" da interface pública, é possível indicá-lo com a seguinte convenção:

1. Por padrão, todos os atributos são públicos. Python assume que "aqui somos todos adultos" e sabemos ler a documentação de nosso código e usá-lo bem.

2. Se quisermos indicar que um atributo de membro de uma classe é privado, o faremos adicionando o símbolo '_' na frente do nome do atributo. Por exemplo:

'MiClase._atributoPrivado'. Isto não torna o atributo privado como em outras linguagens de programação, é apenas uma dica para o programador, para que ele saiba usá-lo (embora se ele quiser, ele pode fazer isso).

3. Se quisermos "proteção extra", usaremos 2 traços '__', em vez de um: p. ex. ***'MiClase.__otroAtribPrivado'.*** Isso também impedirá que o intérprete a exiba quando chamamos a função ***help*** para mais informações sobre a classe, etc.

Portanto, Python fornece uma implementação conceitual de modificadores de acesso público, protegido e privado, mas não como outras linguagens como C# , Java, C++... etc, é simplesmente um encapsulamento "simulado".

Para acessar esses dados encapsulados, devem ser criados métodos públicos para atuar como interfaces. Em outros idiomas, nós os chamaríamos **getters** e **setters** e isto é o que dá origem às propriedades, que nada mais são do que atributos protegidos com interfaces de acesso:

```
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()
```

```
Eu sou um atributo inatingível do exterior.
Eu sou um método inalcançável do exterior.
```

Mas esta simulação de **getter** e **setters** é exatamente isso, uma simulação.

Vejamos outro exemplo de encapsulamento em 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.")
```

Neste exemplo, criamos um carro de classe com um conjunto de atributos particulares: **largo**, **ancho**, **ruedas**, **peso**, **color**, **is_enMarcha**. Em outra linguagem de programação com encapsulamento real, não seríamos capazes de alterar os valores desses atributos.

Em Python poderíamos, mas como os declaramos como privados, entenderemos que o programador não quer que o valor desses métodos seja alterado manualmente, então ele nos forneceu um método, **arrancar()**, o que muda o valor do atributo **is_enMarcha**. Supõe-se que, se quisermos alterar o valor desse atributo, devemos idealmente utilizar o método implementado para esse fim e não fazê-lo manualmente.

Polimorfismo

Se falamos de herança, temos que falar de polimorfismo. Este termo refere-se à capacidade de todos os objetos pertencentes à mesma família de classes, ou seja, herdados da mesma classe, serem chamados com os métodos da classe pai que eles sobrecarregaram, mas eles se comportarão com as chamadas aos seus próprios métodos sobrecarregados.

O polimorfismo é uma propriedade de herança pela qual objetos de diferentes subclasses podem responder à mesma ação.

O polimorfismo está implícito em Python, já que todas as classes são subclasses de uma superclasse comum chamada **Object**.

Por exemplo, a seguinte função aplica um desconto ao preço de um produto:

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

Graças ao polimorfismo não temos que verificar se um objeto tem ou não o atributo **pvp**, tentamos simplesmente acessar e, se existir, premiar:

```
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
```

Exemplo 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
```

Neste exemplo, estamos, em primeiro lugar, definindo a classe **Cuadrado**, que herda de **Poligono**. Criamos uma tupla com dois **Triangulo** e um **Cuadrado** e looping através dele **for**. Como podemos ver, chamamos o método **show** para cada elemento do tuple. Logicamente, cada chamada produz o resultado específico para o tipo de polígono ao qual ela pertence. Note que isto em **Python** é trivial, pois sempre lidamos diretamente com referências a objetos, por isso a chamada para **show** é sempre aquele que corresponde ao 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)
```