



Taller de Programación Web

Herencia, Polimorfismo y Encapsulamiento



Subsecretaría de
Empleo
Chaco Gobierno de todos



Ministerio de
Producción, Industria y Empleo
Chaco Gobierno de todos

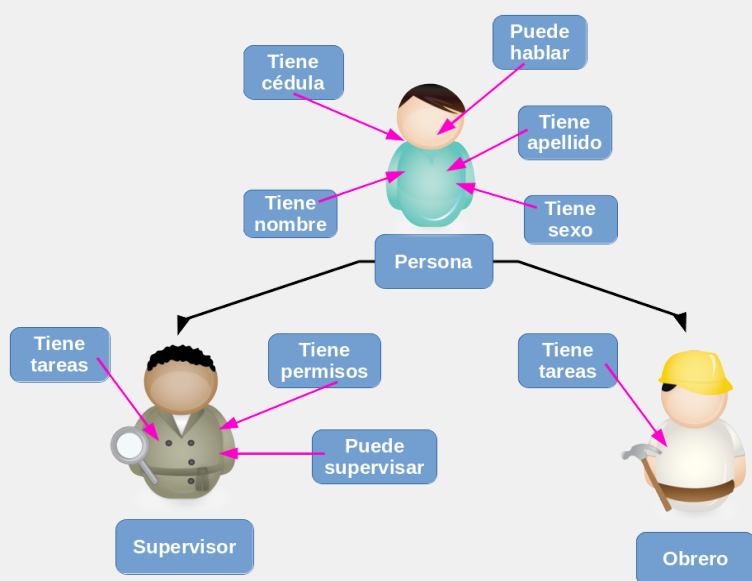


CHACO
Gobierno de todos



Herencia

Hay tres conceptos que son básicos para cualquier lenguaje de programación orientado a objetos: el **encapsulamiento**, la **herencia** y el **polimorfismo**.



En un lenguaje orientado a objetos cuando hacemos que una clase (subclase) **hereda** de otra clase (superclase) estamos haciendo que la subclase contenga todos los atributos y métodos que tenía la superclase.

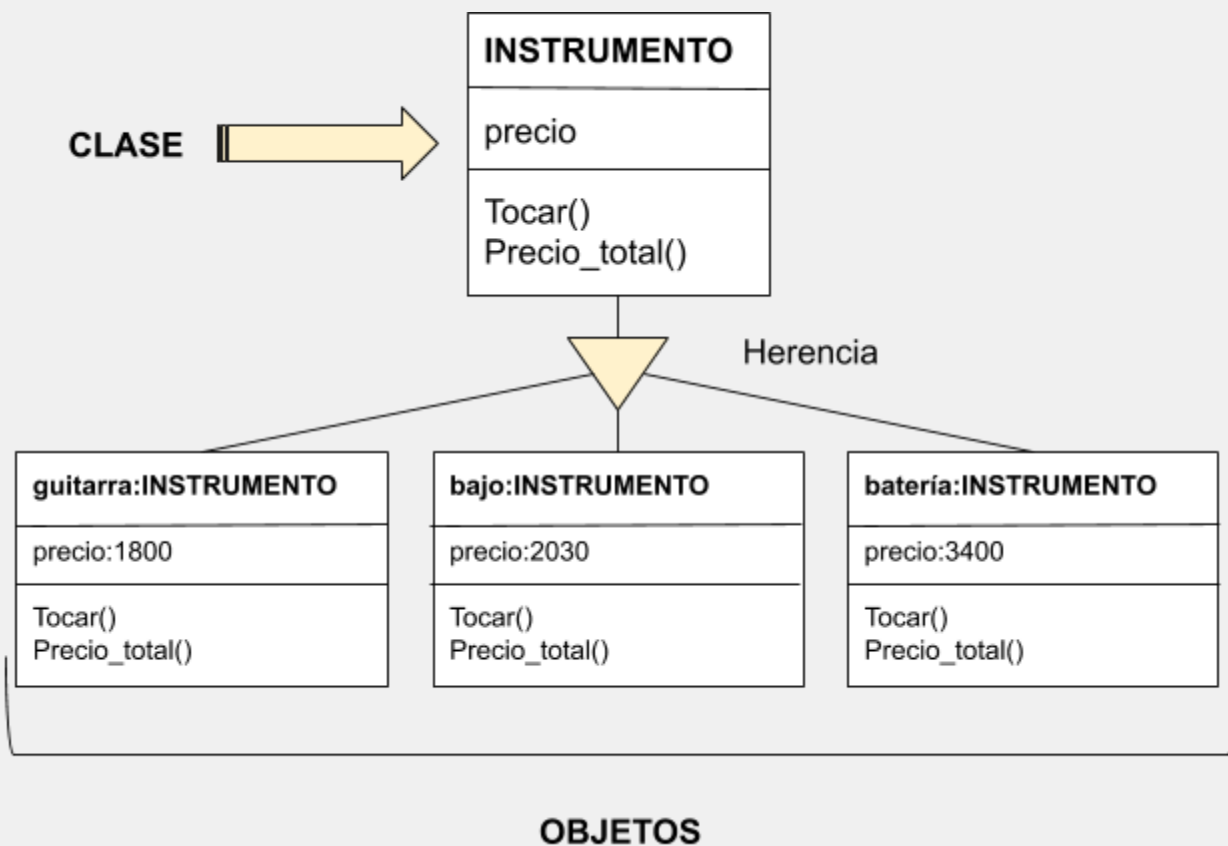
Al acto de heredar de una clase también se le llama a menudo “extender una clase”.



Herencia simple

Supongamos que queremos modelar los instrumentos musicales de una banda, tendremos entonces una **clase Guitarra**, una **clase Batería**, una **clase Bajo**, etc. Cada una de estas clases tendrá una serie de atributos y métodos, pero ocurre que, por el hecho de ser instrumentos musicales, estas clases compartirán muchos de sus atributos y métodos; un ejemplo sería el método **tocar()**.

Es más sencillo crear un tipo de objeto Instrumento con las atributos y métodos comunes e indicar al programa que Guitarra, Batería y Bajo son tipos de instrumentos, haciendo que hereden de Instrumento.





Para indicar que una clase hereda de otra se coloca el nombre de la clase de la que se hereda entre paréntesis después del nombre de la clase:

Código

Definir la clase Instrumento con sus métodos y atributos y crear clases que hereden de ella.

```
class Instrumento:
    def __init__(self, precio):
        self.precio = precio

    def tocar(self):
        print("Estamos tocando música")

    def precio(self):
        print("El precio de este instrumento es: $", self.precio)

class Bateria(Instrumento):
    pass

class Guitarra(Instrumento):
    pass
```

Como Bateria y Guitarra heredan de Instrumento, ambos tienen un método **tocar()** y un método **precio()**, y se inicializan pasando un único parámetro, precio. Pero, ¿qué ocurriría si quisiéramos especificar un nuevo parámetro **tipo_cuerda** a la hora de crear un objeto Guitarra? Bastaría con escribir un nuevo método **__init__** para la clase Guitarra que se ejecutaría en lugar del **__init__** de Instrumento. Esto es lo que se conoce como sobrescribir métodos.



Ahora bien, puede ocurrir en algunos casos que necesitemos sobrescribir un método de la clase padre, pero que en ese método queramos ejecutar el método de la clase padre porque nuestro nuevo método no necesite más que agregar un par de instrucciones extra. En ese caso usamos la sintaxis ***SuperClase.metodo(self, args)*** para llamar al método de igual nombre de la clase padre. Por ejemplo, para llamar al método `__init__` de `Instrumento` desde `Guitarra` usaríamos ***Instrumento.__init__(self, precio)***.

Código

Definir la clase `Instrumento` con sus métodos y atributos y crear clases que hereden de ella.

```
class Instrumento:
    def __init__(self, precio):
        self.precio = precio

    def tocar(self):
        print("Estamos tocando música")

    def correr(self):
        print("El precio de este instrumento es: $", self.precio)

class Bateria(Instrumento):
    pass

class Guitarra(Instrumento):
    def __init__(self, tipo_cuerda, precio):
        Instrumento.__init__(self, precio)
        self.tipo_cuerda = tipo_cuerda
```



Observe que en este caso si es necesario especificar el parámetro **self**.

Herencia múltiple

En Python, a diferencia de otros lenguajes como Java o C#, se permite la herencia múltiple, es decir, una clase puede heredar de varias clases a la vez. Por ejemplo, podríamos tener una clase Cocodrilo que heredara de la clase Terrestre, con métodos como caminar() y atributos como velocidad_caminar y de la clase Acuático, con métodos como nadar() y atributos como velocidad_nadar. Para indicar que una clase hereda de múltiples clases basta con enumerar las clases de las que se hereda separándolas por comas:

Código

Definir diferentes clases con sus métodos y atributos, luego implementar herencia múltiple en una de ellas.

```
class Terrestre:
    def __init__(self):
        pass

    def desplazar(self):
        print("El animal anda")

class Acuatico:
    def __init__(self):
        pass

    def desplazar(self):
        print("El animal nada")

class Cocodrilo(Terrestre, Acuatico):
    pass
```



En el caso de que alguna de las clases padre tuvieran métodos con el mismo nombre y número de parámetros las clases sobrescribirían la implementación de los métodos de las clases más a su derecha en la definición.

En el siguiente ejemplo, como Terrestre se encuentra más a la izquierda, sería la definición de **desplazar()** de esta clase la que prevalecerá, y por lo tanto si llamamos al método **desplazar** de un objeto de tipo **Cocodrilo** lo que se imprimiría sería “El animal anda”.

Código

Definir diferentes clases con sus métodos y atributos, luego implementar herencia múltiple en una de ellas.

```
class Terrestre:
    def __init__(self):
        pass

    def desplazar(self):
        print("El animal anda")

class Acuatico:
    def __init__(self):
        pass
    def desplazar(self):
        print("El animal nada")

class Cocodrilo(Terrestre, Acuatico):
    pass

c = Cocodrilo()
```



```
c.desplazar()
```

Resultado

El animal anda

Lo mismo ocurre para el caso de constructor, en caso de que se requiera el uso de el constructor de alguna clase específica puede usarse la sintaxis anteriormente vista, *SuperClase.metodo(self, args)*.

Código

Definir diferentes clases con sus métodos y atributos, luego implementar herencia múltiple en una de ellas.

```
class Terrestre:
    def __init__(self):
        print("Este es el constructor de la clase Terrestre")

    def desplazar(self):
        print("El animal anda")

class Acuatico:
    def __init__(self):
        print("Este es el constructor de la clase Acuático")

    def desplazar(self):
        print("El animal nada")

class Cocodrilo(Terrestre, Acuatico):
```




```
pass
```

```
c = Cocodrilo()
```

Resultado

Este es el constructor de la clase Terrestre

Código

Definir diferentes clases con sus métodos y atributos, luego implementar herencia múltiple en una de ellas.

```
class Terrestre:
    def __init__(self):
        print("Este es el constructor de la clase Terrestre")

    def desplazar(self):
        print("El animal anda")

class Acuatico:
    def __init__(self):
        print("Este es el constructor de la clase Acuático")

    def desplazar(self):
        print("El animal nada")

class Cocodrilo(Terrestre, Acuatico):
```



```
def __init__(self):  
    Acuatico.__init__(self)  
  
c = Cocodrilo()
```

Resultado

Este es el constructor de la clase Acuático

En el segundo ejemplo se define cual de los constructores de las superclases se va a utilizar, permitiéndonos especificar qué métodos usará de cada una de la clases padre sin importar quien esté más a su derecha en la definición de la clase

Herencia multinivel en Python

Por otro lado, también podemos heredar de una clase derivada. Esto se llama herencia multinivel y puede ser de cualquier profundidad en Python.

En la herencia multinivel, las características de la clase base y la clase derivada se heredan en la nueva clase derivada.

Código

Definir diferentes clases con sus métodos y atributos, luego implementar herencia multinivel entre ellas.

```
class Figura:  
    def __init__(self, area):  
        self.area = area  
  
    def retornar_area(self):  
        print("El área de la figura es:", self.area)
```



```
class Poligono(Figura):  
    def __init__(self, lados, area):  
        Figura.__init__(self, area)  
        self.lados = lados  
  
    def retornar_lados(self):  
        print("Los lados del poligono son:", self.lados)  
  
class Cuadrilatero(Poligono):  
  
    def __init__(self, area):  
        Poligono.__init__(self, 4, area)
```

Resultado

Aquí por ejemplo, **Poligono** se deriva de **Figura**, y **Cuadrilatero** se deriva de **Poligono** por lo que **Cuadrilatero** tendrá los métodos y atributos tanto de la clase **Poligono** como de la clase **Figura**. Si en la clase **Poligono** alguno de los métodos de **Figura** es redefinido **Cuadrilatero** heredará el método redefinido.

Nota En el diseño de jerarquías de herencia no siempre es del todo fácil decidir cuándo una clase debe extender a otra. La regla práctica para decidir si una clase (S) puede ser definida como heredera de otra (T) es que debe cumplirse que "S es un T". Por ejemplo, Perro es un Animal, pero Vehículo no es un Motor.



Método de resolución de orden en Python

Cada clase en Python se deriva de la clase **object**. Es el tipo más básico en Python.

Así que técnicamente, todas las demás clases, ya sean integradas o definidas por el usuario, son clases derivadas y todos los objetos son instancias de la clase **object**.

Código

Evaluar distintos tipos de datos para verificar que son instancias de la clase `object`.

```
print(issubclass(list, object))
```

```
print(isinstance(5.5, object))
```

```
print(isinstance("Hola", object))
```

Resultado

True

True

True

En el escenario de herencia múltiple, cualquier atributo especificado se busca primero en la clase actual. Si no se encuentra, la búsqueda continúa en clases primarias en profundidad, de izquierda a derecha, sin buscar la misma clase dos veces.

Por lo tanto, en el ejemplo anterior de la clase **Cuadrilatero** el orden de búsqueda es [**Cuadrilatero**, **Poligono**, **Figura**, **object**]. Este orden también se denomina linealización de la clase **Cuadrilatero** y el conjunto de reglas que se utiliza para encontrar este orden se denomina Orden de resolución de métodos, MRO por sus siglas en inglés (Method Resolution Order).

El MRO de una clase puede verse como el atributo `__mro__` o el método `mro()`. El primero devuelve una tupla mientras que el segundo devuelve una lista.



Código

Definir diferentes clases con sus métodos y atributos, luego implementar herencia multinivel entre ellas. Luego imprimir el Orden de Resolución de métodos.

```
class Figura:
    def __init__(self, area):
        self.area = area

    def retornar_area(self):
        print("El área de la figura es:", self.area)

class Poligono(Figura):
    def __init__(self, lados, area):
        Figura.__init__(self, area)
        self.lados = lados

    def retornar_lados(self):
        print("Los lados del poligono son:", self.lados)

class Cuadrilatero(Poligono):

    def __init__(self, area):
        Poligono.__init__(self, 4, area)

print(Cuadrilatero.__mro__)
print(Cuadrilatero.mro())
```

Resultado

```
(<class '__main__.Cuadrilatero'>, <class '__main__.Poligono'>, <class
'__main__.Figura'>, <class 'object'>)
[<class '__main__.Cuadrilatero'>, <class '__main__.Poligono'>, <class
'__main__.Figura'>, <class 'object'>]
```



Aquí hay un ejemplo de herencia múltiple un poco más complejo y su visualización junto con el MRO.

Código

Definir diferentes clases con sus métodos y atributos, luego implementar herencia multinivel entre ellas. Luego imprimir el Orden de Resolución de métodos.

```
class X:
    pass

class Y:
    pass

class Z:
    pass

class A(X, Y):
    pass

class B(Y, Z):
    pass

class M(B, A, Z):
    pass

print(M.mro())
```

Resultado

```
[<class '__main__.M'>, <class '__main__.B'>, <class '__main__.A'>, <class '__main__.X'>, <class '__main__.Y'>, <class '__main__.Z'>, <class 'object'>]
```