

PROGRAMACION 2

Herencia

Polimorfismo

Esc. Sup. De Comercio N° 49 Cap. Gral. J.J. de Urquiza

Técnico Superior en Desarrollo de Software



HERENCIA

¿Para qué sirve la herencia?

- Para la reutilización de código en caso de crear objetos similares

Sobreescritura de métodos

Si tomamos el ejemplo de los vehículos, la clase moto hereda de la clase vehículos todos sus atributos y métodos, pero también puede tener otros métodos o atributos que la identifican. Es decir, qué características especiales tiene una moto que la distinguen de los otros vehículos. Entonces al declararla los agregamos o redefinimos. Con redefinirlos queremos decir que quizás alguno de los métodos ya definidos en la clase padre pueden ser necesarios pero, necesitamos agregarle o cambiarle algo que sea específico de las motos.

Entonces qué hacemos?

Repetimos el método (con el mismo nombre) dentro de la subclase y hacemos las modificaciones que se necesitan.

Ahora surgen la duda cuando llamamos al método (ambos tienen el mismo nombre) a cuál se hará referencia al del padre o al del hijo?

Se referencia al del hijo, ya que cuando definimos un método con el mismo nombre en el hijo se sobreescribe el método heredado.

- Herencia:
 - Sobre escritura de métodos
 - Herencia simple y múltiple

Siguiendo con el ejemplo de vehículos que pasaría si tenemos que definir vehículos eléctricos. Definimos una clase eléctricos que tiene como característica la autonomía de carga que posee. Ahora necesitamos definir una subclase automovil eléctrico que tendrá características del vehículo y de eléctrico. Python nos permite realizar los que se llama herencia múltiple, entonces dentro de los paréntesis ponemos separados con comas todas las clases de las cuales va a heredar.

Hay otros lenguajes que no admiten la herencia multiple pero Python sí

Seguimos con el ejemplo de la clase anterior de la clase vehículos.

Vamos a construir la clase moto que heredaba de vehículos pero en principio la teníamos vacía.

Vimos en la clase anterior que una clase además de los métodos que hereda puede tener sus propios métodos.

Vamos a crear un comportamiento particular para la clase moto

```
class Moto(Vehículos):
    hcaballito=""
    def caballito(self):
        self.hcaballito="Voy haciendo el caballito"
```

Si en la clase moto creamos un método estado

```

1 def estado(self):
2     print ("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn Marcha: ",
3         self.enmarcha, "\nAcelarando: ", self.acelera, "\nFrenando: ", self.frena)
4
5
6 class Moto(Vehiculos):
7     hcaballito=""
8     def caballito(self):
9         self.hcaballito="Voy haciendo el caballito"
10
11     def estado(self):
12         print ("Marca: ", self.marca, "\nModelo: ", self.modelo, "\nEn Marcha: ",
13             self.enmarcha, "\nAcelarando: ", self.acelera, "\nFrenando: ", self.frena, "\n", self.hcaballito)
14

```

```

miMoto.estado()

```

ya que moto hereda los métodos de vehículo, cuando llamamos aquí al método estado a cuál se estaría refiriendo al de vehículo o al de moto? Hará referencia al de la clase moto ya que siempre se sobrescribe el método que heredas de la clase padre.

```

class Furgoneta(Vehiculos):
    def carga(self, cargar):
        self.cargado=cargar
        if(self.cargado):
            return "La furgoneta está cargada"
        else:
            return "La furgoneta no está cargada"

```

```

miFurgoneta=Furgoneta("Renault", "Kangod")

```

Por ejemplo si queremos crear una clase de autos eléctricos va a tener características propias de los autos pero algunas particularidades de los autos eléctrico

```

class VElectricos():
    def __init__(self):
        self.autonomia=100
    def cargarEnergia(self):
        self.cargando=True

```

```

class AutoElectrico(eléctrico, vehículo):

```

Qué va a suceder cuando definimos una instancia de esta clase?

```

AutoElect1=(AutoElectrico)

```

Tanto la clase vehículo como la clase eléctrico tenían definidos constructores cuál heredará, la de eléctrico o la de vehículo. Por defecto Python da preponderancia a la clase que primero de enumeró dentro de los paréntesis en nuestro caso eléctrico. A esto se llama herencia múltiple hay algunos lenguajes que no soportan este tipo de herencia.

```
print(miFurgoneta.carga(True))

class BicicletaElectrica(VElectricos,Vehiculos):
    pass

miBici=BicicletaElectrica("Orbea", "HC1030")
```

Aquí tenemos una herencia múltiple

super()

Si queremos alterar la sobreescritura de métodos y llamar al método padre podemos utilizar super()

```
1 class Persona():
2
3     def __init__(self, nombre, edad, Lugar_residencia):
4
5         self.nombre=nombre
6
7         self.edad=edad
8
9         self.lugar_residencia=Lugar_residencia
10
11     def descripcion(self):
12
13         print("Nombre: ", self.nombre, " Edad: ", self.edad, " Residencia: ", self.lugar_residencia)
14
15 class Empleado(Persona):
16
17     def __init__(self, salario, antigüedad):
18
19         super().__init__("Antonio", 55, "España")
20
21         self.salario=salario
22
23         self.antigüedad=antigüedad
24
```

Principio de sustitución

Un empleado es siempre una persona, pero una persona no siempre es un empleado. Es decir, le herencia. Este principio de sustitución se utiliza para corroborar que las clases y herencias fueron bien creadas

isinstance()

Esta función nos dice si un objeto es instancia de una clase determinada.

`Isinstance(nombre_de_la_instancia, clase)`

Nos devuelve true o false. Esto es muy útil cuando tenemos muchas clases, subclases y herencias para comprobar instancia de qué clase es.

POLIMORFISMO

Polimorfismo (del griego significa Muchas formas) cuando se da el poliformismo en programación quiere decir que un objeto puede cambiar de forma dependiendo del contexto en el que se utiliza

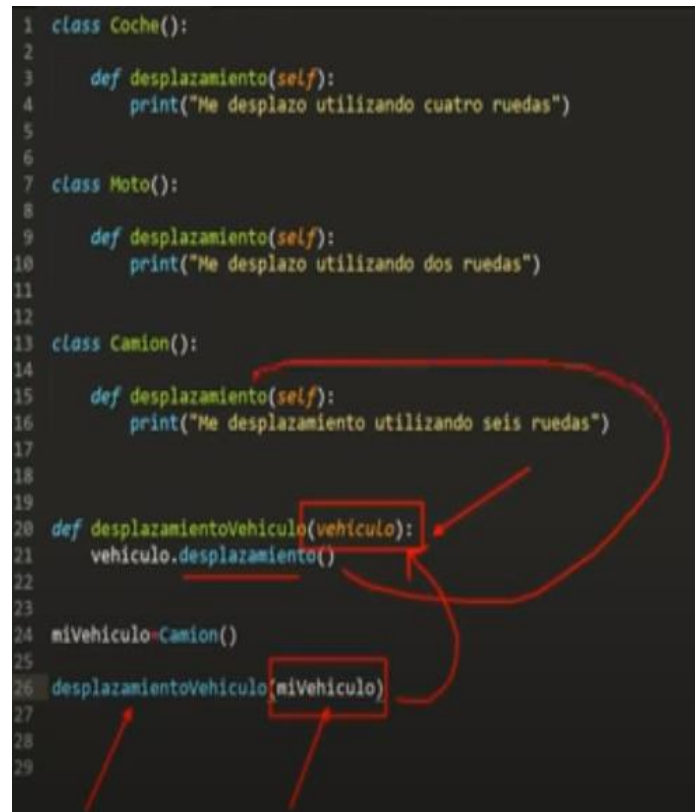
El polimorfismo es la técnica que nos posibilita que al invocar un determinado método de un objeto, podrán obtenerse distintos resultados según la clase del objeto. Esto se debe a que distintos objetos pueden tener un método con un mismo nombre, pero que realice distintas operaciones.

```
class Coche():
    def desplazamiento(self):
        print("utiliza cuatro ruedas")
class Moto():
    def desplazamiento(self):
        print("utiliza dos ruedas")
class Camion():
    def desplazamiento(self):
        print("utiliza seis ruedas")

mivehiculo=Moto()
mivehiculo.desplazamiento()

mivehiculo2=Coche()
mivehiculo2.desplazamiento()
```

Utilizo el mismo método comportamiento, pero pertenecen a clases distintas. Detecta a qué clase pertenece el método y llama al que corresponde



```
1 class Coche():
2
3     def desplazamiento(self):
4         print("Me desplazo utilizando cuatro ruedas")
5
6 class Moto():
7
8     def desplazamiento(self):
9         print("Me desplazo utilizando dos ruedas")
10
11
12 class Camion():
13
14     def desplazamiento(self):
15         print("Me desplazo utilizando seis ruedas")
16
17
18
19
20 def desplazamientoVehiculo(vehiculo):
21     vehiculo.desplazamiento()
22
23
24 miVehiculo=Camion()
25
26 desplazamientoVehiculo(miVehiculo)
27
28
29
```

Pero en lugar de estar definiendo todas estas instancias podemos utilizar polimorfismo. Y definir un método que reciba un objeto por parámetro. La función va a utilizar ese objeto que recibe por parámetro para llamar al método desplazamiento que corresponde.

Pero a qué método se está llamando? A qué se refiere vehículo?

El *polimorfismo* nos permite que ese objeto pueda adaptarse. Cambiar de forma de acuerdo al contexto de que se trate y va a saber a qué método nos estamos refiriendo

Además esto es posible a que python es dinámico, es decir en tiempo de ejecución es cuando se determina el tipo de un objeto. Veamos otro ejemplo:

```
class gato():
    def hablar(self):
        print("MIAU")
class perro():
    def hablar(self):
        print("GUAU")
def escucharMascota(animal):
    animal.hablar()
if __name__ == '__main__':
    g = gato()
    p = perro()
    escucharMascota(g)
    escucharMascota(p)
```

En otros lenguajes que son más tipeados como por ejemplo Java es más complejo usar polimorfismo