PROGRAMACIÓN AVANZADA CON PYTHON

(CEFIRE CTEM)

Programación orientada a objetos en Python (parte 2)

Autora: María Paz Segura Valero (segura_marval@gva.es)

CONTENIDO

1. Introducción

En la unidad anterior hicimos una introducción a los elementos básicos de la **Programación Orientada a Objetos** (clases, objetos, atributos y métodos) y también explicamos cómo conseguir el encapsulamiento de datos en Python.

Durante esta unidad vamos a dar un paso más e introduciremos los mecanismos de **polimorfismo**, **herencia** y **delegación** que nos permitirán reutilizar código de otras clases ya existentes previamente.

2. Polimorfismo

Fíjate en el siguiente ejemplo:

```
>>> p1 = Personaje("Esther", 14, "Purita Campos")
>>> print("Hola")
 Hola
>>> print(10)
 10
>>> p1 = Personaje("Esther", 14, "Purita Campos")
>>> print(p1)
 Esther tiene 14 años y es de Purita Campos
```

¿No te llama la atención que una misma función (`print()`) ofrezca resultados diferentes según el tipo de dato que se le pasa como parámetro? Igual imprime una cadena de texto, un número o, incluso, un objeto de la clase `Personaje`. Esto es posible gracias a que Python es un lenguaje de tipado dinámico (no conocemos el tipo de una variable hasta que no se ejecuta el programa) y al polimorfismo.

El **polimorfismo** es una técnica que permite invocar a una función o método y que el resultado sea distinto según el tipo de datos que se le pase.

Así que, si creamos varias clases con un método que tiene el mismo nombre, podremos obtener resultados diferentes según el tipo de objeto.

Ejemplo: Tenemos tres clases de animales (`Oruga`, `Gorrion`, `Ballena`) con un método llamado `mover()` y, evidentemente, cada tipo de animal se mueve de una manera diferente.

Además, disponemos de una función llamada `que_se_mueva()` a la que se le pasa un `animal` como parámetro de entrada y ejecuta el método `mover()` de dicho animal.

```python
class Oruga:
    def mover(self):
        print("Me deslizo por la tierra")

class Gorrion:
    def mover(self):
        print("Vuelo por el aire")

class Ballena:
    def mover(self):
        print("Nado por el mar")

def que_se_mueva(animal):
    animal.mover()
```

Creamos tres animales diferentes y llamamos a la función `que_se_mueva()` que ejecuta el método `mover()` del objeto correspondiente

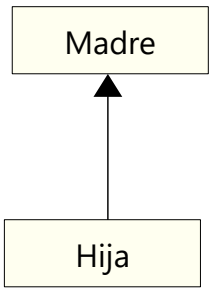
```
>>> o = Oruga(); que_se_mueva(o)
 Me deslizo por la tierra
>>> g = Gorrion(); que_se_mueva(g)
 Vuelo por el aire
>>> b = Ballena(); que_se_mueva(b)
 Nado por el mar
```

3. Herencia

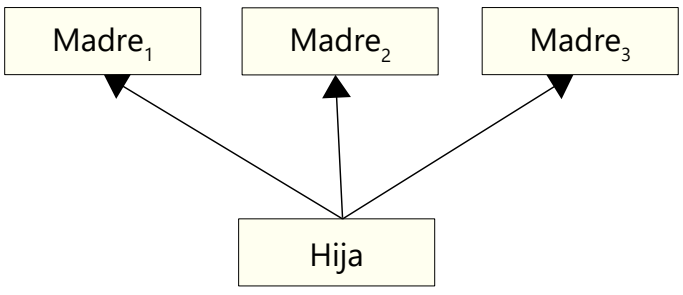
La **herencia** es el mecanismo por el que podemos crear clases nuevas a partir de otras clases ya existentes. A la clase ya existente se le conoce como **clase base** o **clase madre** y a la clase nueva como **clase derivada** o **clase hija**.

Existen dos tipos de herencia:

- *Herencia simple*: cuando una clase derivada hereda de una sola clase base.
- *Herencia múltiple*: cuando una clase derivada hereda de varias clases base.

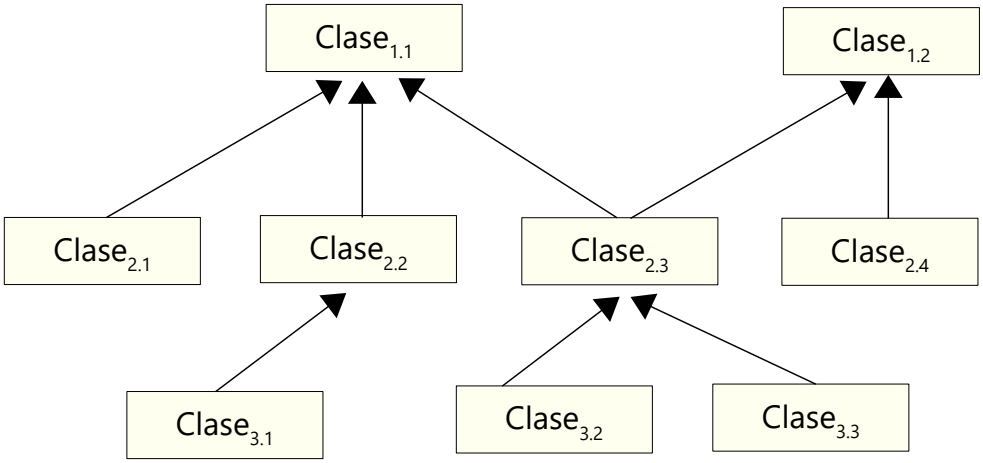


Herencia simple



Herencia múltiple

Una clase dada puede actuar como clase base de otras y como clase hija a la vez. De esta forma podemos crear niveles de jerarquía o **especialización** de clases.

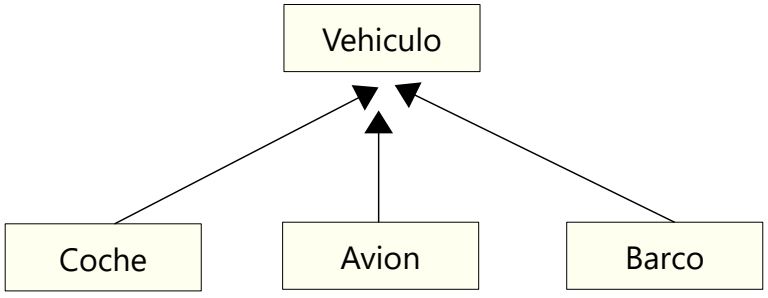


Jerarquía o especialización de clases

3.1. Herencia simple

En la **herencia simple** las clases hijas heredan los atributos y métodos de la clase madre y pueden redefinirlos e, incluso, añadir nuevos.

Ejemplo: Las clases `Coche`, `Avion` y `Barco` son clases derivadas de la clase base `Vehiculo`.



Para crear una clase derivada simplemente hará falta indicar la clase base de la que heredamos, con la siguiente sintaxis:

```
class clase_derivada(clase_base):
```

Ejemplo:

```
#clase base
class Vehiculo:
    def __init__(self, matricula):
        self.matricula = matricula

#clases derivadas
class Coche(Vehiculo):
    pass

class Avion(Vehiculo):
    pass

class Barco(Vehiculo):
    pass
```

Si creamos un objeto de tipo `Coche` también contiene el atributo `matricula`.

```
>>> v = Vehiculo("vv"); v.matricula
'vv'
>>> c = Coche("cc"); c.matricula
'cc'
```

De hecho, si intentamos crear un objeto `Coche` sin pasarle un valor para la `matricula`, da un error.

```
>>> c = Coche()
 Traceback (most recent call last):
   File "<pyshell>", line 1, in <module>
 TypeError: __init__() missing 1 required positional argument: 'matricula'
```

Normalmente las clases hijas suelen tener atributos adicionales a los de la clase madre, así que deberemos redefinir el **constructor** de la clase hija y, dentro de él, llamar al constructor de la clase madre para que haga su trabajo.

Para ello, necesitamos conocer una función importante: **super()**. Esta función devuelve una referencia a la clase madre directa de una clase dada. Así podemos acceder a sus métodos sin problemas.

Sintaxis:

```
super().nombre_metodo(parámetros)
```

Donde ***nombre_metodo*** es el nombre del método de la clase base que queremos ejecutar y ***parámetros*** es la lista de valores que espera dicho método.

Ejemplo: Añadimos nuevos atributos a las clases hijas y deberemos tenerlos en cuenta a la hora de crear nuevos objetos.

```
#clase base
class Vehiculo:
    def __init__(self, matricula):
        self.matricula = matricula

#clases derivadas
class Coche(Vehiculo):
    def __init__(self, matricula, cilindrada, energia):
        super().__init__(matricula)
        self.cilindrada = cilindrada
        self.energia = energia

class Avion(Vehiculo):
    def __init__(self, matricula, motores):
        super().__init__(matricula)
        self.motores = motores

class Barco(Vehiculo):
    def __init__(self, matricula, eslora, bodega):
        super().__init__(matricula)
        self.eslora = eslora
        self.bodega = bodega
```

El constructor de la clase base se encarga de inicializar los atributos heredados. Así reutilizamos código.

Creamos objetos de cada clase. El número de atributos que hay que informar es diferente.

```
>>> v = Vehiculo("vv")
>>> c = Coche("cc", 100, "eléctrica")
>>> a = Avion("aa", 4)
>>> b = Barco("bb", 1500, 'sí')
```

3.2. Herencia múltiple

La **herencia múltiple** es la técnica que permite crear una clase derivada a partir de varias clases base. En este tipo de herencia la clase derivada hereda todos los atributos y métodos de sus clases base.

Para crear una clase derivada simplemente hará falta indicar la lista de clases base de las que heredamos, con la siguiente sintaxis:

class *clase_derivada*(*clase_base*$_1$, *clase_base*$_2$, ..., *clase_base*$_n$):

Hay que tener cuidado de que no haya conflicto de nombres de atributos o métodos entre las clases base. Si lo hubiera, se da prioridad a la clase declarada más a la izquierda.

Ejemplo: Definimos la clase **Sirena** que deriva de las clases y **Mitologico**.

```
#clases base
class Pez:
    def __init__(self, agua):
        self.agua = agua #Tipo de agua que habita

class Mitologico:
    def __init__(self, origen):
        self.origen = origen #Origen del personaje mitológico

#clase derivada
class Sirena(Pez, Mitologico):
    def __init__(self, nombre, agua, origen):
        super().__init__(agua)
        Mitologico.__init__(self, origen)
        self.nombre = nombre

    def __str__(self):
        return self.nombre + ' vive en agua ' + self.agua + ' y es de la mitología ' + self.origen
```

```
>>> s = Sirena("Ariel", "salada", "clásica")
>>> print(s)
```

Fíjate que en el constructor de la clase derivada la función **super()** hace referencia a la clase base **Pez**, ya que es la que está más hacia la izquierda en el orden de definición de clases base de **Sirena**. Así que, tenemos que añadir la llamada al constructor de la clase base **Mitologico** directamente (y especificando el primer parámetro como **self**, cosa que no ocurre al llamar a **super()**).

También podríamos haber realizado las llamadas a los constructores de las clases base así:

```
Pez.__init__(self, agua)
Mitologico.__init__(self, origen)
```

3.3. Algunos métodos útiles

En este apartado se presentan dos métodos que pueden ser de utilidad cuando trabajamos con herencia:

Método	Descripción

issubclass()	Este método permite comprobar si una clase es hija de otra clase.
	Sintaxis: **issubclass(*clase_hija, clase_madre*)**
	
	Ejemplos:
	`>>> %Run vehiculos_v1.py`
	`>>> issubclass(Coche, Vehiculo)`
	`True`
	`>>> issubclass(int, Vehiculo)`
	`False`

isinstance()	Este método permite comprobar si un objeto es instancia de una clase.
	Sintaxis: **isinstance(*nombre_objeto, clase*)**
	
	Ejemplos:
	`>>> v = Vehiculo("vv")`
	`>>> isinstance(v, Vehiculo)`
	`True`
	`>>> isinstance(v, Coche)`
	`False`

4. Delegación

En algunas ocasiones necesitamos crear algún atributo en nuestra clase que sea un objeto de otra clase pero no crear una clase derivada. Para ello se utiliza el mecanismo de **Delegación** que permite incluir instancias de otras clases y delegar funciones en ellas.

Ejemplo: Dada la clase *Persona*, con la que trabajamos en la unidad anterior, creamos la clase *Asociacion* que permite gestionar las asociaciones existentes en una zona. La clase *Asociacion* tiene un atributo llamado *representante* que es un objeto de la clase *Persona*.

```
#Importamos la clase "Persona"
from persona import Persona

#Creamos la clase "Asociacion"
class Asociacion:
    #métodos redefinidos
    def __init__(self, cif, nombre, fecha, representante):
        self.CIF = cif
        self.nombre = nombre
        self.fecha_creacion = fecha
        self.representante = representante

    def __str__(self):
        return 'La "' + self.nombre + '" se creó en ' + self.fecha_creacion + \
               ' y su representante es ' + self.representante.__str__() + '.'

    def compara_representantes(self, asociacion):
        if (self.representante == asociacion.representante):
            respuesta = "SI"
        else:
            respuesta = "NO"
        print("La asociación <" + self.CIF + "> y la asociación <" + asociacion.CIF + "> " + \
              respuesta + " comparten el mismo representante.")
```

Si te fijas, cuando utilizamos el atributo *representante* en los métodos de la clase *Asociacion* estamos ejecutando métodos (delegando) de la clase *Persona* para que sea dicha clase la que se encargue de gestionar sus objetos.

Vamos a probar la clase. Para ello creamos tres asociaciones, con sus respectivos representantes, y ejecutamos el método `compara_representantes()` para conocer si hay algún representante común entre ellos.

```
print("ASOCIACIONES EXISTENTES:")
r1 = Persona("11111111H", "Armando")
a1 = Asociacion("1", "Asociación de vecinos", "01-01-2014", r1)
print(a1)

r2 = Persona("22222222J", "Pepita")
a2 = Asociacion("2", "Programando juntos", "01-01-2016", r2)
print(a2)

a3 = Asociacion("3", "Club de cocina", "01-01-2020", r2)
print(a3)

print("\nCOMPARANDO REPRESENTANTES")
a1.compara_representantes(a2)
a1.compara_representantes(a3)
a2.compara_representantes(a3)
```

```
ASOCIACIONES EXISTENTES:
La "Asociación de vecinos" se creó en 01-01-2014 y su representante es (11111111H, Armando).
La "Programando juntos" se creó en 01-01-2016 y su representante es (22222222J, Pepita).
La "Club de cocina" se creó en 01-01-2020 y su representante es (22222222J, Pepita).

COMPARANDO REPRESENTANTES
La asociación <1> y la asociación <2> NO comparten el mismo representante.
La asociación <1> y la asociación <3> NO comparten el mismo representante.
La asociación <2> y la asociación <3> SI comparten el mismo representante.
```

Podríamos añadir atributos más complejos, por ejemplo, una lista de personas afiliadas a la asociación. Entonces deberíamos añadir métodos que los gestionasen adecuadamente. Algunos de ellos podrían ser los siguientes: `alta_afiliado()`, `baja_afiliado()`, `imprimir_afiliados()`, etc.

Puedes ver el ejemplo completo en el fichero **delegacion_completo.py** del aula virtual.

5. Fuentes de información

- Programación orientada a objetos de *José Domingo Muñoz*: https://plataforma.josedomingo.org/pledin/cursos/python3/curso/u50/
- Herencia en Python: https://www.codigofuente.org/herencia-en-python/
- Programación orientada a objetos: https://entrenamiento-python-basico.readthedocs.io/es/latest/leccion9/poo.html