HERENCIA EN PYTHON

La **herencia** es un proceso mediante el cual se puede crear una clase **hija** que hereda de una clase **padre**, compartiendo sus métodos y atributos. Además de ello, una clase hija puede sobrescribir los métodos o atributos, o incluso definir unos nuevos.

Se puede crear una clase hija con tan solo pasar como parámetro la clase de la que queremos heredar. En el siguiente ejemplo vemos cómo se puede usar la herencia en Python, con la **clase Perro** que **hereda de Animal.**

Se refiere a la habilidad: puede usar todas las funciones de una clase existente y extender estas funciones sin reescribir la clase original.

- **Clase padre ("clase base"**, **"clase principal"** o **"superclase")**
  Clase de la que desciende o deriva una clase, las **clases hijas (descendientes)** "**subclase" "clase derivada"**, heredan (incorporan) automáticamente los atributos y métodos de la clase padre.

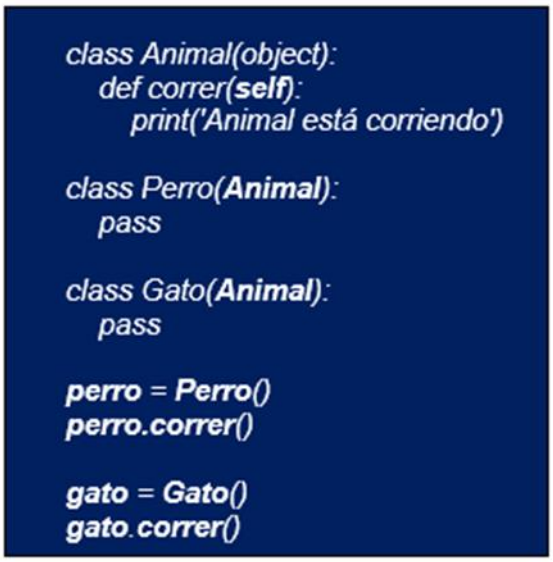
Subclase: Clase descendiente de otra. Hereda automáticamente los atributos y métodos de su superclase. Es una especialización de otra clase. Admiten la definición de nuevos atributos y métodos para aumentar la especialización de la clase

¿Y para que queremos la herencia? Dado que una clase hija hereda los atributos y métodos de la clase padre, nos puede ser muy útil cuando tengamos clases que se parecen entre sí, pero tienen ciertas particularidades. En este caso en vez de definir un montón de clases para cada animal, podemos tomar los elementos comunes y crear una clase Animal de la que hereden el resto, respetando por tanto la filosofía **DRY**. Realizar estas abstracciones y buscar el denominador común para definir una clase de la que hereden las demás.

DRY (Don't Repeat Yourself), consiste en no repetir código de manera innecesaria. Cuanto más código duplicado exista, más difícil será de modificar y más fácil será crear inconsistencias. Las clases y la herencia a no repetir código.

En algunos lenguajes POO, una subclase puede heredar varias clases base. Pero en circunstancias normales, una subclase solo puede tener una clase base.
Para lograr una herencia múltiple, se puede lograr a través de una herencia múltiple.
Hay dos formas principales de implementar el concepto de herencia: herencia de implementación y herencia de interfaz.

1. La **herencia de implementación** se refiere a la capacidad de usar las propiedades y métodos de la clase base sin codificación adicional.
2. La **herencia de interfaz** significa que solo se utilizan los nombres de las propiedades y los métodos, pero la subclase debe proporcionar la capacidad de implementarla



La clase padre `Animal` que tendrá todos los atributos y métodos genéricos que los animales pueden tener:

- Tenemos la **especie** ya que todos los animales pertenecen a una.
- Y la **edad**, ya que todo ser vivo nace, crece, se reproduce y muere.

Y los métodos o funcionalidades:

- Tendremos el método **hablar**, que cada animal implementará de una forma. Los perros ladran, las abejas zumban y los caballos relinchan.
- Un método **moverse**. Unos animales lo harán caminando, otros volando.
- Y por último un método **descríbeme** que será común.

Definimos la clase padre, con una serie de atributos comunes para todos los animales como hemos indicado.

Si queremos pasar parámetros a la instancia c, tenemos que usar el constructor, entonces, ¿cómo heredar el constructor y cómo definir sus propias propiedades en la subclase?

Método de construcción de la clase heredada:

1. La escritura de la clase clásica: el nombre de la clase padre.**init**(self, parámetro 1, parámetro 2, ...)

2. La escritura de clases de nuevo estilo: super (subclase, self).**init**(Parámetro 1, parámetro 2, ...)

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad

    # Método genérico, pero con
implementación particular
    def hablar(self):
        # Método vacío
        pass

    # Método genérico pero con
implementación particular
    def moverse(self):
        # Método vacío
        pass

    # Método genérico con la misma
implementación
    def describeme(self):
```

Tenemos ya por lo tanto una clase genérica Animal, que generaliza las características y funcionalidades que todo animal puede tener. Ahora creamos una clase Perro que hereda del Animal. Como primer ejemplo vamos a crear una clase vacía, para ver como los métodos y atributos son heredados por defecto.

hemos creado una clase nueva que tiene todo el contenido que la clase padre tiene, pero aquí viene lo que es de verdad interesante. Vamos a crear varios animales concretos y sobrescribir algunos de los métodos que habían sido definidos en la clase Animal, como el hablar o el moverse, ya que cada animal se comporta de una manera distinta.

Podemos incluso crear nuevos métodos que se añadirán a los ya heredados, como en el caso de la Abeja con picar().

```
# Perro hereda de Animal
class Perro(Animal):
    pass

mi_perro = Perro('mamífero', 10)
mi_perro.describeme()
# Soy un Animal del tipo Perro
```

```python
class Perro(Animal):
    def hablar(self):
        print("Guau!")
    def moverse(self):
        print("Caminando con 4 patas")

class Vaca(Animal):
    def hablar(self):
        print("Muuu!")
    def moverse(self):
        print("Caminando con 4 patas")

class Abeja(Animal):
    def hablar(self):
        print("Bzzzz!")
    def moverse(self):
        print("Volando")

    # Nuevo método
    def picar(self):
        print("Picar!")
```

Por lo tanto, ya podemos crear nuestros objetos de esos animales y hacer uso de sus métodos que podrían clasificarse en tres:

- Heredados directamente de la clase padre: describeme()
- Heredados de la clase padre pero modificados: hablar() y moverse()
- Creados en la clase hija por lo tanto no existentes en la clase padre: picar()

```python
mi_perro = Perro('mamífero', 10)
mi_vaca = Vaca('mamífero', 23)
mi_abeja = Abeja('insecto', 1)

mi_perro.hablar()
mi_vaca.hablar()
# Guau!
# Muuu!

mi_vaca.describeme()
mi_abeja.describeme()
# Soy un Animal del tipo Vaca
# Soy un Animal del tipo Abeja

mi_abeja.picar()
# Picar!
```

Uso de super()

En pocas palabras, la función `super()` nos permite acceder a los métodos de la clase padre desde una de sus hijas.

Tal vez queramos que nuestro Perro tenga un parámetro extra en el constructor, como podría ser el dueño. Para realizar esto tenemos dos alternativas:

- Podemos crear un nuevo __init__ y guardar todas las variables una a una.
- O podemos usar super() para llamar al __init__ de la clase padre que ya aceptaba la especie y edad, y sólo asignar la variable nueva manualmente.

```
class Perro(Animal):
    def __init__(self, especie, edad, dueño):
        # Alternativa 1
        # self.especie = especie
        # self.edad = edad
        # self.dueño = dueño

        # Alternativa 2
        super().__init__(especie, edad)
        self.dueño = dueño
mi_perro = Perro('mamífero', 7, 'Luis')
mi_perro.especie
mi_perro.edad
mi_perro.dueño
```

Herencia múltiple

La herencia múltiple es similar, pero una clase **hereda de varias clases** padre en vez de una sola.

Veamos un ejemplo. Por un lado, tenemos dos clases Clase1 y Clase2, y por otro tenemos la Clase3 que hereda de las dos anteriores. Por lo tanto, heredará todos los métodos y atributos de ambas.

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3(Clase1, Clase2):
    pass
```

Es posible también que una clase herede de otra clase y a su vez otra clase herede de la anterior.

```
class Clase1:
    pass
class Clase2(Clase1):
    pass
class Clase3(Clase2):
    pass
```

las clases hijas heredan los métodos de las clases padre, pero también pueden reimplementarlos de manera distinta. Entonces, si llamo a un método que todas las clases tienen en común ¿a cuál se llama? Pues bien, existe una forma de saberlo.

La forma de saber a qué método se llama es consultar el **MRO** o *Method Order Resolution*. Esta función nos devuelve una tupla con el orden de búsqueda de los métodos. Como era de esperar se empieza en la propia clase y se va subiendo hasta la clase padre, de izquierda a derecha.

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3(Clase1, Clase2):
    pass

print(Clase3.__mro__)
# (<class '__main__.Clase3'>, <class '__main__.Clase1'>, <class
'__main__.Clase2'>, <class 'object
```

Todas las clases en Python heredan de una clase genérica `object`, aunque no lo especifiquemos explícitamente.
Podemos tener una clase heredando de otras tres. Fíjate en que el **MRO** depende del orden en el que las clases son pasadas: 1, 3, 2.

```
class Clase1:
    pass
class Clase2:
    pass
class Clase3:
    pass
class Clase4(Clase1, Clase3, Clase2):
    pass
print(Clase4.__mro__)
# (<class '__main__.Clase4'>, <class '__main__.Clase1'>, <class
'__main__.Clase3'>, <class '__m
```

POLIMORFISMO

El término polimorfismo tiene origen en las palabras *poly* (muchos) *morfo* (formas), y aplicado a la programación hace referencia a que los objetos pueden tomar diferentes formas.
La técnica de polimorfismo de la POO significa la capacidad de tomar más de una forma. Una operación puede presentar diferentes comportamientos en diferentes instancias. El comportamiento depende de los tipos de datos utilizados en la operación. El polimorfismo es ampliamente utilizado en la aplicación de la herencia.

Utilizando polimorfismo podemos invocar un mismo método de diferentes objetos y obtener diferentes resultados según la clase de estos.
Esto significa que podemos **llamar a un método exactamente igual a otro** y el intérprete automáticamente **detectará a cuál de ellos nos referimos según diversos parámetros**, por ejemplo, el tipo de dato que pasamos como argumento al momento de llamarlo, la clase a la que pertenece, o hasta podemos especificarle a que método nos referimos. El polimorfismo esta estrictamente ligado al concepto de **Herencia**
Para acudir al método de la clase padre desde la clase hija recordemos que podemos usar la ***función Super()***