Clase 14 - Herencias

# **Herencia**

## **¿Qué es?**

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 sobreescribir los métodos o atributos, o incluso definir unos nuevos.

## **¿De qué se trata?**

Se puede crear una clase hija con tan solo pasar como parámetro la clase de la que queremos heredar.

## 

**¿Cómo funciona?**

Dado que una clase hija hereda los atributos y métodos de la padre, nos puede ser muy útil cuando tengamos clases que se parecen entre sí pero tienen ciertas particularidades.

En este caso, siguiendo el ejemplo anterior, en vez de definir muchas 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.**

**Principio de DRY**

**Don't repeat yourself**

Cuanto más código duplicado exista, más difícil será de modificar y más fácil será crear inconsistencias.

Las clases y la herencia ayudan a no repetir código de manera innecesaria.

Realizar **estas abstracciones y buscar el denominador común para definir una clase de la que hereden las demás,** es una tarea de lo más compleja en el mundo de la programación.

**Atributos**

* **Especie:** todos los animales pertenecen a una.
* **Edad:** momento cronológico.

**Métodos o funcionalidades**

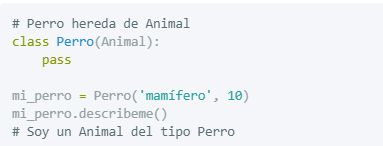
* Método **hablar:** que cada animal implementará de una forma.
* Método **moverse:**  Unos animales lo harán caminando, otros volando.
* Método **descríbeme** que será común a todos los animales.

**Clase vacía**

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 cómo los métodos y atributos son heredados por defecto.



**Creando clases**

Con tan solo un par de líneas de código, hemos creado una clase nueva que tiene todo el contenido que la clase padre tiene.

A continuación vamos a crear varios animales concretos y sobreescribir 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.

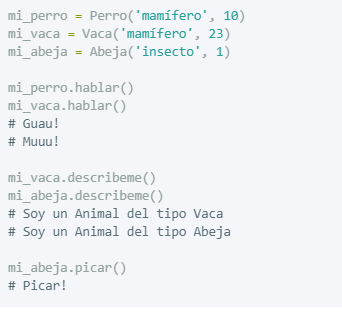


Podemos, incluso, crear nuevos métodos que se añadirán a los ya heredados, como en el caso de la **Abeja** con **picar()**.

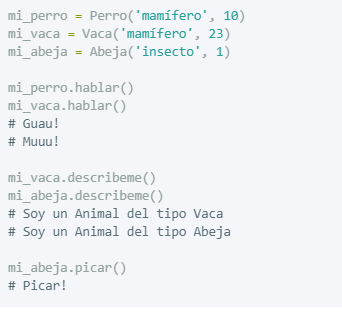
🐝



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: **descríbeme.**
* 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.**

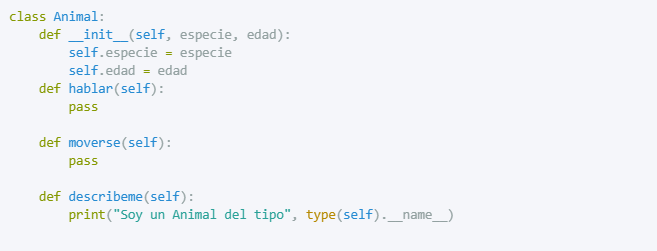
****

**super()**

**¿Para qué sirve?**

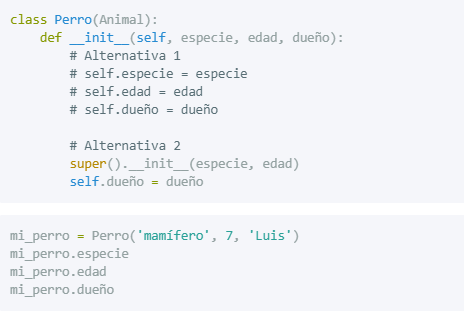
La función **super** nos permite acceder a los métodos de la clase padre desde una de sus hijas.

Volvamos al ejemplo de Animal y Perro…

****

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

1. Podemos crear un nuevo **\_init\_** y guardar todas las variables una a una.
2. 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.

****

**Herencia múltiple**

Hemos visto cómo se podía crear una clase padre que heredaba de una clase hija, pudiendo hacer uso de sus métodos y atributos.

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

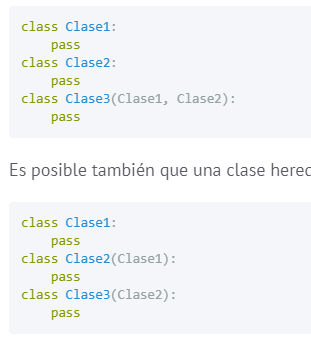
En Python es posible realizar herencia múltiple.

Anteriormente hemos visto cómo se podía crear una clase padre que heredaba de una clase hija, pudiendo hacer uso de sus métodos y atributos.

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

**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.



Hasta ahora sabíamos que las clases hijas heredan los métodos de las clases padre, pero también pueden reimplementarlos de manera distinta. Este punto nos plantea el siguiente interrogante:

Si llamo a un método que todas las clases tienen en común ¿A cuál se llama?

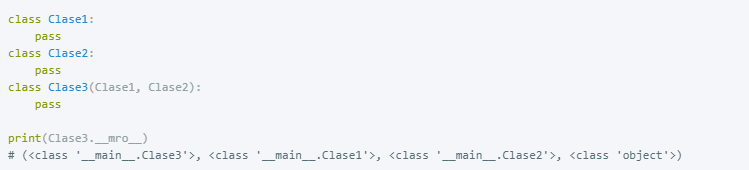
Pues bien, existe una forma de saberlo.

**Method Order Resolution**

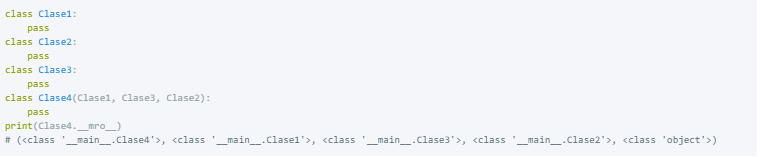
Sabremos a qué método se llama, consultando al MRO. **Esta función nos devuelve una tupla con el orden de búsqueda de los métodos.**

Como era de esperar se empieza en la propia clase y se va subiendo hasta la clase padre, de izquierda a derecha.

Una curiosidad es que al final del todo vemos la clase object. Aunque pueda parecer raro, es correcto ya que en realidad todas las clases en Python heredan de una clase genérica object, aunque no lo especifiquemos explícitamente.



También podemos tener una clase heredando de otras tres. Vemos que el MRO depende del orden en el que las clases son pasadas: 1, 3, 2.



**Poliformismo**

**¿De qué se trata?**

El término aplicado a la programación hace referencia a que los objetos pueden tomar diferentes formas. ¿Pero qué significa esto?

Significa que **objetos de diferentes clases pueden ser accedidos utilizando el mismo interfaz, mostrando un comportamiento distinto (tomando diferentes formas) según cómo sean accedidos.**

**¿Para qué se utiliza?**

Si la técnica de polimorfismo, siendo una propiedad de la POO (Programación Orientada a Objetos), implica la capacidad de tomar más de una forma, una operación puede presentar diferentes comportamientos en diferentes instancias.

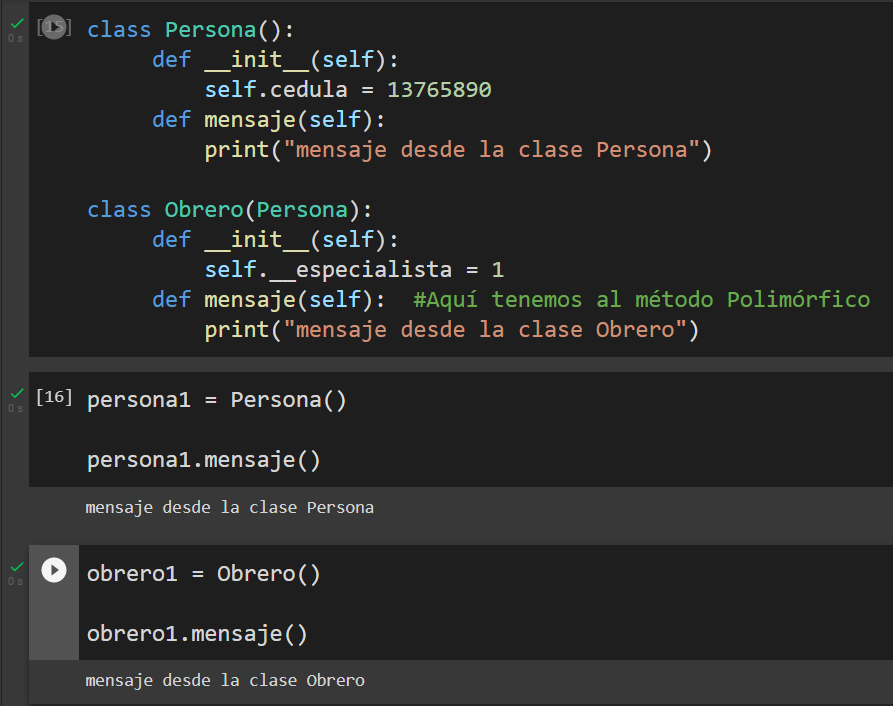
**El comportamiento depende de los tipos de datos utilizados en la operación.** El polimorfismo es ampliamente utilizado en la aplicación de la herencia.

Veamos cómo funciona …

**¿Cómo se utiliza?**

Podemos sustituir un método proveniente de la Clase Padre, en la Clase Hija. Se debe definir un método con el mismo nombre y parámetros, pero debe tomar otra conducta.

Es básicamente lo que veníamos haciendo sin saber que se llamaba Polimorfismo. 🤯

****

**Duck typing**

**¿Qué es?**

El término polimorfismo visto desde el punto de vista de Python es complicado de explicar sin hablar del duck typing. Es un concepto que aplica a ciertos lenguajes orientados a objetos y que tiene origen en la siguiente frase:

If it walks like a duck and it quacks like a duck, then it must be a duck

(Si camina como un pato y habla como un pato, entonces tiene que ser un pato)

¿Y qué relación tienen los patos con la programación? Pues bien, se trata de un símil en el que los patos son objetos y hablar/andar métodos. Es decir, que **si un determinado objeto tiene los métodos que nos interesan, nos basta, siendo su tipo irrelevante.**

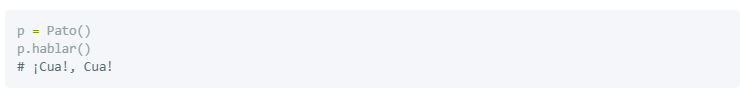
**¿Cómo funciona?**

Una vez entendido el origen del concepto, veamos lo que realmente significa esto en Python. En pocas palabras, **a Python le dan igual los tipos de los objetos, lo único que le importan son los métodos.**

Definamos una clase **pato** con un método **hablar ( ).**



Y llamamos al método de la siguiente forma:



**Método hablar ()**

En Python **no es necesario especificar los tipos**, simplemente decimos que el parámetro de entrada tiene el nombre **x.**

Hasta aquí nada nuevo, pero vamos a definir una función **llama\_hablar ( )**, que llama al método hablar ( ) del objeto que se le pase.



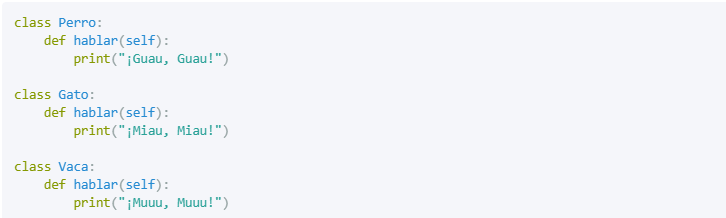
Cuando Python entra en la función y evalúa **x.hablar(**), le da igual el tipo al que pertenezca **x** siempre y cuando tenga el método **hablar()**.



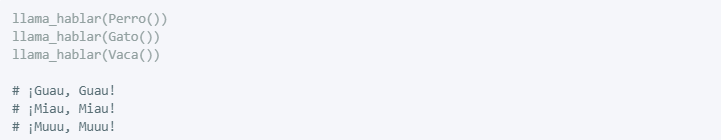
Esto es el duck typing en todo su esplendor. 🤩

Definamos tres clases de animales distintas que implementan el método **hablar().**

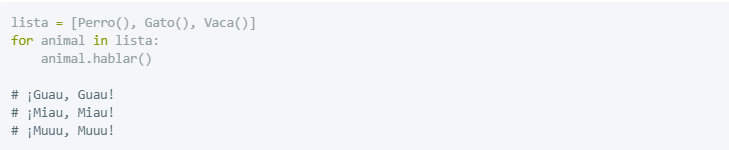
Nótese que no existe **herencia** entre ellas, son clases totalmente independientes. De haberla estaríamos hablando de **polimorfismo.**



Y, como es de esperar, la función **llama\_hablar()** funciona correctamente con todos los objetos.



Otra forma de verlo, es iterando una lista con diferentes animales, donde animal toma los valores de cada objeto animal.



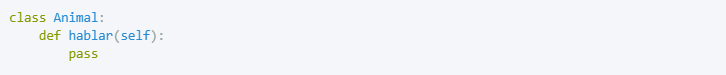
**Conclusiones**

* Python es un lenguaje que soporta el duck typing, lo que hace que el tipo de los objetos no sea tan relevante, siendo más importante lo que sus métodos pueden hacer.
* Otros lenguajes como Java, no soportan el duck typing, pero se puede conseguir un comportamiento similar cuando los objetos comparten un interfaz (si existe herencia entre ellos). Este concepto relacionado es el **polimorfismo.**
* El duck typing está en todos lados, desde la función **len()** hasta el uso del operador **\***

**Volviendo al Poliformismo**

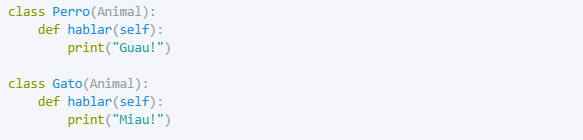
Al ser un lenguaje con tipado dinámico y permitir duck typing, en Python no es necesario que los objetos compartan un interfaz, simplemente basta con que tengan los métodos que se quieren llamar.

Supongamos que tenemos una clase **Animal** con un método **hablar().**



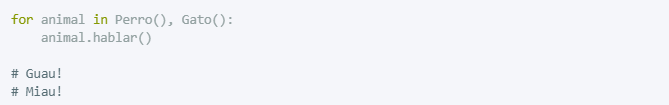
**hablar () y Poliformismo**

Por otro lado tenemos otras dos clases, **Perro**, **Gato** que heredan de la anterior. Además, implementan el método **hablar()** de una forma distinta.



**hablar () y Poliformismo**

A continuación creamos un objeto de cada clase y llamamos al método **hablar()**. Podemos observar que cada animal se comporta de manera distinta al usar **hablar().**



En el caso anterior, la variable animal ha ido “tomando las formas” de Perro y Gato. Sin embargo, nótese que al tener tipado dinámico este ejemplo hubiera funcionado igual sin que existiera herencia entre Perro y Gato, como sucede en duck typing.

**Herencia y Encapsulamiento**

**Por ultimo!**

A través de la aplicación del concepto de Encapsulamiento, podemos indicarle a Python por ejemplo, que no queremos que un método sea heredado por una clase hija.

Veamos a continuación el notebook: [*Herencia y Encapsulamiento.ipynb*](https://colab.research.google.com/drive/1fVM8GtAPn7LP9HMsvaOnC9HyMqoUa9fc)