

MÓDULO 4: Parte 2

Objetos dentro de Objetos.

Herencia.

Polimorfismo

1000
PROGRAMADORES

>Objetos dentro de Objetos

Ahora que hemos implementado **clases**, y hemos instanciado **objetos** a partir de ellas. Podemos notar que no hay una diferencia sustancial al momento de emplear uno de estos objetos o una variable “*simple*”. Esto se debe a que en **Python**, **TODO ES UN OBJETO**.

En retrospectiva, cuando utilizamos variables de los tipos de datos **int**, **float**, **str** o **bool**, estábamos trabajando también con objetos. Estos tipos de datos están predefinidos y disponibles para el usuario, de manera que no tengamos que implementarlos por nuestra cuenta.

>Objetos dentro de Objetos

Así que, cuando necesitemos resolver un problema que requiera de un tipo de dato que no se encuentre predefinido, o bien necesite de alguna **característica** o **comportamiento** que no se encuentre implementado previamente, **siempre podremos crear una nueva clase**.

Tomando en cuenta estas afirmaciones, ahora resulta natural que, estas **clases**, se puedan poner en **colecciones** o utilizarlas dentro de otras clases (por ejemplo, como un atributo de otra clase).

>Objetos dentro de Objetos

En base a estos conceptos proponemos el siguiente ejercicio.

Describiremos un **catálogo de películas** para analizar cómo se diseña e implementa un anidamiento de objetos:



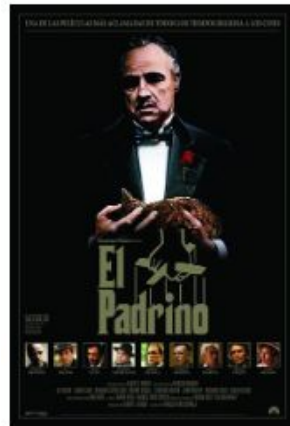
>Objetos dentro de Objetos

Comenzaremos definiendo la clase **Película** para poder instanciar todas las películas que deseamos en variables independientes

```
class Pelicula:
    def __init__(self, titulo, duracion, lanzamiento):
        self.titulo = titulo
        self.duracion = duracion
        self.lanzamiento = lanzamiento

    def __str__(self):
        return self.titulo+' '+str(self.lanzamiento)
```

```
peli = Pelicula("El Padrino", 175, 1972)
```



>Objetos dentro de Objetos

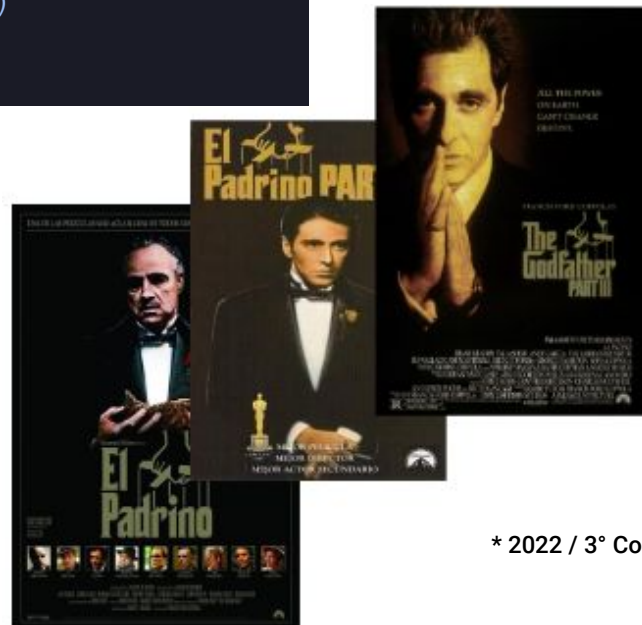
```
class Catalogo:
    def __init__(self, peliculas=None):
        if peliculas != None:
            self.peliculas = peliculas
        else:
            self.peliculas = []
    def agregar(self, p):
        self.peliculas.append(p)
    def __str__(self):
        cadena = '\nCatálogo:'
        for pelicula in self.peliculas:
            cadena += '\n'+str(pelicula)
        return cadena
```

A continuación definimos la clase **Catálogo**, que contará con una lista de **Películas** como atributo.

>Objetos dentro de Objetos

```
pe1i1 = Pelicula("El Padrino", 175, 1972)
pe1i2 = Pelicula("El Padrino II", 202, 1974)
pe1i3 = Pelicula("El Padrino III",162, 1991)
catalogo = Catalogo([pe1i1,pe1i2,pe1i3])
print(catalogo)
```

Así podemos *anidar una clase dentro de otra*. Siendo que al menos un atributo pertenece a otra clase

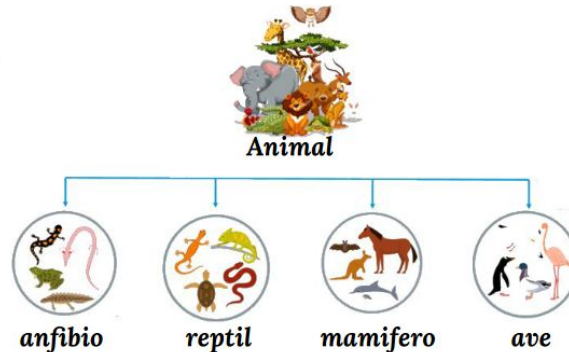


>Herencia

La herencia es una poderosa característica que otorga la capacidad de crear una nueva clase extendiendo una clase ya existente, y esto permite que una clase herede los atributos y métodos de la clase principal. Nos permite crear clases más especializadas que reutilicen código de una clase general.

>Herencia

- La clase de la que hereda una clase se llama padre o superclase. Una clase que hereda de una superclase se llama hija o subclase.
- Existe una relación jerárquica entre clases similar a las relaciones o categorizaciones que conocemos de la vida real.



>Herencia

Para heredar primero necesitaremos una clase padre, a modo de ejemplo emplearemos la clase **Animal**. No realizaremos **encapsulamiento** ya que es la primera implementación de **herencia** que haremos, pero puede realizarse si lo necesitamos:

>Herencia

```
class Animal:
    def __init__(self, especie, edad):
        self.especie = especie
        self.edad = edad
    def hablar(self):
        pass
    def moverse(self):
        pass
    def __str__(self):
        return "Soy un Animal del tipo "+str(type(self).__name__)
```

Por supuesto, como los métodos **hablar()** y **moverse()** están vacíos, las instancias de la clase **Animal** no realizarán nada al invocarlos.

```
animal1 = Animal("Canis lupus familiaris",5)
```

>Herencia

En **Python**, al momento de heredar debemos emplear la siguiente sintaxis:

```
class ClaseHija(ClasePadre):
```

Como vemos deberemos crear una nueva clase y especificar entre paréntesis, a continuación del nombre de la clase que heredará, el nombre de la clase padre. Entonces teniendo en cuenta la clase **Animal** antes implementada, heredamos a una clase **Perro**:

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

>Herencia

Podemos notar que, además de realizar la herencia implementamos los métodos **hablar()** y **moverse()** que encontramos en la clase padre.

Entonces ¿qué ocurrirá si un objeto ***firulais*** de la clase perro realiza el comportamiento **hablar()**?

```
firulais = Perro("Pastor Aleman",4)  
firulais.hablar()
```

Guau!


Lógicamente ***firulais*** ladra, puesto que es un **Perro** y hemos ***especializado*** el comportamiento **hablar()** para todo **Perro**

>Herencia

Podemos decir también que los atributos y métodos de la clase padre fueron heredados a la clase hija. Es decir, que cualquier objeto de la clase **Perro** cuenta con los atributos **especie** y **edad**, además del método **__str__**. Que a pesar de no haber sido implementado en la clase **Perro** puede ser invocado de igual manera.

>Herencia

```
print(firulais.especie)  
print(firulais.edad)  
print(firulais)
```



```
Pastor Aleman  
4  
Soy un Animal del tipo Perro
```

Vemos que podemos mostrar por consola a los atributos de la instancia ***firulais***, y por supuesto también podemos mostrar al mismo ***firulais*** dado que en la clase padre de **Perro**, es decir en **Animal**, fue definido el método **__str__**.

De esta manera podemos emplear la herencia para reducir sustancialmente el código repetido, y emplear la herencia para dedicarnos exclusivamente a la **especialización** de las clases.

>Herencia

De la misma manera podemos heredar a otras clases. Es decir, podemos especializar el comportamiento siempre que lo necesitemos mediante la herencia.

>Herencia

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

Podemos notar que el comportamiento **moverse()** es idéntico para **Perro** y **Vaca**, ambos se mueven “Caminando con 4 patas”, lo cual perfectamente puede suceder en la realidad.

>Herencia

Objetos de diferentes clases realizan un mismo comportamiento, pero el hecho de realizar **herencia** radica en aquellos comportamientos que son **específicos** de una clase hija. Es decir, el método **hablar()** es la especialización que poseen ambas clases, puesto que ambas especies realizan sonidos diferentes (perros ladran y vacas mugen).

```
lola = Vaca("Bos taurus",2)  
lola.hablar()
```

Muuu!

>Herencia

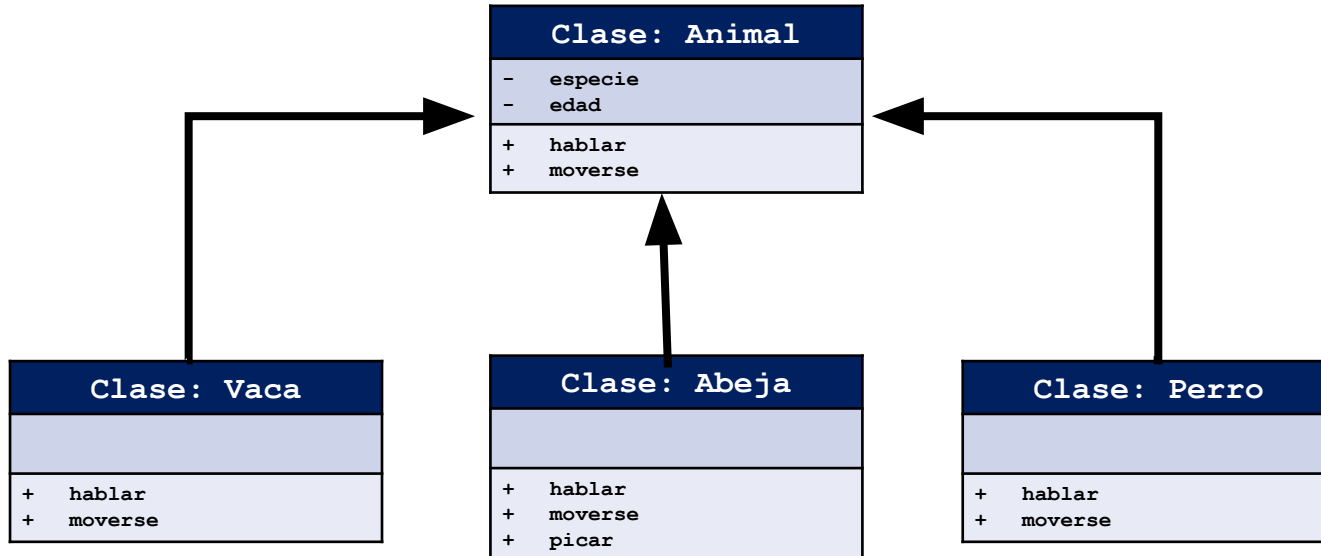
Cabe destacar que no nos encontramos limitados a implementar únicamente los métodos **hablar()** y **moverse()**, sino que podemos agregar nuevos comportamientos propios de la clase hija. Por ejemplo:

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

Como vemos el método **picar()** es propio de la clase **Abeja**.

>Herencia

A partir de estas clases podemos establecer la siguiente jerarquía de clases, donde **Vaca**, **Perro** y **Abeja** heredan de **Animal**



>Duck Typing en Python

El término *polimorfismo* visto desde el punto de vista de Python es complicado de explicar sin hablar del **duck typing**.

El **duck typing** o **tipado de pato** es un concepto relacionado con la programación 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".

Lo que se podría traducir al español como: *Si camina como un pato y habla como un pato, entonces tiene que ser un pato.*

>Duck Typing en Python

- ¿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.

En pocas palabras, **en Python nos dan igual los tipos de objetos, siempre y cuando se implementen los métodos necesarios.**

Duck Typing en Python

Para ejemplificar este concepto definiremos una clase **Pato** y otra clase **Gato**. Ambas contarán con el comportamiento **hablar()**.

```
class Pato:
    def hablar(self):
        print("Cuac")
```

```
class Gato:
    def hablar(self):
        print("Miau")
```

A continuación definiremos una función que necesita del método **hablar()**.

```
def llamar_amigos(x):
    x.hablar()
    x.hablar()
```

Veremos la practicidad de **duck typing**, al momento de *invocar* la función **llamar_amigos**. Donde nos es indiferente conocer el tipo del objeto que usaremos como argumento de dicha función. Lo único importante, es que el objeto cuente con el método **hablar()**.

```
plucky = Pato()
llamar_amigos(plucky)
```

```
michi = Gato()
llamar_amigos(michi)
```

> Duck Typing en Python

Veremos la practicidad de **duck typing**, al momento de *invocar* la función **llamar_amigos**. Donde nos es indiferente conocer el tipo del objeto que usaremos como argumento de dicha función. Lo único importante, es que el objeto cuente con el método **hablar()**.

>Duck Typing en Python

```
plucky = Pato()  
llamar_amigos(plucky)
```

```
michi = Gato()  
llamar_amigos(michi)
```

A pesar de que obtengamos un resultado diferente en cada caso, lo cual es lógico considerando la naturaleza de cada objeto. Es importante resaltar que siempre tendremos una salida para la función **llamar_amigos**, siempre que hayamos implementado el método **hablar()** en la clase sobre la cual instanciamos nuestros objetos.

Como aclaración final, debemos mencionar que **duck typing** es una característica válida en **Python** sólo porque este se trata de un lenguaje de tipado dinámico, **y no todos los lenguajes de programación soportan este mecanismo.**

>Polimorfismo

En el ámbito de **P00** el polimorfismo hace referencia a la habilidad que tienen objetos de diferentes clases para responder a métodos con el mismo nombre pero con implementaciones diferentes.

Por ejemplo, para las clases **Perro**, **Vaca** y **Abeja** podemos decir que tanto el método **hablar** como el método **moverse** emplean el **polimorfismo**, puesto que cada clase responde de una manera en particular.

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

```
class Vaca(Animal):  
    def hablar(self):  
        print("Muuu!")  
    def moverse(self):  
        print("Caminando")
```

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

>Polimorfismo

El polimorfismo nos brindará cierta independencia al momento de realizar un comportamiento, puesto que nuestros objetos son abstracciones de la realidad y es importante que puedan resolver una misma tarea acorde a la naturaleza del mismo objeto. A final de cuentas, existen múltiples interpretaciones para resolver una misma tarea.

A person is holding a yellow sticky note with the word "CODE" written on it in blue ink. The person is smiling and pointing towards the camera. The background is blurred, showing a desk with a laptop and other items. The entire image has a green overlay.

CODE

🏠 web: <http://milprogramadores.unsa.com.ar>

📍 telegram: <https://t.me/milprogramadoressaltenios>

💖 centro de ayuda: <http://ayudamilprogramadores.com/>