Esta clase va a ser

- grabada

Clase 14. PYTHON

Herencias

Temario

13	14	15
Programación orientada a objetos II	Herencias	Scripts, Módulos y Paquetes
✓ POO	✓ Herencia	✓ Scripts
✓ Clases	✓ Herencia múltiple	✓ Módulo
✓ Atributos	✓ Poliformismo	✓ Paquete
✓ Métodos		

Objetivos de la clase

- **Aplicar** Herencias.
- **Conocer** los polimorfismos.
- **Ejecutar** Herencias múltiples.

Herencia

CODERHOUSE

¿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.

Fuente: ElLibroDePython

¿De qué se trata?

```python
# Definimos una clase padre
class Animal:
    pass

# Creamos una clase hija que hereda de la padre
class Perro(Animal):
    pass
```

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

Fuente: ElLibroDePython

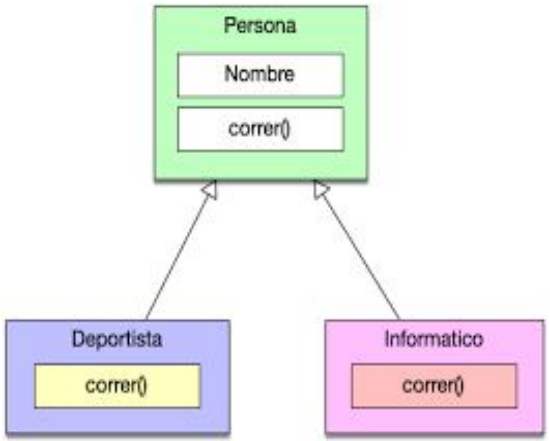
¿Cómo funciona?

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.**

Fuente: ElLibroDePython

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.

Fuente: ElLibroDePython

PARA RECORDAR

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.

Ejemplo en vivo

Vamos a definir una clase padre **Animal** que tendrá todos los atributos y métodos genéricos que los animales pueden tener.

```python
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):
        print("Soy un Animal del tipo", type(self).__name__)
```

Atributos

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

Fuente: ElLibroDePython

```python
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):
        print("Soy un Animal del tipo", type(self).__name__)
```

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.

Fuente: ElLibroDePython

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.👉

Fuente: ElLibroDePython

Clase vacía

Como primer ejemplo vamos a crear una clase vacía, para ver cómo los métodos y atributos son heredados por defecto.

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

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

Fuente: ElLibroDePython

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.

```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!")
```

Fuente: ElLibroDePython

Creando clases

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

```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!")
```

Fuente: ElLibroDePython

Creando clases

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

👉

```
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!
```

Fuente: ElLibroDePython

Creando clases

- ✓ 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.**

```
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!
```

Fuente: ElLibroDePython

super()

¿Para qué sirve?

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

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

    def moverse(self):
        pass

    def describeme(self):
        print("Soy un Animal del tipo", type(self).__name__)
```

Volvamos al ejemplo de Animal y Perro...

Fuente: ElLibroDePython

¿Para qué sirve?

```
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
```

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

Fuente: ElLibroDePython

¿Para qué sirve?

```
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
```

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.

Fuente: ElLibroDePython

Break

¡10 minutos y volvemos!

Herencia múltiple

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.

Fuente: ElLibroDePython

Para pensar

¿Qué diferencias se pueden detectar?

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

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

Contesta mediante el chat de Zoom

Herencia múltiple

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.

Fuente: ElLibroDePython

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
```

Fuente: ElLibroDePython

Ejemplo

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

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

Fuente: ElLibroDePython

PARA RECORDAR

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.

Fuente: ElLibroDePython

Method Order Resolution

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.

```
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'>)
```

Fuente: ElLibroDePython

Method Order Resolution

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.

```
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 '__main__.Clase2'>, <class 'object'>)
```

Fuente: ElLibroDePython

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.**

Fuente: ElLibroDePython

¿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 👉

Fuente: ElLibroDePython

¿Cómo se utiliza?

```
class Persona():
        def __init__(self):
            self.cedula = 13765890
        def mensaje(self):
            print("mensaje desde la clase Persona")

    class Obrero(Persona):
        def __init__(self):
            self.__especialista = 1
        def mensaje(self):  #Aquí tenemos al método Polimórfico
            print("mensaje desde la clase Obrero")
```

```
[16] persona1 = Persona()

    persona1.mensaje()
```

```
mensaje desde la clase Persona
```

```
obrero1 = Obrero()

obrero1.mensaje()
```

```
mensaje desde la clase Obrero
```

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. 🤯

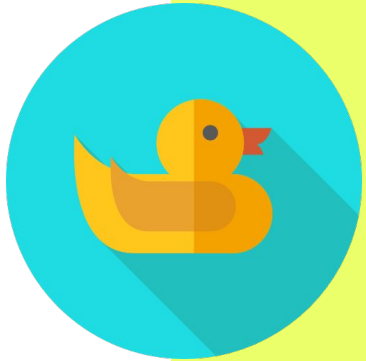
Duck typing

¿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)



Fuente: ElLibroDePython

PARA RECORDAR

¿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.**

```
class Pato:
    def hablar(self):
        print("¡Cua!, Cua!")
```

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

Y llamamos al método de la siguiente forma:

```
p = Pato()
p.hablar()
# ¡Cua!, Cua!
```

Fuente: ElLibroDePython

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.

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

Fuente: ElLibroDePython

Método hablar ()

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()**.

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

Esto es el duck typing en todo su esplendor. 🤩

Fuente: ElLibroDePython

Método hablar ()

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

```
class Perro:
    def hablar(self):
        print("¡Guau, Guau!")

class Gato:
    def hablar(self):
        print("¡Miau, Miau!")

class Vaca:
    def hablar(self):
        print("¡Muuu, Muuu!")
```

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

Fuente: ElLibroDePython

Método hablar ()

```
llama_hablar(Perro())
llama_hablar(Gato())
llama_hablar(Vaca())

# ¡Guau, Guau!
# ¡Miau, Miau!
# ¡Muuu, Muuu!
```

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.

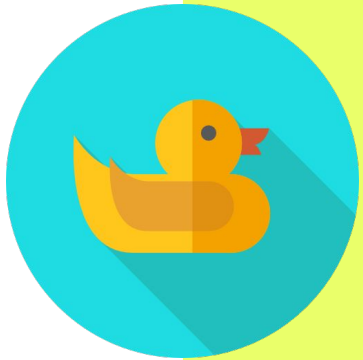
```
lista = [Perro(), Gato(), Vaca()]
for animal in lista:
    animal.hablar()

# ¡Guau, Guau!
# ¡Miau, Miau!
# ¡Muuu, Muuu!
```

Fuente: ElLibroDePython

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 ***



Fuente: ElLibroDePython

CODERHOUSE

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().**

```
class Animal:
    def hablar(self):
        pass
```

Fuente: ElLibroDePython

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.

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

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

Fuente: ElLibroDePython

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().**

```
for animal in Perro(), Gato():
    animal.hablar()

# Guau!
# Miau!
```

Fuente: ElLibroDePython

PARA RECORDAR

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

CODERHOUSE

¡Por último!

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*

Herencia múltiple

Crea una herencia múltiple, trabajando con Mamífero, Cetáceo, AnimalMarino.

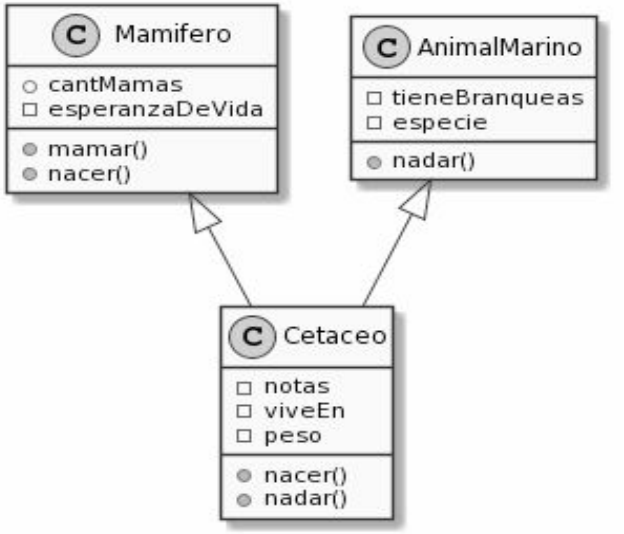
Duración: **20 minutos**

CODERHOUSE

ACTIVIDAD EN CLASE

Herencia múltiple

Realiza la herencia múltiple respetando este diagrama de Clases UML. Y crear dos Cetáceos.



Especie	Longitud (m)	Peso (kg)	Notas
Cochito o Vaquita marina	1,2-1,5	30-55	Cetáceos más pequeño. en Peligro crítico de extinción.
Delfín común	2,4	70-110	
Delfín mular	3,7	150-650	
Narval	5	800-1600	
Ballena franca pigmea	6,5	3000-3500	Misticetos más pequeños
Orca	9,7	2600-9000	Delfínidos más grandes
Yubarta	13,7-15,2	25 000-40 000	
Cachalote	14,9-20	13 000-14 000	Odontoceto más grandes
Ballena azul	30	110 000	Animal más grande de todos

Fuente: ElLibroDePython

CODERHOUSE

Segunda pre-entrega

En la clase que viene se presentará la consigna de la primera parte del Proyecto final, que nuclea temas vistos entre las clases 12 y 15. Recuerda que tendrás 7 días para subirla en la plataforma.

MATERIAL AMPLIADO

Recursos multimedia

Ejercicios

- ✓ Ejercicios de Repaso
- ✓ Ejercicios Clase En Vivo

Disponible en nuestro repositorio.

¿Preguntas?

Resumen
de la clase hoy

- ✓ Herencia: definición, uso.
- ✓ Principio de DRY.
- ✓ Clases: creación, atributos, métodos y funcionalidades.
- ✓ Herencia Múltiple.
- ✓ Poliformismo.
- ✓ Duck Typing.

CODERHOUSE

Muchas gracias.

#DemocratizandoLaEducación