

Introducción a la programación con Python Herencia y Polimorfismo

Alexis Rodríguez

Marcel Morán C

Esquema

- Mi libro de animales
- ¿Que es herencia?
- Sintaxis de herencia
- Mi libro de animales con herencia
- ¿Que es una clase abstracta?
- Sintaxis de clase abstracta
- Orden de resolución de métodos (MRO)
- Polimorfismo en acción
- Sintaxis de polimorfismo
- Ejemplo de polimorfismo

Examen

- **23/12/2024 desde 2:30 - 4:30 pm**
- Lectura y Talleres 1 a 7
- Formato de examen Teoría y Práctica
 - Teoría (9 preguntas) 16 puntos
 - Opcion multiple, verdadero o false
 - Respuesta corta
 - ¿Cuál es la diferencia entre un compilador y un intérprete?
 - Práctica (3 preguntas) 34
 - Conceptos básicos de programación (asignación, loops, recursión) e.j calcular el término de una secuencia fibonacci (10 puntos)
 - Conceptos básicos de programación (Archivos, estructuras) e.j crea un diario(10 puntos)
 - Conceptos OOP (14 puntos)

Un programa de animales



Un libro de animales

```
class Perro:
```

```
    def __init__(self, edad, nombre, sonido):  
        self.edad = edad  
        self.nombre = nombre  
        self.sonido = sonido
```

```
    def describete(self):  
        print("Hola me llamo", self.nombre, "tengo", self.edad, "años y",  
self.sonido)
```

```
max = Perro(10, "Max", "ladr@",)  
max.describete()  
>>> Hola me llamo Max tengo 10 años y ladr@
```



Un libro de animales

```
class Gato:
```

```
    def __init__(self, edad, nombre, sonido):  
        self.edad = edad  
        self.nombre = nombre  
        self.sonido = sonido
```

```
    def describete(self):  
        print("Hola me llamo", self.nombre, "tengo", self.edad, "años y",  
self.sonido)
```

```
gato = Gato(15, "Mineta", "maull@",)  
gato.describete()  
>>> Hola me llamo Mineta tengo 15 años y maull@
```



Un libro de animales

```
class Cuervo:
```

```
    def __init__(self, edad, nombre, sonido):  
        self.edad = edad  
        self.nombre = nombre  
        self.sonido = sonido
```

```
    def describete(self):  
        print("Hola me llamo", self.nombre, "tengo", self.edad, "años y",  
self.sonido)
```

```
cuervo = Cuervo(5, "Itachi", "grazn@",)  
cuervo.describete()  
>>> Hola me llamo Itachi tengo 5 años y grazn@
```



Un libro de animales

```
class Caballo:
```

```
    def __init__(self, edad, nombre, sonido):  
        self.edad = edad  
        self.nombre = nombre  
        self.sonido = sonido
```

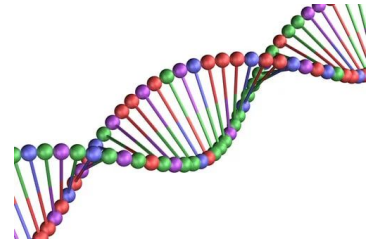
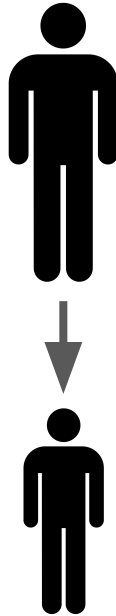
```
    def describete(self):  
        print("Hola me llamo", self.nombre, "tengo", self.edad, "años y",  
self.sonido)
```

```
caballo = Caballo(8, "Spirit", " relinch@")  
caballo.describete()  
>>> Hola me llamo Spirit tengo 8 años y relinch@
```



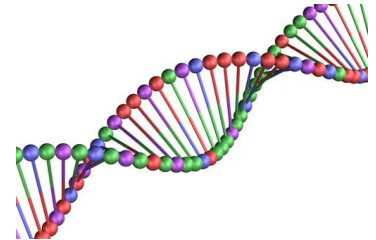
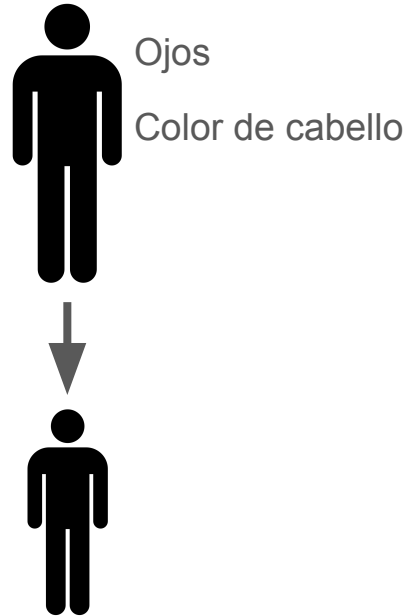
Programación Orientada a Objetos

- Es un derivado de una clase que hereda propiedades de su clase
- La clase padre hereda sus propiedades a sus hijos



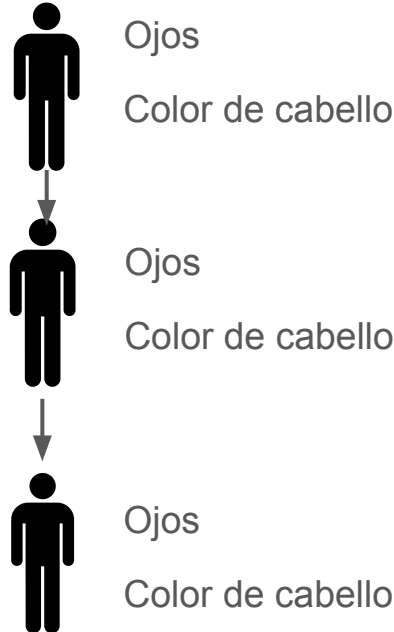
Programación Orientada a Objetos

- Es un derivado de una clase que hereda propiedades de su clase
- La clase padre hereda sus propiedades a sus hijos



Programación Orientada a Objetos

- Es un derivado de una clase que hereda propiedades de su clase
- La clase padre hereda sus propiedades a sus hijos
- Clases superiores



Sintaxis de Herencia

- Asumiendo que tenemos nuestro clase Padre
- **class** Hijo(**Padre**):
- **super().__init__**(argumentos_n1, argumentos_n2)

Un libro de animales

```
class Animal:
```

```
    def __init__(self, edad, nombre, sonido):  
        self.edad = edad  
        self.nombre = nombre  
        self.sonido = sonido
```

```
    def describete(self):  
        print("Hola me llamo", self.nombre, "tengo", self.edad, "años y",  
self.sonido)
```



Un libro de animales

```
class Perro(Animal):  
    def __init__(self, edad, nombre, sonido):  
        super().__init__(edad, nombre, sonido)
```

```
class Gato(Animal):  
    def __init__(self, edad, nombre, sonido):  
        super().__init__(edad, nombre, sonido)
```

```
class Cuervo(Animal):  
    def __init__(self, edad, nombre, sonido):  
        super().__init__(edad, nombre, sonido)
```

```
class Caballo(Animal):  
    def __init__(self, edad, nombre, sonido):  
        super().__init__(edad, nombre, sonido)
```



Un libro de animales

```
perro = Perro(10, "max", "ladr@")
gato = Gato(15, "Mineta", "maull@")
cuervo = Cuervo(5, "Itachi", "grazn@")
caballo = Caballo(8, "Spirit", "ladr@")
```

```
perro.describete()
gato.describete()
cuervo.describete()
caballo.describete()
```

```
>>>Hola me llamo max tengo 10 años y ladr@
>>>Hola me llamo Mineta tengo 15 años y maull@
>>>Hola me llamo Itachi tengo 5 años y grazn@
>>>Hola me llamo Spirit tengo 8 años y ladr@
```

Clases abstractas - El problema

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

    def describete(self):
        print("Hola me llamo", self.nombre, "tengo",
self.edad, "años y", self.sonido)

    def hablar(self):
        print("Hola me llamo", self.nombre, "tengo",
self.edad, "años y", self.sonido)
```

```
animal_1 = Animal(10, "max", "ladr@")
animal_2 = Animal(5, "Itachi", "grazn@")
```

```
animal_1.describete()
animal_2.describete()
```

```
animal_1.hablar()
animal_2.hablar()
```



```
Hola me ll.
Hola me ll.
Hola me ll.
Hola me ll.
grazn@
```



```
ños y ladr@
años y grazn@
ños y ladr@
años y
```


Clases abstractas - La solución

```
from abc import ABC, abstractmethod
```

```
class Animal(ABC):  
    def __init__(self, edad, nombre, sonido):  
        self.edad = edad  
        self.nombre = nombre  
        self.sonido = sonido
```

```
@abstractmethod  
def describete(self):  
    pass
```

```
@abstractmethod  
def hablar(self):  
    pass
```

```
animal_1 = Animal(10, "max", "ladr@")  
animal_2 = Animal(5, "Itachi", "grazn@")
```

```
animal_1.describete()  
animal_2.describete()
```

```
animal_1.hablar()  
animal_2.hablar()
```



TypeError

Traceback (most recent call last)

```
~\AppData\Local\Temp\ipykernel_26580\1939502414.py in <module>  
----> 1 animal_1 = Animal(10, "max", "ladr@")  
      2 animal_2 = Animal(5, "Itachi", "grazn@")  
      3  
      4  
      5 animal_1.describete()
```

TypeError: Can't instantiate abstract class Animal with abstract methods describete, hablar

Clases abstractas - Subclases

```
class Gato(Animal):  
    def __init__(self, edad, nombre, sonido):  
        super().__init__(edad, nombre, sonido)  
  
    def describete(self):  
        print("Hola me llamo", self.nombre, "tengo",  
self.edad, "años y", self.sonido)  
  
    def hablar(self):  
        print("Hola me llamo", self.nombre, "tengo",  
self.edad, "años y", self.sonido)
```

```
gato = Gato(15, "Mineta", "maull@")  
gato.describete()  
gato.hablar()
```



```
Hola me llamo Mineta tengo 15  
años y maull@  
Hola me llamo Mineta tengo 15  
años y maull@
```

Clases abstractas

- Una clase abstracta es aquella que implemente un método abstracto
- Permite a los hijos implementar definir los métodos abstractos
- Clases abstractas **no pueden ser instanciadas**



wuah

kro



jegi

miau



Sintaxis de clases abstractas

- from **abc** import **ABC, abstractmethod**
- decorador/decorator encima del método **@abstractmethod**
- **class nombre(ABC)**



wuah

kro



jegi

miau



Orden de resolución de métodos (MRO)

```
class Padre:  
    def __str__(self):  
        return 'Hola yo soy un objeto de clase  
Padre'
```

```
class Hijo(Padre):  
    def __str__(self):  
        return 'Hola yo soy un objeto de clase hijo'
```

```
class Nieto(Hijo):  
    def __str__(self):  
        return 'Hola yo soy un objeto de clase  
nieto'
```

Sobrescribir

The diagram illustrates the Method Resolution Order (MRO) for the classes Padre, Hijo, and Nieto. It shows three class definitions. The 'Padre' class has a '__str__' method. The 'Hijo' class inherits from 'Padre' and also has a '__str__' method. The 'Nieto' class inherits from 'Hijo' and also has a '__str__' method. Three arrows originate from the right side of the image, pointing to the '__str__' method of each class. The top arrow points to the 'Padre' class, the middle arrow points to the 'Hijo' class, and the bottom arrow points to the 'Nieto' class. The word 'Sobrescribir' (Override) is placed to the right of the arrows, indicating that the child classes are overriding the method from their parents.


Orden de resolución de métodos (MRO)

```
un_ejemplo_nieto = Nieto()
```

```
representation = un_ejemplo_nieto.__str__()  
print(representation)
```

```
Hola yo soy un objeto de clase nieto
```

Método dentro de
clase Nieto



Cómo accedemos a funciones que estén en clases superiores?

```
print(Nieto.__mro__)
```

```
(__main__.Nieto, __main__.Hijo, __main__.Padre, object)
```

Orden de resolución de métodos (MRO)

```
un_ejemplo_nieto = Nieto()

print(Nieto.__mro__)

(__main__.Nieto, __main__.Hijo, __main__.Padre, object)
```

```
representation_str = super(Nieto, un_ejemplo_nieto).__str__()
print(representation_str)
```

Método dentro de
clase Hijo

Hola yo soy un objeto de clase hijo

```
representation_str = super(Hijo, un_ejemplo_nieto).__str__()
print(representation_str)
```

Método dentro de
clase Padre

Hola yo soy un objeto de clase padre

Polimorfismo en acción

Miembro de un equipo



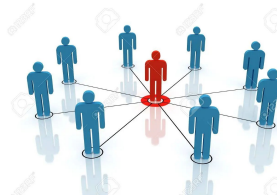
Hola yo soy
un miembro
de un equipo

Trabajador



Hola yo soy
un trabajador

Empleado



Sintaxis de polimorfismo en Python

```
class MiembroEquipo:
    def __init__(self, nombre equipo):
        self.nombre_equipo = nombre_equipo

    def describete(self):
        print(self.__str__())

    def __str__(self):
        return 'Soy un miembro de un equipo, el nombre de mi equipo es ' + self.nombre_equipo

class Trabajador:
    def __init__(self, salario, titulo_trabajo):
        self.salario = salario
        self.titulo_trabajo = titulo_trabajo

    def describete(self):
        print(self.__str__())

    def __str__(self):
        return 'Soy un trabajador, mi titulo de trabajo es ' + self.titulo_trabajo + ' y mi salario es ' + self.salario
```

Sintaxis de polimorfismo en Python

```
class Empleado(MiembroEquipo, Trabajador):  
    def init (self, nombre equipo, salario, titulo_trabajo):  
        MiembroEquipo. init (self, nombre equipo)  
        Trabajador.__init__(self, salario, titulo_trabajo)  
  
    def str (self):  
        miembro equipo str =super(LiderEquipo, self). str ()  
        trabajador equipo str =super(MiembroEquipo, self). str ()  
        return miembro_equipo_str + '. Ademas, ' + trabajador_equipo_str
```

```
empleado= Empleado('Inteligencia Artificial', 2000, 'programador en Inteligencia Artificial')  
empleado.describete()
```

Soy un miembro de un equipo, el nombre de mi equipo es Inteligencia Artificial.
Ademas, Soy un trabajador, mi titulo de trabajo es programador en Inteligencia Artificial y mi salario es 2000

Conclusión

- Uno de los principios OOP se aplica con la herencia
- Una clase hereda propiedades de otra clase o clase superior
- Sintaxis de python para heredar los atributos y métodos de otras `class nombre (Clase Superior)`
- Clases abstractas no pueden ser instanciadas pero permiten la implementación de sus clases
- `from abc import ABC, abstractmethod @abstractmethod`
- Polimorfismo se refiere a que hay varias maneras de ejecutar la misma acción
- Sobre escribir ocurre cuando un método en una clase hijo lleva el mismo nombre que una en la clase padre.
- No existe sobre carga de métodos en Python.
- El atributo `__mro__` permite acceder a la lista de búsqueda de atributos
- Usando la función `super()` podemos acceder a métodos dentro de superclases

Retroalimentación

- Para retroalimentación dirigirse al siguiente enlace <https://forms.gle/HXPRwxJdEizL25fz5> .
- Déjanos saber qué podemos hacer para mejorar el curso

