

Programación Avanzada

IIC2233 2025-2

Cristian Ruz - Pablo Araneda - Francisca Ibarra - Tamara Vidal - Daniela Concha



Anuncios

28 de agosto de 2025



1. Hoy tenemos la Experiencia 1 de OOP.
2. No hay control de salida (por esta vez)
3. Ya está disponible la ECA. Recuerden llenarla todas las semanas (tiene de domingo a martes) para la bonificación extra a fin de semestre ✨.

Herencia en OOP



Herencia en OOP

- ¿Cuál es la diferencia entre atributos de clase y atributos de instancia?
- ¿Qué significa heredar en OOP?
- ¿Cuál es la relación entre una *superclase* y una *subclase*?
- ¿Por qué y cuándo usamos el método `super()`?
- ¿Hay algún límite en la herencia en OOP?

Ejemplo de herencia

```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    def saludar(self):
        print("Es un honor saludarte! 🧑")
```

```
class MaestroAgua(Persona):
    def __init__(self, nombre, sabe_curar):
        super().__init__(nombre)
        self.sabe_curar = sabe_curar

    def agua_control(self):
        print("Te voy a congelar!")

    def superataque(self):
        if self.sabe_curar:
            self.saludar()
            print("Sana sana colita de rana 🐸")
        else:
            print("Lo siento 😞")
```

```
class MaestroFuego(Persona):
    def __init__(self, nombre, controla_rayos):
        super().__init__(nombre)
        self.controla_rayos = controla_rayos

    def fuego_control(self):
        print("Recibe mi bola de fuego!")

    def superataque(self):
        if self.controla_rayos:
            self.saludar()
            print("Pika pika... chu ⚡")
        else:
            print("Todavía no sé tirar rayos ☁")
```

Polimorfismo



Polimorfismo

- ¿Qué implica realizar *overriding* en Python?
- *Overloading* no existe en Python.
- ¿Y los operadores aritméticos en Python?

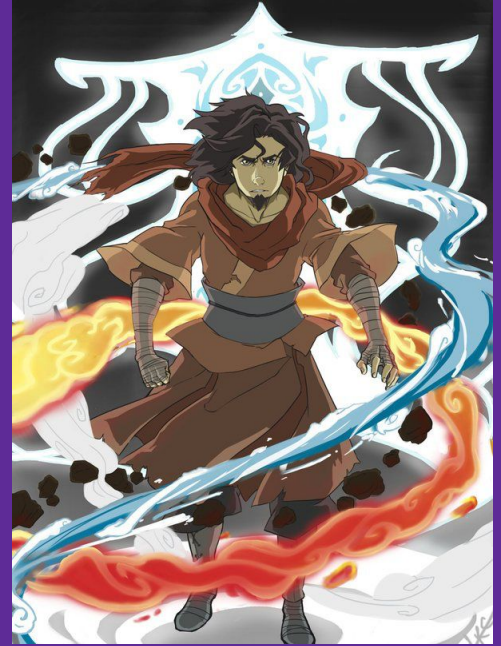
Ejemplo de *overriding* y “*overloading*”

```
class Persona:  
    def __init__(self, nombre):  
        self.nombre = nombre  
  
    def entrenar(self):  
        print("¿Y si no voy?")
```

```
class MaestroAgua(Persona):  
    def __init__(self, nombre):  
        super().__init__(nombre)  
  
    def entrenar(self):  
        print("Me voy a una cascada 🌊")
```

```
class MaestroFuego(Persona):  
    def __init__(self, nombre):  
        super().__init__(nombre)  
  
    def entrenar(self):  
        print("Necesito un volcán 🌋")  
  
    def entrenar(self, tiempo: int):  
        if tiempo < 60:  
            print(f"No alcanzo buu 😓")  
  
    def entrenar(self, maestro: str):  
        print(f"¡Aprenderé mucho con {maestro}!")
```


Multiherencia



Multiherencia

- ¿Cuál es la diferencia entre herencia simple y multiherencia?
- ¿Cuáles son sus ventajas?
- ¿Qué significa MRO y como afecta en la herencia?
- ¿Qué es el problema del diamante?

Ejemplo de multiherencia y MRO

```
class Aleman:
    def __init__(self):
        self.pais = "Alemania"

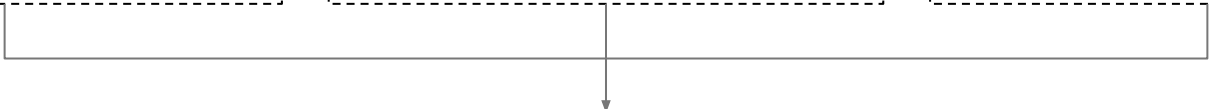
    def nacionalidad(self):
        print("Soy alemán!")
```

```
class Ingles:
    def __init__(self):
        self.pais = "Inglaterra"

    def nacionalidad(self):
        print("Soy inglés!")
```

```
class Chileno:
    def __init__(self):
        self.pais = "Chile"

    def nacionalidad(self):
        print("Soy chileno!")
```



```
class CiudadaniaMultiple(Aleman, Ingles, Chileno):
    def __init__(self, nombre):
        super().__init__()
        self.nombre = nombre

    def informacion_pasaporte(self):
        print(f"País:{self.pais} Nombre:{self.nombre}")
        self.nacionalidad()
```

The diagram illustrates a multiple inheritance hierarchy. Three base classes, `Aleman`, `Ingles`, and `Chileno`, are shown in separate boxes at the top. Arrows from each of these boxes point downwards to a single box containing the `CiudadaniaMultiple` class, indicating that `CiudadaniaMultiple` inherits from all three base classes.

Clases abstractas



Clases abstractas

- ¿Qué es una clase abstracta?
- ¿Cuándo es necesario usarla?
- ¿Cuál es el uso del módulo abc?

Decoradores



Decoradores

- ¿Qué son los objetos de primera clase?
- ¿Qué problema resuelven los decoradores?
- ¿Qué devuelve un decorador?
- ¿Qué diferencia hay entre acceder a un método normal y a una `property`?
- ¿Qué ventajas tiene usar `@property` frente a acceder directamente a atributos?

Decoradores

```
import time
# Esta es la funcion decoradora
def agregar_tiempo(func):
    def dormir(n: int, m: int) -> int:
        time.sleep(2)
        print("Espere 2 segundos")
        return func(n, m)
    return dormir

# Aplicamos el decorador a sumar_algo
@agregar_tiempo
def sumar_algo(n: int, m: int) -> int:
    return n + m

print("Empieza")
print(sumar_algo(3, 2))
print("Termina")
```


Decoradores

sin decoradores

```
class Planta:
    def __init__(self, nombre: str) -> None:
        self.nombre = nombre
        self.calidad = "bueno"

    def obtener_calidad(self) -> str:
        print("accediendo al getter")
        return "demasiado " + self.calidad

    def cambiar_calidad(self, nueva_calidad: str)
        self.calidad = nueva_calidad
        print(f"Parece que ahora soy un {self.nombre}
{self.calidad}.")

p = Planta("Zapallo")
print(p.calidad)
print(p.obtener_calidad())

p.cambiar_calidad("meh")

# ver el cambio
print(p.obtener_calidad())

# cuál es el output?
```

con decoradores @properties

```
class Planta:
    def __init__(self, nombre: str) -> None:
        self.su_nombre = nombre
        self.su_calidad = "bueno"

    @property
    def calidad(self) -> str:
        print("accediendo al getter")
        return "demasiado " + self.su_calidad

    @calidad.setter
    def calidad(self, nueva_calidad: str) -> None:
        print("accediendo al setter")
        self.su_calidad = nueva_calidad
        print(f"Parece que ahora soy un {self.su_nombre}
{self.su_calidad}.")

p = Planta("Zapallo")
print(p.su_calidad) # "bueno"

p.calidad = "regular"

print(p.calidad)

# cuál es el output?
```

Veamos una pregunta de Evaluación Escrita

Tema: OOP (Midterm 2024-1)

7. En función del siguiente código, ¿cuál o cuáles clases se pueden instanciar sin provocar un error al momento de ejecutar el código?

- A) Solo la clase A
- B) Solo la clase B
- C) Solo la clase C
- D) Las clases B y C
- E) Ninguna se puede instanciar

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

```
class A(ABC):
```

```
    @abstractmethod
```

```
    def saludar(self):
```

```
        pass
```

```
    @abstractmethod
```

```
    def despedir(self):
```

```
        pass
```

```
class B(A):
```

```
    def saludar(self):
```

```
        print("Konnichiwa")
```

```
class C(B):
```

```
    def despedir(self):
```

```
        print("Sayonara")
```

Veamos una pregunta de Evaluación Escrita

Tema: OOP (Midterm 2024-1)

7. En función del siguiente código, ¿cuál o cuáles clases se pueden instanciar sin provocar un error al momento de ejecutar el código?

- A) Solo la clase A
- B) Solo la clase B
- C) Solo la clase C**
- D) Las clases B y C
- E) Ninguna se puede instanciar

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

```
class A(ABC):
```

```
    @abstractmethod
```

```
    def saludar(self):
```

```
        pass
```

```
    @abstractmethod
```

```
    def despedir(self):
```

```
        pass
```

```
class B(A):
```

```
    def saludar(self):
```

```
        print("Konnichiwa")
```

```
class C(B):
```

```
    def despedir(self):
```

```
        print("Sayonara")
```

Programación Avanzada

IIC2233 2025-2

Cristian Ruz - Pablo Araneda - Francisca Ibarra - Tamara Vidal - Daniela Concha

