

Programación Avanzada

IIC2233 2023-2


Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattán



Anuncios

1. Hoy comenzamos la primera actividad.
 2. Ayer se subió la primera tarea
 3. Encuesta de Carga Académica.
¡Respóndanla!
-

Área de bienestar

Nati es nuestra
jefa de bienestar 

bienestar.iic2233@gmail.com

1. Solicitudes de apoyo y problemas personales.
2. Puede dar extensiones de plazo para las tareas en casos conversados.

¿Dónde quedamos?



OOP



- Paradigma de programación
- Interacción entre objetos

Properties ✨

- Encapsular atributos del objeto
- Manejar el acceso o modificación de uno o varios atributos

Properties, ¿Cómo funcionan?

```
class Planta:
    def __init__(self, nombre):
        self._nombre = nombre
        self._calidad = 'bueno'

    @property
    def calidad(self):
        return self._calidad

    @calidad.setter
    def calidad(self, nueva_calidad):
        self._calidad = nueva_calidad
        print(f'Parece que ahora soy un {self._nombre} {self._calidad}.')

p = Planta('Zapallo')
p.calidad = 'muy bueno'
```

Parece que ahora soy un Zapallo muy bueno.

Properties, ¿Cómo funcionan?

```
class Planta:
    def __init__(self, nombre):
        self._nombre = nombre
        self._calidad = 'bueno'

    @property
    def nombre(self):
        return self._nombre

    @nombre.setter
    def nombre(self, nuevo_nombre):
        print(f'Soy un {self.nombre} y nunca seré un {nuevo_nombre}.')

p = Planta('Zapallo', 50)
p.nombre = 'Tomate'
```

Soy un Zapallo y nunca seré un Tomate.

¿Y ahora qué viene?



Herencia



- Relación de **especialización** y **generalización** entre clases
- Una clase (*subclase*) **hereda** **atributos** y **comportamientos** de otra clase (*superclase*)

Herencia

Contexto: Suponga un mundo de **fantasía** 🧙‍♂️ donde se puede controlar los elementos de la naturaleza 💧 🔥 🌿 🌬️.

¿Qué objetos hay?

Herencia

Contexto: Suponga un mundo de **fantasía** 🧙‍♂️ donde se puede controlar los elementos de la naturaleza 💧 🔥 🌿 ☁️.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas?

Herencia

Contexto: Suponga un mundo de **fantasía** 🧙‍♂️ donde se puede controlar los elementos de la naturaleza 💧 🔥 🌿 🌬️.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas?

¿Depende! ¿MaestroAgua?

¿MaestroFuego? ¿MaestroViento?

Herencia

```
class Persona:

    def __init__(self, nombre):
        self.nombre = nombre

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

```
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")
```

```
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 😭")
```

Polimorfismo

- Utilizar objetos de distinto tipo con la misma **interfaz**
 - Se hace con ***overriding*** y ***overloading*** (Este último no está disponible en python 😞)
-

Polimorfismo, volviendo al ejemplo

Contexto: Suponga un mundo de **fantasía** 🧙‍♂️ donde se puede controlar los elementos de la naturaleza 💧 🔥 🌿 🌬️.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas?

¿Depende! ¿MaestroAgua?
¿MaestroFuego? ¿MaestroViento?

¿Qué acción es común a todos, pero cada uno lo hace de forma distinta?

Polimorfismo, volviendo al ejemplo

Contexto: Suponga un mundo de **fantasía** 🧙‍♂️ donde se puede controlar los elementos de la naturaleza 💧 🔥 🌿 🌬️.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas?

¿Depende! ¿MaestroAgua?
¿MaestroFuego? ¿MaestroViento?

¿Qué acción es común a todos, pero cada uno lo hace de forma distinta? **Entrenar**

Polimorfismo

```
class Persona:

    def entrenar(self):
        pass

class MaestroAgua(Persona):

    def entrenar(self):
        print("Me voy a una cascada")

class MaestroFuego(Persona):

    def entrenar(self):
        print("Necesito un volcán")
```

Multiherencia



Una clase puede heredar de más de una superclase

Multiherencia, volviendo al ejemplo

Contexto: Suponga un mundo de **fantasía** 🧙‍♂️ donde se puede controlar los elementos de la naturaleza 💧 🔥 🌿 🌬️.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas?

¿Depende! ¿MaestroAgua?
¿MaestroFuego? ¿MaestroViento?

¿Qué acción es común a todos, pero cada uno lo hace de forma distinta? **Entrenar**

¿Y si alguien controla 2 elementos?

Multiherencia, volviendo al ejemplo

Contexto: Suponga un mundo de **fantasía** 🧙‍♂️ donde se puede controlar los elementos de la naturaleza 💧 🔥 🌿 🌬️.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas?

¿Depende! ¿MaestroAgua?

¿MaestroFuego? ¿MaestroViento?

¿Qué acción es común a todos, pero cada uno lo hace

¿Y si alguien controla 2 elementos? **Multiherencia**



Multiherencia

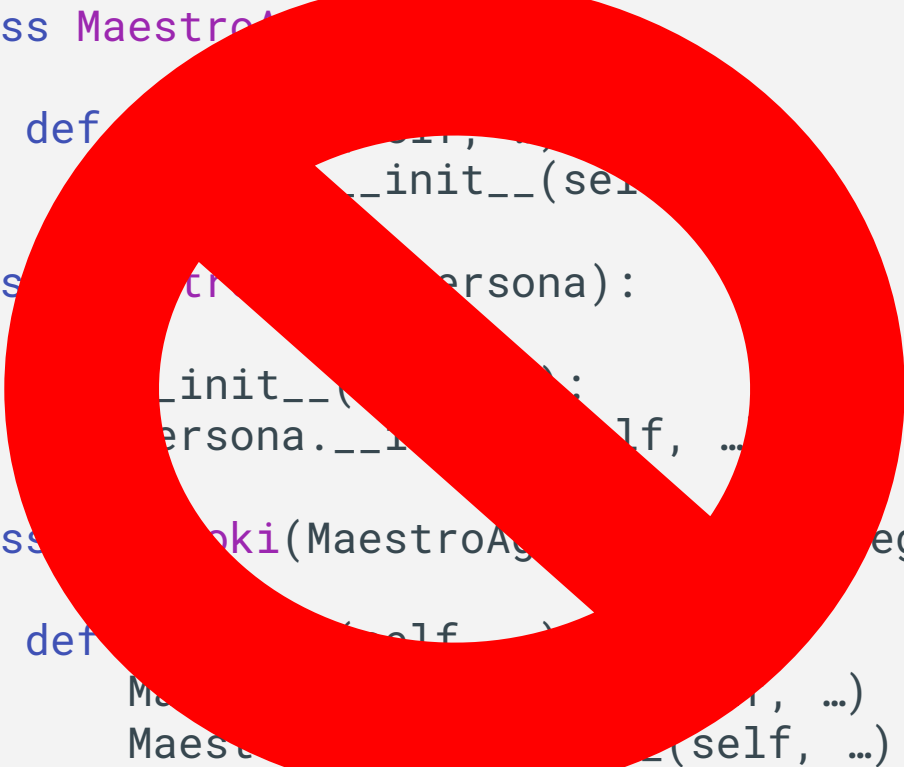
```
class MaestroAgua(Persona):  
    def __init__(self, ...):  
        Persona.__init__(self, ...)  
  
class MaestroFuego(Persona):  
    def __init__(self, ...):  
        Persona.__init__(self, ...)  
  
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, ...):  
        MaestroAgua.__init__(self, ...)  
        MaestroFuego.__init__(self, ...)
```

Multiherencia

```
class MaestroAgente:
    def __init__(self, ...):
        ...

class Persona:
    def __init__(self, ...):
        ...

class Agente(MaestroAgente, Persona):
    def __init__(self, ...):
        MaestroAgente.__init__(self, ...)
        Persona.__init__(self, ...)
```



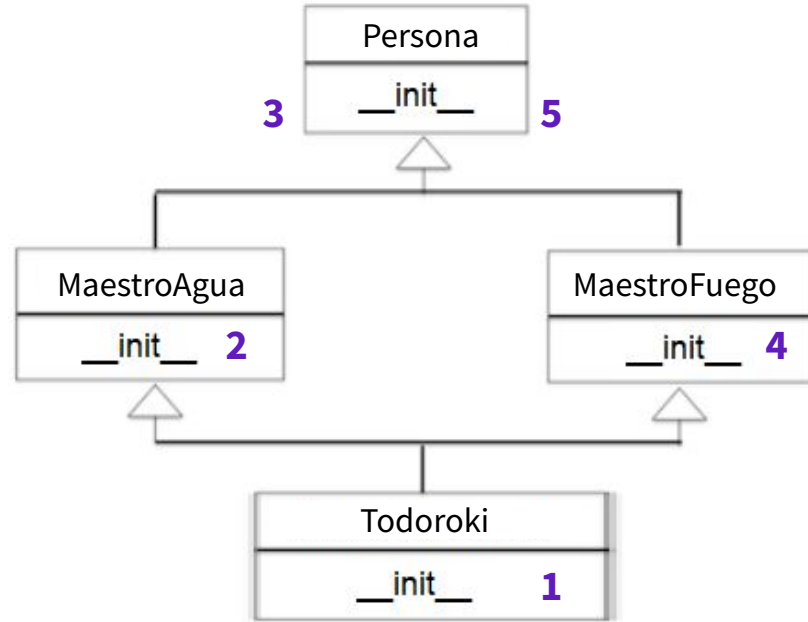
**Tendremos el
problema del
diamante** 💎

🤔 ¿Por qué?

Multiherencia

Sigamos el hilo de ejecución al instanciar a Todoroki, **sin usar super()**

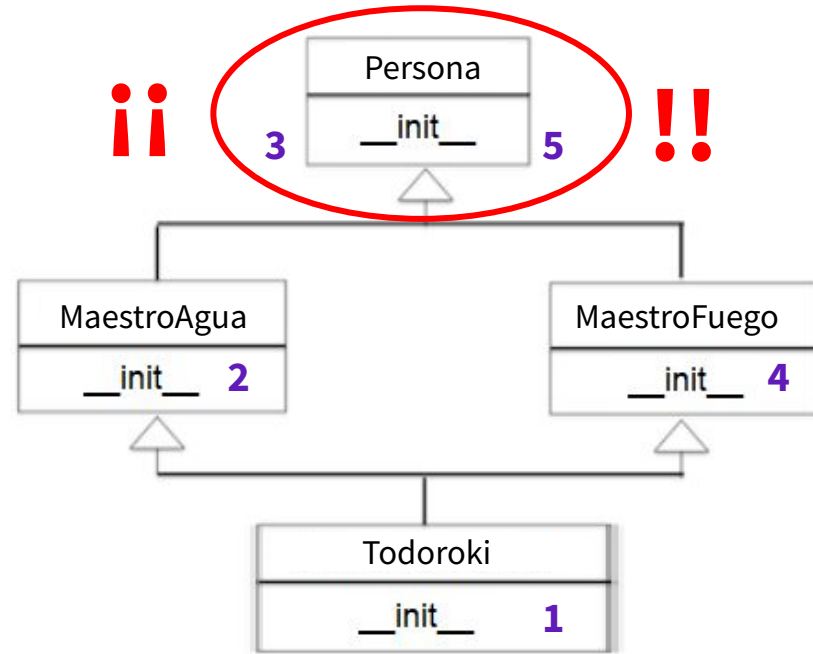
```
class MaestroAgua(Persona):  
    def __init__(self, ...):  
        Persona.__init__(self, ...)  
  
class MaestroFuego(Persona):  
    def __init__(self, ...):  
        Persona.__init__(self, ...)  
  
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, ...):  
        MaestroAgua.__init__(self, ...)  
        MaestroFuego.__init__(self, ...)
```



Multiherencia

Sigamos el hilo de ejecución al instanciar a Todoroki, **sin usar super()**

```
class MaestroAgua(Persona):  
    def __init__(self, ...):  
        Persona.__init__(self, ...)  
  
class MaestroFuego(Persona):  
    def __init__(self, ...):  
        Persona.__init__(self, ...)  
  
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, ...):  
        MaestroAgua.__init__(self, ...)  
        MaestroFuego.__init__(self, ...)
```



Multiherencia

```
class MaestroAgua(Persona):  
    def __init__(self, ...):  
        super().__init__(...)  
  
class MaestroFuego(Persona):  
    def __init__(self, ...):  
        super().__init__(...)  
  
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, ...):  
        super().__init__(...)
```

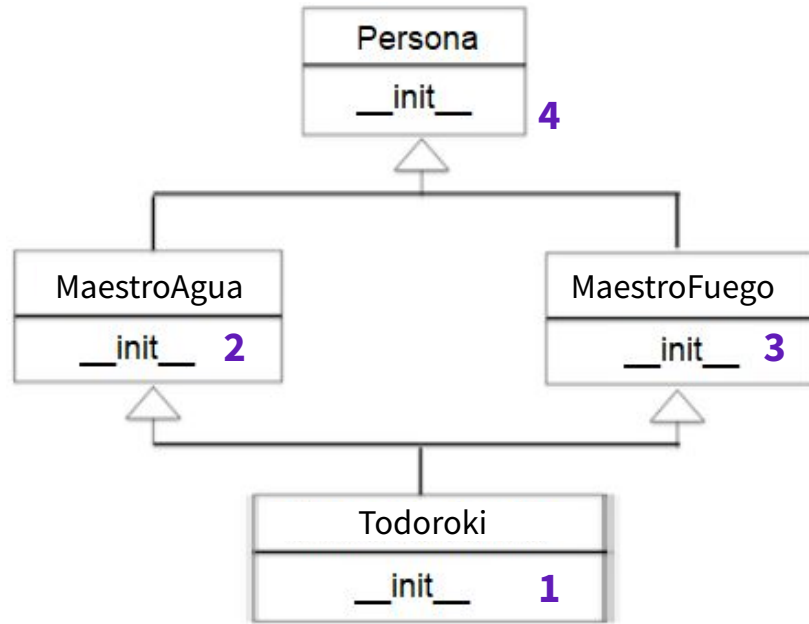
**Esta es la
solución**

🤔 ¿Por qué?

Multiherencia

Sigamos el hilo de ejecución al instanciar a Todoroki pero **haciéndolo bien esta vez**

```
class MaestroAgua(Persona):  
    def __init__(self, ...):  
        super().__init__(...)  
  
class MaestroFuego(Persona):  
    def __init__(self, ...):  
        super().__init__(...)  
  
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, ...):  
        super().__init__(...)
```



Multiherencia

Sigan

```
class M
    def

class M
    def

class T
    def
```



PERFECTLY BALANCED

AS ALL THINGS SHOULD BE

sta vez

```
MaestroFuego
    __init__ 3
```

Multiherencia



¿Cómo paso diferentes argumentos a mis padres solo con **un super**?

```
class MaestroAgua:

    def __init__(self, curar):
        self.puede_curar = curar

class MaestroFuego:

    def __init__(self, rayos):
        self.controla_rayos = rayos
```

```
class Todoroki(MaestroFuego, MaestroAgua):
```

```
    def __init__(self, vida, puede_curar, controla_rayos):
        super().__init__(puede_curar, controla_rayos)
```

Nos saldrá un error 🤖



Multiherencia - operadores * y **

Solución: Uso de “*” y “**” + **super()** en las clases padres:

```
class MaestroAgua:

    def __init__(self, curar, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.puede_curar = curar
```

```
class MaestroFuego:

    def __init__(self, rayos, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.controla_rayos = rayos
```

```
class Todoroki(MaestroFuego, MaestroAgua):

    def __init__(self, vida, puede_curar, controla_rayos):
        super().__init__(curar=puede_curar, rayos=controla_rayos)
```

¿Cómo funciona todo esto?

Multiherencia - operadores * y **

Explicación paso a paso:

```
class MaestroAgua:

    def __init__(self, curar, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.puede_curar = curar
```

```
class MaestroFuego:

    def __init__(self, rayos, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.controla_rayos = rayos
```

```
class Todoroki(MaestroFuego, MaestroAgua):
```

```
    def __init__(self, vida, puede_curar, controla_rayos):
        super().__init__(curar=puede_curar, rayos=controla_rayos)
```

1. El **super()** manda los 2 argumentos a **MaestroFuego** como *keywords*

Multiherencia - operadores * y **

Explicación paso a paso:

```
class MaestroAgua:
```

```
    def __init__(self, curar, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self.puede_curar = curar
```

```
class MaestroFuego:
```

```
    def __init__(self, rayos, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self.controla_rayos = rayos
```

```
class Todoroki(MaestroFuego, MaestroAgua):
```

```
    def __init__(self, vida, puede_curar, controla_rayos):  
        super().__init__(curar=puede_curar, rayos=controla_rayos)
```

2. **rayos** es cargado en el primer argumento. **curar** queda guardado dentro de ****kwargs**

Multiherencia - operadores * y **

Explicación paso a paso:

```
class MaestroAgua:

    def __init__(self, curar, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.puede_curar = curar
```

```
class MaestroFuego:

    def __init__(self, rayos, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.controla_rayos = rayos
```

```
class Todoroki(MaestroFuego, MaestroAgua):

    def __init__(self, vida, puede_curar, controla_rayos):
        super().__init__(curar=puede_curar, rayos=controla_rayos)
```

3. **super()** manda los argumentos de ***args** y ****kwargs** a la siguiente clase. En este caso, MaestroAgua

Multiherencia - operadores * y **

Explicación paso a paso:

```
class MaestroAgua:
```

```
    def __init__(self, curar, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self.puede_curar = curar
```

```
class MaestroFuego:
```

```
    def __init__(self, rayos, *args, **kwargs):  
        super().__init__(*args, **kwargs)  
        self.controla_rayos = rayos
```

```
class Todoroki(MaestroFuego, MaestroAgua):
```

```
    def __init__(self, vida, puede_curar, controla_rayos):  
        super().__init__(curar=puede_curar, rayos=controla_rayos)
```

4. **curar** es cargado en el primer argumento. ****kwargs queda vacío**

Multiherencia - operadores * y **

No olviden, *args* y *kwargs* es solo una convención, lo importante es el “*” y “**”

```
class MaestroAgua:

    def __init__(self, curar, *uwu, **bro):
        super().__init__(*uwu, **bro)
        self.puede_curar = curar

class MaestroFuego:

    def __init__(self, rayos, *nana, **nani):
        super().__init__(*nana, **nani)
        self.controla_rayos = rayos

class Todoroki(MaestroFuego, MaestroAgua):

    def __init__(self, vida, puede_curar, controla_rayos):
        super().__init__(curar=puede_curar, rayos=controla_rayos)
```

Multiherencia - Reflexión

 ¿Siempre hay que usar `super ()` cuando hacemos multiherencia?

> No  . Depende de cada caso.

Si es que el problema del diamante genera un error en la ejecución del código:

> Es necesario recurrir al uso de `super ()`.

Si necesitamos llamar a métodos de 2 o más padres, y utilizar sus `return`:

> Es necesario evaluar si con `ClasePadre.metodo(. . .)` está todo listo o bien utilizar `super ()`. Dependerá del caso a caso.

Clases Abstractas



- Clase que no se instancia directamente
- Contiene uno o más métodos abstractos
- Subclases implementan métodos abstractos

Clase abstracta, volviendo al ejemplo

Contexto: Suponga un mundo de **fantasía** 🧙 donde se puede controlar los elementos de la naturaleza 💧 🔥 🌿 🌬️.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas?

¿Depende! ¿MaestroAgua?
¿MaestroFuego? ¿MaestroViento?

¿Qué acción es común a todos, pero cada uno lo hace de forma distinta? **Entrenar**

Oye.... pero cómo fuerza que todos deban entrenar ...

Clase abstracta, volviendo al ejemplo

Contexto: Suponga un mundo de **fantasía** 🧙 donde se puede controlar los elementos de la naturaleza 💧 🔥 🌿 🌬️.

¿Qué objetos hay? **Personas**

¿Qué características tienen estas personas?

¿Depende! ¿MaestroAgua?
¿MaestroFuego? ¿MaestroViento?

¿Qué acción es común a todos, pero cada uno lo hace de forma distinta? **Entrenar**

Oye.... pero cómo fuerza que todos deban entrenar ... **Clase abstracta**

Clase abstracta

```
from abc import ABC, abstractmethod

class Persona(ABC):

    def __init__(self, nombre):
        self.nombre = nombre

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


Hablemos de la Tarea 1

Pasemos a la Actividad 1 🚗

Programación Avanzada

II C2233 2023-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattán

