

Programación Avanzada

IIC2233 2024-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattán



Anuncios

1. Hoy es la Actividad 2, recuerden que se entrega a las 23:59.
2. El viernes 30 se publica la Tarea 2.
3. La ECA se encuentra disponible para responder de domingo a martes.
4. Prioricen git en sus entregas 🕒

OOP

- Paradigma de programación
- Interacción entre objetos

Programación Orientada a Objetos

Objeto: Colección de datos que además tiene comportamientos asociados

¿Cómo los representamos en Python? ¡Clases!

Programación Orientada a Objetos

Objeto: Colección de datos que además tiene comportamientos asociados

¿Cómo los representamos en Python? ¡Clases!

```
class Planta:
    def __init__(self, nombre: str, resistencia: int) -> None:
        self.nombre = nombre           # Zapallo
        self.agua = 0
        self.resistencia = resistencia # 50 unidades

    def __str__(self) -> str:
        return f'{self.nombre} - {self.agua}/{self.resistencia}'
        # Zapallo - 0/50

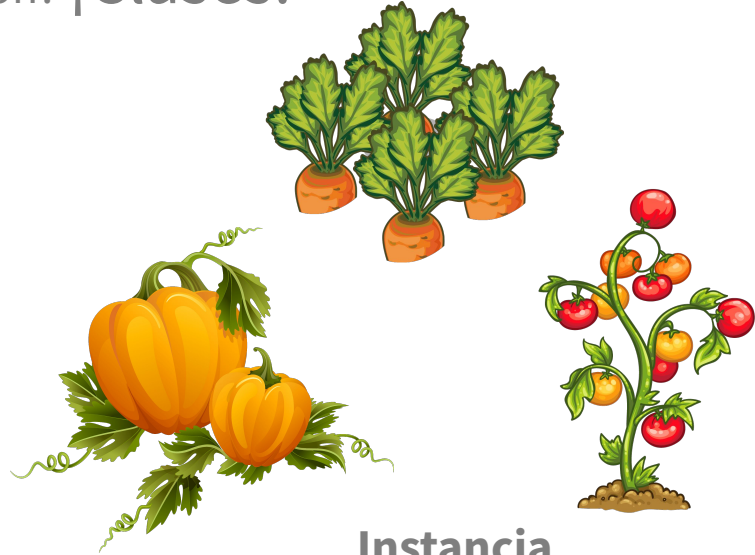
    def regar(self, cantidad: int) -> None:
        self.agua += cantidad
        if self.agua >= self.resistencia:
            self.agua = self.resistencia
```

Programación Orientada a Objetos

Objeto: Colección de datos que además tiene comportamientos asociados
¿Cómo los representamos en Python? ¡Clases!



Clase



Instancia

Programación Orientada a Objetos

Interacción entre Objetos: Podemos utilizar los métodos y propiedades de un objeto en otro.

```
class Huerto:
    def __init__(self) -> None:
        self.plantas = []

    def plantar(self, planta: Planta) -> None:
        if planta not in self.plantas:
            planta.regar(5)
            self.plantas.append(planta)
```



OOP

Atributos de instancia
vs.
Atributos de clases

Atributos de instancia y de clase

Atributo de **instancia**

- Relacionados a una Instancia en particular.
- Necesitamos referencia a su instancia para usarlos.
- Ya los hemos usado en OOP.
- Necesitan un `self`.

Atributo de **clase**

- Compartidos por todas las instancias de una clase.
- Su modificación se refleja en todas las instancias.
- Basta una referencia a la clase o una instancia para usarlos.
- Se definen fuera del inicializador `__init__`.
- No se les antepone un `self`.

Atributos de instancia y de clase

```
class Planta:
    id_max = 0 # Atributo de clase
    def __init__(self, nombre: str) -> None:
        self.nombre = nombre
        self.id = Planta.id_max # Atributo de instancia
        Planta.id_max += 1      # Modificamos el atributo de clase

menta = Planta("Menta")
menta.id # 0
menta.id_max # 1

rosa = Planta("Rosa")
rosa.id # 1
rosa.id_max # 2
```

Atributos de instancia y de clase

Entendamos mejor cómo funcionan los atributos de instancia y de clase con el siguiente ejemplo:

```
class SerVivo:

    planeta_origen = 'Tierra' # Atributo de clase

    def __init__(self, tipo:str) -> None:
        self.tipo = tipo
```

Atributos de instancia y de clase

```
ser_vivo_1 = SerVivo('Terrícola')
ser_vivo_2 = SerVivo('?')
print('Valores iniciales:')
print(f'\tser_vivo_1 tipo = {ser_vivo_1.tipo}')
print(f'\tser_vivo_1 planeta_origen = {ser_vivo_1.planeta_origen}')
print(f'\tser_vivo_2 tipo = {ser_vivo_2.tipo}')
print(f'\tser_vivo_2 planeta_origen = {ser_vivo_2.planeta_origen}')
```

```
Valores iniciales:
    ser_vivo_1 tipo = Terrícola
    ser_vivo_1 planeta_origen = Tierra
    ser_vivo_2 tipo = ?
    ser_vivo_2 planeta_origen = Tierra
```

Atributos de instancia y de clase

```
# Realizamos cambios solo sobre ser_vivo_1
# Recordemos que: ser_vivo_2 = SerVivo('')

# Modificamos ATTR CLS a través de Instancia
ser_vivo_1.planeta_origen = 'Venus'
ser_vivo_1.tipo = 'Venusiano'

# Modificamos ATTR CLS a través de CLS
SerVivo.planeta_origen = 'Marte'
```

Valores después de actualizaciones:

```
ser_vivo_1 tipo = Venusiano
ser_vivo_1 planeta_origen = Venus
ser_vivo_2 tipo = ?
ser_vivo_2 planeta_origen = 
```

**¿Cuál será el planeta de origen
de ser_vivo_2?**



Atributos de instancia y de clase

```
# Realizamos cambios solo sobre ser_vivo_1
# Recordemos que: ser_vivo_2 = SerVivo('?')

# Modificamos ATTR CLS a través de Instancia
ser_vivo_1.planeta_origen = 'Venus'
ser_vivo_1.tipo = 'Venusiano'

# Modificamos ATTR CLS a través de CLS
SerVivo.planeta_origen = 'Marte'
```

Valores después de actualizaciones:

```
ser_vivo_1 tipo = Venusiano
ser_vivo_1 planeta_origen = Venus
ser_vivo_2 tipo = ?
ser_vivo_2 planeta_origen = Marte
```

¡¡ES MARTE!!

Atributos de instancia y de clase

```
# Modificamos ATTR CLS a través de Instancia
ser_vivo_1.planeta_origen = 'Venus'
ser_vivo_1.tipo = 'Venusiano'

# Modificamos ATTR CLS a través de CLS
SerVivo.planeta_origen = 'Marte'

# Instanciamos un nuevo Ser Vivo
ser_vivo_3 = SerVivo('Marciano')
```

Valores después de actualizaciones:

```
ser_vivo_1 tipo = Venusiano
ser_vivo_1 planeta_origen = Venus
ser_vivo_2 tipo = ?
ser_vivo_2 planeta_origen = Marte
ser_vivo_3 tipo = Marciano
ser_vivo_3 planeta_origen = 
```

**¿Cuál será el planeta de origen
del NUEVO ser vivo?**



Atributos de instancia y de clase

```
# Modificamos ATTR CLS a través de Instancia
ser_vivo_1.planeta_origen = 'Venus'
ser_vivo_1.tipo = 'Venusiano'

# Modificamos ATTR CLS a través de CLS
SerVivo.planeta_origen = 'Marte'

# Instanciamos un nuevo Ser Vivo
ser_vivo_3 = SerVivo('Marciano')
```

Valores después de actualizaciones:

```
ser_vivo_1 tipo = Venusiano
ser_vivo_1 planeta_origen = Venus
ser_vivo_2 tipo = ?
ser_vivo_2 planeta_origen = Marte
ser_vivo_3 tipo = Marciano
ser_vivo_3 planeta_origen = Marte
```

¡¡ES MARTE TAMBIÉN!!

Properties

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

Properties: ¿Cómo funcionan?

```
class Planta:
    def __init__(self, nombre: str) -> None:
        self._nombre = nombre
        self._calidad = 'bueno'

    @property
    def calidad(self) -> str:
        return self._calidad

    @calidad.setter
    def calidad(self, nueva_calidad: str) -> None:
        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: str) -> None:
        self._nombre = nombre
        self._calidad = 'bueno'

    @property
    def nombre(self) -> str:
        return self._nombre

    @nombre.setter
    def nombre(self, nuevo_nombre: str) -> None:
        print(f'Soy un {self.nombre} y nunca seré un {nuevo_nombre}.')


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

Soy un Zapallo y nunca seré un Tomate.

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**

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?
 ¿MaestroTierra?  ¿MaestroViento?

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 ☁")
```

Herencia: Atributos de instancia y de clase

```
class SerVivo:
    planeta_origen = 'Marte' # Atributo de clase
    def __init__(self, tipo:str) -> None:
        self.tipo = tipo
```

```
class UltraPlanta(SerVivo):
    id_max = 0 # Atributo de clase

    def __init__(self, nombre:str, tipo:str = 'Terrícola') -> None:
        super().__init__(tipo)
        self.nombre = nombre
        # Atributo de instancia
        self.id = UltraPlanta.id_max
        # Modificamos el atributo de clase (UltraPlanta)
        UltraPlanta.id_max += 1
        if SerVivo.planeta_origen == 'Marte':
            # Modificamos atributo de la clase Madre (SerVivo)
            SerVivo.planeta_origen = 'Tierra'
```

Herencia: Atributos de instancia y de clase

```
ultra_planta = UltraPlanta('Spike')
print(f'ultra_planta.tipo = {ultra_planta.tipo}')
print(f'ultra_planta.planeta_origen = {ultra_planta.planeta_origen}')
print(f'ultra_planta.nombre = {ultra_planta.nombre}')

ser_vivo_4 = SerVivo('MegaTerricola')
print(f'ser_vivo_4.tipo = {ser_vivo_4.tipo}')
print(f'vivo_4.planeta_origen = {ser_vivo_4.planeta_origen}')
```

```
ultra_planta.tipo = Terricola
ultra_planta.planeta_origen = Tierra
ultra_planta.nombre = Spike
ser_vivo_4.tipo = MegaTerricola
ser_vivo_4.planeta_origen = Tierra
```

Polimorfismo

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

Polimorfismo - *Overriding*

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?
 ¿MaestroTierra?  ¿MaestroViento?

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

Polimorfismo - *Overriding*

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?
 ¿MaestroTierra?  ¿MaestroViento?

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


Polimorfismo - *Overriding*

```
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 🔥")
```

Polimorfismo - *Overloading*

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?

 ¿MaestroTierra?  ¿MaestroViento?

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

¿Qué acción puede variar según los datos que me lleguen? **Luchar**

Polimorfismo - *Overloading*

```
class Persona:
    def luchar(self):
        print("No hay enemigos 🙄")

    def luchar(self, enemigo: Persona):
        print("Puedo ganarle a otra persona")

    def luchar(self, enemigos: list[Persona]):
        if len(enemigos) < 2:
            print("Puedo ganar 😁")
        else:
            print("Nop, son muchos enemigos")
```


En Python, el último método definido es el que será considerado.

En otros lenguajes que si tienen implementado *overloading*, se ejecutará el método correspondiente según los argumentos que llegan.

Multiherencia

Una clase puede heredar de más de una superclase

Multiherencia

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?
 ¿MaestroTierra?  ¿MaestroViento?

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

¿Qué acción puede variar según los datos que me lleguen? **Luchar**

¿Y si alguien controla 2 elementos?

Multiherencia

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!**

💧 ¿Maestr

🌿 ¿Maestr

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

¿Qué acción puede variar según los datos que me llegu

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



Multiherencia

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

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

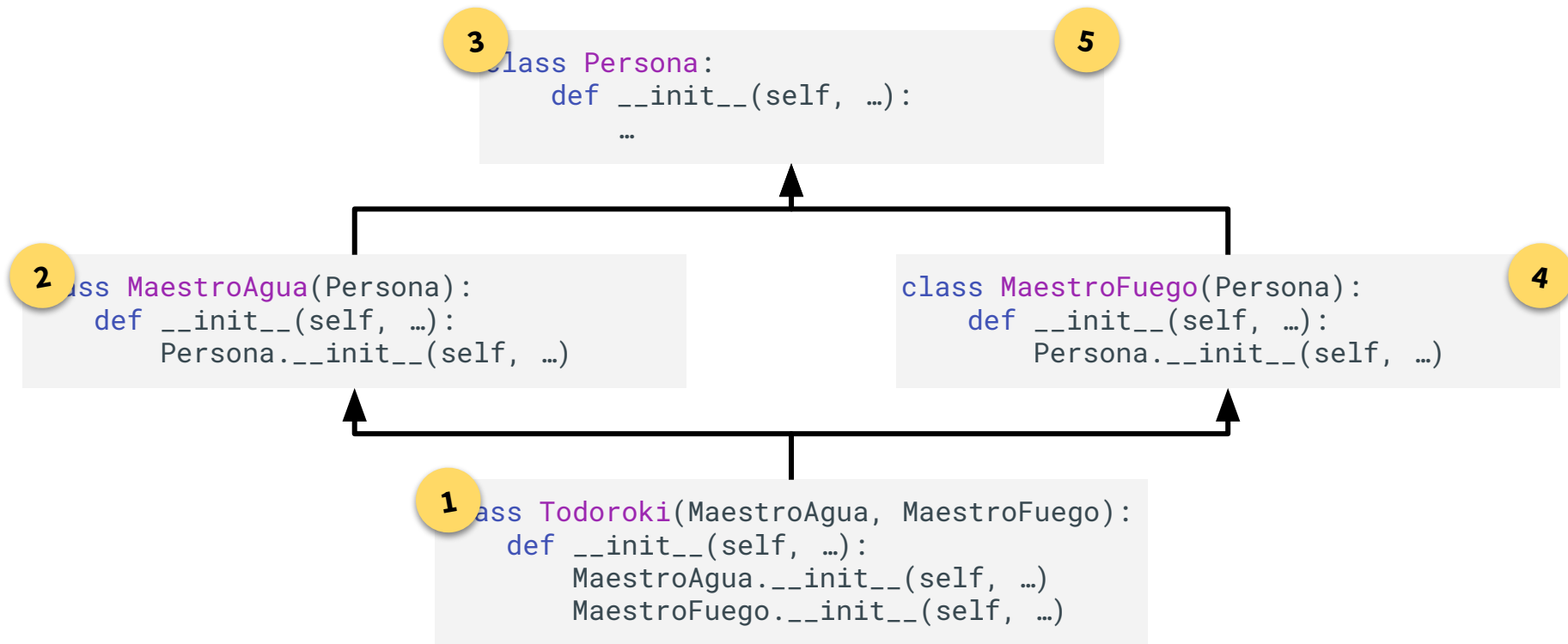
```
class MaestroAgua(  
    def __init__(s  
        Persona.__
```

```
sona):  
    ...):  
    __init__(self, ...)
```

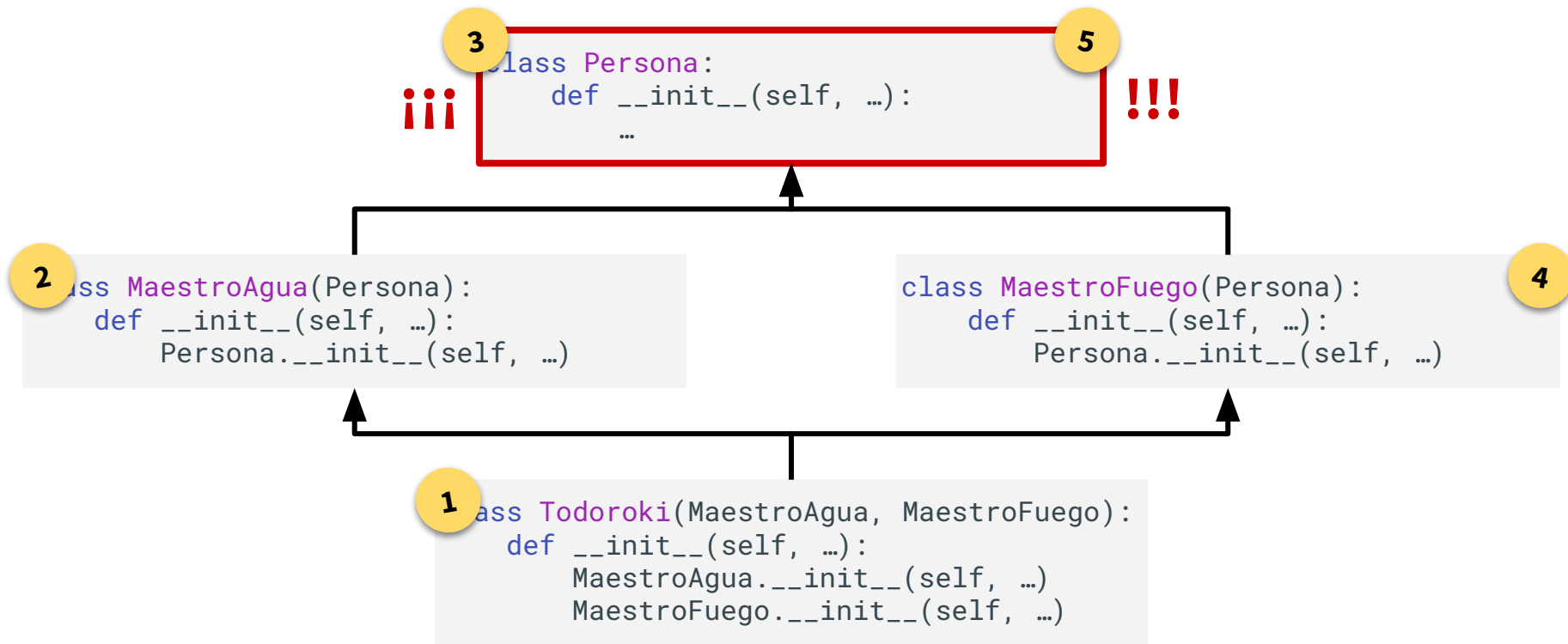
Tendremos el problema del diamante 💎
¿Por qué? 🤔

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

Multiherencia



Multiherencia



Multiherencia

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

```
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

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

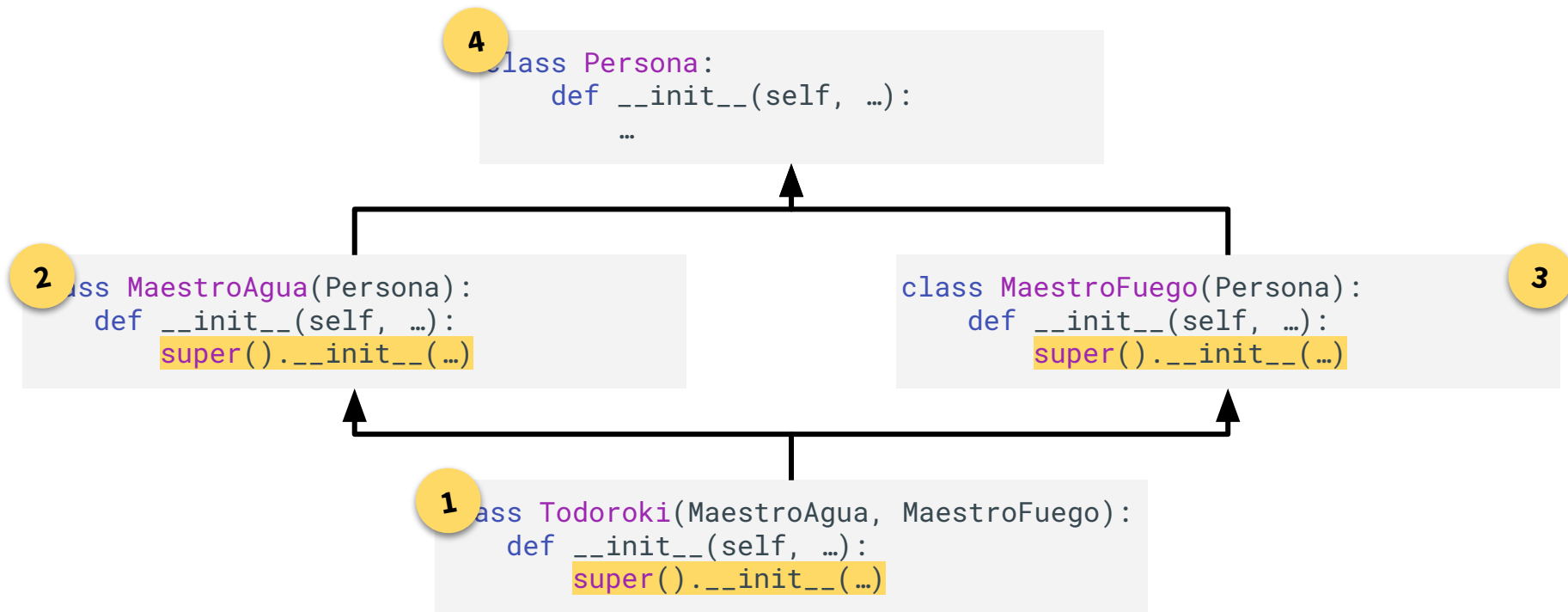
```
class MaestroAgua(  
    def __init__(s  
        super().__init__(...)
```

La solución sería utilizar `super()`
¿Por qué? 🤔

```
Persona):  
    ...):  
    __init__(...)
```

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

Multiherencia



Multi

PERFECTLY BALANCED

2

ass Ma
def

3

AS ALL THINGS SHOULD BE

Multiherencia

Pero, ¿cómo paso diferentes argumentos a mis padres solo con un super? 🤔

```
class MaestroAgua(Persona):  
    def __init__(self, curar):  
        self.puede_curar = curar
```

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

```
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, puede_curar, controlar_rayos):  
        super().__init__(puede_curar, controla_rayos)
```

Multiherencia

Pero, ¿cómo paso diferentes argumentos a mis padres solo con un super? 🤔

```
class MaestroAgua(  
    def __init__(s  
        self.puede
```

Nos saldrá un error
¿Por qué? 🤔

```
sona):  
    rayos):  
    rayos = rayos
```

```
class Todotoki(MaestroAgua, MaestroFuego):  
    def __init__(self, puede_curar, controlar_rayos):  
        super().__init__(puede_curar, controla_rayos)
```

Multiherencia: operadores * y **

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

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

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

```
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, puede_curar, controlar_rayos):  
        super().__init__(curar=puede_curar, rayos=controlar_rayos)
```

Multiherencia: operadores * y **

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

```
class MaestroAgua(Persona):  
    def __init__(self, curar, *args, **kwargs):  
        super().__init__(  
            self.puede
```

¿Cómo funciona todo esto?

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

```
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, puede_curar, controlar_rayos):  
        super().__init__(curar=puede_curar, rayos=controlar_rayos)
```


Multiherencia: operadores * y **


Explicación paso a paso

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

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

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

```
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, puede_curar, controlar_rayos):  
        super().__init__(curar=puede_curar, rayos=controlar_rayos)
```



Multiherencia: operadores * y **

Explicación paso a paso

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

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


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

```
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, puede_curar, controlar_rayos):  
        super().__init__(curar=puede_curar, rayos=controlar_rayos)
```

Multiherencia: operadores * y **

Explicación paso a paso

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



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

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


```
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, puede_curar, controlar_rayos):  
        super().__init__(curar=puede_curar, rayos=controla_rayos)
```

Multiherencia: operadores * y **

Explicación paso a paso

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

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



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

```
class Todoroki(MaestroAgua, MaestroFuego):  
    def __init__(self, puede_curar, controlar_rayos):  
        super().__init__(curar=puede_curar, rayos=controlar_rayos)
```

Ejemplo de consumo de `*args` y `**kwargs`

Junto a la presentación, les hemos subido unos un *jupyter notebook* donde pueden apreciar el comportamiento que tienen los `*args` y `**kwargs` cuando se produce el problemas del diamante.

Código complementario

Multiherencia: Reflexión

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

- No , depende de cada caso.

Si es que el problema del diamante provoca que un método se ejecute 2 o más veces:

- 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.

Programación Avanzada

IIC2233 2024-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Lucas Van Sint Jan - Francisca Cattán

