

Programación Avanzada

IIC2233 2024-1

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Dante Pinto - Francisca Cattán



Anuncios

1. Hoy es la Actividad 1, el puntaje se calculará con test privados.
 2. El martes 09 se publica la Tarea 2.
 3. El viernes de la próxima semana es el *Midterm*.
 4. La ECA se encuentra disponible para responder de domingo a martes.
-

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, resistencia):
        self.nombre = nombre          # Zapallo
        self.agua = 0
        self.resistencia = resistencia # 50 unidades

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

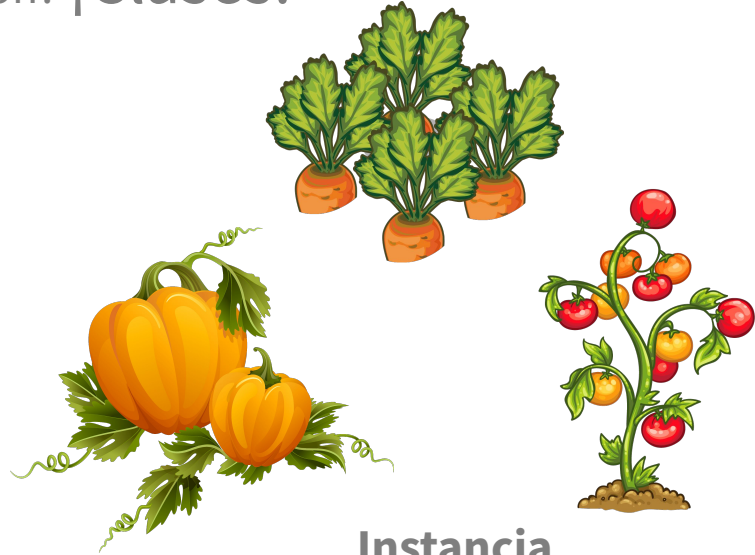
    def regar(self, cantidad):
        self.agua += cantidad
        if self.agua >= self.resistencia:
            self.agua = self.resistencia
```

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):
        self.plantas = []

    def plantar(self, planta):
        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):
        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
```

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')
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 ☁")
```

Polimorfismo

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

Polimorfismo

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

```
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

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**

¿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

¿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.__
```

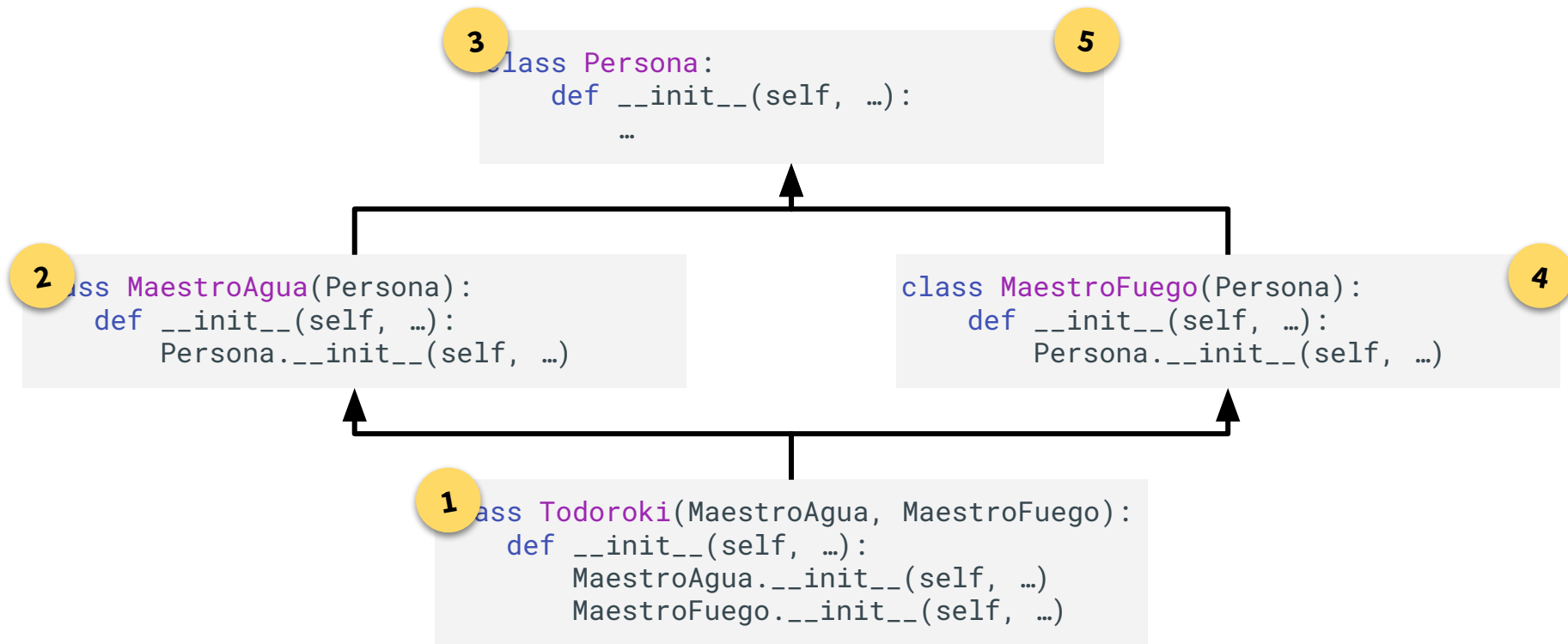
```
sona):  
    ...):  
    __init__(self, ...)
```

Tendremos el problema del diamante 💎

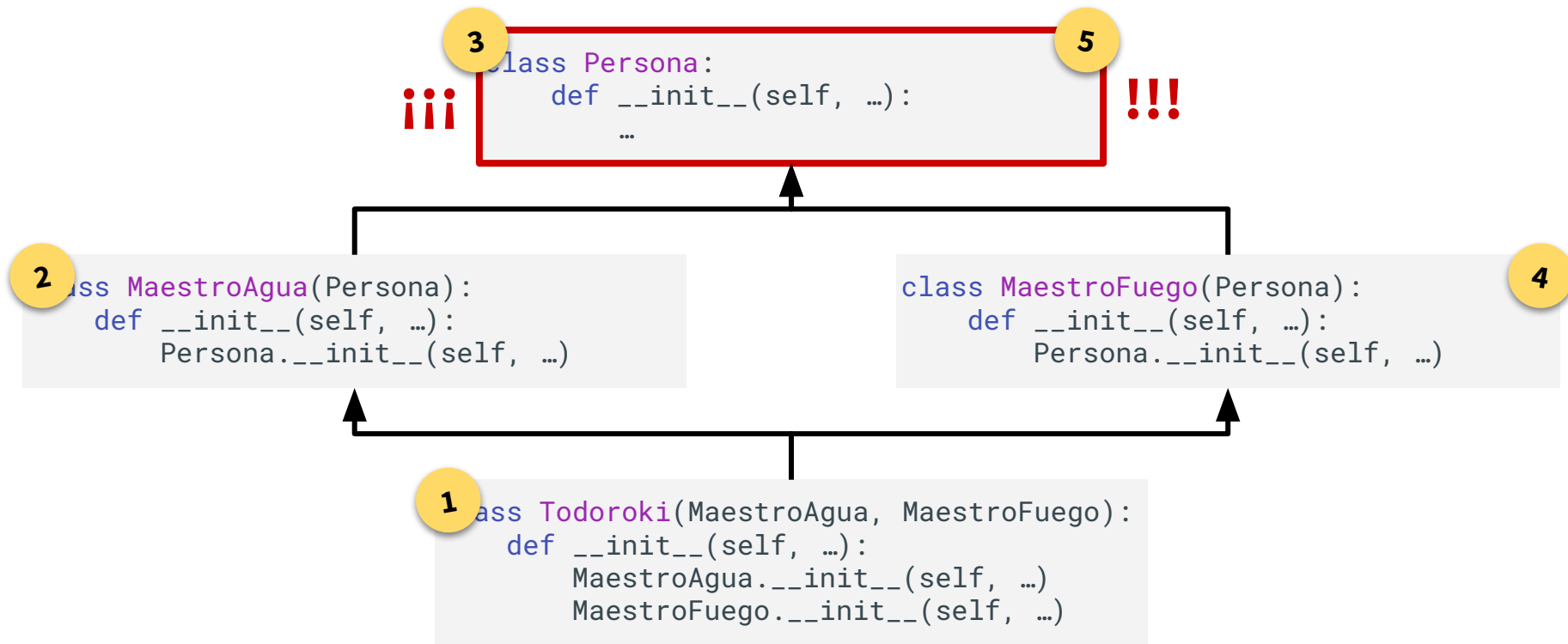
¿Por qué? 🤔

```
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, ...):
```

La solución sería utilizar `super()`

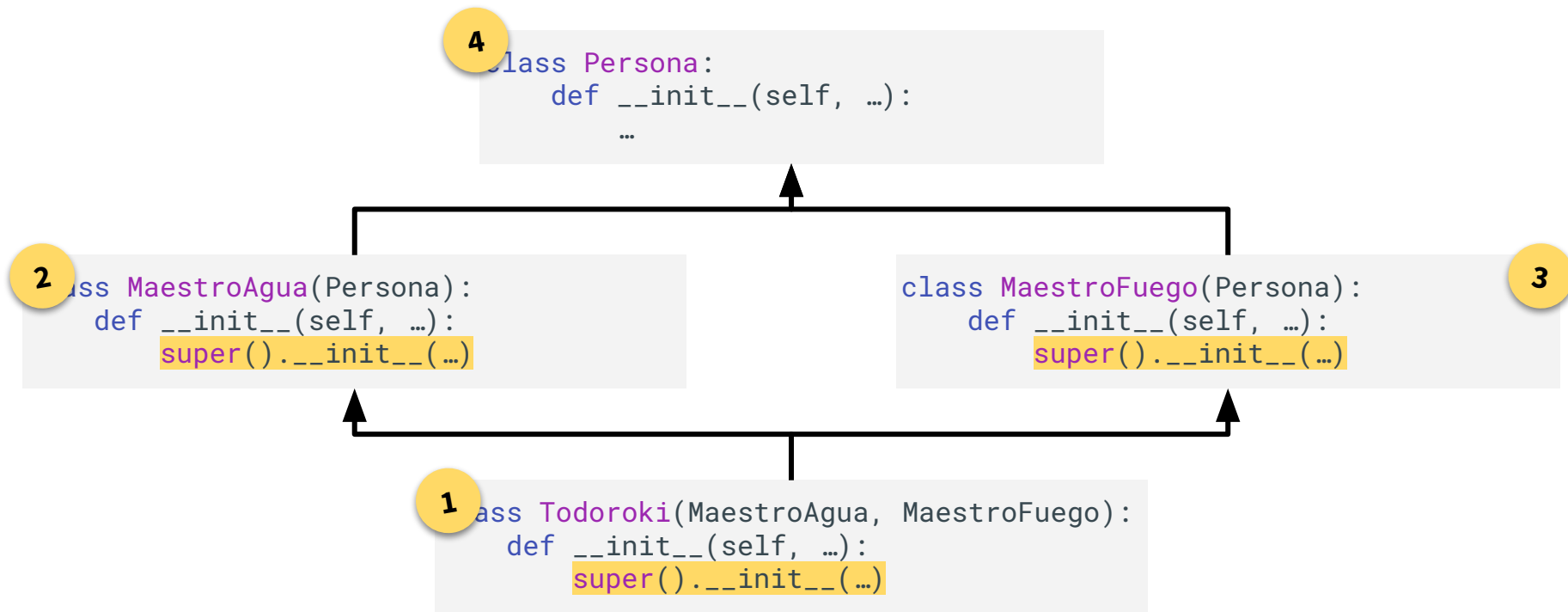
¿Por qué? 🤔

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

```
Persona):  
    def __init__(...):  
        super().__init__(...)
```

```
class Todopoki(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? 🤔

Nos saldrá un error

¿Por qué? 🤔

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

```
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)
```

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

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**

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

Oye, pero... ¿Cómo fuerza que todos deban entrenar?

Clase abstracta

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**

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

Oye, pero... ¿Cómo fuerzo que todos deban entrenar? **Clases abstractas**

Clase abstracta

```
from abc import ABC, abstractmethod

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

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

Actividad 1

—

Primeros pasos de la Actividad

1. Actualizar repositorio.

1.1. Si aún no clonan: `git clone https://github.com/IIC2233/syllabus.git`

1.2. Si ya clonaron: `git pull`

2. Vayan a la carpeta “Actividades”, luego “AC1”, copien el contenido **a su repo personal**.

3. Desarrollen la Actividad **en su repo personal**.

4. Recuerden subir su commit al terminar:

```
git add Actividades/AC1/clases.py
git commit -m 'AC1 subida'
git push
```


Programación Avanzada

IIC2233 2024-1

Hernán Valdivieso - Daniela Concha -

- Dante Pinto - Francisca Cattán

