



# Herencia y polimorfismo

Herencia y polimorfismo

***Utilizar el concepto de herencia para la resolución de un problema de baja complejidad acorde al lenguaje Python.***

***Representar un problema de orientación de objetos mediante un diagrama de clases para su implementación en Python.***

- Unidad 1:  
Introducción a la programación orientada a objetos con Python
- Unidad 2:  
Tipos de métodos e interacciones entre objetos
- Unidad 3:  
Herencia y polimorfismo
- Unidad 4:  
Manejo de errores y archivos



Te encuentras aquí



## ¿Qué aprenderás en esta sesión?

- *Describe el concepto de polimorfismo bajo el paradigma de orientación a objetos para resolver un problema.*
- *Utiliza herencia de clases como medio de implementación de polimorfismo para resolver un problema.*

# ¿Cómo interactúan las clases?



***/\* Herencia \*/***

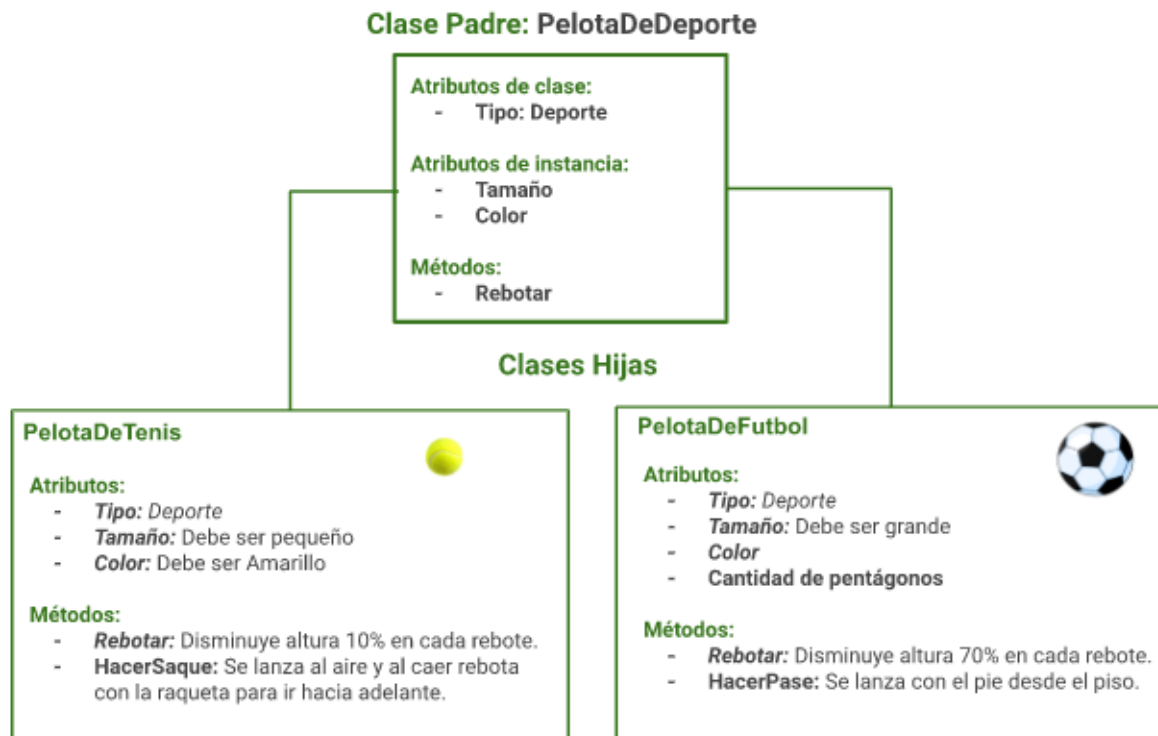
# ¿Qué es herencia?

Mecanismo que permite derivar una clase, para crear una jerarquía de clases que comparten los mismos atributos y métodos.

- La clase padre define un conjunto de atributos y métodos que describen un comportamiento común del conjunto de clases hijas que le heredan.
- Las clases hijas heredan tanto los atributos como el comportamiento de la clase padre, pudiendo, además, añadir otros atributos y otros comportamientos.
- Los comportamientos (métodos) heredados pueden ser reescritos para modificar la lógica según se requiera en la clase hija, esto último se verá más adelante.

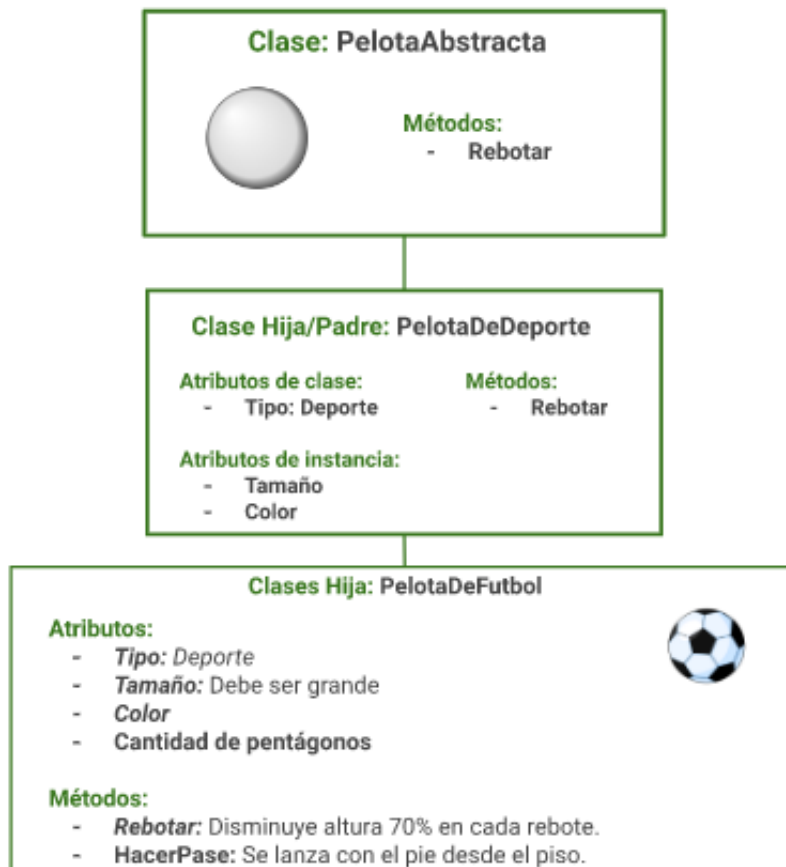
# ¿Qué es herencia?

## Ejemplo



# ¿Qué es herencia?

## Ejemplo





# Herencia simple

Consiste en que una clase hija hereda solamente desde una clase padre, por lo que solamente hay dos clases involucradas en la relación.

- Se debe definir una clase cualquiera, la cual será la clase padre y la herencia se creará en el momento en que otra clase (hija) la use como argumento en su definición.
- De esta forma, las instancias de la clase hija poseerán tanto los atributos como los métodos de la clase padre.

## Clase Padre: PelotaDePlastico

### Atributos de clase:

- Tipo: Plástico
- Duración: Corta

### Atributos de instancia:

- Tamaño
- Color

### Métodos:

- **Rebotar:** Disminuye altura 50% en cada rebote.

## Clase Hija: PelotaDeJuguete

### Atributos:

- *Tipo:* Plástico
- *Duración:* Corta
- *Tamaño*
- *Color*

### Métodos:

- *Rebotar*



# Herencia simple

## Ejemplo

```
class PelotaDeDeporte():
    def __init__(self, color: str):
        self.__color = color

    @property
    def color(self) -> str:
        return self.__color

class PelotaDeFutbol(PelotaDeDeporte):
    def mostrar_color(self):
        print(f"Mi color es {self.color}")

pdf = PelotaDeFutbol("Blanco y Negro")

# Salida: Mi color es Blanco y Negro
pdf.mostrar_color()
```

# Herencia múltiple

Consiste en que una clase hija hereda más de una clase padre, es decir, la clase hija poseerá todos los atributos y métodos de todas las clases heredadas.

- Para que una clase herede desde más de una clase padre, se debe incluir todas las clases desde las cuales se desee heredar, como argumentos en la definición de la clase hija.
- Si un atributo o método se encuentra en más de una de las clases heredadas, se considerará los definidos en la primera clase heredada de derecha a izquierda.

**Clase Padre: PelotaDeDeporte**

**Atributos de clase:**

- Tipo: Deporte

**Métodos:**

- Rebotar

**Clase Padre: PelotaDePlastico**

**Atributos de clase:**

- Tipo: Plástico
- Duración: Corta

**Métodos:**

- **Rebotar:** Disminuye altura 50% en cada rebote.

**Clase Hija: PelotaDePingPong**

**Atributos:**

- Tipo: Deporte
- Duración: Corta

**Métodos:**

- **Rebotar:** Disminuye altura un 10% en cada rebote



# Herencia múltiple

## Ejemplo

```
class PelotaDeDeporte():  
    tipo = "Deporte"  
  
class PelotaDePlastico():  
    tipo = "Plástico"  
  
class PelotaDePingPong(PelotaDeDeporte, PelotaDePlastico):  
    pass  
  
# Salida: "Deporte"  
print(PelotaDePingPong.tipo)
```

**Cuando se utilizan  
varios tipos de herencia  
a la vez, se habla  
de herencia híbrida.**



La herencia múltiple es una característica que no está disponible en todos los lenguajes que utilizan el paradigma orientado a objetos, pero Python es uno de los que sí permite utilizarla.



Si se heredan varias clases y varias de ellas tienen constructores, al crear una instancia de la clase hija se ejecutará solamente el constructor de la primera clase heredada (de izquierda a derecha) que tenga un constructor.



# Ejercicio guiado

## "Herencia simple y múltiple"





# Herencia simple y múltiple

Desde un emprendimiento, se te ha solicitado comenzar el diseño de la estructura de clases para sus productos. Por ahora, se considera solamente el caso de los chocolates, donde existen, por el momento, solo los chocolates amargos (en el futuro habrá más tipos específicos, pero considera que no pueden existir chocolates “a secas”). Todos los chocolates deben tener un porcentaje de cacao específico, de acuerdo a su tipo, siendo el de los chocolates amargos entre 75% y 85%.

A su vez, también existe una variedad de chocolate amargo sin gluten. Se debe considerar que existirán en el futuro otros productos, además de los chocolates, que también serán sin gluten.

*Recuerda que una clase abstracta puede implementar, además de al menos 1 método abstracto, métodos no abstractos. Es decir, si la clase abstracta define un constructor con los atributos de instancia, las clases que implementan la clase abstracta (que la hereden), también poseerán dicho constructor y atributos de instancia.*



# Herencia simple y múltiple

## *Solución*

### Paso 1

En un archivo **sin\_gluten.py**, definir la clase **SinGluten**, la cual contiene solamente un atributo de clase **tipo\_producto** con el valor “Sin Gluten”.

```
class SinGluten():  
    tipo_producto = "Sin Gluten"
```



# Herencia simple y múltiple

## Solución

### Paso 2

En un archivo **chocolate.py**, importar la clase **SinGluten**, además de la clase **ABC** y el decorador **abstractmethod** desde **abc**. A continuación, crear la clase abstracta **Chocolate**.

```
from sin_gluten import SinGluten
from abc import ABC, abstractmethod

class Chocolate(ABC):
```



# Herencia simple y múltiple

## Solución

### Paso 3

Dentro de esta clase, definir el método abstracto **validar\_porc\_cacao**, que recibe un número flotante por parámetro.

```
@abstractmethod
def validar_porc_cacao(self, porc: float) -> float:
    pass
```



# Herencia simple y múltiple

## Solución

### Paso 4

Dentro de la misma clase, definir el constructor de la clase, el cual asigna un valor entregado por parámetro al atributo **porc\_cacao**. Este valor debe estar corregido mediante el método **validar\_porc\_cacao**.

```
def __init__(self, porc_cacao: float):  
    self.porc_cacao = self.validar_porc_cacao(porc_cacao)
```



# Herencia simple y múltiple

## Solución

### Paso 5

A continuación, en el mismo archivo, se crea la clase **ChocolateAmargo**, la cual hereda la clase **Chocolate**. Se define un método **validar\_porc\_cacao**, donde; si el valor entregado es menor a 0.75, se retorna 0.75; si el valor entregado es mayor a 0.85, se retorna 0.85. En cualquier otro caso, se retorna el valor entregado.

```
class ChocolateAmargo(Chocolate):  
    def validar_porc_cacao(self, porc: float):  
        return min(max(0.75, porc), 0.85)
```



# Herencia simple y múltiple

## Solución

### Paso 6

Finalmente, en el mismo archivo, se crea la clase **ChocolateAmargoSinGluten**, la cual hereda tanto de **ChocolateAmargo** como de **SinGluten**.

```
class ChocolateAmargoSinGluten(ChocolateAmargo, SinGluten):  
    pass
```

Ejemplo de instancia de chocolate amargo:

```
c = ChocolateAmargo(0.3)  
  
# Salida: 0.75  
print(c.porc_cacao)
```




**/\* Polimorfismo \*/**



# ¿Qué es el Polimorfismo?

En la vida real, el polimorfismo se refiere a la capacidad de “tener varias formas”.

En programación orientada a objetos, el polimorfismo se refiere a este mismo concepto, considerando que la “forma” corresponde al **comportamiento de un objeto**, es decir, el polimorfismo se refiere a que el comportamiento del objeto cambia, según la clase de origen del objeto desde el cual se está llevando a cabo.

**PelotaDeJuguete**

**Métodos:**

- **Rebotar:** Disminuye altura **50%** en cada rebote.

**PelotaDeFutbol**

**Métodos:**

- **Rebotar:** Disminuye altura **70%** en cada rebote.

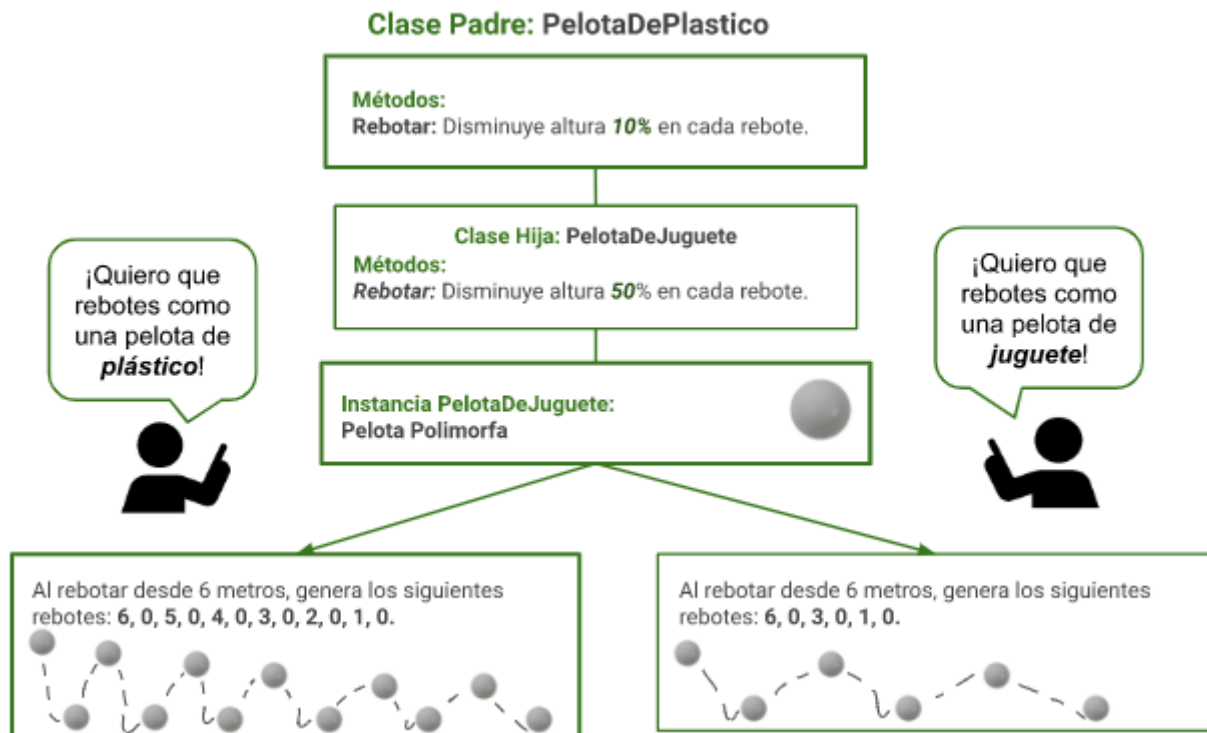
# Heredar una clase y aplicar polimorfismo

El polimorfismo también es posible de implementar en una **jerarquía de clases**. En ella, la clase padre actúa como interfaz de un conjunto de clases, donde cada **clase hija** define una forma específica de un método definido previamente en su clase padre.

Supongamos un caso de herencia simple entre una **clase padre** y una **clase hija**, donde ambas han definido un mismo método. Al crearse una instancia de la clase hija, y realizar en ella un llamado al método definido en ambas clases, el método que se ejecutará será el definido en la clase hija.

# Heredar una clase y aplicar polimorfismo

## Ejemplo



# Polimorfismo en una clase heredada

En Python, para hacer uso del comportamiento de la clase padre desde una clase hija, se debe hacer uso del método **built-in super()**.

Este recibe como primer argumento el tipo de la clase hija, y como segundo argumento una instancia de la clase hija. Luego, al llamado de **super()** se puede concatenar el llamado del método que se desea utilizar desde la clase padre.

**{desafío}**  
**latam\_**

```
class PelotaDePlastico() -> None:
    def __init__(self):
        self.rebotes = []
    def rebotar(self, altura):
        self.rebotes = []
        while altura > 0:
            self.rebotes += [int(altura), 0]
            altura //= 1.1
class PelotaDeJuguete(PelotaDePlastico) -> None:
    def rebotar(self, altura):
        self.rebotes = []
        while altura > 0:
            self.rebotes += [altura, 0]
            altura //= 2
pdj = PelotaDeJuguete()
pdj.rebotar(5)
# Salida: [5, 0, 1, 0]
print(pdj.rebotes)
# Se hace llamado al método del padre PelotaDePlastico
super(type(pdj), pdj).rebotar(5)
# Salida: [5, 0, 2, 0, 1, 0]
print(pdj.rebotes)
```

Ejercicio guiado

"Polimorfismo en subclases"



# Polimorfismo en subclases

Desde la empresa “Juegos por comida”, te han solicitado programar el prototipo de una batalla entre un jugador y un monstruo. Para ello, debes considerar que tanto los jugadores como los monstruos poseen puntos de vida (HP), puntos de ataque (ATK) y puntos de defensa (DF), y opcionalmente un arma (solo especificar armas al crear jugadores), que son asignados al momento de crearse. Además, tanto los jugadores como los monstruos pueden generar ataques y defenderse.

Para esta demostración, se le solicita generar un script `demo.py` que genere un jugador con 500 de HP, 10 de ATK, 5 de DF y una espada. El jugador debe enfrentarse al monstruo “Bégimo”, el cual tiene 1.000 de HP, 1 de ATK y 8 de DF. Ambos deben enfrentarse alternadamente en turnos de ataque-defensa, hasta que alguno de los dos muera (tenga HP igual o menor a 0).



# Polimorfismo en subclases

*Para el enfrentamiento entre ambos, debe considerar lo siguiente:*

Los ataques generan un puntaje de ataque (número entero).

- En el caso de los jugadores que tienen un arma, se debe retornar la cantidad de puntos de ATK más un número al azar entre 1 y 5, en caso contrario solo se retorna los puntos de ATK.
- En el caso de los monstruos, debe retornar los puntos de ATK más el 1% del HP actual (retorna un número entero).

La acción de defensa recibe un ataque (número entero) y disminuye el HP.

- En el caso de los jugadores, al ataque recibido se le debe restar un número al azar entre 1 y los puntos de DF, y el resultado de ello (forzar a ser un número entero mayor a 0) se debe restar al HP.
- En el caso de los monstruos, al ataque recibido se le debe restar los puntos de DF y el 0.1% del HP actual, y el resultado de ello (forzar a ser un número entero mayor a 0) se debe restar al HP.



# Polimorfismo en subclases

## Solución

Independiente de quien inicie el ataque, el oponente debe defenderse del ataque recibido, y luego atacar de vuelta al atacante (en caso de que aún tenga HP), quien a su vez se defenderá de este ataque y luego volverá a atacar (en caso de que aún tenga HP), y así sucesivamente hasta que alguno de los dos muera (su HP sea menor o igual a 0).

### Aspectos técnicos

Se solicita que tanto Jugador como Monstruo hereden de una clase común Personaje, con el fin de reutilizar el constructor desde aquella clase. Debe considerar que no pueden existir instancias de Personaje, y que todas las clases que hereden de ella deben implementar su propio método ataque y defensa. Por el momento, no se debe aplicar encapsulamiento de los atributos.

*Si lo deseas, puedes agregar `print()` donde estimes conveniente para visualizar los resultados de cada ataque (Puedes usar `instancia.__class__.__name__`).*





# Polimorfismo en subclases

## Solución

### Paso 1

En un archivo personaje.py, crear la clase abstracta Personaje con un constructor que recibe por parámetro hp, atk, df, y arma, y los asigna en los respectivos atributos de instancia. Ésta será clase padre tanto de Jugador como de Monstruo.

```
from abc import ABC, abstractmethod
class Personaje(ABC):
    def __init__(self, hp: int, atk: int, df: int, arma: str="") -> None:
        self.hp = hp
        self.atk = atk
        self.df = df
        self.arma = arma
```



# Polimorfismo en subclases

## Solución

### Paso 2

A continuación, en la misma clase, definir los métodos abstractos ataque y defensa.

```
@abstractmethod
def ataque(self) -> int:
    pass

@abstractmethod
def defensa(self, ataque: int) -> None:
    pass
```



# Polimorfismo en subclases

## Solución

### Paso 3

En un archivo llamado jugador.py, importar random y Personaje, y definir la clase Jugador que herede de Personaje.

```
import random
from personaje import Personaje

class Jugador(Personaje):
```



# Polimorfismo en subclases

## Solución

### Paso 4

A continuación, en el mismo archivo, definir el método ataque. Retornar el ataque de la instancia más un número al azar entre 1 y 5 en caso de que exista un arma, o solo el ataque de la instancia en caso contrario.

```
def ataque(self) -> int:  
    return (self.ataque + random.randint(1, 5)  
            if self.arma else self.ataque)
```



# Polimorfismo en subclases

## Solución

### Paso 5

A continuación, dentro de la misma clase, definir el método defensa, el cual recibe como argumento un ataque. Dentro del método, al hp de la instancia se debe asignar el hp actual menos la resta entre el ataque recibido y un número al azar entre 1 y la defensa de la instancia (en caso de que sea un número positivo).

```
def defensa(self, ataque: int) -> None:  
    self.hp -= max(ataque - random.randint(1, self.df), 0)
```



# Polimorfismo en subclases

## Solución

### Paso 6

En otro archivo llamado monstruo.py, importar Personaje y definir la clase Monstruo heredando Personaje.

```
from personaje import Personaje
class Monstruo(Personaje):
```

### Paso 7

A continuación, dentro de la misma clase, definir el método ataque. Retornar el ataque de la instancia más el 1% del hp de la instancia (como número entero).

```
def ataque(self) -> int:
    return self.atk + int(self.hp * 0.01)
```



# Polimorfismo en subclases

## Solución

### Paso 8

A continuación, dentro de la misma clase, definir el método defensa, el cual recibe por parámetro un ataque. El método asigna al hp de la instancia el valor actual menos la resta entre el ataque y la suma entre la defensa de la instancia y el 0.1% del hp actual (como número entero, en caso que sea mayor a 0).

```
def defensa(self, ataque: int) -> None:  
    self.hp -= max(ataque - (self.df + int(self.hp*0.001)), 0)
```

# Polimorfismo en subclases

## Solución

### Paso 9

Finalmente, en un archivo llamado demo.py, importe las clases Jugador y Monstruo

```
from jugador import Jugador
from monstruo import Monstruo
```

### Paso 10

En el mismo archivo anterior, genera el jugador y el monstruo solicitados dentro de una lista. Definir además una variable auxiliar con valor 0 (esta variable almacenará el ataque

```
enfrentados = [Jugador(500, 10, 5, "espada"), Monstruo(1000, 1, 8)]
atk = 0
```





# Polimorfismo en subclases

## Solución

### Paso 11

Iniciar un ciclo while que se ejecutará mientras no exista elementos dentro de la lista enfrentados con HP igual o menor a 0. Dentro del while, iniciar otro ciclo for que recorra la lista de enfrentados. Dentro del ciclo for, ejecutar la defensa del elemento iterado, en caso de que exista un valor para la variable auxiliar atk.

```
while any(e.hp <= 0 for e in enfrentados) == False:
    for e in enfrentados:
        if atk:
            e.defensa(atk)
```



# Polimorfismo en subclases

## Solución

### Paso 12

A continuación, dentro del ciclo for, en caso de que el elemento iterado aún tenga HP, generar un ataque, que se almacenará en la variable auxiliar atk.

```
if e.hp > 0:  
    atk = e.ataque()  
else:  
    print(f";Oh no!, el {e.__class__.__name__} ha muerto :(")
```



# ¿Qué es la herencia?



¿Cómo se debe heredar una  
clase en Python?



¿Cuál es la ventaja del  
Polimorfismo en la  
programación orientada  
a objetos?



En Python, si una clase hija ha sobrescrito un método de una clase padre, ¿cómo puede una instancia de la clase hija hacer uso del método original de su clase padre?





## Próxima sesión...

- *Codifica un programa utilizando herencia y sobreescritura de métodos para resolver un problema.*

**{desafío}**  
**latam\_**

*Academia de  
talentos digitales*

