Herencia y polimorfismo

Sobreescritura de métodos

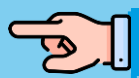
{desafío} latam_

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.

{desafío} latam_

- 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?

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

¿Qué es herencia y polimorfismo?

{desafío} latam_

¿Qué entendemos por sobreescritura?

/* Sobreescritura */

Sobreescritura de métodos

- Es una forma de aplicar polimorfismo.
- Sirve para dar un comportamiento específico en una clase hija, diferenciándolo del comportamiento de su clase padre.
- Requiere de una relación de herencia entre dos clases, ya que consiste en volver a definir el método definido en una clase padre en la clase hija que la hereda, manteniendo el nombre del método original de la clase padre.

Sobreescritura de métodos

Ejemplo

Heredar una clase y aplicar sobreescritura de métodos

Para aplicar la sobreescritura, se debe definir dentro de la clase hija el método que se desea sobreescribir de la clase padre, manteniendo el nombre del método.

Una vez generada una instancia de la clase hija, al llamar al método sobreescrito se ejecutará el código definido en el método de la clase hija, en lugar del definido en la clase padre.

La clase hija hereda todos los métodos de la clase padre, por lo que si los métodos de la clase padre no son sobreescritos en la clase hija, de todas formas son posibles de ser llamados desde las instancias de la clase hija, ejecutándose en ese caso el método definido en la clase padre.

Sobreescritura de constructor y propiedades

La sobreescritura de métodos también se puede aplicar al constructor o a las propiedades.

De esta forma, se puede aplicar validaciones específicas para una clase hija que no aplican en el caso de la clase padre.

```
class PelotaDeDeporte():
    def __init__(self, color: str) -> None:
        self.__color = color
    @property
    def color(self) -> str:
        return self.__color
    @color.setter
    def color(self, color) -> None:
        self.__color = color
class PelotaDeTenis(PelotaDeDeporte):
    def __init__(self) -> None:
        self.__color = "Amarillo"
    @property
    def color(self) -> str:
        return self.__color
    @color.setter
    def color(self, color: str) -> None:
        pass
pdt = PelotaDeTenis()
pdt.color = "Rojo"
# Salida: Amarillo
print(pdt.color)
```

Sobreescritura haciendo llamado a método de clase padre

Al sobrescribir un método en una clase hija, también es posible ejecutar en su definición el método de la clase padre, de tal forma que al hacer el llamado al método desde la clase hija se ejecute tanto el **método de la clase padre** como la lógica específica sobreescrita en el **método de la clase hija**.

Esto es muy útil para el caso de los constructores, donde una clase hija hereda todos los atributos de la clase padre, pero además incorpora atributos propios.

Para hacer el llamado al método de la clase padre, se puede hacer mediante `super()`, sin argumentos .

Sobreescritura haciendo llamado a método de clase padre

Ejemplo

```
class PelotaDeDeporte():
   def __init__(self, color: str):
       self.__color = color
   @property
   def color(self):
       return self.__color
   @color.setter
   def color(self, color):
       self.__color = color
class PelotaDeFutbol(PelotaDeDeporte):
   def __init__(self, color: str, cantidad_hexagonos: int):
       super().__init__(color) # se ejecuta constructor de
PelotaDeDeporte
       self.__cantidad_hexagonos = cantidad_hexagonos
   @property
   def cantidad_hexagonos(self):
       return self.__cantidad_hexagonos
pdf = PelotaDeFutbol("Blanco y Negro", 15)
# Salida: Blanco y Negro
print(pdf.color)
# Salida: 15
print(pdf.cantidad_hexagonos)
```

{desafío} latam_

Herencia múltiple

"el problema del diamante"

Si un método es heredado (desde una clase A) y sobrescrito por dos clases (B y C), las cuales luego en conjunto son heredadas por una cuarta clase (D), no queda claro cuál de los métodos será ejecutado al ser llamado desde la última clase de la jerarquía (D).

Esto Python lo resuelve según el **orden en el cual se heredan las clases**, de forma tal que se preservará en la clase hija lo definido en la primera clase heredada (declarada de izquierda a derecha) que lo posee. Esta misma regla se aplica a todas las clases superiores de la jerarquía, siendo esto gestionado por el MRO ("Method Resolution Order") de Python.

Sobreescritura de constructor en herencia múltiple

Una forma de lograr que se ejecute el constructor de más de una clase heredada al momento de crear una instancia de la clase hija, es haciendo el **llamado al constructor directamente desde las clases padres**.

Cada llamado debe recibir como argumentos `self` y luego todos los argumentos requeridos en cada caso. Esta forma de hacer uso del constructor en una clase heredada también es posible en herencia simple.

Sobreescritura de constructor en herencia múltiple

Ejemplo

```
class PelotaDeDeporte():
   def __init__(self, tamaño: int):
       print("Creando pelota de deporte")
       self.tamaño = tamaño

class PelotaDePlastico():
   def __init__(self, material: str):
       print("Creando pelota de plástico")
       self.material = material

class PelotaDePingPong(PelotaDeDeporte, PelotaDePlastico):
   def __init__(self, tamaño: int, material: str, timbre: str):
       PelotaDeDeporte.__init__(self, tamaño)
       PelotaDePlastico.__init__(self, material)
       print("Creando pelota de ping pong")
       self.timbre = timbre
# Salida:
# Creando pelota de deporte
# Creando pelota de plástico
# Creando pelota de ping pong
pdpp = PelotaDePingPong(4, "celuloide", "POWERTI")
```

Sobreescritura de constructor en herencia múltiple

Otra forma

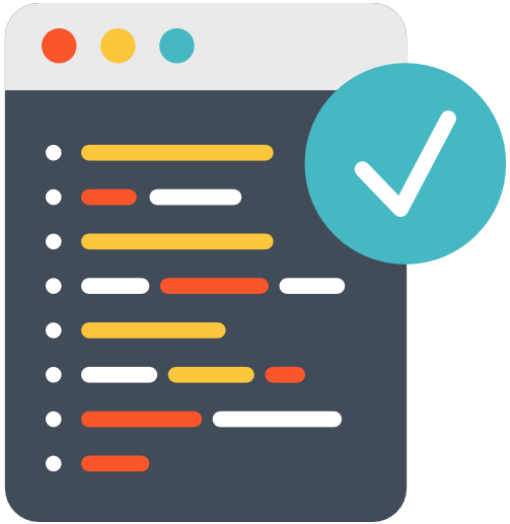
Se puede realizar mediante el uso de **`super()`**, pero se debe tener algunas consideraciones:

"Teniendo como ejemplo una clase hija C que hereda, de izquierda a derecha, las clases A y B, y que se desee ejecutar los constructores padres en el orden A, B al momento de crear una instancia de C"

Parámetro especial **kwargs

Se utiliza para establecer un número variable de argumentos opcionales.

Se denominan argumentos de “clave” (a diferencia de **`*args`** que corresponde a “arguments”, una lista ordenada de argumentos), porque los argumentos están organizados dentro de un diccionario, donde las claves corresponden a los parámetros, y los valores a los argumentos asociados a esos parámetros.



Parámetro especial **kwargs

Ejemplo

- Al hacer la creación de la instancia, se debe especificar cada argumento con el nombre del parámetro al que corresponde, de forma tal que puedan ser almacenados en el parámetro **`**kwargs`**.
- El orden de salida de los print ha cambiado, ya que la clase **PelotaDeDeporte** hace un llamado al constructor de la siguiente clase heredada (**PelotaDePlastico**) antes de ejecutar su propio print.

```
class PelotaDeDeporte():
    def __init__(self, tamaño: int, **kwargs):
        super().__init__(**kwargs)
        print("Creando pelota de deporte")
        self.tamaño = tamaño
class PelotaDePlastico():
    def __init__(self, material: str, **kwargs):
        super().__init__(**kwargs)
        print("Creando pelota de plástico")
        self.material = material

class PelotaDePingPong(PelotaDeDeporte, PelotaDePlastico):
    def __init__(self, timbre: str, **kwargs):
        super().__init__(**kwargs)
        print("Creando pelota de ping pong")
        self.timbre = timbre
# Salida:
# Creando pelota de plástico
# Creando pelota de deporte
# Creando pelota de ping pong
pdpp = PelotaDePingPong(tamaño=4, material="celuloide",
timbre="POWERTI")
```

`isinstance`

Distinguir la clase de una instancia

- Recibe como primer argumento un objeto cualquiera.
- Recibe como segundo argumento un tipo.
- Retornará True en caso de que el objeto entregado en el primer argumento sea del tipo del tipo entregado en el segundo argumento, y False en el caso contrario.

Si quieres conocer más sobre cómo opera esta función, puedes revisar la documentación oficial en este enlace.

isinstance

Ejemplo

```
class PelotaDeDeporte():
    def __init__(self, color: str) -> None:
        self.__color = color
    @property
    def color(self) -> str:
        return self.__color
    @color.setter
    def color(self, color) -> None:
        self.__color = color
class PelotaDeFutbol(PelotaDeDeporte):
    def __init__(self, color: str, cantidad_hexagonos: int) -> None:
        super().__init__(color)
        self.__cantidad_hexagonos = cantidad_hexagonos
    @property
    def cantidad_hexagonos(self) -> int:
        return self.__cantidad_hexagonos
    def hacer_pase(self, destino: str, fuerza: int) -> tuple:
        return (destino, fuerza * 0.5)
class PelotaDeTenis(PelotaDeDeporte):
    def __init__(self) -> None:
        self.__color = "Amarillo"
    @property
    def color(self) -> str:
        return self.__color
    @color.setter
    def color(self, color: str) -> None:
        pass
    def hacer_saque(self, altura: int, fuerza: int) -> tuple:
        return (altura, altura * fuerza)
```

isinstance

Ejemplo

```
from pelota import PelotaDeFutbol, PelotaDeTenis

pdf = PelotaDeFutbol("Blanco y Negro", 15)
pdt = PelotaDeTenis()
pelotas = [pdf, pdf, pdt, pdt, pdf]

for p in pelotas:
   if isinstance(p, PelotaDeTenis) == False:
       p.color = "Roja"
   if isinstance(p, PelotaDeFutbol):
       p.hacer_pase("jugador 2", 3)
   elif isinstance(p, PelotaDeTenis):
       p.hacer_saque(2, 3)
```

isinstance

Definición de la clase padre

Al crear una instancia de una clase heredada se puede acceder a un método creado en su clase padre, la instancia (self) evaluada mediante isinstance puede ser tanto de la clase padre, como de cualquier clase hija que no haya sobrescrito el método donde se hace la evaluación. Esto permite condicionar, entre otros, valores que se pueden asignar a atributos de la instancia.

```python
class PelotaDeDeporte():
    def __init__(self, color: str) -> None:
        if isinstance(self, PelotaDeTenis):
            self.__color = "Amarillo"
        else:
            self.__color = color

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

class PelotaDeTenis(PelotaDeDeporte):
    pass

p = PelotaDeTenis("Rojo")

# Salida: Amarillo
print(p.color)
```

Ejercicio guiado

"Polimorfismo y sobreescritura de métodos"

{desafío} latam_

Polimorfismo y sobreescritura de métodos

Desde la empresa “Juegos por comida”, te han solicitado modificar el demo realizado previamente, consistente en la batalla entre un jugador y un monstruo. Te solicitan que, previo al enfrentamiento, el monstruo debe mostrar el diálogo “GRAAAWR”.

- Considera que todos los monstruos son “personajes no jugadores”, y que todos los “personajes no jugadores” pueden realizar diálogos.
- Los “personajes no jugadores” necesitan un nombre para crearse. Este nombre se debe concatenar al principio de cada diálogo, por ejemplo: Bégimo: "GRAAAWR".

Polimorfismo y sobreescritura de métodos

- También han solicitado que al morir alguno de los enfrentados, se muestre en pantalla el mensaje “¡Felicidades!, ¡Haz ganado la batalla”, en caso de que el vencedor sea el jugador, y el mensaje “¡Oh no!, haz perdido la batalla :(“ en caso de que el vencedor sea el monstruo.

Aspectos técnicos

En esta etapa, se solicita aplicar encapsulamiento a los atributos, además no se debe permitir crear instancias de Monstruo que tengan armas, por lo que se debe sobrescribir el constructor de la clase base.

*Se hará uso de **kwargs y super() para manejar el llamado a múltiples constructores en el caso de herencia múltiple.*

Polimorfismo y sobreescritura de métodos

Solución

Paso 1

En un archivo **npc.py**, definir la clase NPC (del inglés “non player character”). En ella, definir un constructor que reciba por parámetro nombre, el cual se asigna a un atributo privado de instancia, y argumentos opcionales de palabra clave, los cuales se deben usar como argumento en el llamado al constructor de **super()**.

```
class NPC():
    def __init__(self, nombre: str, **kwargs) -> None:
        super().__init__(**kwargs)
        self.__nombre = nombre
```

Polimorfismo y sobreescritura de métodos

Solución

Paso 2

A continuación, en la misma clase, definir el método **mostrar_dialogo**, el cual recibe por parámetro un texto, y luego lo muestra por pantalla en la forma solicitada usando print.

```
def mostrar_dialogo(self, mensaje: str) -> None:
        print(f"{self.__nombre}: {mensaje}")
```

Polimorfismo y sobreescritura de métodos

Solución

Paso 3

En el archivo **personaje.py**, refactorizar la clase abstracta Personaje con un constructor que recibe por parámetro **hp**, **atk**, **df**, y ****kwargs**, y los asigna en los respectivos atributos privados de instancia y constructor de **super()**.

```
from abc import ABC, abstractmethod
class Personaje(ABC):
   def __init__(self, hp: int, atk: int, df: int, **kwargs) -> None:
       super().__init__(**kwargs)
       self.__hp = hp
       self.__atk = atk
       self.__df = df
```

Polimorfismo y sobreescritura de métodos

Solución

Paso 4

En la misma clase, definir la propiedad hp, ya que requiere ser accedida desde fuera de la clase (por el script demo, y por las clases hijas).

```
@property
def hp(self) -> int:
    return self.__hp
```

Paso 5

En la misma clase, definir el método setter de hp, ya que requiere ser modificado desde fuera de la clase (desde las clases hijas).

```
@hp.setter
def hp(self, hp) -> None:
    self.__hp = hp
```

Polimorfismo y sobreescritura de métodos

Solución

Paso 6

Seguidamente en la misma clase, definir la propiedad atk, ya que requiere ser accedida desde fuera de la clase (desde las clases hijas).

```
@property
def atk(self) -> int:
    return self.__atk
```

Paso 7

A continuación, en la misma clase, definir la propiedad df, ya que requiere ser accedida desde fuera de la clase (desde las clases hijas).

```
@property
def df(self) -> int:
    return self.__df
```

Polimorfismo y sobreescritura de métodos

Solución

Paso 8

Mantener la definición de los métodos abstractos ataque y defensa.

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

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

Polimorfismo y sobreescritura de métodos

Solución

Paso 9

En el archivo **jugador.py**, sobreescribir el constructor de la clase Jugador, de forma tal que dentro de éste se haga el llamado al constructor de la clase padre, y a continuación se asigne el valor de arma al atributo de instancia.

```
import random
from personaje import Personaje

class Jugador(Personaje):
   def __init__(self, hp: int, atk: int, df: int, arma: str = None) -> None:
       super().__init__(hp, atk, df)
       self.__arma = arma
```

Polimorfismo y sobreescritura de métodos

Solución

Paso 10

Mantener los métodos ataque y defensa.

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

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

Polimorfismo y sobreescritura de métodos

Solución

Paso 11

En el archivo **monstruo.py**, refactorizar la clase Monstruo, de forma tal que herede de Personaje y de NPC respectivamente (debe agregar import). Sobrescribir el constructor, el cual debe recibir por parámetro una lista de argumentos de palabra clave, y usarlo como argumento para el llamado al constructor de **super()**.

```
from personaje import Personaje
from npc import NPC

class Monstruo(Personaje, NPC):
   def __init__(self, **kwargs) -> None:
       super().__init__(**kwargs)
```

Polimorfismo y sobreescritura de métodos

Solución

Paso 12

A continuación, en el mismo archivo, mantener los métodos ataque y defensa.

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

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

Polimorfismo y sobreescritura de métodos

Solución

Paso 13

En el script **demo.py**, luego de importar las clases requeridas, crea una instancia de Monstruo, especificando con el nombre de cada parámetro los argumentos requeridos según lo solicitado. Luego, desde la instancia, haz el llamado al método **mostrar_dialogo** para mostrar el mensaje solicitado.

```
from jugador import Jugador
from monstruo import Monstruo

m = Monstruo(hp=1000, atk=1, df=8, nombre="Bégimo")
m.mostrar_dialogo("GRAAAWR")
```

Polimorfismo y sobreescritura de métodos

Solución

Paso 14

Modificar la lista enfrentados para que almacene la instancia de Monstruo creada.

```
enfrentados = [Jugador(500, 10, 5, "espada"), m]
atk = 0
```

Polimorfismo y sobreescritura de métodos

Solución

Paso 15

Modificar la condición que evalúa el hp del enfrentado recorrido, de forma que en caso de que haya muerto se evalúe el tipo de su clase (mediante isinstance) para mostrar en pantalla el mensaje solicitado.

```
while any(e.hp <= 0 for e in enfrentados) == False:
   for e in enfrentados:
       if atk:
           e.defensa(atk)
       if e.hp > 0:
           atk = e.ataque()
       else:
           if isinstance(e, Monstruo):
               print("¡Felicidades!, ¡Haz ganado la batalla")
           elif isinstance(e, Jugador):
               print("¡Oh no!, haz perdido la batalla :(")
```

¿En qué consiste la sobrescritura de métodos?

¿Cómo se aplica la sobrescritura de un método en la clase heredada?

Desde una instancia de una clase hija, ¿cómo se puede ejecutar el método
sobrescrito en ella en lugar del definido en su clase padre?

¿Qué hace la función isinstance?

Próxima sesión...

- *Desafío guiado.*

{desafío}
latam_

{desafío}
latam_

Academia de
talentos digitales