Sesión 4.2 - Herencia y composición.

- Herencia en Python
  - Métodos de la clase derivada
  - Acceso a la clase base.
  - Ejemplo de herencia en Python
  - Herencia múltiple.
  - Métodos de la clase derivada.
  - Ejemplo de herencia múltiple en Python.
- Composición.
  - Ejemplo de composición de clases en Python
- Herencia vs. Composición
- Clases y métodos abstractos.
- Aplicaciones
  - Errores Personalizados
  - Enum

Herencia en Python

En Python, la herencia es un mecanismo que permite crear una clase nueva a partir de una clase existente, heredando todos los atributos y métodos de la clase existente. La clase existente se conoce como clase base o clase padre, y la nueva clase se conoce como clase derivada o clase hija.

La herencia se define utilizando la siguiente sintaxis:

```
class ClaseDerivada(ClaseBase):
    # Definición de la clase derivada
```

En este ejemplo, `ClaseDerivada` es la nueva clase que estamos creando, y `ClaseBase` es la clase existente que estamos heredando.

Métodos de la clase derivada

La clase derivada hereda todos los métodos y atributos de la clase base. Además, la clase derivada también puede tener sus propios métodos y atributos.

```
class Cliente:
    'Representa un cliente genérico.'
    def pagar(self, n):
        "Cobra dinero del cliente"
        print('Cobro del cliente:', n, '€')

class ClienteVip(Cliente):
    'Representa un cliente VIP'
```

```
    def pagar(self, n):
        "Cobra dinero del cliente VIP (con descuento)."
        print('Cobro del cliente', n*0.9, '€ [10% de descuento]')
```

Si la clase derivada tiene un método con el mismo nombre que un método de la clase base, el método de la clase derivada reemplaza al método de la clase base. Esto se conoce como **sobrescritura de métodos**.

Acceso a la clase base.

`super()` es una función incorporada en Python que permite llamar a un método de la clase base desde la clase derivada. Esto es útil cuando queremos agregar funcionalidad a un método de la clase base sin reemplazarlo completamente.

La sintaxis para utilizar `super()` es la siguiente:

```
class ClaseDerivada(ClaseBase):
    def metodo(self):
        super().metodo()
        # Agregar funcionalidad adicional
```

En este ejemplo, `metodo()` es un método de la clase derivada que llama al método de la clase base utilizando `super()`.

Ejemplo de herencia en Python

A continuación se muestra un ejemplo de cómo se puede utilizar la herencia en Python:

```
class Animal:
    "Representa un animal genérico."

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

    def sonido(self):
        "Sin sonido, es genérico."
        pass

class Perro(Animal):
    "Representa a un perro."

    def __init__(self, nombre, edad, raza):
        super().__init__(nombre, edad)  # Inicia la clase Animal
        self.raza = raza

    def sonido(self):
        "Especializamos el sonido para un perro."
        return "Guau!"
```

```python
class Gato(Animal):
    "Representa a un gato."

    def __init__(self, nombre, edad, color):
        super().__init__(nombre, edad)
        self.color = color

    def sonido(self):
        "Especializamos el sonido para un gato."
        return "Miau!"
```

En este ejemplo, la clase `Animal` es la clase base, y las clases `Perro` y `Gato` son las clases derivadas. Las clases derivadas heredan los atributos y métodos de la clase base, y también tienen sus propios atributos y métodos.

Ejercicio de clase.

Crea una clase base llamada `Vehiculo` que tenga los siguientes atributos y métodos:

- Atributos:
  - `marca`: la marca del vehículo.
  - `modelo`: el modelo del vehículo.
  - `año`: el año de fabricación del vehículo.
  - `velocidad`: velocidad (al inicio 0) en km/h.
- Métodos:
  - `acelerar()`: aumenta la velocidad del vehículo en 5 km/h.
  - `frenar()`: disminuye la velocidad del vehículo en 5 km/h.

Luego, crea dos clases hijas llamadas `Automovil` y `Motocicleta` que hereden de `Vehiculo` y tengan los siguientes atributos y métodos adicionales:

- `Automovil`
  - Atributos:
    - `n_puertas`: el número de puertas del automóvil.
    - `puertas`: una lista de puertas abiertas o cerradas.
  - Métodos:
    - `abrir_puerta(n)`: abre una puertas del automóvil.
    - `cerrar_puerta(n)`: cierra una puerta del automóvil.
    - `acceder_puerta(n)`: prueba a acceder a una puerta.

      > **Nota**: Si el automóvil está en movimiento todas las puertas se cierran y no se pueden abrir.

- `Motocicleta`
  - Atributos:
    - `cilindrada`: la cilindrada de la motocicleta en centímetros cúbicos

- Métodos:
  - `acelerar()`: modifica el método de `Vehiculo` haciendo que acelere una décima parte de la cilindrada.

Herencia múltiple.

En Python, es posible crear una clase derivada que herede de varias clases base al mismo tiempo. Esto se conoce como herencia múltiple.

La herencia múltiple se define utilizando la siguiente sintaxis:

```
class ClaseDerivada(ClaseBase1, ClaseBase2, ..., ClaseBaseN):
    # Definición de la clase derivada
```

En este ejemplo, `ClaseDerivada` es la nueva clase que estamos creando, y `ClaseBase1`, `ClaseBase2`, ..., `ClaseBaseN` son las clases existentes que estamos heredando.

Métodos de la clase derivada.

La clase derivada hereda todos los métodos y atributos de todas las clases base. Si una clase base tiene un método con el mismo nombre que otro método de otra clase base, el método se resuelve siguiendo el **MRO (Method Resolution Order)** de la clase derivada.

Ejemplo de herencia múltiple en Python.

A continuación, se muestra un ejemplo genérico de cómo se puede utilizar la herencia múltiple en Python:

```
class A:
    def metodo_a(self):
        print("Método A")

class B:
    def metodo_b(self):
        print("Método B")

class C(A, B):
    def metodo_c(self):
        print("Método C")

objeto = C()
objeto.metodo_a() # Imprime "Método A"
objeto.metodo_b() # Imprime "Método B"
objeto.metodo_c() # Imprime "Método C"
```

En este ejemplo, la clase `C` hereda de las clases `A` y `B` al mismo tiempo. La clase `C` tiene su propio método `metodo_c`, pero también hereda los métodos `metodo_a` de la clase `A` y `metodo_b` de la clase `B`.

Composición.

Composición de clases en Python

La composición de clases es otra forma de crear relaciones entre clases en Python, además de la herencia. En la composición, una clase incluye una instancia de otra clase como uno de sus atributos.

La composición se define creando una instancia de la clase que se quiere incluir como atributo en la clase compuesta:

```
class ClaseCompuesta:
    def __init__(self):
        self.otra_clase = OtraClase()
```

En este ejemplo, `ClaseCompuesta` es la clase actual que estamos creando, y OtraClase es la clase que estamos incluyendo como atributo.

Ejemplo de composición de clases en Python

A continuación, se muestra un ejemplo de cómo se puede utilizar la composición de clases en Python:

```
class Motor:
    "Representa el motor de un vehículo."
    def __init__(self, cilindrada):
        self.cilindrada = cilindrada

    def arrancar(self):
        "Brumbrumbrumbrum"
        print("Motor arrancado")

class Coche:
    "Representa un coche."

    def __init__(self, marca, modelo, cilindrada):
        self.marca = marca
        self.modelo = modelo
        self.motor = Motor(cilindrada)

    def arrancar(self):
        "Arranca el coche."
        print(f"Arrancando el coche {self.marca} {self.modelo}")
        self.motor.arrancar()

coche = Coche("Ford", "Focus", 1600)
coche.arrancar() # Imprime "Arrancando el coche Ford Focus" y "Motor
arrancado"
```

En este ejemplo, la clase Coche incluye una instancia de la clase Motor como uno de sus atributos. El constructor de Coche recibe tres argumentos: marca, modelo y cilindrada. Dentro del constructor, se crea una instancia de Motor con la cilindrada proporcionada, que se almacena en el atributo motor de Coche. El

método arrancar de Coche imprime un mensaje que indica que el coche está arrancando, y luego llama al método arrancar de la instancia de Motor.

Herencia vs. Composición

Veamos una comparativa entre ambos métodos:

	Herencia de clases	**Composición de clases**
Definición	Se define a partir de una **clase** existente.	Se define a partir de **objetos** de otras clases.
Relación	Existe una relación de "es un" entre la clase derivada y la clase base.	Existe una relación de "tiene un" entre la clase y los objetos que se componen de ella.
Flexibilidad	Menos flexible, ya que los cambios en la clase base afectan a las clases derivadas.	Más flexible, ya que los cambios en las clases de los componentes no afectan a la clase principal.
Reutilización de código	Promueve la reutilización de código, ya que las clases derivadas heredan los atributos y métodos de la clase base.	También promueve la reutilización de código, ya que se pueden utilizar objetos de otras clases para construir una nueva clase.
Complejidad	Puede llevar a una jerarquía de clases compleja y difícil de mantener.	Se corre el riesgo de que el código se disperse en múltiples módulos cuando se utilizan muchos objetos de otras clases.

La elección entre herencia y composición depende de las necesidades específicas del problema que se está resolviendo. Si las subclases comparten muchos atributos y métodos con la superclase, la herencia puede ser una buena opción. Si se necesita más flexibilidad o se desea evitar la duplicación de código, la composición puede ser la mejor opción.

En general, es importante tener en cuenta que la herencia y la composición no son mutuamente excluyentes y pueden ser utilizadas juntas en una solución de programación orientada a objetos.

Clases y métodos abstractos.

En Python, una **clase abstracta** es una clase que no puede ser instanciada directamente, sino que **sirve como una plantilla para definir otras clases**. La clase abstracta define los métodos y atributos que deben estar presentes en sus clases derivadas, pero no proporciona una implementación para ellos. Las clases derivadas deben proporcionar una implementación para estos métodos y atributos.

Python proporciona una biblioteca de clases abstractas en el módulo `abc`. Para definir una clase abstracta, debemos hacer uso de la clase `ABC` (**Abstract Base Class**) como una superclase y decorar los métodos que queremos que sean abstractos con el decorador `@abstractmethod`.

A continuación se muestra un ejemplo de cómo definir una clase abstracta en Python:

```
from abc import ABC, abstractmethod

class FiguraGeometrica(ABC):
```

```
    "Figura geométrica genérica."

    @abstractmethod
    def area(self):
        "Calcula el área de la figura."
        pass

    @abstractmethod
    def perimetro(self):
        "Calcula el perímetro de la figura."
        pass
```

En este ejemplo, `FiguraGeometrica` es una clase abstracta que define dos métodos abstractos `area()` y `perimetro()`. Estos métodos no tienen una implementación concreta en la clase abstracta y se espera que sean implementados en las clases derivadas. Por ejemplo:

```
class Cuadrado(FiguraGeometrica):
    "Representa un triángulo."

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

    def area(self):
        return self.lado**2

    def perimetro(self):
        return 4*self.lado
```

Los métodos abstractos deben ser implementados en las subclases para que estas puedan ser instanciadas. Si una subclase no implementa un método abstracto, entonces también se considera una clase abstracta y no se puede instanciar.

Los métodos abstractos **se utilizan para definir una interfaz común** que debe ser implementada por varias clases diferentes. Esto permite que varias clases diferentes implementen la misma interfaz, lo que facilita la reutilización y la modularidad del código.

Ejercicio de clase

Completa el ejemplo anterior. Crea una clase abstracta llamada `FiguraGeometrica` que tenga los siguientes atributos y métodos:

- Atributos:
  - `nombre`: el nombre de la figura.
- Métodos:
  - `area()`: calcula el área de la figura.
  - `perimetro()`: calcula el perímetro de la figura.

Luego, crea dos clases hijas llamadas `Circulo` y `Rectangulo` que hereden de `FiguraGeometrica` y tengan los siguientes atributos y métodos adicionales:

- `Circulo`
    - Atributos:
        - `radio`: el radio del círculo.
    - Métodos:
        - `area()`: sobrescribe el método `area()` de `FiguraGeometrica` para calcular el área del círculo.
        - `perimetro()`: sobrescribe el método `perimetro()` de `FiguraGeometrica` para calcular el perímetro del círculo.
- `Rectangulo`
    - Atributos:
        - `base`: la base del rectángulo.
        - `altura`: la altura del rectángulo.
    - Métodos:
        - `area()`: sobrescribe el método `area()` de `FiguraGeometrica` para calcular el área del rectángulo.
        - `perimetro()`: sobrescribe el método `perimetro()` de `FiguraGeometrica` para calcular el perímetro del rectángulo.

Aplicaciones

A continuación un par de aplicaciones que usan herencia en la **Python Standard Library**.

Errores Personalizados

Es posible crear nuestras propias excepciones personalizadas. Esto es útil cuando queremos lanzar una excepción específica que no está disponible en Python de manera nativa. Por ejemplo, podríamos querer crear una excepción para cuando un usuario intenta acceder a un archivo que no existe en nuestro programa.

Para crear una excepción personalizada en Python, debemos definir una clase que herede de la clase `Exception`. Esta clase puede tener cualquier nombre que deseemos, pero es una buena práctica incluir la palabra "Error" en el nombre para indicar que se trata de una excepción.La clase debe tener un constructor que puede tomar cualquier cantidad de argumentos que deseemos, aunque lo común es incluir un mensaje de error que se muestra cuando se lance la excepción.

Veámoslo con un ejemplo. Supongamos que estás creando un programa para manejar la venta de entradas para un evento. Una de las reglas de tu programa es que una persona solo puede comprar un máximo de 10 entradas. Si alguien intenta comprar más de 10 entradas, quieres lanzar una excepción personalizada para manejar este caso.

```
class MaxTicketsError(Exception):
    "Error que se lanza cuando se sobrepasa el límite de tickets
permitido."
    def __init__(self, mensaje):
        self.mensaje = mensaje
```

```
        super().__init__(self.mensaje)  # Inicializa la clase base
Excepction
```

En este ejemplo, hemos creado una clase llamada `MaxTicketsError` que hereda de la clase `Exception`. En su constructor, tomamos un argumento `message` que se utiliza para establecer el mensaje de error que se mostrará cuando se lance la excepción.

```
def comprar_tickets(num_tickets):
    "Simula comprar tickets"
    if num_tickets > 10:
        raise MaxTicketsError("Solo se pueden comprar hasta 10 entradas por
persona")
    else:
        print(f"¡Has comprado {num_tickets} entradas!")

try:
    buy_tickets(15)
except MaxTicketsError as e:
    print(e.mensaje)
```

También podemos crear excepciones personalizadas con más argumentos. Supongamos que estamos construyendo un programa de análisis de datos y necesitamos calcular la correlación entre dos listas. Si las dos listas tienen diferentes longitudes, debemos lanzar una excepción personalizada que muestre las dos listas y el mensaje de error.

```
class UnequalLengthError(Exception):
    "Error que se genera cuando se requiere dos listas de la misma
longitud."

    def __init__(self, message, list1, list2):
        self.message = message
        self.list1 = list1
        self.list2 = list2
        super().__init__(self.message)

    def __str__(self):
        return f"{self.message}. Lista 1: {self.list1}. Lista 2:
{self.list2}"
```

En este ejemplo, hemos creado una clase llamada `UnequalLengthError` que hereda de la clase `Exception`. En su constructor, tomamos tres argumentos: `message`, `list1` y `list2`. `message` se utiliza para establecer el mensaje de error que se mostrará cuando se lance la excepción. `list1` y `list2` son las dos listas que se pasan a la función de correlación y queremos incluir en nuestro mensaje de error si tienen longitudes diferentes.

Además, hemos agregado un método `__str__` a la clase para personalizar la salida del objeto de excepción.

Ahora podemos usar esta excepción personalizada en nuestro programa para manejar el caso en que alguien intenta calcular la correlación de dos listas de diferentes longitudes:

```
def correlation(list1, list2):
    if len(list1) != len(list2):
        raise UnequalLengthError("Las listas tienen diferentes longitudes",
list1, list2)
    else:
        # Calcula la correlación
        pass

try:
    correlation([1, 2, 3], [4, 5])
except UnequalLengthError as e:
    print(e)
```

Enum

`enum` es un módulo de la biblioteca estándar de Python que permite definir enumeraciones, es decir, un conjunto de constantes con nombre y valor que pueden ser asignados a variables. Las enumeraciones son útiles cuando se necesita trabajar con un conjunto fijo de valores que no cambian, como días de la semana, meses del año, colores, etc.

La forma más básica de crear una enumeración es mediante el uso de la clase Enum del módulo enum. Aquí un ejemplo:

```
from enum import Enum

class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

color_1 = Color(1)  # Color Rojo
color_2 = Color.GREEN  # Color Verde
```

En este ejemplo, se define la enumeración `Color` con tres constantes: `RED`, `GREEN` y `BLUE`. Cada constante tiene un valor numérico asociado, que en este caso son `1`, `2` y `3` respectivamente.

Una vez que se ha definido una enumeración, se pueden usar las constantes de la misma como si fueran atributos de la clase. Por ejemplo, para acceder al valor de la constante `RED`, se puede escribir `Color.RED.value`. Sin embargo, lo común es utilizarlas directamente desde

El módulo enum también proporciona otras características útiles, como la capacidad de crear enumeraciones a partir de secuencias o diccionarios, la creación de enumeraciones con valores personalizados, la comparación de enumeraciones y la iteración a través de ellas.

Además, con enum también se pueden definir miembros con nombres que no son válidos en Python, mediante el uso de decoradores especiales, y también se pueden crear enumeraciones con comportamientos personalizados, como por ejemplo, para definir métodos y propiedades específicas de la enumeración.

> Recursos de `enum`:
>
> - Documentación.
> - Tutorial.
> - Libro de recetas.