Herencia, Polimorfismo y Excepciones en Python

La Programación Orientada a Objetos en Python nos permite crear código reutilizable y estructurado mediante conceptos fundamentales como la herencia y el polimorfismo. Estos pilares, junto con el manejo adecuado de excepciones, son esenciales para desarrollar aplicaciones robustas y mantenibles.

En esta presentación, exploraremos cómo implementar estos conceptos en Python, desde la creación de jerarquías de clases hasta el manejo personalizado de errores, con ejemplos prácticos que podrás aplicar en tus propios proyectos.

por Kibernum Capacitación

Preguntas de Activación de Contenidos

¿Conoces algún ejemplo de la vida real que funcione por herencia (como padres e hijos, especies, profesiones)? ¿Cómo lo trasladarías a clases en programación?

¿Qué tipos de errores has encontrado al programar? ¿Cómo los resolviste?

¿Por qué crees que es útil agrupar errores y definir su comportamiento en estructuras como try-except?

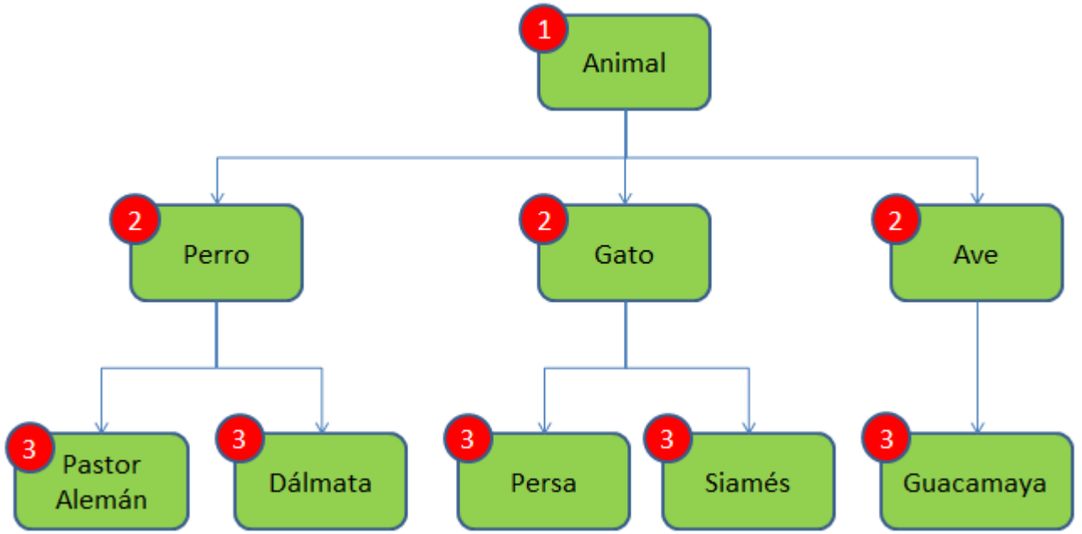
Fundamentos de la Herencia

¿Qué es la herencia?

La herencia es uno de los pilares fundamentales de la Programación Orientada a Objetos (POO). Permite crear nuevas clases basadas en clases existentes, reutilizando su código y extendiendo sus funcionalidades.

En Python, cuando una clase hereda de otra:

- Se le llama subclase o clase hija
- La clase de la cual hereda se llama superclase o clase padre




```python
class Animal:
    def hablar(self):
        print("El animal hace un sonido")

class Perro(Animal):
    def hablar(self):
        print("El perro ladra")
```

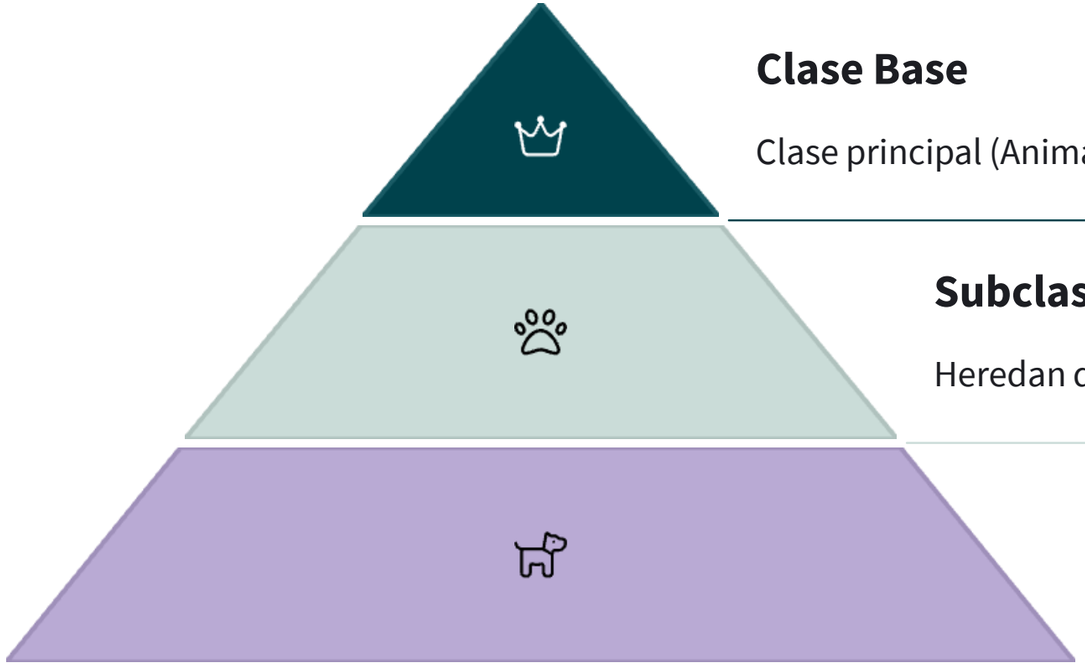
La herencia nos permite construir sobre código existente, evitando la duplicación y facilitando el mantenimiento. Una subclase puede acceder a los métodos y atributos de su clase padre, además de definir los suyos propios o sobrescribir los heredados.

Jerarquía de Clases

```python
class Animal:
    pass

class Mamifero(Animal):
    pass

class Gato(Mamifero):
    pass
```



Clase Base

Clase principal (Animal)

Subclases

Heredan de la clase base (Perro, Gato, Ave)

Sub-subclases

Heredan de las subclases (Pastor Alemán, Persa, Guacamaya)

Puedes tener varias capas de herencia, formando lo que se conoce como jerarquía de clases. Es como un "árbol genealógico" donde cada clase hereda características de sus "antepasados". Esta estructura permite organizar el código de manera lógica y reutilizable.

Herencia Múltiple y MRO

Herencia Múltiple

En Python, una clase puede heredar de más de una clase padre simultáneamente, adquiriendo características de todas ellas. Esto permite combinar funcionalidades de diferentes fuentes en una sola clase.

Method Resolution Order (MRO)

Cuando hay métodos con el mismo nombre en las clases padre, Python usa el MRO para decidir cuál método usar. Puedes ver este orden usando **NombreClase.__mro__**.

```
class A:
    pass

class B:
    pass

class C(A, B):  # Herencia múltiple
    pass
```

Consideraciones

La herencia múltiple es poderosa pero puede complicar el código si no se usa con cuidado. Es importante entender el MRO para evitar comportamientos inesperados en tu programa.

Polimorfismo

```
def hacer_hablar(animal):
    animal.hablar()
```

Sobrecarga

```
def saludar(nombre="amigo"):
    print(f"Hola, {nombre}")
```

Métodos Mágicos (Dunder)

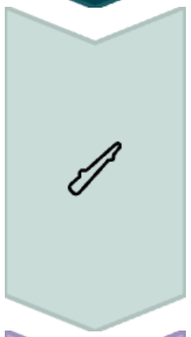
```
class Persona:
    def __init__(self, nombre):
        self.nombre = nombre

    def __str__(self):
        return f"Hola, soy {self.nombre}"
```

Polimorfismo y Métodos Mágicos

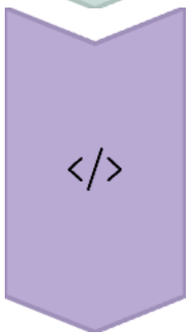
Polimorfismo

Distintas clases pueden tener métodos con el mismo nombre pero comportamientos diferentes. Permite tratar objetos de diferentes clases de manera uniforme.



Métodos Mágicos (Dunder)

Son métodos especiales que empiezan y terminan con doble guion bajo (__). Permiten definir comportamientos para operadores y funciones integradas.



Sobrecarga

En Python se logra usando valores por defecto o *args, ya que Python no admite sobrecarga de métodos como otros lenguajes.

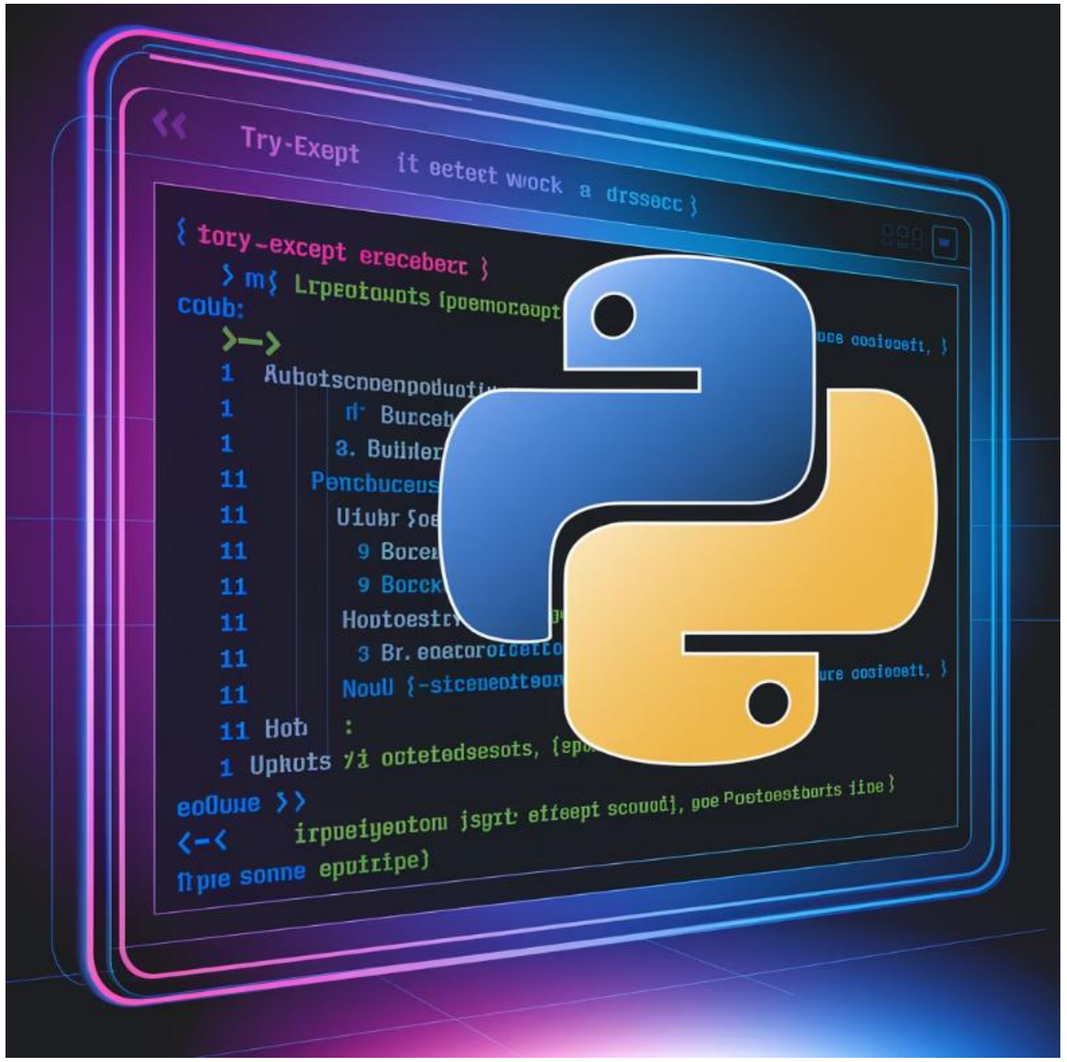
Introducción a las Excepciones

¿Qué es una excepción?

Una excepción es un error que ocurre durante la ejecución de un programa. Si no se maneja, el programa se detiene abruptamente.

Python ofrece herramientas para capturar y manejar excepciones, evitando que tu aplicación se caiga por completo y permitiendo una respuesta controlada ante situaciones inesperadas.

Las excepciones no son solo mensajes de error: son objetos que pertenecen a una jerarquía de clases, lo que permite manejarlas de forma estructurada y específica según el tipo de error que ocurra.



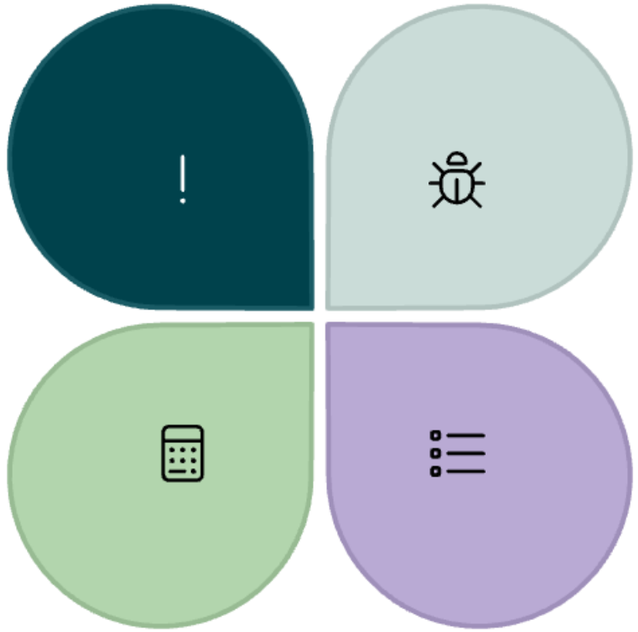
Jerarquía de Excepciones

BaseException

La clase más general de todas las excepciones.

ValueError

Ocurre cuando una función recibe un argumento del tipo correcto pero con un valor inapropiado.



Exception

Hereda de BaseException y es la clase base más común que usamos al capturar errores.

IndexError

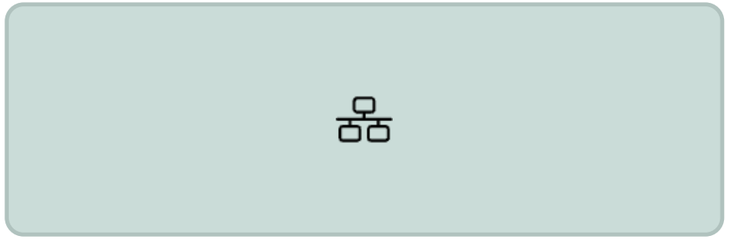
Se produce al intentar acceder a un índice que no existe en una secuencia.

Esta jerarquía permite capturar excepciones específicas o grupos de excepciones relacionadas, dando flexibilidad al manejo de errores en nuestras aplicaciones.

Manejo de Excepciones con try-except-finally

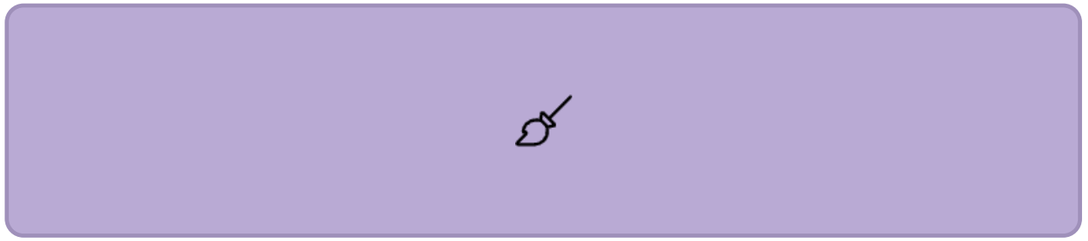
try

Bloque de código que podría generar una excepción



except

Captura y maneja la excepción si ocurre



finally

Se ejecuta siempre, haya o no excepción

El manejo adecuado de excepciones permite que nuestros programas sean más robustos, capaces de recuperarse de errores y proporcionar información útil al usuario o al desarrollador sobre lo que salió mal.

Manejo de Excepciones con try-except-finally

```
try:
    numero = int(input("Ingresa un número: "))
except ValueError:
    print("Eso no es un número válido.")
finally:
    print("Este bloque siempre se ejecuta.")
```

Excepciones Personalizadas

Definir la Clase de Excepción

Crea una nueva clase que herede de Exception o alguna de sus subclases. Esto permite crear errores específicos para tu aplicación.

Implementar Funcionalidad

Añade atributos y métodos específicos que proporcionen información detallada sobre el error ocurrido en el contexto de tu aplicación.

Lanzar y Capturar

Usa "raise" para lanzar tu excepción personalizada cuando sea necesario y captúrala con bloques try-except en los lugares apropiados.

```
class ErrorPersonalizado(Exception):
    pass
```

Actividad Práctica: Sistema de Gestión de Usuarios

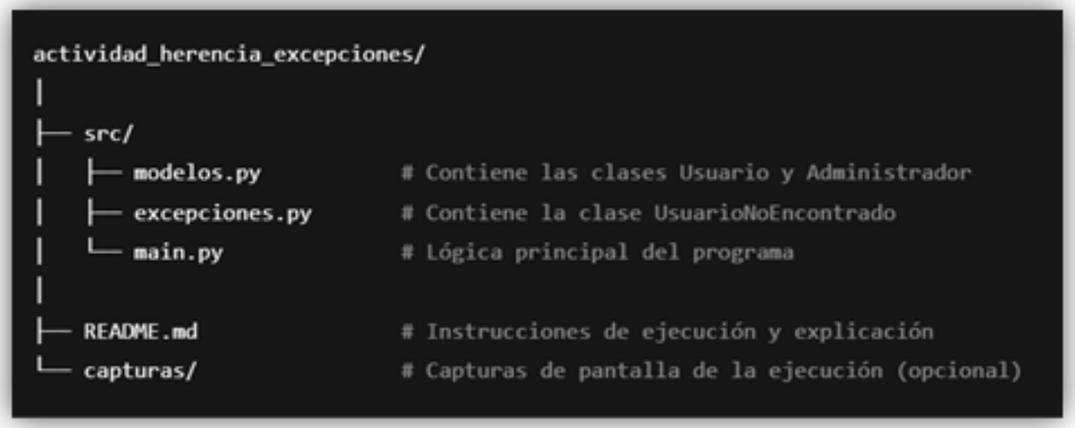
Objetivo:

Aplicar los conceptos de **herencia**, **jerarquía de clases** y **manejo de excepciones personalizadas**, desarrollando un pequeño sistema de administración de usuarios con validación de errores.

Contexto:

En un sistema de gestión, los administradores deben tener la capacidad de eliminar usuarios registrados. Cuando el nombre no se encuentra, se debe capturar una excepción personalizada que permita manejar el error de forma controlada.

Estructura de carpetas sugeridas:



Actividad Práctica: Sistema de Gestión de Usuarios

Comandos de ejecución sugeridos (desde terminal)

```
cd actividad_herencia_excepciones/src
python main.py
```

Asegúrate de que Python esté instalado (python --version) y que estés usando Python 3.8 o superior.

Paso a paso detallado:

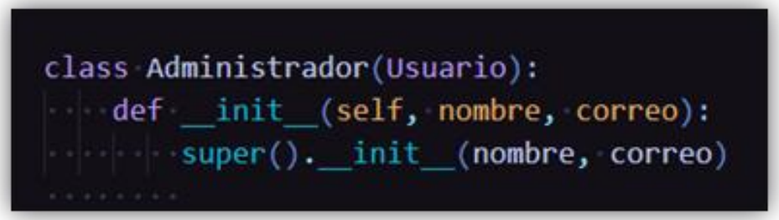
1 Crear una clase Usuario

```
class Usuario:
    def __init__(self, nombre, correo):
        self.nombre = nombre
        self.correo = correo
```

Explicación: Esta clase representa a un usuario genérico. El constructor (__init__) inicializa el nombre y el correo electrónico.

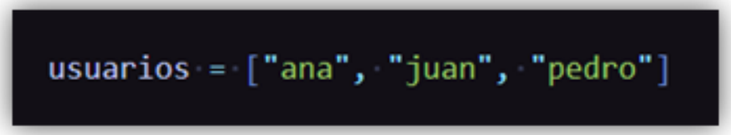
Actividad Práctica: Sistema de Gestión de Usuarios

2. Crear una subclase Administrador que herede de Usuario



Explicación: Usamos super() para llamar al constructor de la clase padre (Usuario). Esto es una práctica recomendada para mantener la jerarquía correctamente.

3 Definir una lista simulada de usuarios



Explicación: Esta lista actuará como la base de datos simulada de usuarios registrados.

Actividad Práctica: Sistema de Gestión de Usuarios

4. Crear una excepción personalizada UsuarioNoEncontrado

```
class UsuarioNoEncontrado(Exception):
    def __init__(self, nombre):
        super().__init__(f"El usuario '{nombre}' no fue encontrado.")
```

Explicación: Creamos una subclase de Exception para definir un error propio que podemos lanzar y capturar más adelante.

5. Agregar método eliminar_usuario(nombre) a la clase Administrador

```
class Administrador(Usuario):
    def __init__(self, nombre, correo):
        super().__init__(nombre, correo)

    def eliminar_usuario(self, nombre, lista_usuarios):
        if nombre not in lista_usuarios:
            raise UsuarioNoEncontrado(nombre)
        lista_usuarios.remove(nombre)
        print(f"✅ Usuario '{nombre}' eliminado exitosamente.")
```

Explicación:

- Se verifica si el nombre está en la lista.
- Si no está, se **lanza la excepción personalizada**.
- Si está, se elimina y se imprime un mensaje de confirmación.

Actividad Práctica: Sistema de Gestión de Usuarios

6. Ejecutar el código con manejo de excepciones (try-except)

```
admin = Administrador("admin1", "admin@correo.com")

try:
    admin.eliminar_usuario("maria", usuarios)
except UsuarioNoEncontrado as e:
    print(f"Error: {e}")
```

Explicación:

- Se crea un administrador y se intenta eliminar un usuario que **no está** en la lista.
- El error es **capturado y mostrado** al usuario con un mensaje claro.

Enlace a Material Complementario

https://www.youtube.com/watch?v=E1ND60YHecg

🎥 Este video ofrece una explicación didáctica y en español sobre la **herencia**, el **polimorfismo**, y el **manejo de excepciones** en Python. Incluye ejemplos prácticos paso a paso, ideales para quienes están comenzando a aplicar estos conceptos en sus proyectos.

Reflexiones sobre Programación Orientada a Objetos

¿Qué ventajas encuentras al usar herencia para organizar clases relacionadas? ¿En qué tipo de proyectos crees que es más útil?

¿Cómo influye el polimorfismo en la flexibilidad del código cuando trabajas con múltiples subclases?

¿Qué beneficios ves en usar excepciones personalizadas en lugar de depender únicamente de errores genéricos como ValueError o KeyError?