


Lectura Herencia, Polimorfismo y Excepciones en Python

 por Kibernet Capacitación S.A.

```

311e))
P>.fbnre (taor))
xiter inheritance
miora
oairice))
xsoert i & (polyiitaom
xolyeerice)

```

```

333
{faler))
Pj ~ ftoee))
Plee))>v))
Pj ~ ftabaitancer (tee)
Plee))>v))
Pj ~ ftabaitancer (tee)

```

```

333
{faler))
Plee))'vlt)
Pj ~ fteeritance)

```

¿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.

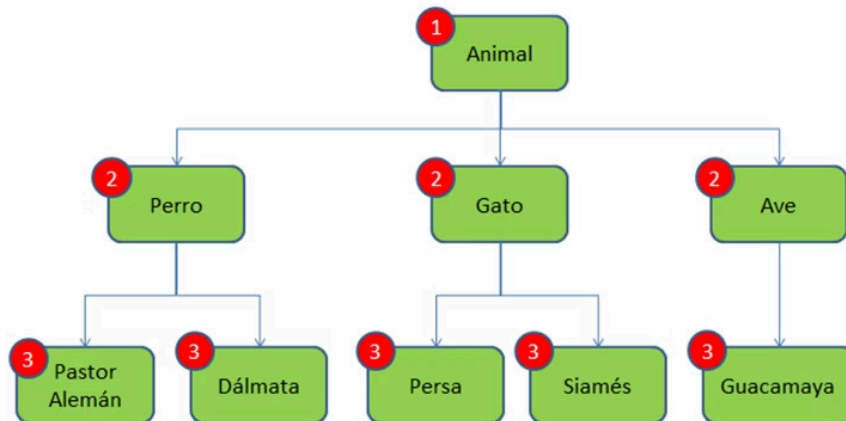
En Python, cuando una clase hereda de otra:

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

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

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

En este ejemplo, Perro hereda de Animal y sobrescribe el método hablar.



Jerarquía de clases en herencia

Esta imagen muestra cómo funciona la herencia en programación orientada a objetos con una jerarquía de clases.

1. Animal es la clase padre o base. Es general, como decir: "todos estos son animales".
2. Perro, Gato y Ave heredan de Animal. Son subclases: toman lo común de Animal y le agregan sus propias cosas.
3. Las razas específicas (Pastor Alemán, Persa, Guacamaya, etc.) heredan de sus respectivas clases (Perro, Gato, Ave). Son sub-subclases. Toman lo de su "mamá" (Perro, Gato...) y lo de su "abuelo" (Animal).

Puedes tener varias capas de herencia:

```
class Animal:
    .. pass

class Mamifero(Animal):
    .. pass

class Gato(Mamifero):
    .. pass
    .. ..
```

Esto se conoce como jerarquía de clases, donde una clase hereda de otra que también hereda de otra, formando un "árbol genealógico" de clases.

Herencia Múltiple y MRO

En Python, una clase puede heredar de más de una clase:

```
class A:
    ...
    pass

class B:
    ...
    pass

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

Cuando hay métodos con el mismo nombre en las clases padre, Python usa el MRO (Method Resolution Order) para decidir cuál método usar.

Puedes ver el orden así:

```
print(C.__mro__)
```

Polimorfismo y Sobrecarga

Polimorfismo: distintas clases pueden tener métodos con el mismo nombre pero comportamientos diferentes.

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

Esta función funciona con cualquier clase que tenga un método hablar.

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

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

Dunder Methods (Métodos Mágicos)

Los dunder methods son métodos especiales que empiezan y terminan con doble guion bajo __. Por ejemplo:

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

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

¿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.

Python ofrece herramientas para capturar y manejar excepciones, evitando que tu aplicación se caiga por completo.

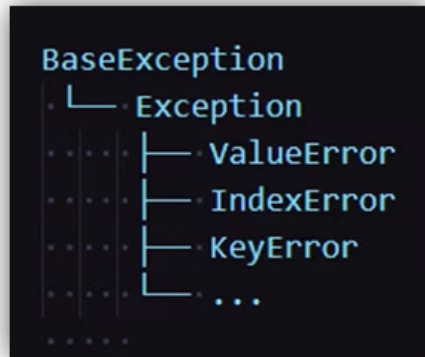
Jerarquía de excepciones

Cuando ocurre un error en tiempo de ejecución, Python lanza una excepción. Estas excepciones no son solo mensajes de error: en realidad, son objetos que pertenecen a una jerarquía de clases.

¿Qué significa jerarquía?

Piensa en esto como una familia de errores, donde todos los errores tienen un "antepasado común". En Python, ese ancestro es la clase `BaseException`.

Python tiene una jerarquía de clases para las excepciones:



Manejo de Excepciones en Python

¿Qué significa esto?

- BaseException es la clase más general de todas.
- Exception hereda de BaseException y es la clase base más común que usamos al capturar errores.
- Las demás (ValueError, IndexError, etc.) heredan de Exception.

Manejo 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

Puedes crear tus propias excepciones:

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

Buenas prácticas para manejar excepciones



Captura solo las excepciones que esperas

Evita usar except genéricos que puedan ocultar errores inesperados



Proporciona mensajes de error útiles

Los mensajes claros ayudan a identificar y solucionar problemas



No uses except: sin especificar el tipo de error

Esto puede ocultar bugs y hacer el debugging más difícil



Usa finally si necesitas cerrar archivos o liberar recursos

Garantiza que los recursos se liberen correctamente incluso si hay errores

Actividad Práctica Guiada: Herencia y Excepciones en Python

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 sugerida:

```
actividad_herencia_excepciones/  
├──  
├── src/  
│   ├── modelos.py      # Contiene las clases Usuario y Administrador  
│   ├── excepciones.py  # Contiene la clase UsuarioNoEncontrado  
│   └── main.py          # Lógica principal del programa  
├── README.md            # Instrucciones de ejecución y explicación  
└── capturas/            # Capturas de pantalla de la ejecución (opcional)
```

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 de la actividad práctica

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.

Crear una subclase Administrador que herede de Usuario

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

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

Definir una lista simulada de usuarios

```
usuarios = ["ana", "juan", "pedro"]
```

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

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.

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.

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.