



Programación Orientada a Objetos (POO)

Centro de Servicios y Gestión Empresarial
SENA Regional Antioquia



www.sena.edu.co

Contextualización

Contenido Temático:

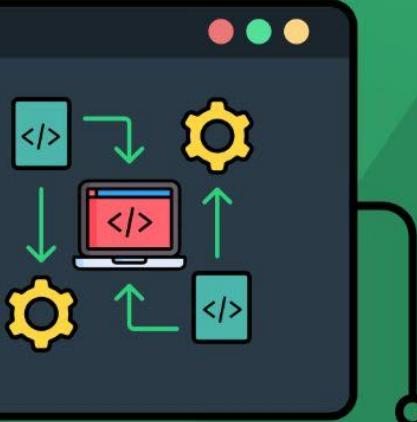


- **Fundamentos de POO**
 - Clases, atributos, métodos y objetos
 - Métodos (constructores, de instancia, de clase, estáticos y mágicos)
- **Principios fundamentales de la POO**
 - Encapsulamiento, Herencia y Polimorfismo
- **Abstracción y diseño de clases**
 - Interfaces y clases abstractas
 - Interfaces formales y virtuales
 - Clases abstractas (también llamadas clases virtuales)
 - Colecciones como estructuras abstractas de datos
- **Avanzado**
 - Uso y creación de decoradores (relevante en lenguajes como Python)
 - Manejo de excepciones

¿QUÉ SON LOS PARADIGMAS DE PROGRAMACIÓN?

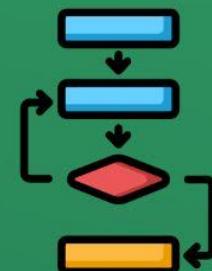


Los paradigmas son los diferentes estilos de usar la programación para resolver un problema.



PROGRAMACIÓN ESTRUCTURADA

Programación secuencial con la que todos aprendemos a programar. Usa ciclos y condicionales.



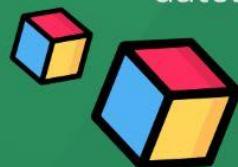
PROGRAMACIÓN REACTIVA

Observa flujos de datos asíncronos y reacciona frente a sus cambios.



PROGRAMACIÓN ORIENTADA A OBJETOS

Divide los componentes del programa en objetos que tienen datos y comportamiento y se comunican entre sí.



PROGRAMACIÓN FUNCIONAL

Divide el programa en tareas pequeñas que son ejecutadas por funciones.



Fuente:
<https://ed.team/programacion>

Programación Orientada a Objetos



La Programación Orientada a Objetos (POO) es un paradigma de programación que organiza el código en clases y objetos, inspirándose en cómo funciona el mundo real. En lugar de pensar en funciones sueltas y datos separados, en **POO** se piensa en **entidades** (objetos) que tienen **propiedades** (atributos) y **comportamientos** (métodos).

Ventajas de POO

- Es más rápido y más fácil de ejecutar
- Proporciona una estructura clara para los programas.
- Ayuda a mantener el código SECO "No se repita", y hace que el código sea más fácil de mantener, modificar y depurar
- Hace posible crear aplicaciones reutilizables completas con menos código y un tiempo de desarrollo más corto

Elementos Claves en POO

Concepto	Descripción (Python)
Clase	Plantilla o molde para crear objetos. Se define con la palabra clave <code>class</code> .
Objeto	Instancia de una clase. Se crea llamando la clase como una función: <code>obj = Clase()</code> .
Atributos	Variables que pertenecen a una clase o a un objeto. Se acceden con <code>self.atributo</code> .
Métodos	Funciones definidas dentro de una clase. Usan <code>self</code> como primer parámetro.
Encapsulamiento	Oculta detalles internos. Se usan convenciones como <code>_protégido</code> y <code>__privado</code> .
Herencia	Una clase puede heredar atributos y métodos de otra. Se indica como <code>class Hija(Padre)</code> .
Polimorfismo	Permite que diferentes clases tengan métodos con el mismo nombre pero comportamiento distinto.
Abstracción	Resalta lo esencial y oculta lo complejo. Se implementa con clases y métodos abstractos (módulo <code>abc</code>).

Clases y objetos

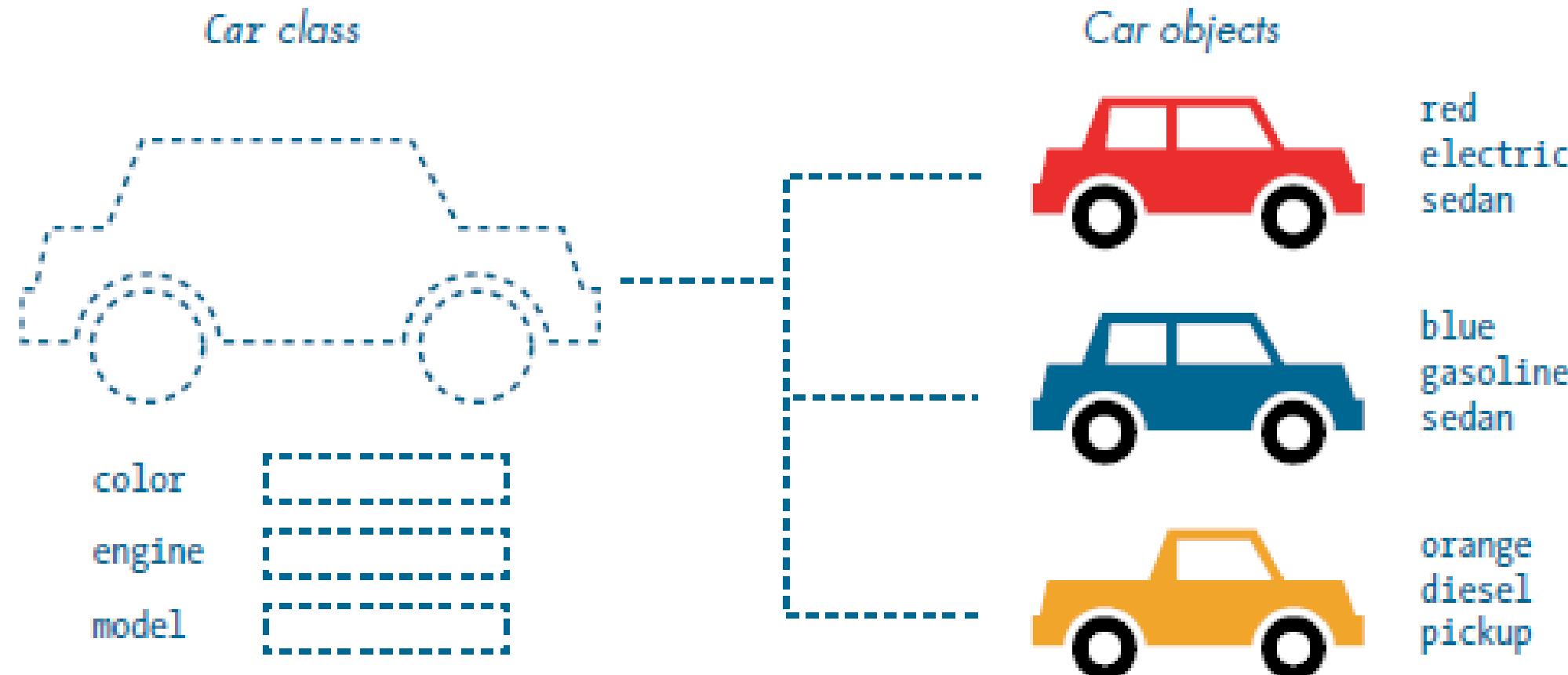
Clases, Atributos, Métodos y Objetos



La POO en Python se basa en la definición de **clases**, que actúan como **moldes o planos** para crear **objetos**. Cada **objeto** puede tener:

- **Atributos** (*también llamados propiedades o variables de instancia*): describen el estado del objeto.
- **Métodos**: *funciones asociadas* a la clase que definen el comportamiento del objeto.

Clases, Atributos, Métodos y Objetos



Clases, Atributos, Métodos y Objetos



```
# Definición de la clase Carro
class Carro:
    # Constructor de la clase Carro
    # Se definen los atributos de la clase Carro
    def __init__(self, color, motor, modelo):
        self.color = color          # Atributo
        self.motor = motor          # Atributo
        self.modelo = modelo         # Atributo

    # Método para mostrar la información del carro
    def mostrar_info(self):
        print(f"Color: {self.color} | Motor: {self.motor} | Modelo: {self.modelo}")

# Objetos: instancias de la clase Carro
carro1 = Carro("rojo", "eléctrico", "sedán")

# Mostrar información de cada carro
carro1.mostrar_info()
```

Métodos

Constructores, Instancia, Clase, Estáticos y Mágicos

Métodos

Un **método** es una *función* que está asociada a una clase y que puede actuar sobre los atributos del objeto (acceder, modificarlos, entre otros).

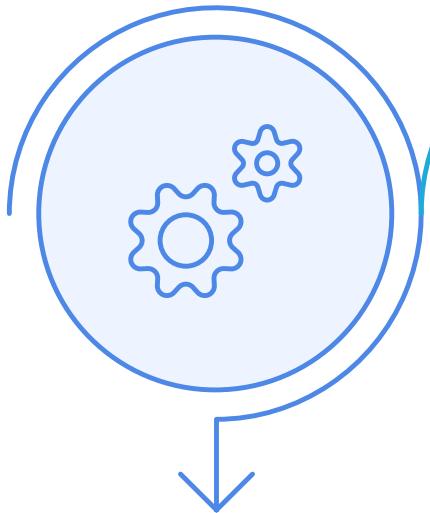
Ejemplo:

```
# Definición de la clase Carro
class Carro:
    # Constructor de la clase Carro
    def __init__(self, color, motor, modelo):
        self.color = color          # Atributo
        self.motor = motor          # Atributo
        self.modelo = modelo         # Atributo

    # Método para mostrar la información del carro
    def mostrar_info(self):
        print(f"Color: {self.color} | Motor: {self.motor} | Modelo: {self.modelo}")
```

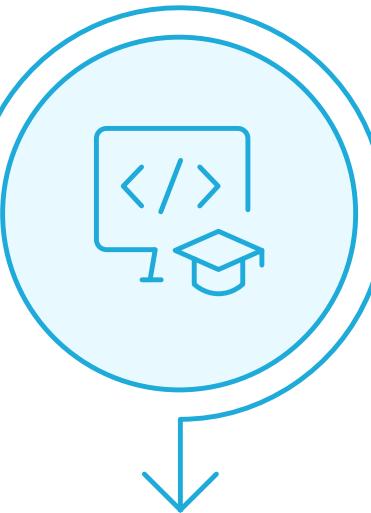


Tipos de Métodos



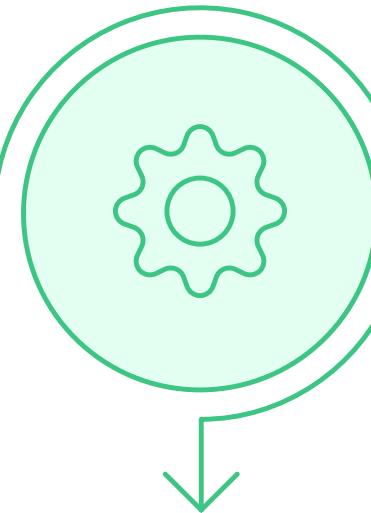
Método de Instancia

Opera sobre una instancia específica (usa `self`).



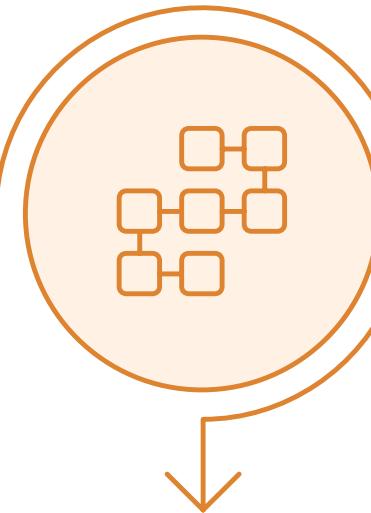
Método de Clase

Opera sobre la clase misma (usa `@classmethod` y `cls`).



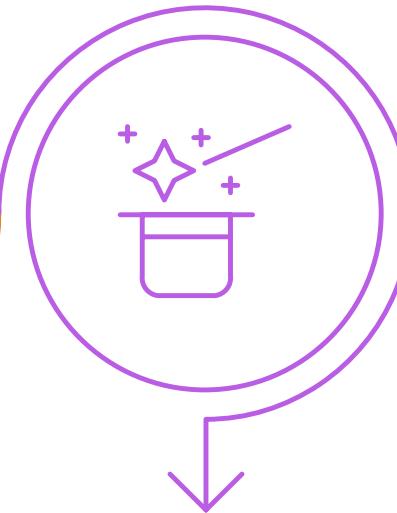
Método Estático

No depende ni de instancia ni de clase. (usa `@staticmethod`)



Método Constructor

Se llama automáticamente al crear un objeto (`__init__`).



Métodos Mágicos

Métodos especiales con dobles guiones bajos (`__str__`, `__len__`, `__add__`, etc.).

Tipos de Métodos



```
class Auto:

    # Método constructor: se ejecuta al crear un objeto
    def __init__(self, color, motor, modelo):
        self.color = color          # Atributo de instancia
        self.motor = motor
        self.modelo = modelo

    # Método de instancia: usa 'self' para acceder a atributos del objeto
    def mostrar_info(self):
        print(f"Auto: {self.color}, Motor: {self.motor}, Modelo: {self.modelo}")

    # Método de clase: usa 'cls', actúa sobre la clase, no sobre instancias
    @classmethod
    def auto_estandar(cls):
        return cls("gris", "gasolina", "sedán")
```

Tipos de Métodos



```
class Auto:

    # Método constructor: se ejecuta al crear un objeto
    def __init__(self, color, motor, modelo):
        self.color = color          # Atributo de instancia
        self.motor = motor
        self.modelo = modelo

    # Método estático: no usa ni 'self' ni 'cls'
    @staticmethod
    def es_modelo_deportivo(modelo):
        return modelo.lower() in ["coupe", "convertible", "sport"]

    # Método mágico: define cómo se representa el objeto como texto
    def __str__(self):
        return f"{self.color.capitalize()} {self.modelo} con motor {self.motor}"
```



Decoradores

Decoradores comunes en Python

Decorador	¿Qué hace?
@property	Convierte un método en un atributo de solo lectura (permite acceso como obj.atributo sin paréntesis).
@staticmethod	Indica que el método no necesita self ni cls . Se comporta como una función "dentro" de la clase.
@classmethod	Indica que el método recibe cls en lugar de self , útil para crear instancias alternativas o acceder/modificar propiedades de clase.

Principios fundamentales de la POO

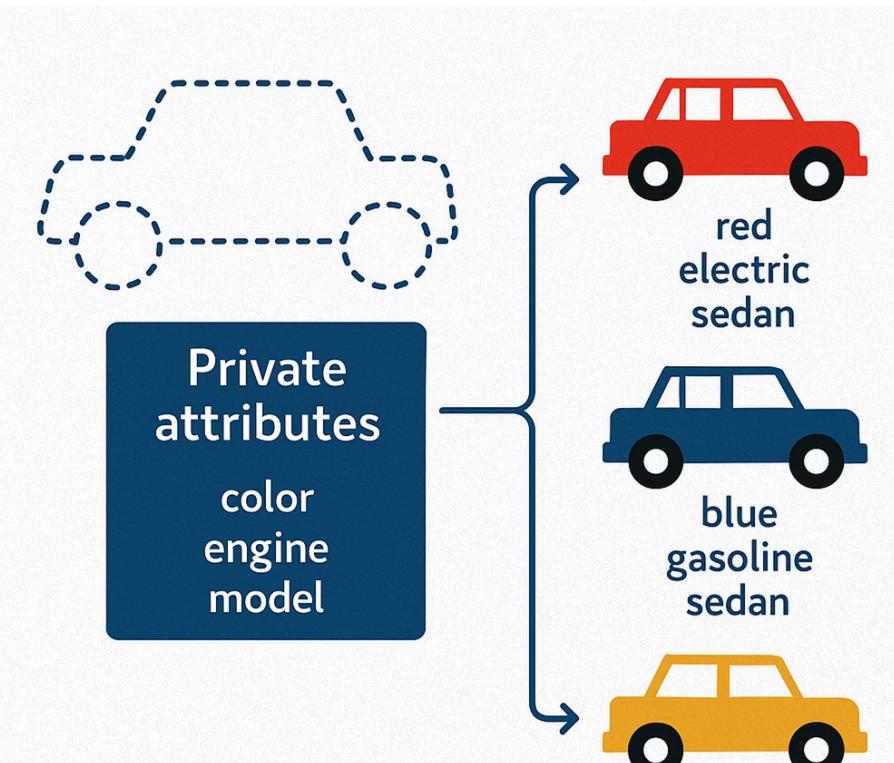
Encapsulamiento, Herencia y Polimorfismo

Encapsulamiento

Es el principio que permite **ocultar los detalles internos** de un objeto y proteger sus datos, exponiendo solo lo necesario a través de métodos públicos.

En Python, esto se logra usando convenciones como:

- **_atributo**: protegido (convención).
- **__atributo**: privado (name mangling).



Encapsulamiento

```
class Auto:
    def __init__(self, color, motor, modelo):
        self.color = color
        self.motor = motor
        self.modelo = modelo
```

```
# Crear un auto
auto = Auto("rojo", "eléctrico", "sedán")

# Modificar atributos
auto.motor = "piedra"
auto.color = ""
```



```
class Auto:
    def __init__(self, color, motor, modelo):
        self.__color = color
        self.__motor = motor
        self.__modelo = modelo

    # Propiedad para acceder al color (getter)
    @property
    def color(self):
        return self.__color

    # Propiedad para modificar el color (setter)
    @color.setter
    def color(self, color):
        if color: # Validación simple
            self.__color = color
        else:
            print("Color no válido")
```



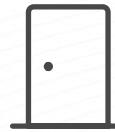
Encapsulamiento



! Control de estado limitado



Sin validación de datos



Atributos expuestos



Sin Encapsulamiento



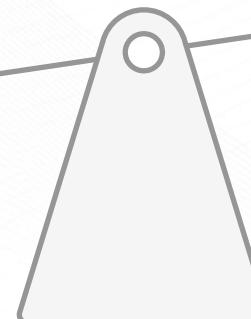
Mayor control de estado



Validación de datos



Atributos protegidos

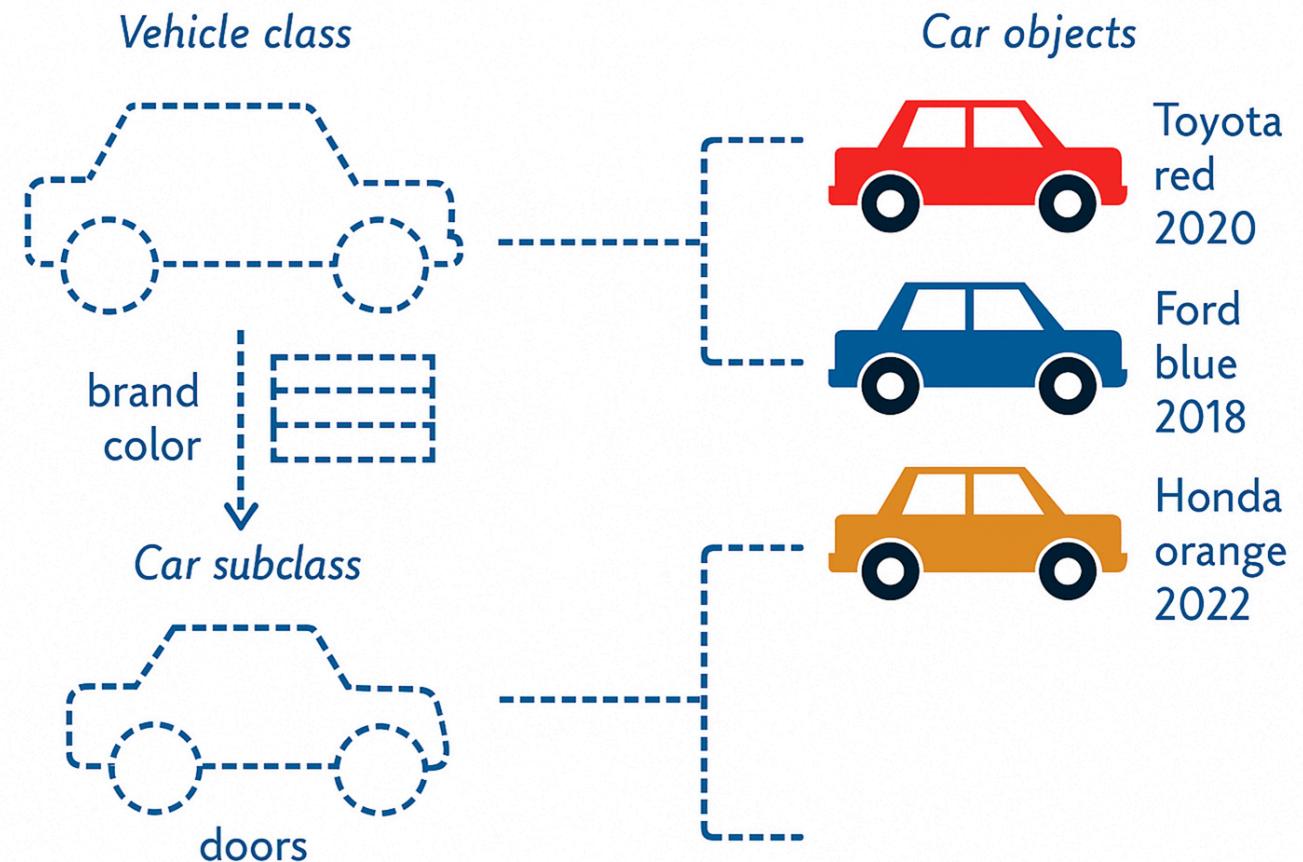


Con Encapsulamiento

Comparando la seguridad y el control en el diseño de objetos.

Herencia

Permite que una clase herede **atributos** y **métodos** de otra. La clase que hereda se llama **subclase**, y la clase base es la **superclase**.



Herencia

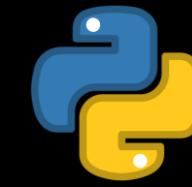
```
# Clase base
class Auto:
    def __init__(self, color, motor):
        self.color = color
        self.motor = motor

    def descripcion(self):
        return f"Color: {self.color}, Motor: {self.motor}"

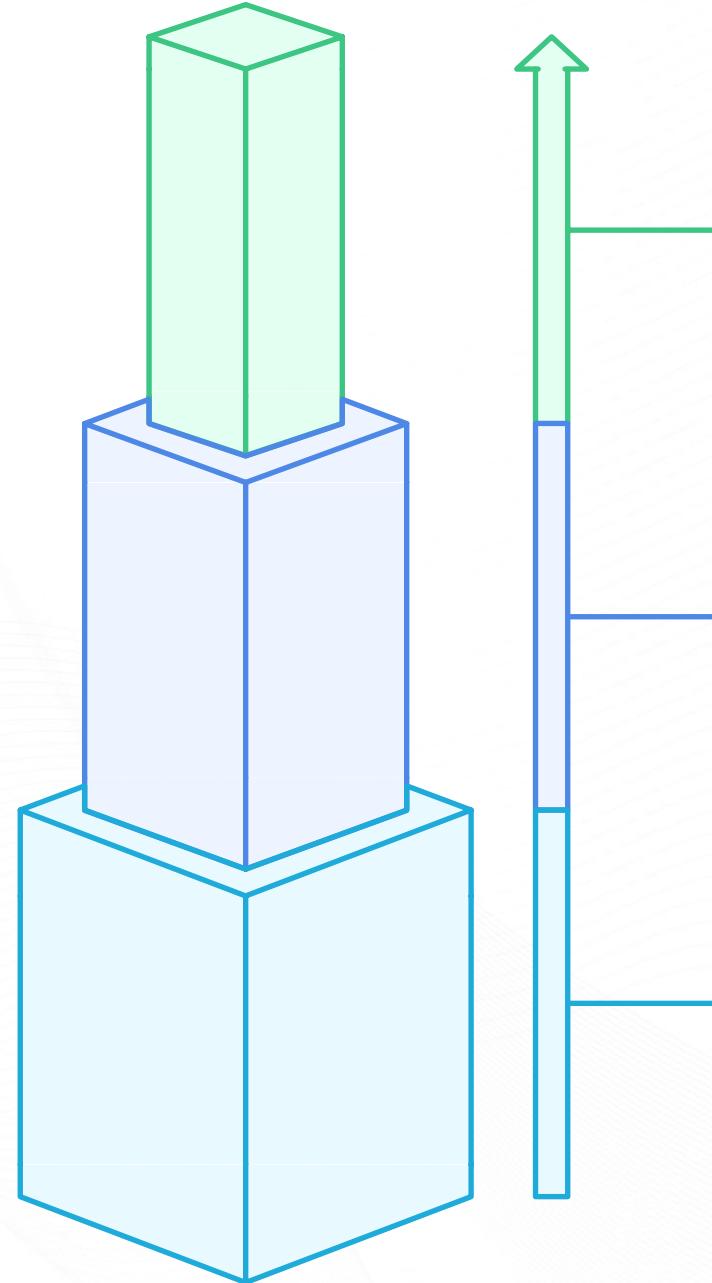
# Subclase: Automóvil (2 puertas, rápido)
class Automovil(Auto):
    def __init__(self, color, motor):
        super().__init__(color, motor) # Hereda atributos de Auto
        self.puertas = 2

    def velocidad(self):
        return "Este automóvil es rápido."

    def descripcion(self):
        base = super().descripcion()
        return f"{base}, Puertas: {self.puertas}. {self.velocidad()}"
```



Herencia



Reutilización de Código

Permite a las clases heredar atributos y métodos

Jerarquías de Clases

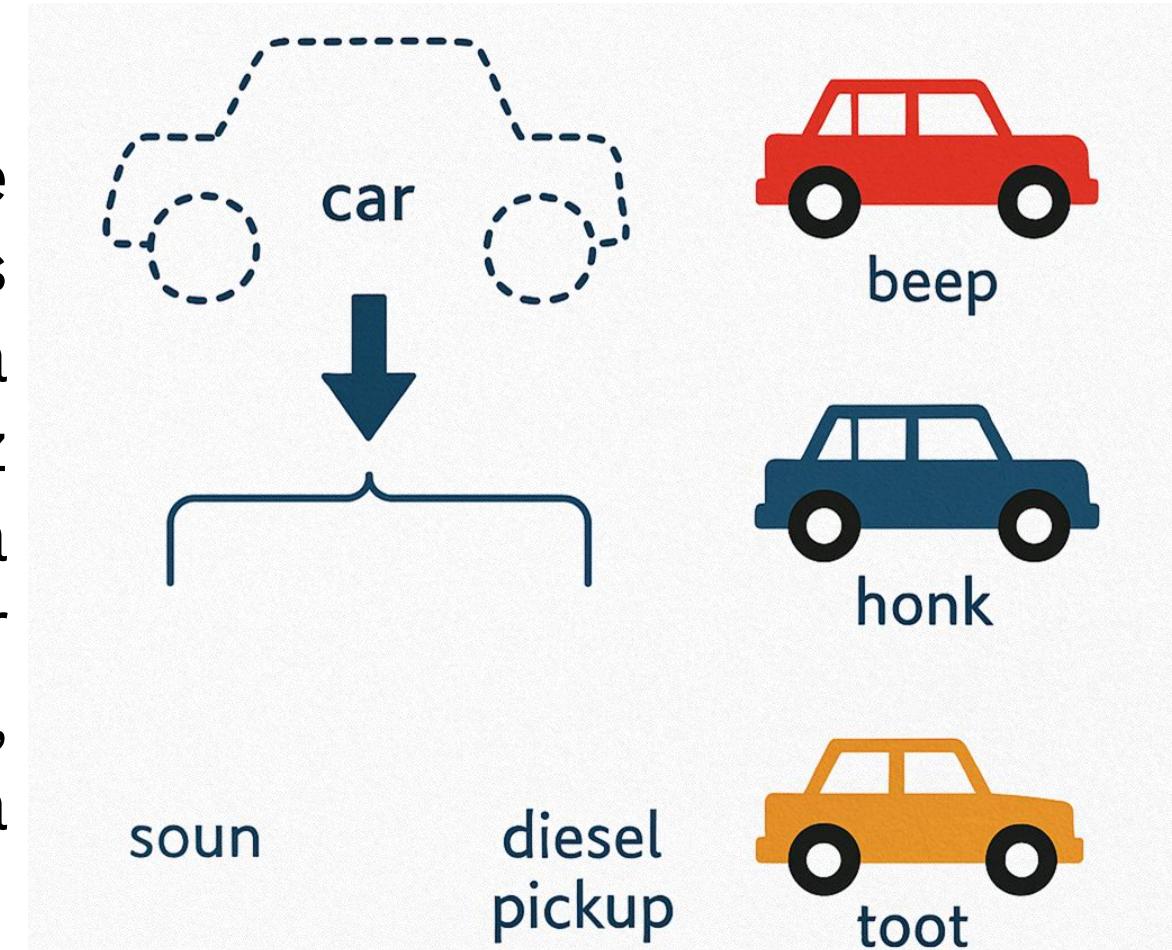
Simplifica el diseño y la implementación de programas complejos

Diseño Simplificado

Mejora la claridad y eficiencia del desarrollo de software

Polimorfismo

Se refiere a la capacidad de diferentes clases de ser tratadas como instancias de la misma clase a través de una interfaz común. Esto permite que una función o método pueda operar en diferentes tipos de objetos, facilitando la extensibilidad y la reutilización del código.



Polimorfismo

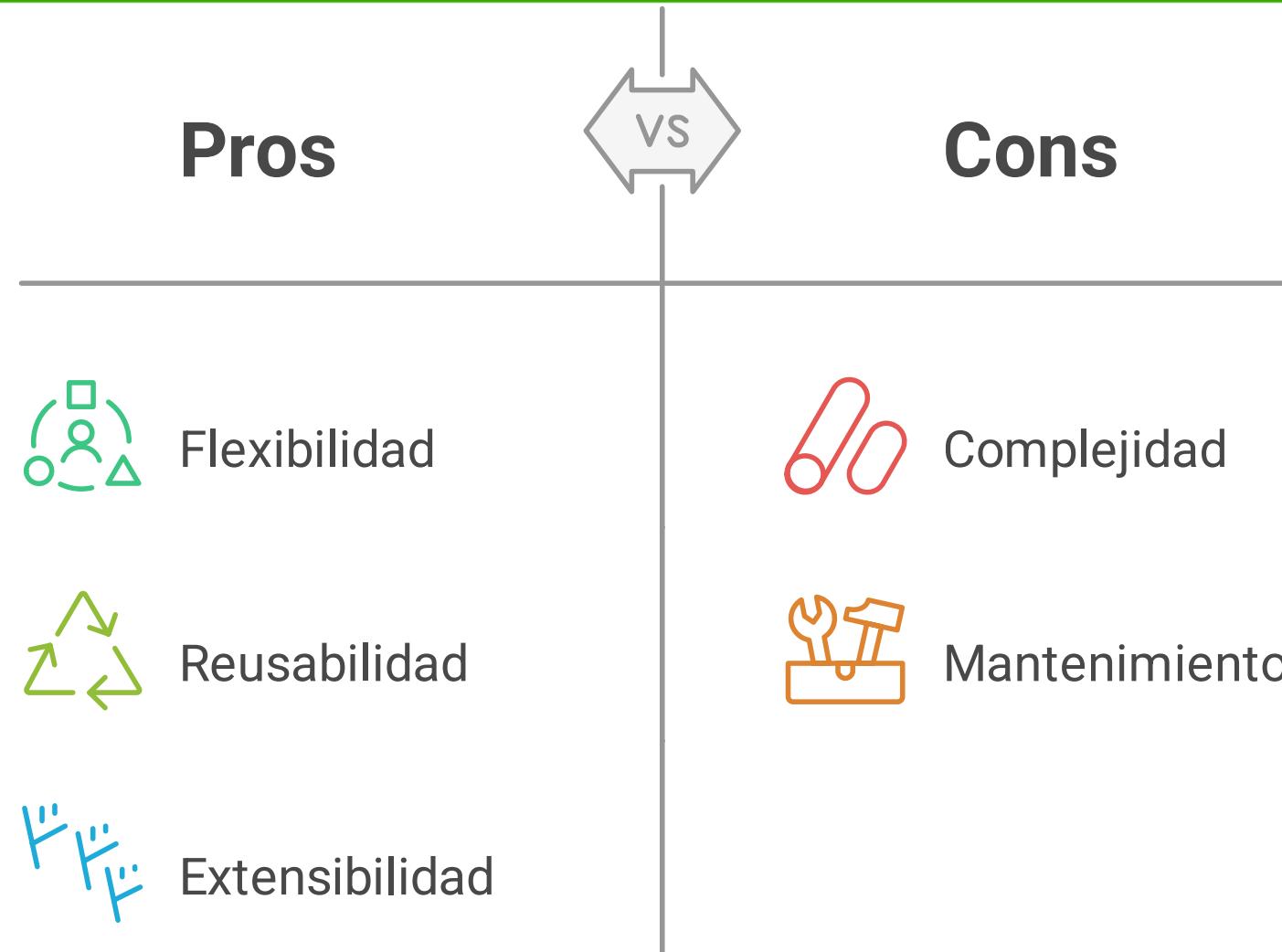
```
# Subclases que sobreescriben el método  
sonar()  
class SedanElectrico(Auto):  
    def sonar(self):  
        return "Beep"
```

```
# Clase base  
class Auto:  
    def __init__(self, color, motor, modelo):  
        self.color = color  
        self.motor = motor  
        self.modelo = modelo  
  
    def sonar(self):  
        return "Sonido genérico de auto"
```

```
# Subclases que sobreescriben el método  
sonar()  
class SedanGasolina(Auto):  
    def sonar(self):  
        return "Honk"
```



Polimorfismo



Interfaces / clases abstractas

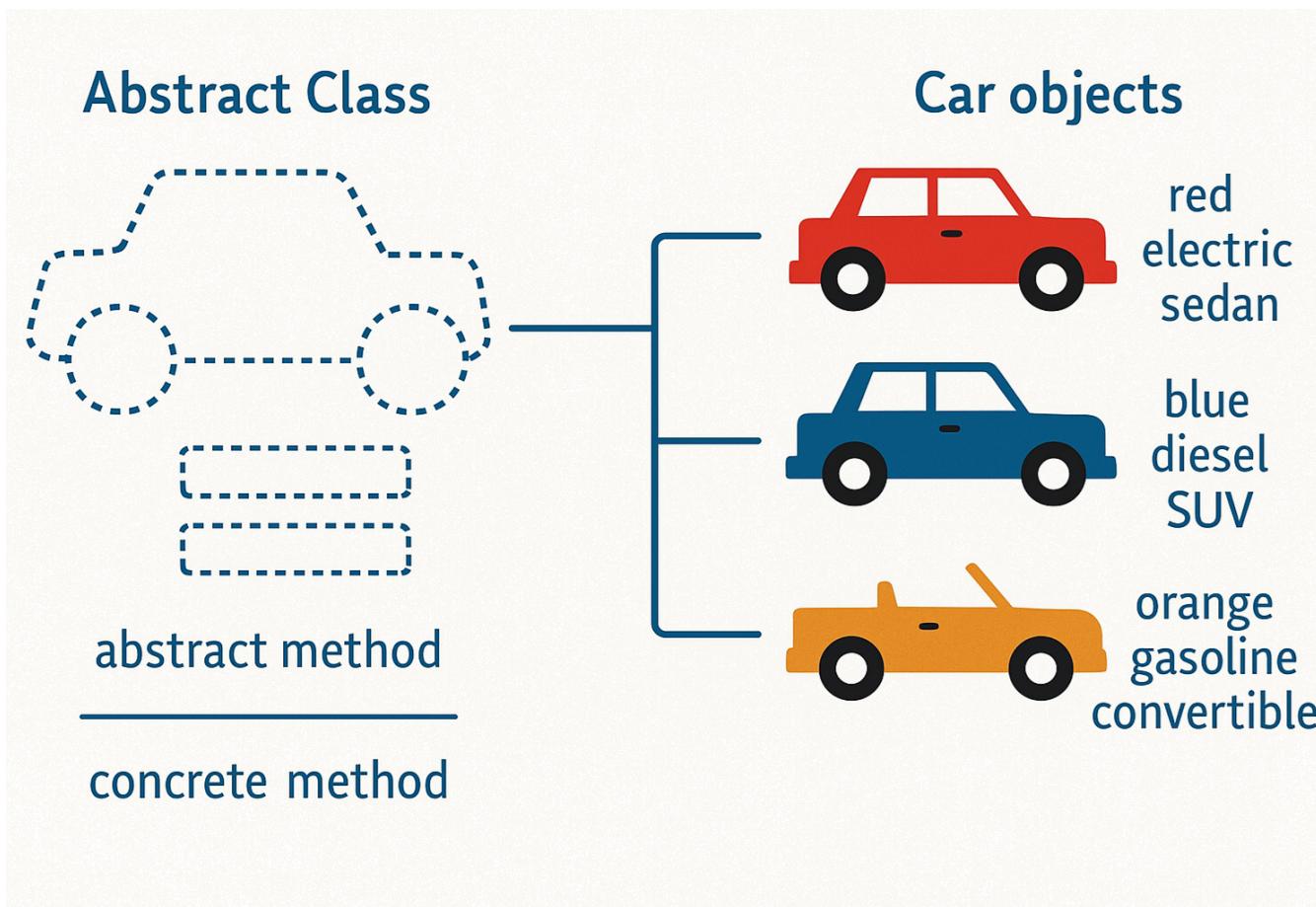
Interfaces / clases abstractas

En Python, las clases abstractas y las interfaces se implementan a través del módulo **abc** (*Abstract Base Classes*) y funcionan como estructuras base que definen un contrato obligatorio para las clases hijas.

No pueden ser instanciadas directamente y contienen métodos abstractos (sin implementación) y opcionalmente métodos concretos (con lógica definida).

Estos mecanismos aseguran la implementación coherente de comportamientos comunes, facilitando el polimorfismo, la modularidad y la escalabilidad en el diseño orientado a objetos.

Interfaces / clases abstractas



- Una **clase abstracta** es una clase que:
- No se puede **instanciar directamente**.
 - Sirve como **modelo base** para otras clases.
 - Puede contener:
 - **Métodos abstractos** (que deben ser implementados por las subclases).
 - **Métodos concretos** (ya implementados).

Interfaces / clases abstractas

```
from abc import ABC, abstractmethod  
  
# Clase abstracta  
class Vehiculo(ABC):  
    # Método abstracto que deben implementar las  
    # subclases  
    @abstractmethod  
    def sonar(self):  
        pass  
  
# Subclases que implementan el método  
# abstracto  
class SedanElectrico(Vehiculo):  
    def sonar(self):  
        return "Beep"  
  
class SuvDiesel(Vehiculo):  
    def sonar(self):  
        return "Honk"
```



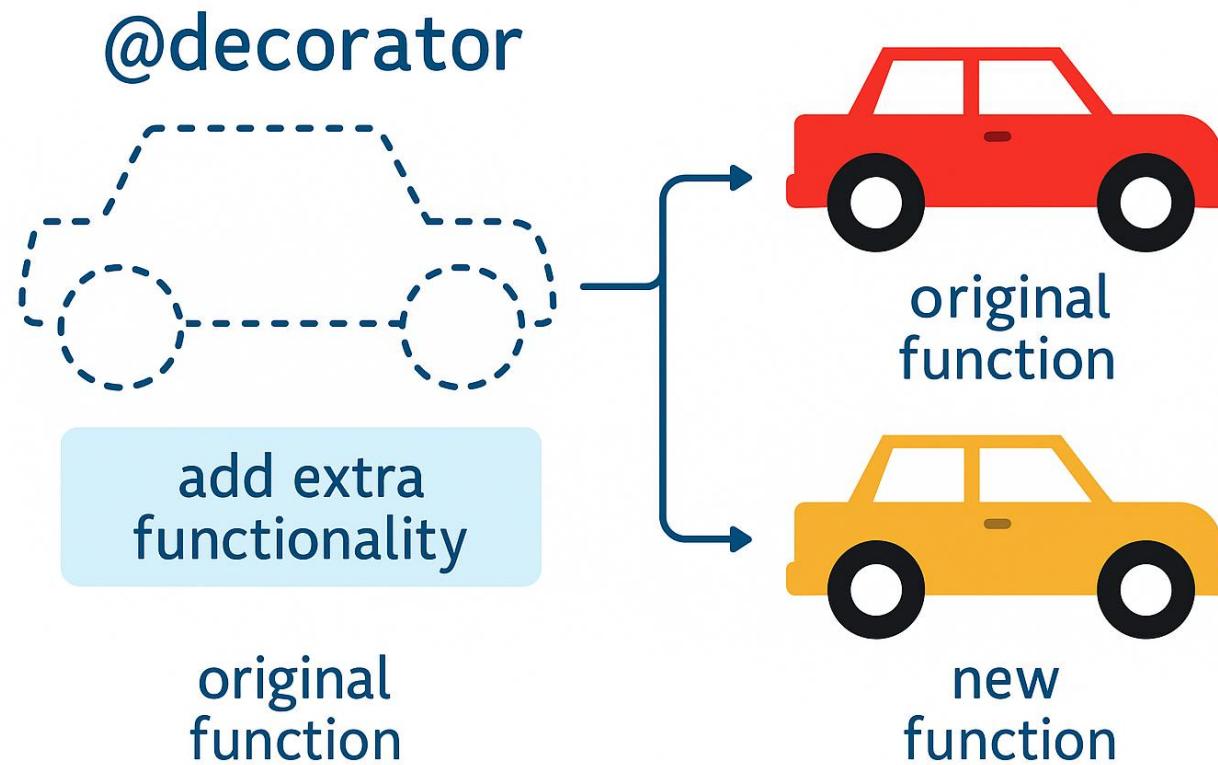
Decoradores y Manejo de excepciones

Decoradores

Un **decorador** en Python es una función que envuelve a otra función o método para modificar o extender su comportamiento sin alterar su código original.

Se implementa como una función que recibe otra función como argumento, y devuelve una nueva función con funcionalidades adicionales. Se utiliza comúnmente con el **símbolo @** justo antes de la definición de una función o método.

Decoradores



¿Para qué sirven?

- Añadir funcionalidad sin alterar la función original
- Control de acceso, verificación de permisos
- Registro de logs, medición de tiempos
- Decoradores mas comunes como `@staticmethod`, `@classmethod`, `@property`

Decoradores

```
# Sintaxis:  
# @decorador  
def decorador(funcion):  
    def funcion_decorada():  
        # Código adicional  
        funcion()  
        # Código adicional  
    return funcion_decorada  
  
# Uso del decorador  
@decorador  
def funcion():  
    # Código de la función original  
    pass  
  
# Llamada a la función decorada  
funcion()
```

```
# Ejemplo de decorador No. 1  
def decorador(funcion):  
    def funcion_decorada():  
        print("Antes de la función")  
        funcion()  
        print("Después de la función")  
    return funcion_decorada  
  
# Aplicando el decorador a una función  
@decorador  
def saludar():  
    print("Hola, mundo!")  
  
# Llamando a la función decorada  
saludar()
```



Manejo de Excepciones

Python utiliza **try-except** para manejar errores y excepciones en **tiempo de ejecución**, evitando que el programa se **detenga abruptamente**.

Sintaxis:

```
try:  
    # Código que puede generar un error  
except Exception as e:  
    # Código que se ejecuta si ocurre la excepción  
else:  
    # Código que se ejecuta si no ocurre la excepción  
finally:  
    # Código que se ejecuta siempre, haya o no ocurrido la excepción
```

Manejo de Excepciones

```
class Division:  
    def dividir(self, dividendo, divisor):  
        try:  
            resultado = dividendo / divisor  
        except ZeroDivisionError:  
            mensaje = "Error: división por cero"  
        except TypeError:  
            mensaje = "Error: tipos de datos incompatibles"  
        else:  
            mensaje = f"Resultado: {resultado}"  
        finally:  
            mensaje += " \nOperación de división finalizada \n"  
        return mensaje  
  
calc = Division()  
print(calc.dividir(10, 2)) # 5.0  
print(calc.dividir(10, 0)) # Error ZeroDivisionError  
print(calc.dividir("sdf", 8)) # Error TypeError
```

Manejo de Excepciones

En Python, se utiliza **raise** para lanzar excepciones personalizadas.

Sintaxis:

```
# Crear excepciones personalizadas
raise NombreDeLaExpcion("Mensaje de error personalizado")
```

```
class Division:  
    def dividir(self, dividendo, divisor):  
        try:  
            if dividendo < 0 or divisor < 0:  
                raise ValueError("Los números deben ser positivos")  
            resultado = dividendo / divisor  
        except ZeroDivisionError:  
            mensaje = "Error: división por cero"  
        except TypeError:  
            mensaje = "Error: tipos de datos incompatibles"  
        except ValueError as e:  
            mensaje = f"Error: {e}"  
        else:  
            mensaje = f"Resultado: {resultado}"  
        finally:  
            mensaje += " \nOperación de división finalizada \n"  
        return mensaje  
  
calc = Division()  
print(calc.dividir(10, 2)) # 5.0  
print(calc.dividir(10, 0)) # Error ZeroDivisionError  
print(calc.dividir("sdf", 8)) # Error TypeError  
print(calc.dividir(10, -3))
```

Manejo de Excepciones

Excepción	Causa Común
ZeroDivisionError	División entre cero.
ValueError	Conversión de tipo inválida.
TypeError	Operaciones entre tipos incompatibles.
IndexError	Acceso a un índice fuera de rango.
KeyError	Acceso a una clave inexistente en un diccionario.
AttributeError	Uso de un método/atributo inexistente en un objeto.
FileNotFoundException	Archivo no encontrado.
ImportError	Módulo no encontrado.
MemoryError	Uso excesivo de memoria.
RecursionError	Llamada recursiva sin fin.
Exception	Errores controlados por el usuario – reglas de negocio

Manejo de Excepciones

Bloque	Función	Ejemplo
try:	Contiene el código que puede generar un error.	try: resultado = 10 / 0
except TipoError:	Captura errores específicos (ej. ZeroDivisionError).	except ZeroDivisionError:
except Exception as e:	Captura cualquier error y lo almacena en e.	except Exception as e: print(e)
else:	Se ejecuta solo si no hay errores en el try.	else: print("Operación exitosa")
finally:	Se ejecuta siempre, ocurra o no un error.	finally: print("Ejecución finalizada.")
raise TipoError("Mensaje")	Lanza manualmente una excepción.	raise ValueError("Error personalizado")



G R A C I A S

Presentó: Alvaro Pérez Niño

Instructor Técnico

Correo: aperezn@sena.edu.co

<http://centrodesserviciosygestionempresarial.blogspot.com/>

Línea de atención al ciudadano: 01 8000 910270

Línea de atención al empresario: 01 8000 910682



www.sena.edu.co