ORS4TECH

POO DESDE CERO

— Guía Complementaria

al Curso de Automatización y Scripting
con Python para Administración de Sistemas

POO desde Cero – Guía Complementaria al "Curso de Automatización y Scripting con Python para Administración de Sistemas"

Este es un curso **complementario**, creado como apoyo al curso principal de **Automatización y Scripting con Python para Administración de Sistemas** (el que compartí junto a este).

Pero ojo, te lo digo desde ya

Si no tienes una base en Programación Orientada a Objetos (POO) o sientes que esos conceptos te cuestan, **te recomiendo empezar por aquí primero**.

Este material fue diseñado especialmente para ayudarte a comprender, paso a paso, cómo funciona la lógica de clases, objetos, atributos, métodos, herencia, encapsulamiento y más. Todo explicado con ejemplos prácticos, claros y aplicables al mundo real.

Como mencioné en el curso principal:

"No necesitas memorizar todo. Lo importante es que entiendas cómo funciona."

Y eso es lo que harás aquí: **entender**, no solo copiar.

Si no planeas dedicarte 100% a la programación, pero sí quieres usar Python como herramienta como lo hacen muchos profesionales en IT, ciberseguridad, analistas de datos o entusiastas de la inteligencia artificial entonces esta base te va a dar el empujón que necesitas.

Así que si estás comenzando o te falta confianza en la parte de objetos y clases, **este es el mejor punto de partida**. Cuando termines aquí, ahora sí, lánzate de lleno al otro curso con más seguridad y herramientas.

Herramientas recomendadas para el curso

Modelo GPT personalizado y entrenado exclusivamente para este curso:

https://chatgpt.com/g/g-682898e5b33c8191b022b82760f4a671-kira4tech

Nota sobre el uso del modelo de IA: Este asistente está basado en un modelo de inteligencia artificial tipo GPT, configurado y entrenado específicamente con el contenido de esta guía. Aun así, como toda IA, puede interpretar mal algunas preguntas o responder fuera de contexto. Si detectas alguna respuesta incorrecta, ambigua o que no refleja el contenido del curso, **por favor notifícalo** para poder ajustar y mejorar el modelo.

Antes de comenzar, vamos a instalar algunas herramientas que serán de **gran utilidad** durante todo el curso:

Visual Studio Code:

Editor de código liviano, personalizable y perfecto para programar en Python.

Extensiones que debes instalar:

- **Python y Python Debugger** (Microsoft)Permiten ejecutar y depurar scripts de Python directamente en VS Code.
- **Spanish Language Pack for Visual Studio Code** (Microsoft) Traduce la interfaz de VS Code al español.
- **Windsurf.** Una inteligencia artificial que te ayuda con el autocompletado del código. La versión gratuita es más que suficiente.
- **Rainbow CSV.** Colorea automáticamente las columnas de archivos .csv para facilitar su lectura y edición.
- **Pylance.** Mejora la experiencia de desarrollo en Python con sugerencias más rápidas e inteligentes.
- **Prettier - Code formatter.** Da formato automáticamente a tu código para que se vea limpio y organizado.
- **indent-rainbow.** Colorea los niveles de indentación para que sea más fácil identificar estructuras en el código.

Iniciar sesión en VS Code y Windsurf

- **Inicia sesión en Visual Studio Code con tu cuenta de Outlook (o Hotmail)**. Esto sirve para sincronizar tus extensiones y configuraciones, lo cual es muy útil si cambias de PC en el futuro.
- **Crea una cuenta en Windsurf** (o inicia sesión con Google/GitHub) para activar el autocompletado. No hace falta pagar nada, la versión gratuita es bastante potente.

Para tomar notas…

Puedes usar el editor que prefieras, pero personalmente te recomiendo **Obsidian**, que es rápido, ligero y perfecto para organizar notas técnicas.

Índice

Capítulo 1: Introducción a la Programación Orientada a Objetos

Parte 1: Conceptos Básicos y Sintaxis

1.1 Conceptos Básicos

La Programación Orientada a Objetos (POO) es un paradigma de programación basado en la idea de "objetos", que pueden contener datos y código: datos en forma de campos (a menudo conocidos como atributos o propiedades) y código en forma de procedimientos (a menudo conocidos como métodos).

Los conceptos fundamentales de la POO son:

- **Clases**: Plantillas para crear objetos. Definen un conjunto de atributos y métodos que los objetos creados a partir de la clase tendrán.
- **Objetos**: Instancias de clases. Representan entidades concretas con atributos específicos y comportamientos definidos por sus métodos.
- **Atributos**: Datos almacenados en objetos.
- **Métodos**: Funciones definidas dentro de una clase que operan sobre los atributos del objeto.

1.2 Clases y Objetos

En Python, una clase se define usando la palabra clave `class`. Aquí hay un ejemplo básico de una clase `Persona`:

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

    def saludar(self):
        return f"Hola, mi nombre es {self.nombre} y tengo {self.edad}
años."
```

Podemos crear objetos de esta clase de la siguiente manera:

```
persona1 = Persona("Juan", 30)
print(persona1.saludar())
```

Explicación del código:

- `class Persona`: Define una clase llamada `Persona`.
- `def __init__(self, nombre, edad)`: Define el método inicializador que asigna los valores de `nombre` y `edad` a los atributos del objeto.
- `def saludar(self)`: Define un método que devuelve un saludo con el nombre y la edad de la persona.

1.3 Sintaxis Básica de Clases en Python

La definición de clases en Python sigue una sintaxis específica:

```
class NombreDeLaClase:
    def __init__(self, parametros):
        # Inicialización de atributos
        self.atributo1 = valor1
        self.atributo2 = valor2

    def metodo(self, parametros):
        # Implementación del método
        pass
```

Ejemplo:

```
class Coche:
    def __init__(self, marca, modelo, año):
        self.marca = marca
        self.modelo = modelo
        self.año = año

    def describir(self):
        return f"Este coche es un {self.marca} {self.modelo} del año
{self.año}."
```

Instanciación y uso:


```
mi_coche = Coche("Toyota", "Corolla", 2020)
print(mi_coche.describir())
```

Parte 2: Ejercicios Prácticos

Ahora que hemos cubierto los conceptos básicos y la sintaxis de las clases en Python, vamos a poner en práctica lo aprendido con algunos ejercicios prácticos.

Ejercicio 1: Sistema de Gestión de Libros

Define una clase `Libro` que tenga los atributos `titulo`, `autor`, y `año_publicacion`. Añade un método `descripcion` que devuelva una cadena con la información del libro.

```
class Libro:
    def __init__(self, titulo, autor, año_publicacion):
        self.titulo = titulo
        self.autor = autor
        self.año_publicacion = año_publicacion

    def descripcion(self):
        return f"'{self.titulo}' por {self.autor}, publicado en
{self.año_publicacion}"
```

Prueba el código creando instancias de `Libro` y llamando al método `descripcion`.

```
libro1 = Libro("Cien Años de Soledad", "Gabriel García Márquez", 1967)
libro2 = Libro("1984", "George Orwell", 1949)
```

```
print(libro1.descripcion())
print(libro2.descripcion())
```

Ejercicio 2: Sistema de Gestión de Usuarios

Define una clase `Usuario` que tenga los atributos `nombre_usuario`, `email`, y `contraseña`. Añade un método `mostrar_informacion` que devuelva una cadena con la información del usuario, excluyendo la contraseña.

```
class Usuario:
    def __init__(self, nombre_usuario, email, contraseña):
        self.nombre_usuario = nombre_usuario
        self.email = email
        self.contraseña = contraseña

    def mostrar_informacion(self):
        return f"Usuario: {self.nombre_usuario}, Email: {self.email}"
```

Prueba el código creando instancias de `Usuario` y llamando al método `mostrar_informacion`.

```
usuario1 = Usuario("juanperez", "juan@example.com", "1234")
usuario2 = Usuario("mariagomez", "maria@example.com", "abcd")

print(usuario1.mostrar_informacion())
print(usuario2.mostrar_informacion())
```

Parte 3: Ejercicios para Realizar

Ahora es tu turno de poner en práctica lo que has aprendido. Aquí tienes algunos ejercicios para realizar por tu cuenta.

Ejercicio 1: Sistema de Gestión de Productos

Define una clase `Producto` que tenga los atributos `nombre`, `precio`, y `cantidad`. Añade un método `mostrar_informacion` que devuelva una cadena con la información del producto.

Ejercicio 2: Sistema de Gestión de Vehículos

Define una clase `Vehiculo` que tenga los atributos `marca`, `modelo`, y `año`. Añade un método `descripcion` que devuelva una cadena con la información del vehículo.

Cuando termines los ejercicios, asegúrate de probar tu código creando instancias de las clases y llamando a los métodos correspondientes.

Capítulo 2: Atributos y Métodos

Parte 1: Tipos de Atributos y Métodos

2.1 Atributos de Instancia y de Clase

Atributos de Instancia:

- Los atributos de instancia son variables que pertenecen a una instancia específica de una clase.
- Se definen dentro del método `__init__` usando `self`.

Ejemplo:

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre  # Atributo de instancia
        self.edad = edad      # Atributo de instancia
```

En este caso:

- `nombre` y `edad` son atributos de instancia.
- Cada instancia de `Persona` tendrá sus propios valores para `nombre` y `edad`.

Cuando creamos una instancia de `Persona`, le asignamos valores específicos a estos


```
persona1 = Persona("Juan", 30)
persona2 = Persona("Ana", 25)

print(persona1.nombre)  # Salida: Juan
print(persona2.nombre)  # Salida: Ana
```

Aquí, `persona1` y `persona2` tienen sus propios valores independientes para `nombre` y `edad`.

Atributos de Clase:

- Los atributos de clase son variables que pertenecen a la clase en sí misma y son compartidos por todas las instancias de la clase.
- Se definen directamente dentro de la clase, fuera de cualquier método.

Ejemplo:

```
class Persona:
    especie = "Homo sapiens"  # Atributo de clase

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

En este caso:

- `especie` es un atributo de clase, compartido por todas las instancias de `Persona`.

Podemos acceder a los atributos de clase a través de la clase o de una instancia:

```
print(Persona.especie)  # Salida: Homo sapiens

persona1 = Persona("Juan", 30)
print(persona1.especie)  # Salida: Homo sapiens
```

Si cambiamos el valor del atributo de clase, afectará a todas las instancias:

```
Persona.especie = "Homo erectus"
print(persona1.especie)  # Salida: Homo erectus
```

2.2 Métodos de Instancia y de Clase

Métodos de Instancia:

- Los métodos de instancia operan sobre una instancia específica de la clase y pueden acceder a los atributos y otros métodos de la instancia usando `self`.

Ejemplo:

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

    def saludar(self):
        return f"Hola, mi nombre es {self.nombre}"
```


Aquí, `saludar` es un método de instancia que utiliza el atributo `nombre` de la instancia.

Métodos de Clase:

- Los métodos de clase operan sobre la clase en sí misma, en lugar de sobre instancias específicas.
- Se definen usando el decorador `@classmethod` y reciben `cls` como primer parámetro en lugar de `self`.

Ejemplo:

```
class Persona:
    especie = "Homo sapiens"

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

    @classmethod
    def cambiar_especie(cls, nueva_especie):
        cls.especie = nueva_especie
```

Aquí, `cambiar_especie` es un método de clase que modifica el atributo de clase `especie`.

Para llamar a un método de clase:

```
Persona.cambiar_especie("Homo erectus")
print(Persona.especie)  # Salida: Homo erectus
```

Métodos Especiales `__str__` y `__repr__`

Objetivo de `__str__`:

- `__str__` define cómo se debe representar el objeto cuando se imprime o se convierte en una cadena usando la función `str()`.
- Su objetivo principal es proporcionar una representación legible y amigable del objeto, útil para mostrar al usuario final.

Ejemplo:

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

    def __str__(self):
        return f"{self.nombre}, {self.edad} años"

persona1 = Persona("Juan", 30)
print(persona1)  # Salida: Juan, 30 años
```


- Aquí, cuando imprimes `persona1`, Python llama automáticamente a `persona1.__str__()` para obtener una representación en cadena del objeto.
- El resultado es `Juan, 30 años`.

Objetivo de `__repr__`:

- `__repr__` proporciona una representación oficial del objeto.
- Su objetivo es devolver una cadena que, idealmente, pueda ser usada para recrear el objeto. Es útil para la depuración y para los desarrolladores.

Ejemplo:

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

    def __repr__(self):
        return f"Persona('{self.nombre}', {self.edad})"

persona1 = Persona("Juan", 30)
print(repr(persona1))  # Salida: Persona('Juan', 30)
```

- Aquí, cuando usas `repr(persona1)`, Python llama a `persona1.__repr__()` para obtener una representación oficial del objeto.
- El resultado es `Persona('Juan', 30)`.

Diferencia y Uso

- Usa `__str__` cuando quieras una representación legible para los usuarios finales.
- Usa `__repr__` cuando quieras una representación detallada y precisa para los desarrolladores.

Varios Métodos en la Clase y el Método `__str__`

Cuando tienes múltiples métodos en una clase, `__str__` se usa específicamente para definir cómo se debe imprimir la instancia de la clase. No afecta otros métodos. Veamos un ejemplo con más de un método:

Ejemplo con Varios Métodos:

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

    def __str__(self):
        return f"{self.nombre}, {self.edad} años"

    def saludar(self):
        return f"Hola, mi nombre es {self.nombre}"

    def cumplir_anios(self):
        self.edad += 1
        return f"Feliz cumpleaños {self.nombre}, ahora tienes
{self.edad} años"

persona1 = Persona("Juan", 30)
print(persona1)  # Salida: Juan, 30 años
print(persona1.saludar())  # Salida: Hola, mi nombre es Juan
print(persona1.cumplir_anios())  # Salida: Feliz cumpleaños Juan,
ahora tienes 31 años
print(persona1)  # Salida: Juan, 31 años
```


- `__str__` define cómo se debe imprimir `persona1` (`Juan, 30 años`).
- `saludar` y `cumplir_anios` son otros métodos que hacen cosas diferentes.
- Imprimir la instancia de la clase (`print(persona1)`) siempre usará `__str__` para determinar la representación en cadena.

¿Qué Sucede si Ambos Métodos Están Presentes?

- `print(obj)` usa `obj.__str__()` si está definido; si no, usa `obj.__repr__()`.
- `repr(obj)` siempre usa `obj.__repr__()`.

Ejemplo con Ambos Métodos:

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

    def __str__(self):
        return f"{self.nombre}, {self.edad} años"

    def __repr__(self):
        return f"Persona('{self.nombre}', {self.edad})"

persona1 = Persona("Juan", 30)
print(persona1)  # Salida: Juan, 30 años (usa __str__)
print(repr(persona1))  # Salida: Persona('Juan', 30) (usa __repr__)
```

- `print(persona1)` llama a `persona1.__str__()`.
- `repr(persona1)` llama a `persona1.__repr__()`.

Ejercicio Práctico para Refuerzo

Ahora que hemos explicado `__str__` y `__repr__`, hagamos un ejercicio para reforzar el conocimiento.

Ejercicio: Sistema de Gestión de Libros

Define una clase `Libro` con los atributos `titulo`, `autor`, y `isbn`. Implementa los métodos `__str__` y `__repr__` para esta clase. Además, añade un método para mostrar información detallada del libro.

Código:

```
class Libro:
    def __init__(self, titulo, autor, isbn):
        self.titulo = titulo
        self.autor = autor
        self.isbn = isbn

    def __str__(self):
        return f"'{self.titulo}' por {self.autor}"

    def __repr__(self):
        return f"Libro('{self.titulo}', '{self.autor}',
'{self.isbn}')"

    def mostrar_informacion(self):
        return f"Titulo: {self.titulo}\nAutor: {self.autor}\nISBN:
{self.isbn}"

# Prueba del ejercicio
libro1 = Libro("1984", "George Orwell", "123-456-789")
print(libro1)  # Salida: '1984' por George Orwell
```

```
print(repr(libro1))  # Salida: Libro('1984', 'George Orwell', '123-
456-789')
print(libro1.mostrar_informacion())
# Salida:
# Titulo: 1984
# Autor: George Orwell
# ISBN: 123-456-789
```

Parte 2: Ejercicios Prácticos

Ejercicio 1: Sistema de Gestión de Usuarios

Define una clase `Usuario` con los atributos `nombre_usuario`, `email`, y `contraseña`. Añade métodos para cambiar la contraseña y mostrar información del usuario sin mostrar la contraseña.

Código:

```
class Usuario:
    def __init__(self, nombre_usuario, email, contraseña):
        self.nombre_usuario = nombre_usuario
        self.email = email
        self.__contraseña = contraseña  # Atributo privado

    def mostrar_informacion(self):
        return f"Usuario: {self.nombre_usuario}, Email: {self.email}"
```



```
    def cambiar_contraseña(self, nueva_contraseña):
        self.__contraseña = nueva_contraseña
```

Prueba:

```
usuario1 = Usuario("juanperez", "juan@example.com", "1234")
print(usuario1.mostrar_informacion())  # Salida: Usuario: juanperez,
Email: juan@example.com
usuario1.cambiar_contraseña("abcd")
```

Ejercicio 2: Sistema de Gestión de Inventario

Define una clase `Producto` con los atributos `nombre`, `precio`, y `cantidad`. Añade métodos para cambiar el precio y mostrar la información del producto.

```
class Producto:
    def __init__(self, nombre, precio, cantidad):
        self.nombre = nombre
        self.precio = precio
        self.cantidad = cantidad

    def mostrar_informacion(self):
        return f"Producto: {self.nombre}, Precio: ${self.precio},
Cantidad: {self.cantidad}"

    def cambiar_precio(self, nuevo_precio):
        self.precio = nuevo_precio
```

Prueba:

```
producto1 = Producto("Laptop", 1000, 5)
print(producto1.mostrar_informacion())  # Salida: Producto: Laptop,
Precio: $1000, Cantidad: 5
producto1.cambiar_precio(900)
print(producto1.mostrar_informacion())  # Salida: Producto: Laptop,
Precio: $900, Cantidad: 5
```

Parte 3: Ejercicios para Realizar

Ejercicio 1: Sistema de Gestión de Empleados

Define una clase `Empleado` con los atributos `nombre`, `puesto`, y `salario`. Añade métodos para cambiar el salario y mostrar la información del empleado.

Ejercicio 2: Sistema de Gestión de Cursos

Define una clase `Curso` con los atributos `nombre`, `codigo`, y `creditos`. Añade métodos para agregar y eliminar estudiantes.

Capítulo 3: Herencia y Polimorfismo

Parte 1: Conceptos Básicos de la Herencia

¿Qué es la Herencia?

La herencia es un mecanismo en la programación orientada a objetos que permite crear una nueva clase basándose en una clase existente. La nueva clase, llamada clase derivada (o subclase), hereda atributos y métodos de la clase base (o superclase). Esto promueve la reutilización del código y la creación de jerarquías de clases.

Clase Base y Clase Derivada

- **Clase Base (Superclase)**: Es la clase original desde la que otras clases heredan.
- **Clase Derivada (Subclase)**: Es la clase que hereda de la clase base.

Ejemplo:

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

    def saludar(self):
        return f"Hola, mi nombre es {self.nombre}"

# Clase derivada
class Estudiante(Persona):
    def __init__(self, nombre, edad, matricula):
        super().__init__(nombre, edad)
        self.matricula = matricula

    def mostrar_matricula(self):
        return f"Mi matrícula es {self.matricula}"
```


En este ejemplo, `Estudiante` es una subclase de `Persona`. Hereda los atributos `nombre` y `edad`, así como el método `saludar`.

Parte 2: Métodos Sobrescritos y Uso de `super()`

Sobrescritura de Métodos

Las subclases pueden sobrescribir (redefinir) métodos de la superclase para proporcionar una funcionalidad específica.

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

    def saludar(self):
        return f"Hola, mi nombre es {self.nombre}"

class Estudiante(Persona):
```

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

    def saludar(self):
        return f"Hola, soy estudiante y mi nombre es {self.nombre}"
```

En este ejemplo, `Estudiante` sobrescribe el método `saludar` de `Persona`.

Uso de `super()`

La función `super()` se usa para llamar a métodos de la superclase desde la subclase. Esto es especialmente útil en el método `__init__` de la subclase para asegurarse de que la superclase se inicialice correctamente.

```
class Estudiante(Persona):
    def __init__(self, nombre, edad, matricula):
        super().__init__(nombre, edad)
        self.matricula = matricula
```

Parte 3: Polimorfismo

¿Qué es el Polimorfismo?

El polimorfismo es una característica fundamental en la programación orientada a objetos que permite a los objetos de diferentes clases ser tratados de la misma manera, aunque esos objetos sean de clases diferentes y puedan tener comportamientos distintos. En términos simples, polimorfismo significa "muchas formas".

Existen dos tipos principales de polimorfismo:

1. **Polimorfismo en tiempo de compilación**: También conocido como sobrecarga de métodos, no es aplicable en Python.
2. **Polimorfismo en tiempo de ejecución**: También conocido como polimorfismo de subtipo, es el enfoque utilizado en Python.

Polimorfismo mediante Métodos

En Python, podemos lograr polimorfismo mediante la sobrescritura de métodos. Vamos a ver cómo funciona esto con ejemplos más detallados.

Ejemplo 1: Formas Geométricas

Supongamos que tienes una clase base `Forma` y dos clases derivadas `Circulo` y `Rectangulo`. Cada una tiene un método `area`, pero la implementación de este método varía según la forma.

```
class Forma:
    def area(self):
        pass

class Circulo(Forma):
```

```
    def __init__(self, radio):
        self.radio = radio

    def area(self):
        return 3.14 * self.radio * self.radio

class Rectangulo(Forma):
    def __init__(self, ancho, alto):
        self.ancho = ancho
        self.alto = alto

    def area(self):
        return self.ancho * self.alto
```

Ahora, podemos tratar diferentes formas de la misma manera utilizando polimorfismo:

```
def mostrar_area(forma):
    print(f"El área es: {forma.area()}")

c = Circulo(5)
r = Rectangulo(4, 6)

mostrar_area(c)  # Salida: El área es: 78.5
mostrar_area(r)  # Salida: El área es: 24
```

Ejemplo 2: Animales

Veamos otro ejemplo con una jerarquía de clases de animales. Imaginemos que tienes una clase base `Animal` y clases derivadas `Perro` y `Gato`. Cada clase derivada tiene un método `hacer_sonido` que produce un sonido diferente.


```
class Animal:
    def hacer_sonido(self):
        pass

class Perro(Animal):
    def hacer_sonido(self):
        return "Guau"

class Gato(Animal):
    def hacer_sonido(self):
        return "Miau"
```

Ahora, podemos crear una función que acepte cualquier objeto `Animal` y llame al método `hacer_sonido`:

```
def imprimir_sonido(animal):
    print(animal.hacer_sonido())

perro = Perro()
gato = Gato()

imprimir_sonido(perro)  # Salida: Guau
imprimir_sonido(gato)   # Salida: Miau
```

Polimorfismo mediante Interfaces

En Python, las interfaces no se definen explícitamente como en algunos otros lenguajes de programación. Sin embargo, podemos usar clases base abstractas para definir métodos que deben ser implementados por las subclases.

```
from abc import ABC, abstractmethod

class Vehiculo(ABC):
    @abstractmethod
    def arrancar(self):
        pass

class Coche(Vehiculo):
    def arrancar(self):
        return "El coche arranca con una llave"

class Moto(Vehiculo):
    def arrancar(self):
        return "La moto arranca con un botón"

def iniciar_vehiculo(vehiculo):
    print(vehiculo.arrancar())

coche = Coche()
moto = Moto()

iniciar_vehiculo(coche)  # Salida: El coche arranca con una llave
iniciar_vehiculo(moto)   # Salida: La moto arranca con un botón
```


Ejercicio Práctico de Polimorfismo

Ahora que hemos visto varios ejemplos, pongamos en práctica lo aprendido con un ejercicio más complejo.

Sistema de Gestión de Empleados

Define una clase `Empleado` con atributos `nombre` y `salario`. Luego, define subclases `Gerente` y `Desarrollador` que sobrescriban el método `trabajar`. Usa polimorfismo para gestionar una lista de empleados y llamar al método `trabajar` en cada uno.

1. **Define la clase base `Empleado`:**
   - Atributos: `nombre`, `salario`.
   - Métodos: `trabajar`.
2. **Define las subclases `Gerente` y `Desarrollador`:**
   - Cada una sobrescribe el método `trabajar`.
3. **Crea una función `mostrar_trabajo` que acepte una lista de empleados y llame al método `trabajar` en cada uno.**

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

    def trabajar(self):
        pass

class Gerente(Empleado):
    def trabajar(self):
        return f"{self.nombre} está gestionando el equipo."

class Desarrollador(Empleado):
    def trabajar(self):
        return f"{self.nombre} está escribiendo código."

def mostrar_trabajo(empleados):
    for empleado in empleados:
        print(empleado.trabajar())

# Crear instancias de empleados
gerente = Gerente("Laura", 5000)
desarrollador = Desarrollador("Pedro", 4000)

# Lista de empleados
empleados = [gerente, desarrollador]

# Mostrar el trabajo de cada empleado
mostrar_trabajo(empleados)
```


Salida esperada:

```
Laura está gestionando el equipo.
Pedro está escribiendo código.
```

Ejercicio para el Lector

1. Define una clase `Producto` con un método `calcular_precio`.

2. Define dos subclases:

- `ProductoFisico`: debe sobrescribir el método `calcular_precio` para calcular el precio de un producto físico.
- `ProductoDigital`: debe sobrescribir el método `calcular_precio` para calcular el precio de un producto digital.

3. Usa polimorfismo para:

- Crear una lista que contenga instancias de `ProductoFisico` y `ProductoDigital`.
- Recorrer la lista y calcular el precio de cada producto utilizando el método `calcular_precio` de la clase correspondiente.

Capítulo 4: Encapsulamiento

El encapsulamiento es un principio clave de la Programación Orientada a Objetos (POO) que implica ocultar los detalles internos de una clase y exponer solo lo necesario. Este enfoque ayuda a proteger los datos y a controlar cómo se accede y modifica la información de un objeto.

4.1 Accesores y Mutadores

Accesores y **mutadores** son métodos que se utilizan para obtener y modificar los valores de los atributos privados de una clase.

- **Accesores**: Métodos que permiten leer los valores de los atributos privados.
- **Mutadores**: Métodos que permiten modificar los valores de los atributos privados.

Ejemplo con Explicación Detallada

Consideremos una clase `Persona` que tiene atributos privados y necesitamos acceder a ellos y modificarlos de manera segura.

```
class Persona:
    def __init__(self, nombre, edad):
        self._nombre = nombre  # Atributo protegido
        self._edad = edad  # Atributo protegido

    # Accesor para obtener el nombre
    def get_nombre(self):
        return self._nombre

    # Mutador para modificar el nombre
    def set_nombre(self, nombre):
        self._nombre = nombre

    # Accesor para obtener la edad
    def get_edad(self):
        return self._edad

    # Mutador para modificar la edad
    def set_edad(self, edad):
        if edad > 0:  # Validación para asegurar que la edad sea
positiva
            self._edad = edad
        else:
            raise ValueError("La edad debe ser positiva")
```


Explicación:

1. **Inicialización**:
    - `__init__(self, nombre, edad)`: El constructor de la clase que inicializa los atributos `nombre` y `edad`.
    - Los atributos `_nombre` y `_edad` están precedidos por un guion bajo _ para indicar que son protegidos.
2. **Métodos Accesores**:

- `get_nombre()`: Devuelve el valor del atributo `_nombre`.
- `get_edad()`: Devuelve el valor del atributo `_edad`.

3. **Métodos Mutadores**:
   - `set_nombre(nombre)`: Modifica el valor del atributo `_nombre`.
   - `set_edad(edad)`: Modifica el valor del atributo `_edad` solo si la edad es positiva; de lo contrario, lanza una excepción.

4.2 Propiedades (`@property`)

Las **propiedades** en Python proporcionan una forma más elegante de acceder y modificar los atributos. En lugar de utilizar métodos accesores y mutadores, puedes usar propiedades que se comportan como atributos pero permiten realizar lógica adicional al acceder o modificar sus valores.

Ejemplo con Explicación Detallada

Volvamos a la clase `Persona` y definamos propiedades para `nombre` y `edad`.

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

    @property
    def nombre(self):
        return self._nombre

    @nombre.setter
    def nombre(self, valor):
        self._nombre = valor

    @property
    def edad(self):
        return self._edad

    @edad.setter
    def edad(self, valor):
        if valor > 0:
            self._edad = valor
        else:
            raise ValueError("La edad debe ser positiva")
```


Explicación:

1. **Propiedad `nombre`**:
   - `@property`: Convierte el método `nombre()` en una propiedad. Puedes acceder a esta propiedad como si fuera un atributo, sin necesidad de paréntesis.
   - `@nombre.setter`: Permite establecer el valor de la propiedad `nombre`. Esto se usa cuando se asigna un nuevo valor a `nombre`.
2. **Propiedad `edad`**:
   - `@property`: Convierte el método `edad()` en una propiedad.
   - `@edad.setter`: Permite establecer el valor de la propiedad `edad`, con validación para asegurar que la edad sea positiva.

4.3 Control de Acceso (Público, Protegido, Privado)

En Python, los atributos y métodos pueden tener diferentes niveles de acceso:

- **Público**: Los atributos y métodos públicos pueden ser accedidos desde cualquier lugar. No llevan prefijo especial.
- **Protegido**: Los atributos y métodos protegidos están destinados a ser accesibles solo dentro de la clase y sus subclases. Se indican con un solo guion bajo _.
- **Privado**: Los atributos y métodos privados están destinados a ser accesibles solo dentro de la propia clase. Se indican con dos guiones bajos __.

Ejemplo con Explicación Detallada

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre  # Atributo público
        self._edad = edad  # Atributo protegido
        self.__identificador = "ID123"  # Atributo privado

    def mostrar_identificador(self):
        return self.__identificador
```

Explicación:

1. **Atributo Público**:
   - `nombre`: Accesible desde cualquier lugar.
2. **Atributo Protegido**:
   - `_edad`: Accesible desde dentro de la clase y sus subclases.
3. **Atributo Privado**:
   - `__identificador`: Accesible solo desde dentro de la clase. Para acceder a este atributo desde fuera de la clase, se debe utilizar un método de la propia clase como `mostrar_identificador()`.

El uso de atributos con un guion bajo inicial (`_nombre`) en Python es una convención para indicar que el atributo es **protegido** (pero no privado) y que no debería ser accedido directamente desde fuera de la clase. Esta convención sugiere que el atributo es interno y está destinado a ser usado solo dentro de la clase o por sus subclases.

Detalles de la Convención

1. **Atributos Protegidos (`_atributo`)**:
   - **Propósito**: Indicar que un atributo es para uso interno y no debería ser accedido directamente desde fuera de la clase. Es una convención de código más que una restricción real.
   - **Acceso**: Aunque el atributo `_nombre` puede ser accedido y modificado directamente, se recomienda usar métodos getters y setters para acceder y modificar estos atributos, lo que proporciona un control adicional.
2. **Atributos Privados (`__atributo`)**:
   - **Propósito**: Usar dos guiones bajos antes del nombre del atributo (`__nombre`) para realizar un **name mangling**, lo que hace que el atributo sea más difícil de acceder desde fuera de la clase.

- **Acceso**: Los atributos privados están diseñados para ser más protegidos y evitar el acceso accidental o intencionado desde fuera de la clase.

Ejercicios

Ejercicio 4.1: Crear una clase Vehiculo con atributos encapsulados

- Define una clase `Vehiculo` con los atributos `marca` y `modelo` como privados. Implementa métodos accesores y mutadores para estos atributos.

Ejercicio 4.2: Implementar una clase Persona con propiedades

- Crea una clase `Persona` con los atributos `nombre` y `edad`. Utiliza propiedades para gestionar el acceso y modificación de estos atributos, incluyendo validaciones necesarias.

Capítulo 5: Composición y Agregación

5.1 Diferencias entre Composición y Agregación

Composición

En composición, una clase contiene una referencia a otra clase y es responsable de la creación y destrucción del objeto contenido. La vida del objeto contenido depende de la vida del objeto contenedor.

Ejemplo Detallado: Composición en una Biblioteca

```
class Pagina:
    def __init__(self, numero, contenido):
        self.numero = numero
        self.contenido = contenido

    def leer(self):
        return f"Página {self.numero}: {self.contenido}"


class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor
        self.paginas = []  # El libro es responsable de sus páginas

    def agregar_pagina(self, numero, contenido):
        pagina = Pagina(numero, contenido)
        self.paginas.append(pagina)

    def leer_libro(self):
        contenido_completo = f"Libro: {self.titulo} por
{self.autor}\n"
        for pagina in self.paginas:
            contenido_completo += pagina.leer() + "\n"
        return contenido_completo


# Uso de la clase Libro y Pagina
libro = Libro("Python para Todos", "Guido van Rossum")
libro.agregar_pagina(1, "Introducción a Python")
libro.agregar_pagina(2, "Variables y Tipos de Datos")
libro.agregar_pagina(3, "Estructuras de Control")

print(libro.leer_libro())
```


En este ejemplo, la clase `Libro` contiene instancias de la clase `Pagina`. La vida de las páginas depende del libro. Si el libro se destruye, las páginas también se destruyen, ya que no tienen sentido sin el libro.

Agregación

En agregación, una clase contiene una referencia a otra clase, pero no es responsable de su creación ni destrucción. La vida del objeto agregado es independiente del objeto que lo contiene.

Ejemplo Detallado: Agregación en una Biblioteca

```
class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor

    def __str__(self):
        return f"{self.titulo} por {self.autor}"


class Biblioteca:
    def __init__(self, nombre):
        self.nombre = nombre
        self.libros = []  # La biblioteca contiene libros, pero no los
crea

    def agregar_libro(self, libro):
        self.libros.append(libro)

    def mostrar_libros(self):
        print(f"Biblioteca: {self.nombre}")
        for libro in self.libros:
            print(f"- {libro}")


# Uso de la clase Biblioteca y Libro
libro1 = Libro("Python para Todos", "Guido van Rossum")
libro2 = Libro("Aprendiendo Python", "Mark Lutz")

biblioteca = Biblioteca("Biblioteca Central")
biblioteca.agregar_libro(libro1)
biblioteca.agregar_libro(libro2)

biblioteca.mostrar_libros()
```


En este ejemplo, la clase `Biblioteca` contiene instancias de la clase `Libro`, pero los libros pueden existir independientemente de la biblioteca. Si la biblioteca se destruye, los libros pueden seguir existiendo en otro contexto.

Comparación y Uso en Proyectos Reales

1. **Composición** se usa cuando una clase no tiene sentido sin otra clase. Por ejemplo:
    - Un `Pedido` y sus `Items de Pedido` en un sistema de ventas.
    - Un `Automóvil` y su `Motor`.
2. **Agregación** se usa cuando una clase puede existir independientemente de otra clase. Por ejemplo:
    - Un `Equipo` y sus `Jugadores` en un sistema deportivo.
    - Una `Empresa` y sus `Empleados`.

Consideraciones Prácticas

- **Mantenibilidad**: Al usar composición y agregación correctamente, el código es más modular y fácil de mantener.
- **Reutilización**: Facilita la reutilización de componentes en diferentes contextos.

- **Flexibilidad**: Proporciona flexibilidad al diseño del sistema, permitiendo cambios y extensiones sin afectar otras partes del código.

Ejercicio Práctico

Ejercicio 1: Composición Crea una clase `Computadora` que tenga una clase `CPU`, `Memoria` y `DiscoDuro`. Cada uno de estos componentes debe tener atributos específicos y métodos para mostrar sus detalles.

Ejercicio 2: Agregación Crea una clase `Universidad` que contenga varias `Facultades`. Cada `Facultad` debe tener un nombre y una lista de `Estudiantes`. Los estudiantes deben poder existir independientemente de las facultades.

Capítulo 6: Manejo de Excepciones

En este capítulo, exploraremos el manejo de excepciones en Python y cómo crear excepciones personalizadas. Veremos cómo usar `try`, `except`, `finally` y `raise`, y cómo diseñar nuestras propias excepciones para gestionar errores específicos en nuestro código.

1. Manejo Básico de Excepciones

El manejo de excepciones en Python se realiza mediante bloques `try` y `except`. La estructura básica es la siguiente:

```
try:
    # Código que puede causar una excepción
    resultado = 10 / 0
except ZeroDivisionError:
    # Código que se ejecuta si ocurre la excepción
    print("No se puede dividir por cero.")
```

Aquí, intentamos realizar una división por cero dentro del bloque `try`. Si se produce una `ZeroDivisionError`, el bloque `except` captura la excepción y ejecuta el código de manejo de errores.

2. Bloque `finally`

El bloque `finally` se usa para ejecutar código que debe ejecutarse independientemente de si ocurre o no una excepción. Este bloque es útil para liberar recursos o realizar tareas de limpieza.

```
try:
    archivo = open('archivo.txt', 'r')
    contenido = archivo.read()
except FileNotFoundError:
    print("El archivo no se encontró.")
finally:
    archivo.close()  # Siempre se ejecuta, incluso si ocurre una
excepción
```

3. Lanzar Excepciones con `raise`

Puedes lanzar una excepción personalizada utilizando la palabra clave `raise`. Esto detiene la ejecución del código y transfiere el control al bloque `except` correspondiente, si existe.

```
def dividir(a, b):
    if b == 0:
        raise ValueError("No se puede dividir por cero.")
    return a / b
```

En este caso, si `b` es cero, se lanza una excepción `ValueError` con un mensaje personalizado.

4. Excepciones Personalizadas

Las excepciones personalizadas permiten definir errores específicos en tu programa, lo que facilita la identificación y manejo de problemas particulares. Para crear una excepción personalizada:

1. **Define una Clase de Excepción:** Hereda de la clase base `Exception`.
2. **Inicializa la Clase Base con Mensajes:** Usa `super().__init__(mensaje)` para pasar el mensaje de error a la clase base.

Ejemplo de Excepción Personalizada:

```
class SaldoInsuficienteError(Exception):
    def __init__(self, mensaje="Saldo insuficiente"):
        super().__init__(mensaje)  # Llama al constructor de Exception

def retirar_dinero(cuenta, cantidad):
    if cantidad > cuenta.saldo:
        raise SaldoInsuficienteError()  # Lanza la excepción
personalizada
    cuenta.saldo -= cantidad

class Cuenta:
    def __init__(self, saldo):
        self.saldo = saldo

cuenta = Cuenta(100)

try:
    retirar_dinero(cuenta, 150)
except SaldoInsuficienteError as e:
    print(f"Error: {e}")  # Imprime el mensaje de la excepción
```


Explicación:

- **`class SaldoInsuficienteError(Exception):`** define una nueva excepción heredando de `Exception`.
- **`def __init__(self, mensaje="Saldo insuficiente"):`** inicializa la excepción, pasando el mensaje a la clase base.
- **`super().__init__(mensaje)`** llama al constructor de la clase base `Exception` para gestionar el mensaje de error.
- **`raise SaldoInsuficienteError()`** lanza la excepción personalizada si el saldo es insuficiente.
- **`except SaldoInsuficienteError as e:`** captura la excepción y accede al mensaje de error.

5. ¿Por Qué Usar Excepciones Personalizadas?

- **Claridad:** Permiten identificar errores específicos y manejarlos de manera adecuada.
- **Mantenimiento:** Facilitan el seguimiento y resolución de errores complejos en programas grandes.

- **Flexibilidad:** Puedes definir comportamientos personalizados para diferentes tipos de errores.

6. Ejemplos Prácticos de Excepciones Personalizadas

1. Sistema de Gestión de Pedidos

Contexto: Imagina que estás desarrollando un sistema para gestionar pedidos en una tienda en línea. Necesitas manejar situaciones como el intento de pedir más artículos de los que están disponibles en inventario.

Código:

```
class ProductoNoDisponible(Exception):
    def __init__(self, mensaje="El producto no está disponible en la
cantidad solicitada."):
        super().__init__(mensaje)

class Pedido:
    def __init__(self, producto, cantidad):
        self.producto = producto
        self.cantidad = cantidad

class Inventario:
    def __init__(self):
        self.productos = {"camisa": 10, "pantalones": 5}

    def realizar_pedido(self, pedido):
        if pedido.cantidad > self.productos.get(pedido.producto, 0):
            raise ProductoNoDisponible()
        self.productos[pedido.producto] -= pedido.cantidad
        return "Pedido realizado con éxito"

# Prueba
inventario = Inventario()
pedido = Pedido("camisa", 15)

try:
    print(inventario.realizar_pedido(pedido))
except ProductoNoDisponible as e:
    print(e)
```


Explicación: Aquí, `ProductoNoDisponible` es una excepción personalizada que se lanza si se intenta pedir más productos de los disponibles en inventario. El método `realizar_pedido` verifica si la cantidad solicitada está disponible y lanza la excepción si no lo está.

2. Sistema de Autenticación de Usuarios

Contexto: Estás desarrollando un sistema de autenticación donde los usuarios deben ingresar una contraseña. Necesitas manejar casos en los que se ingresen contraseñas incorrectas o que no cumplan con los requisitos de seguridad.

Código:

```
class ContraseñaInsegura(Exception):
    def __init__(self, mensaje="La contraseña no cumple con los
requisitos de seguridad."):
        super().__init__(mensaje)

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

class SistemaAutenticacion:
    def verificar_contraseña(self, usuario):
        if len(usuario.contraseña) < 8:
            raise ContraseñaInsegura()
        return "Contraseña segura"

# Prueba
usuario = Usuario("Juan", "12345")

try:
    print(SistemaAutenticacion().verificar_contraseña(usuario))
except ContraseñaInsegura as e:
    print(e)
```

Explicación: `ContraseñaInsegura` es una excepción personalizada que se lanza si la contraseña no cumple con los requisitos de seguridad (en este caso, una longitud mínima de 8 caracteres). El método `verificar_contraseña` verifica si la contraseña es segura y lanza la excepción si no lo es.

3. Sistema de Cuentas Bancarias

Contexto: Estás desarrollando un sistema de gestión de cuentas bancarias. Necesitas manejar errores cuando un usuario intenta retirar más dinero del que tiene en su cuenta.

Código:

```
class SaldoInsuficiente(Exception):
    def __init__(self, mensaje="Saldo insuficiente para realizar la
transacción."):
        super().__init__(mensaje)

class CuentaBancaria:
    def __init__(self, titular, saldo):
        self.titular = titular
        self.saldo = saldo

    def retirar(self, cantidad):
        if cantidad > self.saldo:
            raise SaldoInsuficiente()
        self.saldo -= cantidad
        return f"Retiro exitoso. Saldo restante: {self.saldo}"

cuenta = CuentaBancaria("Maria", 100)

try:
    print(cuenta.retirar(150))
```

```
except SaldoInsuficiente as e:
    print(e)
```

Explicación: `SaldoInsuficiente` es una excepción personalizada que se lanza si el usuario intenta retirar más dinero del que tiene en su cuenta. El método `retirar` verifica si el saldo es suficiente y lanza la excepción si no lo es.

Ejercicios para Practicar

1. **Sistema de Reservas de Hotel:**
   - **Contexto:** Imagina un sistema de reservas de hotel. Define una excepción personalizada para manejar el caso cuando se intenta reservar una habitación que no está disponible.
   - **Tarea:** Crea una clase `HabitacionNoDisponible` y usa esta excepción personalizada en una clase `Hotel` que tenga métodos para reservar habitaciones.

2. **Sistema de Calificación de Cursos:**
   - **Contexto:** Estás desarrollando un sistema para calificar cursos. Define una excepción personalizada para manejar calificaciones que no están dentro del rango válido (por ejemplo, de 0 a 10).
   - **Tarea:** Crea una clase `CalificacionInvalida` y usa esta excepción en un método de una clase `Curso` para calificar estudiantes.

3. **Sistema de Registro de Productos:**
   - **Contexto:** Tienes un sistema para registrar productos en una tienda. Define una excepción personalizada para manejar el caso cuando se intenta registrar un producto con un código ya existente.
   - **Tarea:** Crea una clase `CodigoProductoDuplicado` y usa esta excepción en un método de una clase `Inventario` para registrar nuevos productos.

4. Define una clase `ErrorDeRegistro` que herede de `Exception`. Implementa un método `__init__` que permita pasar un mensaje personalizado al constructor de la clase base. Luego, usa esta excepción en una función que simule el registro de un usuario, lanzando la excepción si el usuario ya está registrado.

   Instrucciones:

   - Define la clase `ErrorDeRegistro` con un mensaje personalizado.
   - Implementa una función `registrar_usuario` que lance `ErrorDeRegistro` si el usuario ya está en una lista de usuarios registrados.
   - Maneja la excepción en un bloque `try` y muestra un mensaje de error adecuado.

5. **Ejercicio 6.2: Manejo de Excepciones en Funciones**

   Crea una clase `Calculadora` con métodos para dividir y multiplicar números. Implementa manejo de excepciones para los casos en que el divisor es cero y para el caso en que los argumentos no son números válidos. Lanza excepciones personalizadas para estos errores y maneja estas excepciones en el código de llamada.

   Instrucciones:

   - Define una clase `Calculadora` con métodos `dividir` y `multiplicar`.
   - Crea excepciones personalizadas para errores de tipo y división por cero.
   - Usa estas excepciones en los métodos de la clase.
   - Maneja las excepciones en el código principal y muestra mensajes de error.

Capítulo 07 : Módulos y Paquetes

1. ¿Qué es un Módulo?

Un **módulo** en Python es simplemente un archivo con extensión `.py` que contiene definiciones y sentencias en Python, como funciones, clases y variables. El propósito de un módulo es organizar y reutilizar el código, permitiendo dividir un programa en partes más pequeñas y manejables.

Ejemplo simple de un módulo:

Supón que creas un archivo llamado `matematica.py` con el siguiente contenido:

```
# matematica.py

def sumar(a, b):
    return a + b

def restar(a, b):
    return a - b
```

Aquí, `matematica.py` es un módulo que contiene dos funciones: `sumar` y `restar`.

2. ¿Cómo Importar un Módulo?

En Python, puedes importar un módulo de diferentes maneras, dependiendo de cómo deseas acceder a su contenido. Aquí te explico las formas más comunes:

Importar todo el módulo

Cuando importas todo el módulo, debes usar el nombre del módulo para acceder a sus funciones o clases.

```
import matematica

resultado = matematica.sumar(5, 3)
print(resultado)  # Imprime: 8
```

Aquí, `matematica.sumar(5, 3)` significa que estás llamando a la función `sumar` que se encuentra dentro del módulo `matematica`.

Importar funciones o clases específicas

Si solo necesitas algunas funciones o clases, puedes importarlas directamente, y no necesitas usar el nombre del módulo cuando las llamas.

```
from matematica import sumar, restar

resultado = sumar(5, 3)
print(resultado)  # Imprime: 8

resultado = restar(5, 3)
print(resultado)  # Imprime: 2
```

Importar un módulo con un alias

Puedes dar un alias al módulo para que su nombre sea más corto o más conveniente de usar.

```
import matematica as mat

resultado = mat.sumar(5, 3)
print(resultado)  # Imprime: 8
```

Importar todo el contenido del módulo

Puedes importar todo el contenido del módulo usando un asterisco `*`. Aunque no es recomendable porque puede causar confusión en nombres de funciones o variables.

```
from matematica import *

resultado = sumar(5, 3)
print(resultado)  # Imprime: 8
```

3. Módulos Integrados en Python

Python viene con muchos módulos integrados que puedes usar sin necesidad de instalarlos. Algunos ejemplos comunes incluyen:

- **`math`**: Proporciona funciones matemáticas como `sqrt`, `sin`, `cos`, etc.
- **`datetime`**: Permite trabajar con fechas y horas.
- **`os`**: Proporciona funciones para interactuar con el sistema operativo.

Ejemplo usando el módulo **`math`**:

```
import math

resultado = math.sqrt(16)
print(resultado)  # Imprime: 4.0
```

4. ¿Qué es un Paquete?

Un **paquete** es una forma de organizar varios módulos en un directorio. Un paquete es simplemente un directorio que contiene un archivo especial llamado `__init__.py`, que puede estar vacío o contener código para inicializar el paquete.

Estructura de un paquete:

```
mi_paquete/
├── __init__.py
├── modulo1.py
└── modulo2.py
```

- `__init__.py`: Este archivo le dice a Python que el directorio debe ser tratado como un paquete.
- `modulo1.py` y `modulo2.py`: Estos son módulos dentro del paquete `mi_paquete`.

5. Cómo Crear y Usar un Paquete

Crear un paquete:

1. **Crea un directorio** llamado `mi_paquete`.
2. **Dentro de este directorio**, crea un archivo `__init__.py`. Este archivo puede estar vacío o contener código que quieras ejecutar cuando se importe el paquete.
3. **Añade otros módulos** en este directorio, como `modulo1.py` y `modulo2.py`.

Contenido de `modulo1.py`:

```
# modulo1.py
def funcion_modulo1():
    return "Hola desde modulo1"
```

Contenido de `modulo2.py`:

```
# modulo2.py
def funcion_modulo2():
    return "Hola desde modulo2"
```

Uso del paquete:

```
from mi_paquete.modulo1 import funcion_modulo1
from mi_paquete.modulo2 import funcion_modulo2

print(funcion_modulo1())  # Imprime: Hola desde modulo1
print(funcion_modulo2())  # Imprime: Hola desde modulo2
```


6. Importaciones Relativas y Absolutas

Aquí es donde algunas veces el tema se pone un poco confuso. Veamos la diferencia:

Importación Absoluta

La **importación absoluta** utiliza la ruta completa desde la raíz del proyecto para importar módulos.

```
from mi_paquete.modulo1 import funcion_modulo1
```

Esto es útil porque es claro de dónde proviene el módulo que estás importando, pero puede volverse tedioso si los nombres de las rutas son largos.

Importación Relativa

La **importación relativa** se utiliza dentro de paquetes para importar otros módulos del mismo paquete sin tener que escribir la ruta completa.

```
# Supongamos que estamos en mi_paquete/modulo2.py
from .modulo1 import funcion_modulo1
```

En este caso, el `.` indica que estás importando desde el mismo paquete (`mi_paquete`). Si usaras `..`, estarías subiendo un nivel en la jerarquía de directorios.

Cuándo usar cada tipo de importación:

- **Importaciones absolutas**: Son claras y explícitas, y es recomendable usarlas cuando estés trabajando en un proyecto grande.
- **Importaciones relativas**: Son más cortas y útiles cuando estás dentro de un paquete y necesitas importar módulos del mismo nivel.

7. Beneficios de Usar Módulos y Paquetes

- **Organización**: Mantienes el código limpio y estructurado.
- **Reusabilidad**: Puedes reutilizar código en diferentes partes de tu proyecto o incluso en otros proyectos.
- **Mantenibilidad**: Es más fácil mantener el código dividido en partes pequeñas y manejables.
- **Modularidad**: Puedes actualizar o mejorar una parte del código sin afectar el resto.

8. Ejemplo Completo de Proyecto con Módulos y Paquetes

Supongamos que tienes una aplicación más grande y necesitas dividirla en módulos y paquetes para manejar diferentes funcionalidades.

Estructura del proyecto:

```
mi_aplicacion/
├── __init__.py
├── usuarios/
│   ├── __init__.py
│   ├── gestion.py
│   └── modelo.py
└── productos/
    ├── __init__.py
    ├── gestion.py
    └── modelo.py
```


- **`usuarios/modelo.py`**:

```
# usuarios/modelo.py
class Usuario:
    def __init__(self, nombre, email):
        self.nombre = nombre
        self.email = email
```

- **`usuarios/gestion.py`**:

```
# usuarios/gestion.py
from .modelo import Usuario

class GestionUsuarios:
    def __init__(self):
        self.usuarios = []

    def agregar_usuario(self, usuario):
        self.usuarios.append(usuario)
```

- **productos/modelo.py**:

```
# productos/modelo.py
class Producto:
    def __init__(self, nombre, precio):
        self.nombre = nombre
        self.precio = precio
```

- **productos/gestion.py**:

```
# productos/gestion.py
from .modelo import Producto

class GestionProductos:
    def __init__(self):
        self.productos = []

    def agregar_producto(self, producto):
        self.productos.append(producto)
```

- **Archivo principal**:

```
# main.py
from usuarios.gestion import GestionUsuarios
from usuarios.modelo import Usuario
from productos.gestion import GestionProductos
from productos.modelo import Producto

gestion_usuarios = GestionUsuarios()
gestion_productos = GestionProductos()

usuario = Usuario("Carlos", "carlos@example.com")
producto = Producto("Laptop", 1500)

gestion_usuarios.agregar_usuario(usuario)
gestion_productos.agregar_producto(producto)
```


Este ejemplo muestra cómo organizar una aplicación usando módulos y paquetes, facilitando la administración y escalabilidad del proyecto.

Capítulo 08 : Introducción al Manejo de Archivos en Python

Trabajar con archivos es una parte fundamental de la programación. Los archivos son utilizados para almacenar datos de forma persistente, ya sea para guardar configuraciones, almacenar información de usuarios, o cualquier otra cosa que necesite ser guardada entre sesiones del programa.

En Python, manejar archivos es sencillo, pero es importante entender los conceptos básicos y las mejores prácticas.

---

1. Abrir y Cerrar Archivos

Abrir un Archivo

En Python, los archivos se abren utilizando la función `open()`. Esta función toma al menos dos argumentos:

1. **Nombre del archivo**: La ruta y el nombre del archivo que quieres abrir.
2. **Modo de apertura**: Un string que indica el propósito de abrir el archivo (leer, escribir, etc.).

```
archivo = open('archivo.txt', 'r')
```


En este ejemplo, `archivo.txt` es el nombre del archivo, y `'r'` es el modo de apertura. Esto significa que estamos abriendo el archivo en modo de **lectura**.

Modos de Apertura Comunes:

- '**r**': Lectura. El archivo debe existir, o se produce un error.
- '**w**': Escritura. Si el archivo no existe, se crea. Si existe, se sobrescribe.
- '**a**': Añadir. Si el archivo no existe, se crea. Si existe, se añade al final del archivo.
- '**b**': Modo binario. Se usa junto con otros modos (`'rb'`, `'wb'`, etc.) para trabajar con archivos binarios.
- '**x**': Creación. Crea un archivo, y si el archivo ya existe, produce un error.

Cerrar un Archivo

Cuando terminas de trabajar con un archivo, es importante cerrarlo usando el método `close()`. Esto asegura que los recursos del sistema se liberen adecuadamente.

```
archivo.close()
```

Ejemplo Completo:

```
archivo = open('archivo.txt', 'r')  # Abre el archivo en modo lectura
contenido = archivo.read()          # Lee el contenido del archivo
print(contenido)                    # Imprime el contenido
archivo.close()                     # Cierra el archivo
```

2. Leer Archivos

Existen varias formas de leer el contenido de un archivo en Python:

Leer Todo el Contenido a la Vez:

```
archivo = open('archivo.txt', 'r')
contenido = archivo.read()  # Lee todo el contenido del archivo
print(contenido)
archivo.close()
```

Leer Línea por Línea:

```
archivo = open('archivo.txt', 'r')
for linea in archivo:
    print(linea.strip())  # .strip() quita los saltos de línea al
final de cada línea
archivo.close()
```

Leer una Sola Línea:

```
archivo = open('archivo.txt', 'r')
linea = archivo.readline()  # Lee la primera línea del archivo
print(linea)
archivo.close()
```

Leer Todas las Líneas como una Lista:


```
archivo = open('archivo.txt', 'r')
lineas = archivo.readlines()  # Devuelve una lista con todas las
líneas
print(lineas)
archivo.close()
```

Cada una de estas opciones es útil en diferentes situaciones, dependiendo de cómo quieras procesar el contenido del archivo.

---

3. Escribir en Archivos

Escribir en archivos es tan sencillo como leerlos, pero debes ser consciente del modo de apertura para no sobrescribir accidentalmente un archivo existente.

Escribir en un Archivo (Modo 'w'):

```
archivo = open('archivo.txt', 'w')
archivo.write('Hola, Mundo!\n')  # Escribe una línea en el archivo
archivo.write('Esta es otra línea.\n')
archivo.close()
```

En este caso, si `archivo.txt` ya existía, su contenido se habrá sobrescrito.

Añadir a un Archivo (Modo 'a'):

```
archivo = open('archivo.txt', 'a')
archivo.write('Esta línea se añade al final.\n')
archivo.close()
```

El modo `'a'` añade el contenido al final del archivo, sin sobrescribir lo que ya estaba allí.

---

4. Gestión de Archivos con `with`

El uso de `with` es una práctica recomendada al trabajar con archivos en Python. El bloque `with` asegura que el archivo se cierre automáticamente, incluso si ocurre una excepción mientras trabajas con él.

```
with open('archivo.txt', 'r') as archivo:
    contenido = archivo.read()
    print(contenido)
```

Cuando se usa `with`, no es necesario llamar a `archivo.close()` explícitamente. El archivo se cerrará automáticamente cuando se salga del bloque `with`.

---

5. Manejo de Errores al Trabajar con Archivos

Es importante manejar posibles errores, como intentar abrir un archivo que no existe.

Ejemplo: Manejo de Errores

```
try:
    with open('archivo_inexistente.txt', 'r') as archivo:
        contenido = archivo.read()
except FileNotFoundError:
    print("El archivo no existe.")
```

En este ejemplo, si el archivo no existe, se capturará la excepción `FileNotFoundError`, y se mostrará un mensaje en lugar de que el programa falle.

---

6. Archivos Binarios

Los archivos binarios contienen datos en un formato que no es texto (por ejemplo, imágenes o archivos ejecutables). Para trabajar con ellos, debes abrirlos en modo binario (`'b'`).

Ejemplo: Leer un Archivo Binario

```
with open('imagen.jpg', 'rb') as archivo:
    contenido = archivo.read()
    # Aquí puedes procesar el contenido binario
```

En este caso, `contenido` es una secuencia de bytes que representa el archivo binario.

---

7. Manejo de Directorios

Python también te permite interactuar con directorios (carpetas) en tu sistema de archivos. Para esto, utilizamos el módulo `os` y, en versiones más modernas, `pathlib`.

Ejemplo: Crear un Directorio

```
import os

os.mkdir('nueva_carpeta')
```

Ejemplo: Listar Contenido de un Directorio

```
import os

contenido = os.listdir('.')
print(contenido)
```

Ejercicio 1: Contador de Palabras

1. **Descripción:**
   - Escribe un programa que abra un archivo de texto existente, lea su contenido y cuente cuántas veces aparece una palabra específica en el archivo.
2. **Instrucciones:**
   - Crea un archivo de texto llamado `texto.txt` y escribe algunas líneas de texto en él.
   - Luego, escribe un programa que:
     - Pregunte al usuario por la palabra que desea buscar.
     - Lea el contenido de `texto.txt`.
     - Cuente cuántas veces aparece esa palabra en el archivo.
     - Imprima el resultado.
3. **Pistas:**
   - Usa el método `str.count()` para contar las apariciones de una palabra en el texto.
   - Recuerda abrir el archivo en modo de lectura y cerrarlo cuando termines, o usa un bloque `with`.

Ejercicio 2: Copia de Archivos

1. **Descripción:**
   - Escribe un programa que copie el contenido de un archivo de texto a otro archivo.
2. **Instrucciones:**

- Crea un archivo de texto llamado `origen.txt` con algo de contenido.
- Luego, escribe un programa que:
  - Lea el contenido de `origen.txt`.
  - Escriba ese contenido en un nuevo archivo llamado `copia.txt`.

3. **Pistas:**
   - Abre ambos archivos, uno para lectura y el otro para escritura.
   - Asegúrate de manejar excepciones para casos donde `origen.txt` no exista.

Ejercicio 3: Lista de Nombres

1. **Descripción:**
   - Escribe un programa que permita al usuario agregar nombres a un archivo de texto y luego muestre todos los nombres almacenados en ese archivo.
2. **Instrucciones:**
   - Crea un archivo de texto llamado `nombres.txt`.
   - Escribe un programa que:
     - Pida al usuario un nombre y lo agregue al archivo `nombres.txt`.
     - Luego, lea y muestre todos los nombres almacenados en el archivo.
   - Permite que el programa siga pidiendo nombres hasta que el usuario decida detenerse.

3. **Pistas:**
   - Usa el modo `'a'` para agregar nombres al archivo.
   - Usa un ciclo `while` para continuar pidiendo nombres hasta que el usuario escriba algo como `"salir"`.

Lectura y Escritura de Archivos JSON

¿Qué es JSON?

JSON (JavaScript Object Notation) es un formato ligero de intercambio de datos que es fácil de leer y escribir tanto para humanos como para máquinas. Es una manera común de almacenar datos estructurados, como diccionarios o listas, en archivos.

Lectura de Archivos JSON

Para leer un archivo JSON en Python, se utiliza la función `json.load()`, que convierte el contenido del archivo en un objeto de Python (generalmente un diccionario o una lista).

Ejemplo:

```
import json

# Leer un archivo JSON
with open('datos.json', 'r') as archivo:
    datos = json.load(archivo)

print(datos)
```

Explicación:

- `with open('datos.json', 'r') as archivo:`: Abre el archivo en modo de lectura ('r') y lo asigna a la variable `archivo`.
- `json.load(archivo)`: Lee el contenido del archivo y lo convierte en un objeto Python (por ejemplo, un diccionario o una lista).

Escritura de Archivos JSON

Para escribir en un archivo JSON, se utiliza `json.dump()`, que convierte un objeto de Python en una cadena JSON y la guarda en el archivo.

Ejemplo:

```
import json

datos = {
    'nombre': 'Juan',
    'edad': 30,
    'ciudad': 'Madrid'
}

# Escribir en un archivo JSON
with open('datos.json', 'w') as archivo:
    json.dump(datos, archivo)
```


Explicación:

- `with open('datos.json', 'w') as archivo:`: Abre el archivo en modo de escritura ('w'). Si el archivo no existe, lo crea; si existe, sobrescribe su contenido.
- `json.dump(datos, archivo)`: Convierte el diccionario `datos` en una cadena JSON y la escribe en el archivo.

Modos de Apertura de Archivos

- `'r'`: Modo lectura (Read). Solo permite leer el archivo.
- `'w'`: Modo escritura (Write). Sobrescribe el archivo si existe, o lo crea si no.
- `'a'`: Modo anexar (Append). Añade datos al final del archivo existente, sin sobrescribir su contenido.

Nota Importante: Cuando trabajas con archivos JSON, **no puedes** usar el modo `'a'` (anexar) para agregar datos. Esto se debe a que JSON debe ser un único objeto

estructurado (como un diccionario o una lista), y el formato JSON no permite simplemente agregar datos al final como lo harías con un archivo de texto o CSV. Para agregar datos a un archivo JSON, debes leer el archivo, modificar los datos en Python y luego escribir todo el contenido de nuevo al archivo.

Ejercicio 1: Sistema de Gestión de Tareas con JSON

Crea un programa en Python que permita gestionar una lista de tareas, donde las tareas se guarden en un archivo JSON.

1. Crear una clase `Tarea` con atributos como `título`, `descripción`, `fecha de vencimiento` y `estado`.
2. Crear una clase `ListaDeTareas` para gestionar la lista de tareas:
    - `agregar_tarea`: Agrega una tarea a la lista.
    - `mostrar_tareas`: Muestra todas las tareas almacenadas.
    - `actualizar_estado_tarea`: Marca una tarea como completada.
3. Guardar las tareas en un archivo JSON.

Lectura y Escritura de Archivos CSV

¿Qué es CSV?

CSV (Comma-Separated Values) es un formato de archivo de texto que utiliza comas para separar valores. Es ampliamente utilizado para almacenar datos tabulares en una estructura simple.

Lectura de Archivos CSV

Para leer un archivo CSV en Python, se utiliza `csv.reader()`, que convierte cada línea del archivo en una lista de valores.

Ejemplo:

```
import csv

# Leer un archivo CSV
with open('datos.csv', 'r') as archivo:
    lector_csv = csv.reader(archivo)
    for fila in lector_csv:
        print(fila)
```

Explicación:

- `csv.reader(archivo)`: Convierte cada línea del archivo en una lista de valores separados por comas.

Escritura de Archivos CSV

Para escribir en un archivo CSV, se utiliza `csv.writer()` junto con los métodos `writerow()` o `writerows()`.

- **`writerow()`**: Escribe una sola fila en el archivo CSV.
- **`writerows()`**: Escribe múltiples filas en el archivo CSV.

Nota: Cuando escribes datos en un archivo CSV usando `csv.writer`, los datos se convierten en cadenas de texto automáticamente. Si quieres almacenar datos como números o fechas en el archivo CSV, los almacenarás como texto, y luego tendrás que convertirlos nuevamente al leerlos si es necesario.

Ejemplo con `writerow()`:

```
import csv

# Datos a escribir
datos = ['Juan', 30, 'Madrid']

# Escribir una fila en un archivo CSV
with open('datos.csv', 'w', newline='') as archivo:
    escritor_csv = csv.writer(archivo)
    escritor_csv.writerow(datos)
```

Ejemplo con `writerows()`:


```
import csv

# Datos a escribir
datos = [
    ['Juan', 30, 'Madrid'],
    ['Ana', 25, 'Barcelona'],
    ['Luis', 35, 'Valencia']
]

# Escribir múltiples filas en un archivo CSV
with open('datos.csv', 'w', newline='') as archivo:
    escritor_csv = csv.writer(archivo)
    escritor_csv.writerows(datos)
```

Explicación:

- `csv.writer(archivo)`: Crea un objeto escritor para el archivo CSV.
- `escritor_csv.writerow(datos)`: Escribe una única fila en el archivo CSV.
- `escritor_csv.writerows(datos)`: Escribe múltiples filas en el archivo CSV.

Modos de Apertura de Archivos CSV

A diferencia de JSON, en los archivos CSV puedes utilizar el modo `'a'` para agregar nuevas filas sin sobrescribir el contenido existente.

Ejemplo de uso del modo `'a'`:

```
import csv

# Nuevos datos a agregar
nuevos_datos = ['Carlos', 28, 'Sevilla']

# Agregar una fila al final del archivo CSV existente
with open('datos.csv', 'a', newline='') as archivo:
    escritor_csv = csv.writer(archivo)
    escritor_csv.writerow(nuevos_datos)
```

Ejercicio 2: Agenda de Contactos con CSV

Crea un programa en Python que permita gestionar una agenda de contactos, donde los contactos se guarden en un archivo CSV.

1. Crear una clase `Contacto` con atributos como `nombre`, `teléfono` y `correo electrónico`.
2. Crear una clase `Agenda` para gestionar la lista de contactos:
    - `agregar_contacto`: Agrega un contacto al archivo CSV.
    - `buscar_contacto`: Busca un contacto en el archivo CSV y muestra su información.
    - `mostrar_contactos`: Muestra todos los contactos almacenados en el archivo CSV.

Ejercicio 3: Exportación de Tareas a CSV

Amplía el primer ejercicio (Sistema de Gestión de Tareas) para que pueda exportar la lista de tareas a un archivo CSV.

1. Crear un método `exportar_csv` en la clase `ListaDeTareas` que lea las tareas desde el archivo JSON y las escriba en un archivo CSV.
2. Asegurarse de que el archivo CSV tenga encabezados adecuados.

Ejercicio 1: Sistema de Gestión de Tareas con JSON

```
import json
from pathlib import Path

class Tarea_no_Existe(Exception):
    def __init__(self, Mensaje="No existe ninguna tarea guardada"):
        super().__init__(Mensaje)

class Lista_de_tarea:
    def __init__(self):
        self.tareas = []

    def agregar_tarea(self, tarea):
        ruta = Path("tareas.json")
        if ruta.exists():
            with open("tareas.json", "r") as archivo:
                self.tareas = json.load(archivo)
            self.tareas.append(tarea)
```

```
        else:
            self.tareas.append(tarea)
        with open("tareas.json", "w") as archivo:
            json.dump(self.tareas, archivo)

    def mostrar_tareas(self):
        try:
            with open("tareas.json", "r") as archivo:
                tareas = json.load(archivo)
                for tarea in tareas:
                    print("----------------------------------------")
                    for key, value in tarea.items():
                        print(f"{key}: {value}")
        except FileNotFoundError:
            raise Tarea_no_Existe()

    def actualizar_estado_tarea(self, titulo):
        try:
            with open("tareas.json", "r") as archivo:
                self.tareas = json.load(archivo)
            for tarea in self.tareas:
                if tarea['Titulo'].lower() == titulo.lower():
                    tarea['Estado'] = "Completado"
            with open("tareas.json", "w") as archivo:
                json.dump(self.tareas, archivo)
        except FileNotFoundError:
            raise Tarea_no_Existe()

class Tarea:
    def __init__(self, titulo, descripcion, fecha_vencimiento,
estado="No completado"):
        self.titulo = titulo
        self.descripcion = descripcion
        self.fecha_vencimiento = fecha_vencimiento
        self.estado = estado

    def convertir_dict(self):
        return {
            'Titulo': self.titulo,
            'Descripcion': self.descripcion,
            'Fecha de Vencimiento': self.fecha_vencimiento,
            'Estado': self.estado
        }

# Ejemplo de uso
lista = Lista_de_tarea()

# Agregar tareas
tarea1 = Tarea("Estudiar", "Estudiar para el examen de matemáticas",
"2024-09-05")
lista.agregar_tarea(tarea1.convertir_dict())

tarea2 = Tarea("Lavar el coche", "Lavar el coche antes del viaje",
"2024-09-06")
lista.agregar_tarea(tarea2.convertir_dict())

# Mostrar tareas
lista.mostrar_tareas()

# Actualizar el estado de una tarea
lista.actualizar_estado_tarea("Estudiar")
```

```
# Mostrar tareas después de la actualización
lista.mostrar_tareas()
```

Ejercicio 2: Agenda de Contactos con CSV

```
import csv

class Agenda:
    def agregar_contacto(self, contacto):
        with open('agenda.csv', 'a', newline='') as archivo:
            escritor_csv = csv.writer(archivo)
            escritor_csv.writerow(contacto)

    def buscar_contacto(self, nombre):
        with open('agenda.csv', 'r') as archivo:
            lector_csv = csv.reader(archivo)
            for fila in lector_csv:
                if fila[0].lower() == nombre.lower():
                    print(f"Nombre: {fila[0]}, Teléfono: {fila[1]},
Correo: {fila[2]}")
                    return
            print("Contacto no encontrado.")

    def mostrar_contactos(self):
        with open('agenda.csv', 'r') as archivo:
            lector_csv = csv.reader(archivo)
            for fila in lector_csv:
                print(f"Nombre: {fila[0]}, Teléfono: {fila[1]},
Correo: {fila[2]}")

# Ejemplo de uso
agenda = Agenda()

# Agregar contactos
agenda.agregar_contacto(['Juan', '123456789', 'juan@mail.com'])
agenda.agregar_contacto(['Ana', '987654321', 'ana@mail.com'])

# Mostrar todos los contactos
agenda.mostrar_contactos()

# Buscar un contacto
agenda.buscar_contacto('Ana')
```

Ejercicio 3: Exportación de Tareas a CSV

```
import json
import csv

class Lista_de_tarea:
    # ... (Resto de la implementación de la clase Lista_de_tarea)

    def exportar_csv(self):
        try:
            with open("tareas.json", "r") as archivo:
                self.tareas = json.load(archivo)

            # Crear encabezados a partir de las claves del primer
elemento
            encabezados = self.tareas[0].keys() if self.tareas else []
```

```
            with open("tareas.csv", "w", newline="") as file:
                escritor_csv = csv.writer(file)
                if encabezados:
                    escritor_csv.writerow(encabezados)
                escritor_csv.writerows([tarea.values() for tarea in
self.tareas])

        except FileNotFoundError:
            raise Tarea_no_Existe()

# Ejemplo de uso
lista = Lista_de_tarea()
lista.exportar_csv()
```

Cierre del Curso Complementario

Si llegaste hasta aquí… **felicitaciones, de verdad.**

No importa si hiciste este curso **antes** o **después** del curso principal: lo que importa es que estás aquí porque **quieres aprender**, y eso ya dice muchísimo de ti. Este curso fue pensado como un refuerzo práctico para que entiendas bien el funcionamiento de la **Programación Orientada a Objetos**, con ejemplos aplicados y con más detalle sobre conceptos que tal vez en el otro curso pasamos más por encima, como:

- Clases y objetos
- Herencia, encapsulamiento, polimorfismo
- Archivos JSON y CSV
- Manejo de errores y estructuras típicas de un programa bien diseñado

Tal vez ya te disté cuenta… **muchas de estas cosas no la usaras todos los días,** tal vez olvides algunas de ellas, otras ni siquiera aparecerán en el curso principal. Pero eso no significa que no sean importantes.

Lo importante es que ahora sabes que **existen**, sabes **para qué sirven**, y si algún día las necesitas, ya **tendrás una base firme para retomarlas, entenderlas y aplicarlas**.

Y si después de este, vas a lanzarte al curso el principal que compartí junto a este, entonces te deseo lo mejor. Dale con calma, a tu ritmo, y **no te frustres si algo no lo entiendes a la primera**.

Recuerda siempre lo que te digo:

No necesitas aprender todo de memoria.

Lo que necesitas es **entender cómo funciona**, por qué se usa, y dónde buscar si se te olvida algo.

Ese es el verdadero poder de un profesional en IT.

Y tú, con este curso, ya estás un paso más cerca.

Nos vemos en el siguiente nivel.

Nelson Arteaga – @Ors4tech