PYTHON

para IA en Ingeniería: de los fundamentos a las soluciones prácticas

Aprendizaje acelerado con ejemplos prácticos y mínima teoría

Volumen 1

Salas García, Javier • Cruz Martínez, Giorgio Mackenzie • Ávila Vilchis, Juan Carlos

Python para IA en Ingeniería: de los fundamentos a las soluciones prácticas

Aprendizaje acelerado con ejemplos prácticos
y mínima teoría

Volumen 1: Fundamentos de Programación en Python

Primera edición: febrero de 2025

ISBN: 9798312064438

Editorial: KDP Amazon

Diseño de la portada: Moran González, Miguel Armando

proyectos@javiersalasg.com

Índice general

Prólogo

"El conocimiento es de valor solo si puede llevarse a la práctica."
– **Anton Chekhov**

Bienvenido a **PYTHON para IA en Ingeniería: de los fundamentos a las soluciones prácticas**, una serie diseñada para proporcionar una ruta directa y eficiente desde los conceptos fundamentales de programación hasta la implementación de soluciones avanzadas con Inteligencia Artificial en diversos contextos de ingeniería.

Esta obra nace como respuesta a una necesidad creciente: proporcionar a ingenieros y profesionales técnicos las herramientas computacionales necesarias para abordar los desafíos contemporáneos, donde la IA se ha convertido en un elemento transformador. El enfoque adoptado en esta serie se distingue por priorizar la aplicación práctica sobre la teoría abstracta, sin sacrificar el rigor necesario para una comprensión sólida de los conceptos.

Estructura de la serie

La serie **PYTHON para IA en Ingeniería** se ha organizado estratégicamente en volúmenes independientes pero interrelacionados, cada uno correspondiente a un capítulo específico del programa completo. Esta estructura modular permite:

- Avanzar a un ritmo personalizado, adaptado a las necesidades específicas del lector.
- Profundizar en temas específicos sin la necesidad de abordar la serie completa.
- Construir conocimiento de manera progresiva, con referencias cruzadas que facilitan la integración de conceptos.
- Aplicar lo aprendido mediante ejercicios prácticos que refuerzan cada concepto presentado.

Los cuatro volúmenes que componen esta serie son:

1. **Fundamentos de Programación en Python** (el presente volumen)
2. **Introducción a la IA y Aprendizaje Automático**
3. **Implementación de Soluciones Avanzadas**
4. **Proyectos Integradores**

Sobre este volumen

El volumen que tiene en sus manos constituye la base sobre la cual se construirá todo el conocimiento posterior. En **Fundamentos de Programación**, se aborda el aprendizaje del lenguaje Python desde sus elementos más básicos, con un enfoque específicamente orientado hacia aplicaciones de Inteligencia Artificial.

A lo largo de este volumen, se presentan:

- **Conceptos fundamentales** de programación explicados de manera concisa y directa.
- **Ejemplos ejecutables** que ilustran cada concepto, disponibles para ser explorados y modificados.
- **Ejercicios prácticos** graduados en complejidad, desde simples aplicaciones de conceptos hasta desafíos que integran múltiples aspectos.
- **Soluciones completas** a todos los ejercicios, con explicaciones detalladas que refuerzan el aprendizaje.

El contenido se ha diseñado para ser accesible tanto para aquellos sin experiencia previa en programación como para quienes desean reafirmar sus conocimientos con un enfoque específico hacia la IA. Las explicaciones evitan deliberadamente la jerga técnica innecesaria, favoreciendo ejemplos concretos que ilustran cada concepto de manera transparente.

Metodología de aprendizaje

Esta serie adopta un enfoque de *aprendizaje acelerado* que se caracteriza por:

- **Mínima teoría, máxima práctica**: Se presentan conceptos teóricos de manera concisa, dedicando la mayor parte del contenido a ejemplos y aplicaciones prácticas.
- **Aprendizaje activo**: Cada concepto se acompaña de código ejecutable que el lector puede experimentar y modificar.
- **Enfoque progresivo**: Los conceptos se construyen secuencialmente, asegurando que cada nuevo elemento se fundamente en conocimientos previamente establecidos.

- **Aplicación inmediata**: Cada técnica o concepto se vincula directamente con aplicaciones relevantes en el contexto de la ingeniería y la IA.

Cómo utilizar este libro

Para aprovechar al máximo este material, se recomienda:

1. **Ejecutar cada ejemplo**: Todos los fragmentos de código incluyen hipervínculos que permiten acceder directamente a los archivos correspondientes, facilitando su ejecución y experimentación.
2. **Experimentar con modificaciones**: La comprensión se consolida al modificar los ejemplos y observar los resultados.
3. **Resolver los ejercicios**: Antes de consultar las soluciones, intentar resolver los ejercicios propuestos aplicando los conceptos aprendidos.
4. **Integrar conceptos**: Buscar activamente conexiones entre diferentes temas, anticipando su aplicación en contextos de IA.

Acceso a recursos digitales

Para acceder a todos los scripts presentados en este libro, así como a contenido digital exclusivo:

1. Visite `https://electrokumo.com/books/` para descargar los archivos de código.
2. Envíe su comprobante de compra a `contact@electrokumo.com`.
3. Recibirá un código de acceso único que le permitirá:
   - Descargar todos los scripts y ejemplos en formato ejecutable.
   - Acceder a la versión electrónica interactiva del libro.
   - Visualizar videos explicativos adicionales que complementan el contenido impreso.

Agradecimientos

Este trabajo no habría sido posible sin la contribución y retroalimentación de numerosos estudiantes y colegas que han participado en los cursos que sirvieron como base para esta serie. Sus preguntas, desafíos y sugerencias han sido fundamentales para refinar el enfoque y contenido presentados.

Dr. Javier Salas García

Esta página se ha dejado intencionalmente en blanco.

Capítulo 1

Fundamentos de Programación en Python

1.1 Introducción a la Programación

Objetivos de Aprendizaje

Al finalizar esta sección, se estará en capacidad de:

- Comprender los conceptos fundamentales de la programación, incluyendo algoritmos, sintaxis y semántica
- Identificar y explicar los diferentes paradigmas de programación, con énfasis en su aplicación en inteligencia artificial
- Configurar correctamente un entorno de desarrollo para Python, incluyendo el manejo de Anaconda y Visual Studio Code
- Aplicar técnicas básicas de depuración para identificar y corregir errores en scripts de Python
- Documentar adecuadamente el código fuente siguiendo las mejores prácticas de la industria

1.1.1 Conceptos básicos de programación

La programación es el proceso de diseñar y escribir instrucciones que una computadora puede ejecutar para realizar tareas específicas. Estas instrucciones, conocidas como código fuente, son escritas en un lenguaje de programación y luego traducidas en instrucciones que el hardware de la computadora puede entender.

Elementos fundamentales de la programación Para comprender la programación, es esencial conocer sus elementos básicos:

- **Algoritmos:** Un algoritmo es una secuencia finita de pasos definidos para resolver un problema o realizar una tarea. Es la base de cualquier programa informático.
- **Lenguajes de programación:** Existen distintos lenguajes de programación, cada uno con su propia sintaxis y reglas. Algunos de los más populares son Python, Java, C++, JavaScript y Ruby.
- **Sintaxis:** Es el conjunto de reglas que define la estructura correcta de las instrucciones en un lenguaje de programación.
- **Semántica:** Se refiere al significado de las instrucciones dentro del código. Un programa puede ser sintácticamente correcto pero semánticamente incorrecto si no produce el resultado esperado.
- **Paradigmas de programación:** Son enfoques o estilos de programación. Entre los más comunes están:
  - Programación imperativa: Se basa en la ejecución secuencial de instrucciones. Ejemplo: C, Python.
  - Programación orientada a objetos (POO): Se basa en la creación de objetos que encapsulan datos y comportamientos. Ejemplo: Java, C++.
  - Programación funcional: Se enfoca en la evaluación de funciones y evita el uso de estados mutables. Ejemplo: Haskell, Lisp.

1.1.2 Entornos de desarrollo integrados (IDE)

Un Entorno de Desarrollo Integrado (IDE) es una aplicación de software que proporciona herramientas para facilitar el desarrollo de programas. Los IDEs varían en complejidad, desde editores de texto con resaltado de sintaxis hasta suites completas con depuradores y herramientas de diseño visual.

Características comunes de un IDE:

- **Editor de código fuente:** Permite escribir y editar el código con funciones como resaltado de sintaxis, autocompletado y sugerencias de código.
- **Compilador o intérprete:** Facilita la traducción del código fuente a lenguaje de máquina o bytecode.
- **Depurador:** Ayuda a encontrar y corregir errores en el código mediante la ejecución paso a paso y la inspección de variables.
- **Herramientas de construcción:** Automatizan el proceso de compilación y enlazado de múltiples archivos de código fuente.
- **Control de versiones:** Se integran con sistemas como Git para ges-

tionar cambios en el código y colaborar en proyectos.

Selección del IDE

Al elegir un entorno de desarrollo integrado (IDE) para programación en Python, se debe considerar:

- La cantidad de memoria RAM disponible en el equipo
- El tipo de proyectos que se desarrollarán
- La experiencia previa con herramientas de desarrollo
- La disponibilidad de extensiones para inteligencia artificial

IDEs populares para Python:

- **PyCharm:** Uno de los IDEs más completos para Python, con gran soporte para desarrollo web, herramientas de depuración y diseño de interfaces gráficas.
- **Visual Studio Code (VS Code):** Editor de código multiplataforma muy popular con extensiones para casi cualquier lenguaje, incluyendo Python.
- **Thonny:** IDE sencillo diseñado para aprender a programar en Python, con una interfaz intuitiva y herramientas para visualizar el flujo de ejecución.
- **IDLE:** El IDE oficial de Python que viene incluido en la instalación, útil para principiantes y para ejecutar scripts rápidamente.

1.1.3 Anaconda y Visual Studio Code

¿Qué es Anaconda?

Anaconda es una distribución de Python que incluye el lenguaje base, bibliotecas de uso común y un gestor de paquetes llamado Conda. Facilita la instalación de paquetes y la gestión de entornos, lo cual es muy útil en proyectos de ciencia de datos e IA.

Instalación de Anaconda: Paso a paso

1. Descarga Anaconda desde su sitio web oficial: https://www.anaconda.com/download
2. Ejecuta el instalador y sigue las instrucciones. Asegúrate de marcar la opción para añadir Anaconda al PATH de tu sistema.

Instalación de Python

Durante la instalación de Python a través de Anaconda, se debe asegurar de:

- Marcar la opción .Adding Anaconda to PATH.en el instalador
- Tener al menos 5GB de espacio libre en disco
- No tener otras versiones de Python instaladas previamente
- Cerrar todos los programas que puedan interferir con la instalación

Creación de un entorno virtual con Python 3.9

1. Abre la línea de comandos (cmd en Windows, Terminal en macOS/Linux).
2. Crea el entorno .entorno1çon Python 3.9:

```
conda create -n entorno1 python=3.9
```

3. Activa el entorno:

```
conda activate entorno1
```

Instalación de paquetes en el entorno

1. Asegúrate de que el entorno .entorno1.esté activado.
2. Instala los paquetes necesarios, por ejemplo:

```
conda install numpy pandas scikit-learn
```

Visual Studio Code (VS Code): Un IDE potente

- VS Code es un editor de código multiplataforma y gratuito.
- Tiene extensiones para Python que facilitan el desarrollo, como resaltado de sintaxis, IntelliSense (autocompletado inteligente) y depuración.
- Puedes integrar VS Code con tu entorno Anaconda para ejecutar y depurar tus programas de IA.

Configuración de VS Code para usar el entorno Anaconda Para seleccionar el intérprete de Python en VS Code, se puede hacer clic en la barra de estado en la esquina inferior derecha de la ventana. En esta barra se mostrará el intérprete de Python actual, que generalmente aparece como "Python"seguido de su versión. Al hacer clic en esta área, se desplegará una lista de todos los intérpretes disponibles en el sistema. En esta lista se podrán identificar los entornos de Anaconda por su ruta característica. Para el entorno creado anteriormente, se deberá buscar una entrada que contenga

.entorno1z que muestre la versión de Python 3.9. La ruta completa variará según el sistema operativo:

En Windows: aparecerá como "Python 3.9.x ('entorno1': conda)çon una ruta similar a `~\anaconda3 \envs\entorno1\python.exe` En macOS/Linux: se mostrará como "Python 3.9.x ('entorno1': conda)çon una ruta similar a /home/TU_USUARIO/anaconda3/envs/entorno1/bin/python

Una vez seleccionado el intérprete correcto, VS Code actualizará automáticamente la configuración y estará listo para ejecutar código Python utilizando el entorno de Anaconda especificado.

Configuración Inicial

Para comenzar a programar en Python, se requiere:

- Anaconda Navigator instalado y actualizado
- Visual Studio Code con las extensiones de Python
- Un entorno virtual creado y activado
- Conocimientos básicos de la terminal o línea de comandos

1.1.4 Depuración de Scripts Python en Visual Studio Code

La depuración es una parte esencial del desarrollo de software que permite ejecutar el código paso a paso para encontrar y corregir errores. Visual Studio Code proporciona herramientas poderosas para la depuración de scripts Python.

Configuración inicial para depuración

1. Asegurarse de tener instalada la extensión de Python para VS Code.
2. Verificar que el intérprete de Python correcto esté seleccionado (el configurado anteriormente con Anaconda).
3. Tener abierto el archivo .py que se desea depurar.

Depuración de Código

Se debe tener especial cuidado con:

- La indentación del código, que es significativa en Python
- El uso correcto de mayúsculas y minúsculas
- La coincidencia de paréntesis y llaves
- La gestión adecuada de las variables globales

Establecer puntos de interrupción (breakpoints)

Un punto de interrupción puede establecerse de dos formas:

1. Haciendo clic en el margen izquierdo junto al número de línea donde se

desea pausar la ejecución.

2. Colocando el cursor en la línea deseada y presionando F9.

El punto de interrupción se visualizará como un punto rojo en el margen.

Iniciar la depuración

Existen tres métodos para iniciar la depuración:

1. Presionar F5 para iniciar la depuración directamente.
2. Seleccionar el menú .Ejecutar ¿Iniciar depuración".
3. Hacer clic en el icono de "playçon el insecto en el panel de depuración.

Controles de depuración

Durante la depuración, se tienen disponibles los siguientes controles principales:

1. **Continuar (F5):** Continúa la ejecución hasta el siguiente punto de interrupción.
2. **Paso a paso por instrucciones (F10):** Ejecuta la siguiente línea de código sin entrar en funciones.
3. **Paso a paso por procedimientos (F11):** Ejecuta la siguiente línea y entra en las funciones.
4. **Paso a paso para salir (Shift+F11):** Sale de la función actual.
5. **Reiniciar (Ctrl+Shift+F5):** Reinicia la sesión de depuración.
6. **Detener (Shift+F5):** Finaliza la sesión de depuración.

Panel de depuración

Durante la sesión de depuración, VS Code muestra cuatro paneles principales:

1. **Variables:** Muestra todas las variables locales y globales y sus valores actuales.
2. **Inspección:** Permite agregar expresiones para monitorear sus valores.
3. **Pila de llamadas:** Muestra la secuencia de llamadas a funciones que llevaron al punto actual.
4. **Puntos de interrupción:** Lista todos los puntos de interrupción establecidos.

Tipos especiales de puntos de interrupción

VS Code ofrece tres tipos avanzados de puntos de interrupción:

1. **Puntos de interrupción condicionales:** Se activan solo cuando se cumple una condición específica.
2. **Puntos de interrupción de expresión:** Se activan cuando una expresión cambia de valor.

3. **Puntos de interrupción de registro:** Registran un mensaje sin detener la ejecución.

Proceso para crear un punto de interrupción condicional:

1. Establecer un punto de interrupción normal haciendo clic en el margen.
2. Hacer clic derecho sobre el punto de interrupción.
3. Seleccionar .ªEdªitar punto de interrupción".
4. Ingresar la condición deseada (por ejemplo: "x ¿10").

Ejemplo práctico de depuración

Consideremos el siguiente script de ejemplo:

Ejemplo 1.1: Cálculo de promedios (calculo_promedios.py)

```
1  def calcular_promedio(numeros):
2      total = 0
3      for num in numeros:
4          total += num
5      return total / len(numeros)
6
7  def procesar_datos(lista):
8      resultados = []
9      for valor in lista:
10         promedio = calcular_promedio(valor)
11         resultados.append(promedio)
12     return resultados
13
14 # Datos de prueba
15 datos = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
16 resultados = procesar_datos(datos)
17 print(resultados)
```

Pasos para depurar el script anterior:

1. Establecer un punto de interrupción en la línea `total += num`.
2. Iniciar la depuración presionando F5.
3. Usar F10 para observar cómo se actualiza el valor de `total` en cada iteración.
4. Utilizar F11 para entrar en la función `calcular_promedio`.
5. Observar las variables en el panel de depuración.

Pasos recomendados para una depuración efectiva:

1. Utilizar la ventana de Inspección"para monitorear expresiones complejas.
2. Implementar puntos de interrupción condicionales para casos específicos.

3. Emplear la consola de depuración para ejecutar comandos y probar expresiones.
4. Mantener visible el panel de variables para seguir los cambios de estado.

Configuración de depuración remota

Para depurar código en una máquina remota o contenedor:

1. Instalar la extensión Remote Development.[en] VS Code.
2. Establecer la conexión con el entorno remoto deseado.
3. Configurar los puntos de interrupción necesarios.
4. Iniciar la sesión de depuración normalmente con F5.

Esta funcionalidad resulta especialmente útil al trabajar con servidores o entornos de desarrollo containerizados.

Conceptos Clave de Programación

La programación se fundamenta en:
- Algoritmos: secuencias de pasos para resolver problemas
- Sintaxis: reglas que definen cómo escribir código válido
- Variables: espacios de memoria para almacenar datos
- Funciones: bloques de código reutilizable
- Estructuras de control: mecanismos para dirigir el flujo del programa

1.2 Estructuras de Datos y Control de Flujo

Objetivos de Aprendizaje

Al finalizar esta sección, se estará en capacidad de:
- Manipular diferentes tipos de datos en Python, comprendiendo sus características y aplicaciones
- Implementar estructuras de control condicionales para crear programas con capacidad de toma de decisiones
- Utilizar estructuras repetitivas para automatizar tareas recurrentes en el procesamiento de datos
- Trabajar eficientemente con estructuras de datos fundamentales como listas, tuplas y diccionarios
- Aplicar operaciones básicas sobre conjuntos para el manejo de datos únicos y sus relaciones

1.2.1 Variables, tipos de datos (estáticos y dinámicos) y operadores

En programación, una variable es un espacio de memoria donde se almacena un valor. Este valor puede cambiar durante la ejecución del programa. En Python, las variables se crean asignándoles un valor, y su tipo se determina dinámicamente.

A continuación, veremos un script básico que muestra cómo se declaran variables de diferentes tipos en Python. Este ejemplo ilustra la sintaxis básica de declaración de variables:

Ejemplo 1.2: Declaración de variables

```
nombre = "Juan"      # Variable de tipo cadena (str)
edad = 30            # Variable de tipo entero (int)
altura = 1.85        # Variable de tipo flotante (float)
es_estudiante = True  # Variable de tipo booleano
(bool)
```

En este script simple, podemos observar cómo Python infiere automáticamente el tipo de dato basándose en el valor asignado. No es necesario declarar explícitamente el tipo como en otros lenguajes.

Tipos de datos:

Los tipos de datos definen la naturaleza de la información que puede contener una variable. Python es un lenguaje con tipado dinámico, lo que significa que el tipo de una variable se infiere en tiempo de ejecución y puede cambiar.

Tipos de datos en Python:

- **Numéricos:** *int* (enteros), *float* (decimales), *complex* (complejos)
- **Cadenas:** *str* (texto)
- **Booleanos:** *bool* (True o False)
- **Colecciones:** *list* (listas), *tuple* (tuplas), *dict* (diccionarios), *set* (conjuntos)

Primeros Pasos en Python

Para familiarizarse con Python, se recomienda:
- Comenzar con scripts simples que realicen operaciones básicas
- Utilizar el modo interactivo de Python para experimentar
- Documentar el código desde el principio
- Practicar con ejercicios de dificultad progresiva

Operadores:

Los operadores son símbolos que realizan operaciones en uno o dos operandos.

Para comprender mejor su funcionamiento, analizaremos el siguiente archivo que demuestra los diferentes tipos de operadores en Python:

Ejemplo 1.3: Operadores básicos en Python (operadores_basicos.py)

```python
# Demostración de operadores aritméticos
x = 10
y = 3

# Operadores aritméticos
suma = x + y            # Suma: 13
resta = x - y           # Resta: 7
multiplicacion = x * y  # Multiplicación: 30
division = x / y        # División: 3.333...
modulo = x % y          # Módulo (resto): 1
potencia = x ** y       # Potencia: 1000

# Operadores de comparación
es_mayor = x > y        # True
es_igual = x == y       # False

# Operadores lógicos
resultado_and = True and False     # False
resultado_or = True or False       # True
resultado_not = not True           # False

print(f"Suma: {suma}")
print(f"Resta: {resta}")
print(f"Multiplicación: {multiplicacion}")
print(f"División: {division}")
print(f"Módulo: {modulo}")
print(f"Potencia: {potencia}")
print(f"Es mayor: {es_mayor}")
print(f"Es igual: {es_igual}")
print(f"AND lógico: {resultado_and}")
print(f"OR lógico: {resultado_or}")
print(f"NOT lógico: {resultado_not}")
```

Este archivo muestra los tres tipos principales de operadores en Python:

- **Operadores aritméticos:** Realizan operaciones matemáticas básicas (+, -, *, /,
- **Operadores de comparación:** Comparan valores y devuelven un resultado booleano (¿, ¡, ==, !=).
- **Operadores lógicos:** Realizan operaciones con valores booleanos (and, or, not).

Para complementar, veamos un script corto que ilustra los operadores de asignación más comunes:

Ejemplo 1.4: Operadores de asignación

```
x = 5              # Asignación simple
x += 3             # Equivalente a: x = x + 3
x -= 2             # Equivalente a: x = x - 2
x *= 4             # Equivalente a: x = x * 4
```

Este script demuestra cómo los operadores de asignación combinada permiten modificar una variable utilizando su valor actual de manera más concisa.

Tipado estático vs dinámico:

- **Tipado estático:** El tipo de una variable se define al declararla y no puede cambiar (como en C++, Java).
- **Tipado dinámico:** El tipo de una variable se infiere en tiempo de ejecución y puede cambiar (como en Python).

Optimización de Diccionarios

Para un uso eficiente de diccionarios:
- Utilizar el método get() para acceso seguro a claves
- Emplear setdefault() para inicialización con valores por defecto
- Aprovechar dict comprehensions para creación concisa
- Considerar defaultdict para valores por defecto automáticos

Para entender mejor cómo funciona el tipado dinámico en Python, analicemos el siguiente archivo:

Ejemplo 1.5: tipos de variables (tipos_variables.py)

```
1  # Ejemplo de tipado dinámico en Python
2  variable = 42               # Inicialmente es un
       entero
3  print(f"Tipo inicial: {type(variable)}")
4
5  variable = "Hola mundo"     # Ahora es una cadena
6  print(f"Tipo después de cambio: {type(variable)}")
7
8  variable = 3.14             # Ahora es un float
9  print(f"Tipo final: {type(variable)}")
10
11 # Demostración de operaciones con diferentes tipos
12 numero_entero = 5
13 numero_flotante = 2.5
14 texto = "Python"
15
16 # Operaciones válidas
17 suma_numeros = numero_entero + numero_flotante
18 multiplicacion_texto = texto * 2
19
```

```
20 print(f"Suma de entero y flotante: {suma_numeros}")
21 print(f"Multiplicación de texto:
       {multiplicacion_texto}")
22
23 # Conversión explícita de tipos (casting)
24 numero_como_texto = str(numero_entero)
25 texto_como_numero = int("123")
26
27 print(f"Número convertido a texto:
       {numero_como_texto}, tipo:
       {type(numero_como_texto)}")
28 print(f"Texto convertido a número:
       {texto_como_numero}, tipo:
       {type(texto_como_numero)}")
```

Este archivo demuestra varias características importantes del sistema de tipos en Python:

- La capacidad de cambiar el tipo de una variable durante la ejecución
- Operaciones entre diferentes tipos de datos
- Conversión explícita entre tipos (casting)
- El uso de la función `type()` para verificar el tipo de una variable

El código muestra cómo una misma variable puede contener diferentes tipos de datos a lo largo de la ejecución del programa, algo que no sería posible en un lenguaje de tipado estático. También ilustra cómo Python maneja automáticamente algunas conversiones entre tipos, como la suma entre enteros y flotantes, y cómo realizar conversiones explícitas cuando son necesarias.

1.2.2 Estructuras de control: condicionales y repetitivas

Las estructuras de control permiten modificar el flujo de ejecución de un programa, posibilitando la toma de decisiones y la repetición de instrucciones según se requiera. En Python, estas estructuras se implementan de manera clara y sintácticamente simple, facilitando la escritura de código legible y mantenible.

Estructuras Condicionales Las estructuras condicionales permiten ejecutar diferentes bloques de código según se cumplan o no determinadas condiciones. En Python, la indentación es fundamental para definir los bloques de código que pertenecen a cada estructura condicional.

Sentencia if La sentencia if es la estructura condicional más básica. Se ejecuta un bloque de código solo si la condición especificada es verdadera.

Ejemplo 1.6: Verificación de número positivo

```
numero = 5
if numero > 0:
print("El número es positivo")
```

En este ejemplo, se evalúa si la variable 'numero' es mayor que cero. La instrucción print se ejecuta únicamente si la condición es verdadera. Se observa que el bloque de código dependiente de la condición se encuentra indentado.

Ejemplo 1.7: Verificación de edad

```
edad = 18
if edad >= 18:
print("Es mayor de edad")
print("Puede realizar trámites legales")
```

En este caso, se muestra cómo un bloque if puede contener múltiples instrucciones. Todas las líneas indentadas después de la condición se ejecutan solo si esta se cumple.

Sentencia if-else La estructura if-else permite especificar un bloque de código alternativo que se ejecutará cuando la condición del if no se cumpla.

Ejemplo 1.8: Validación de contraseña

```
password = "secreto123"
if len(password) >= 8:
print("Contraseña válida")
else:
print("La contraseña debe tener al menos 8
caracteres")
```

Ejemplo 1.9: Control de acceso con verificación múltiple (control_acceso.py)

```
1  """
2  Script para control de acceso que verifica múltiples
       condiciones
3  Incluye validación de usuario y contraseña con
       diferentes niveles de acceso
4  """
5
6  def verificar_acceso(usuario, password):
7      # Definición de credenciales válidas
8      usuarios_admin = {"admin": "admin123", "root":
       "root456"}
9      usuarios_normales = {"usuario1": "pass123",
       "usuario2": "pass456"}
10
11     # Verificación de existencia del usuario
```

```
12      if usuario in usuarios_admin:
13          if password == usuarios_admin[usuario]:
14              return "Acceso concedido - Nivel
        Administrador"
15          else:
16              return "Contraseña incorrecta"
17      else:
18          if usuario in usuarios_normales:
19              if password ==
        usuarios_normales[usuario]:
20                  return "Acceso concedido - Nivel
        Usuario"
21              else:
22                  return "Contraseña incorrecta"
23          else:
24              return "Usuario no encontrado"
25
26  # Ejemplos de uso
27  print(verificar_acceso("admin", "admin123"))      #
        Acceso administrador
28  print(verificar_acceso("usuario1", "pass123"))    #
        Acceso usuario normal
29  print(verificar_acceso("admin", "password"))      #
        Contraseña incorrecta
30  print(verificar_acceso("usuario3", "pass789"))    #
        Usuario no existe
```

Sentencia if-elif-else La estructura if-elif-else permite evaluar múltiples condiciones en secuencia. El bloque elif (else if) se ejecuta cuando la condición anterior es falsa y su propia condición es verdadera. Se pueden incluir tantos bloques elif como sean necesarios.

Ejemplo 1.10: Calificación numérica

```
calificacion = 85
if calificacion >= 90:
print("Calificación: A")
elif calificacion >= 80:
print("Calificación: B")
elif calificacion >= 70:
print("Calificación: C")
else:
print("Calificación: D")
```

Ejemplo 1.11: Días de la semana

```
dia = 3
if dia == 1:
print("Lunes")
elif dia == 2:
print("Martes")
elif dia == 3:
print("Miércoles")
else:
print("Otro día de la semana")
```

Ejemplo 1.12: Sistema de descuentos por compra (sistema_descuentos.py)

```
"""
Sistema de cálculo de descuentos basado en múltiples criterios:
- Monto de compra
- Tipo de cliente
- Temporada de venta
"""

def calcular_descuento(monto, tipo_cliente, es_temporada_baja=False):
    """
    Calcula el descuento aplicable a una compra.

    Parámetros:
        monto (float): Monto total de la compra
        tipo_cliente (str): Tipo de cliente ('regular', 'premium', 'vip')
        es_temporada_baja (bool): Indica si es temporada baja

    Retorna:
        tuple: (descuento_porcentaje, monto_final)
    """
    descuento = 0

    # Descuento por monto de compra
    if monto >= 5000:
        descuento = 20
    elif monto >= 3000:
        descuento = 15
    elif monto >= 1000:
        descuento = 10

    # Descuento adicional por tipo de cliente
    if tipo_cliente == 'vip':
        descuento += 10
    elif tipo_cliente == 'premium':
```

```
34          descuento += 5
35
36      # Descuento por temporada
37      if es_temporada_baja:
38          descuento += 5
39
40      # Limitar descuento máximo a 30%
41      descuento = min(descuento, 30)
42
43      # Cálculo del monto final
44      monto_final = monto * (1 - descuento/100)
45
46      return (descuento, monto_final)
47
48  # Ejemplos de uso
49  casos_prueba = [
50      (1500, 'regular', False),
51      (3500, 'premium', True),
52      (6000, 'vip', False)
53  ]
54
55  for monto, tipo, temporada in casos_prueba:
56      descuento, final = calcular_descuento(monto,
        tipo, temporada)
57      print(f"Compra de ${monto}")
58      print(f"Cliente {tipo} - Temporada baja:
        {temporada}")
59      print(f"Descuento aplicado: {descuento}%")
60      print(f"Monto final: ${final:.2f}\n")
```

Estructuras Repetitivas En programación, se denomina estructura repetitiva o bucle a aquella que permite ejecutar un bloque de código múltiples veces. Se pueden considerar como instrucciones que indican al programa que repita ciertas acciones bajo condiciones específicas.

Bucle while El bucle while se utiliza para ejecutar un bloque de código mientras una condición especificada se mantenga verdadera. Este tipo de estructura se emplea cuando no se conoce con exactitud el número de repeticiones necesarias.

Ejemplo 1.13: Contador simple

```
contador = 1
while contador <= 5:
print(f"Número: {contador}")
contador = contador + 1
```

En el ejemplo anterior, se puede observar cómo el programa imprime números del 1 al 5. La variable contador se incrementa en cada iteración hasta que alcanza el valor límite establecido.

Ejemplo 1.14: Suma hasta límite

```
suma = 0
numero = 1
while suma < 10:
suma = suma + numero
print(f"Sumando {numero}: total = {suma}")
numero = numero + 1
```

En este caso, se realiza una suma continua de números hasta que el total supera o iguala a 10. En cada iteración, se muestra el progreso de la suma.

Ejemplo 1.15: Contador regresivo (contador_regresivo.py)

```
1  """
2  Se implementa un programa que realiza una cuenta
       regresiva desde un número dado
3  """
4
5  def cuenta_regresiva(inicio):
6      """
7      Se realiza una cuenta regresiva desde el número
       inicial hasta 1
8      """
9      numero = inicio
10
11     print(f"Se inicia la cuenta regresiva desde
       {inicio}")
12
13     while numero > 0:
14         print(f"Número: {numero}")
15         numero = numero - 1
16
17     print("Se ha completado la cuenta regresiva")
18
19 # Se ejecuta una cuenta regresiva desde 5
20 cuenta_regresiva(5)
21
22 # Se ejecuta otra cuenta regresiva desde 3
23 print("\nSe inicia otra cuenta regresiva:")
```

```
24  cuenta_regresiva(3)
```

⚠ Errores Comunes en Bucles

Se deben evitar:

- Modificar la lista mientras se itera sobre ella
- Olvidar actualizar variables de control en bucles while
- Crear bucles infinitos por condiciones mal definidas
- Usar índices innecesariamente cuando se puede iterar directamente

Bucle for El bucle for se emplea cuando se requiere iterar sobre una secuencia conocida de elementos. Se utiliza comúnmente para recorrer listas, rangos de números o cadenas de texto, donde el número de iteraciones se encuentra determinado por el tamaño de la secuencia.

Ejemplo 1.16: Recorrido de lista

```
colores = ["rojo", "verde", "azul"]
for color in colores:
print(f"Se ha seleccionado el color {color}")
```

En el ejemplo anterior, se muestra cómo se recorre una lista de colores. Por cada elemento de la lista, se imprime un mensaje indicando el color seleccionado.

Ejemplo 1.17: Secuencia numérica

```
for numero in range(1, 6):
print(f"Se procesa el número {numero}")
```

Ejemplo 1.18: Secuencia de mensajes (secuencia_mensajes.py)

```
1  """
2  Se implementa un programa que muestra una secuencia
       de mensajes
3  para diferentes momentos del día
4  """
5
6  def mostrar_mensajes_del_dia():
7      # Se definen los momentos del día
8      momentos = ["mañana", "tarde", "noche"]
9
10     print("Se inicia la secuencia de mensajes")
11     print("-" * 30)
12
13     # Se procesan los mensajes para cada momento
14     for momento in momentos:
```

```
15          if momento == "mañana":
16              temperatura = 20
17          elif momento == "tarde":
18              temperatura = 25
19          else:
20              temperatura = 18
21
22          print(f"Se registra en la {momento}:")
23          print(f"La temperatura es de {temperatura}
        grados")
24          print("-" * 30)
25
26      print("Se ha completado la secuencia de
        mensajes")
27
28  # Se ejecuta el programa de mensajes
29  mostrar_mensajes_del_dia()
```

Sentencias break y continue Se utilizan dos sentencias especiales para controlar el flujo de los bucles: - break: se emplea para terminar el bucle inmediatamente - continue: se utiliza para saltar a la siguiente iteración del bucle

Ejemplo 1.19: Búsqueda simple

```
numeros = [4, 7, 2, 9, 1]
numero_buscado = 2
for numero in numeros:
if numero == numero_buscado:
print(f"Se ha encontrado el número {numero_buscado}")
break
print(f"Se revisa el número {numero}")
```

Ejemplo 1.20: Filtro de números

```
for numero in range(1, 6):
if numero % 2 == 0:     # Si el número es par
continue            # Se salta a la siguiente iteración
print(f"Se procesa el número impar: {numero}")
```

Ejemplo 1.21: Filtro de valores (filtro_valores.py)

```
"""
Se implementa un programa que filtra valores según diferentes criterios
"""

def filtrar_valores():
    # Se define la lista de valores a procesar
    valores = [15, -3, 8, -1, 12, 4, -6, 10]

    print("Se inicia el proceso de filtrado")
    print("-" * 30)

    # Se procesan solo los números positivos menores a 10
    print("Se analizan los valores:")
    for valor in valores:
        # Se omiten los números negativos
        if valor < 0:
            print(f"Se omite el valor negativo: {valor}")
            continue

        # Se termina si se encuentra un valor mayor a 10
        if valor > 10:
            print(f"Se encuentra valor mayor a 10: {valor}")
            print("Se detiene el proceso")
            break

        # Se procesan los valores válidos
        print(f"Se procesa el valor: {valor}")

    print("-" * 30)
    print("Se ha completado el proceso de filtrado")

# Se ejecuta el proceso de filtrado
filtrar_valores()
```

Bucles anidados Se denomina bucle anidado a la estructura que se forma cuando se coloca un bucle dentro de otro bucle. En esta estructura, por cada iteración del bucle exterior, se ejecuta el bucle interior completo.

Ejemplo 1.22: Calendario simple

```
semanas = ["Semana 1", "Semana 2"]
dias = ["Lunes", "Martes", "Miércoles"]
for semana in semanas:
print(f"Se inicia {semana}:")
for dia in dias:
print(f"Se procesa {dia}")
print("---")
```

Ejemplo 1.23: Matriz básica

```
for fila in range(3):
for columna in range(3):
print("0 ", end="")  # Se utiliza end="" para evitar
el salto de línea
print()  # Se agrega un salto de línea al final de
cada fila
```

Ejemplo 1.24: Patrones básicos (patrones_basicos.py)

```
1  """
2  Se implementa un programa que muestra una secuencia
       de mensajes
3  para diferentes momentos del día
4  """
5
6  def mostrar_mensajes_del_dia():
7      # Se definen los momentos del día
8      momentos = ["mañana", "tarde", "noche"]
9
10     print("Se inicia la secuencia de mensajes")
11     print("-" * 30)
12
13     # Se procesan los mensajes para cada momento
14     for momento in momentos:
15         if momento == "mañana":
16             temperatura = 20
17         elif momento == "tarde":
18             temperatura = 25
19         else:
20             temperatura = 18
21
22         print(f"Se registra en la {momento}:")
23         print(f"La temperatura es de {temperatura}
       grados")
24         print("-" * 30)
25
26     print("Se ha completado la secuencia de
       mensajes")
27
```

```
28 # Se ejecuta el programa de mensajes
29 mostrar_mensajes_del_dia()
```

1.2.3 Listas, tuplas, diccionarios y conjuntos

En la programación, frecuentemente se necesita trabajar con conjuntos de datos relacionados. Por ejemplo, al desarrollar un programa que maneje las calificaciones de un grupo de estudiantes, resultaría poco práctico crear una variable diferente para cada calificación. Para resolver esta situación, Python proporciona diferentes estructuras de datos que permiten agrupar y organizar información de manera eficiente.

Listas Una lista es una estructura de datos que permite almacenar múltiples elementos en una sola variable. Se puede pensar en una lista como en un casillero con compartimentos numerados, donde cada compartimento puede almacenar un valor. Una característica importante de las listas es que pueden contener elementos de diferentes tipos: números, texto, o incluso otras listas.

Para crear una lista en Python, se utilizan corchetes '[]' y los elementos se separan por comas. Cada elemento en la lista ocupa una posición específica, y estas posiciones se numeran comenzando desde 0. Esta numeración se conoce como índice.

Ejemplo 1.25: Creación y acceso a elementos de una lista (lista_basica.py)

```
1  """
2  Se muestra la creación de una lista y cómo acceder a
       sus elementos.
3  Una lista es una estructura que permite almacenar
       múltiples valores en una sola variable.
4  """
5
6  # Se crea una lista de calificaciones de un
       estudiante
7  calificaciones = [8, 9, 7, 10, 8]
8
9  # Se muestra la primera calificación (elemento en
       posición 0)
10 print(f"Primera calificación: {calificaciones[0]}")
11
12 # Se muestra la tercera calificación (elemento en
       posición 2)
13 print(f"Tercera calificación: {calificaciones[2]}")
14
15 # Se muestra la última calificación (elemento en
       posición 4)
16 print(f"Última calificación: {calificaciones[4]}")
```

```
17
18  # Se muestran todas las calificaciones
19  print(f"Todas las calificaciones: {calificaciones}")
```

En el ejemplo anterior, se crea una lista llamada ‘calificaciones‘ que contiene cinco elementos. Para acceder a un elemento específico de la lista, se utiliza el nombre de la lista seguido de corchetes que contienen el índice del elemento deseado. Es importante notar que el primer elemento se accede con el índice 0, por lo que ‘calificaciones[0]‘ nos da el primer elemento, ‘calificaciones[2]‘ el tercer elemento y así sucesivamente. Este sistema de numeración, aunque puede parecer poco intuitivo al principio, es común en muchos lenguajes de programación y tiene ventajas importantes en el manejo eficiente de la memoria.

Una característica fundamental de las listas es que son mutables, es decir, se pueden modificar después de su creación. Esto permite realizar diversas operaciones como cambiar valores existentes, agregar nuevos elementos o eliminar elementos de la lista.

Ejemplo 1.26: Operaciones básicas con listas (operaciones_lista.py)

```
1   """
2   Se presentan las operaciones básicas que pueden
        realizarse con una lista:
3   - Modificar elementos
4   - Agregar nuevos elementos
5   - Obtener la cantidad de elementos
6   """
7
8   # Se crea una lista de calificaciones
9   calificaciones = [8, 9, 7, 10, 8]
10  print(f"Calificaciones originales: {calificaciones}")
11
12  # Se modifica la tercera calificación (posición 2)
13  calificaciones[2] = 8
14  print(f"Calificaciones después de modificar la
        tercera nota: {calificaciones}")
15
16  # Se agrega una nueva calificación al final de la
        lista
17  calificaciones.append(9)
18  print(f"Calificaciones después de agregar una nota:
        {calificaciones}")
19
20  # Se obtiene la cantidad total de calificaciones
21  total_calificaciones = len(calificaciones)
22  print(f"Cantidad total de calificaciones:
        {total_calificaciones}")
```

En este segundo ejemplo se observan tres operaciones fundamentales que pueden realizarse con las listas. La primera es la modificación de un elemento existente, que se realiza asignando un nuevo valor a una posición específica usando su índice. La segunda operación muestra cómo agregar un nuevo elemento al final de la lista utilizando el método ‘append()‘, el cual es muy útil cuando se necesita expandir la lista dinámicamente. La tercera operación demuestra el uso de la función ‘len()‘, que permite conocer cuántos elementos contiene la lista en total. Esta función es particularmente útil cuando se necesita procesar todos los elementos de una lista o verificar si la lista está vacía.

Las listas son especialmente útiles cuando se necesita mantener un orden específico de los elementos y cuando los datos pueden cambiar durante la ejecución del programa. Su flexibilidad para almacenar diferentes tipos de datos y su capacidad de modificación las convierten en una de las estructuras de datos más versátiles en Python.

Las listas en Python ofrecen muchas más operaciones que permiten manipular los datos de formas diferentes. Algunas de las operaciones más utilizadas incluyen la eliminación de elementos, la inserción en posiciones específicas y la ordenación de los elementos.

El método ‘pop()‘ permite eliminar elementos de una lista. Cuando se usa sin argumentos, elimina y devuelve el último elemento. El método ‘insert()‘ permite agregar un elemento en cualquier posición de la lista, desplazando los elementos existentes hacia la derecha. Para ordenar los elementos de una lista, se utiliza el método ‘sort()‘, el cual modifica la lista original ordenando sus elementos. También es posible encontrar la posición de un elemento específico utilizando el método ‘index()‘.

Ejemplo 1.27: Operaciones adicionales con listas (mas_operaciones_lista.py)

```
"""
Se muestran operaciones adicionales que pueden realizarse con listas:
- Eliminar elementos
- Insertar elementos en una posición específica
- Ordenar elementos
- Encontrar elementos
"""

# Se crea una lista de temperaturas registradas durante el día
temperaturas = [24, 28, 25, 21, 27]
print(f"Temperaturas originales: {temperaturas}")

# Se elimina el último elemento de la lista
temperatura_eliminada = temperaturas.pop()
print(f"Se eliminó la temperatura: {temperatura_eliminada}")
print(f"Temperaturas después de eliminar: {temperaturas}")

# Se inserta un elemento en una posición específica
temperaturas.insert(2, 23)  # Se inserta 23 en la posición 2
print(f"Temperaturas después de insertar 23 en posición 2: {temperaturas}")

# Se ordenan las temperaturas de menor a mayor
temperaturas.sort()
print(f"Temperaturas ordenadas de menor a mayor: {temperaturas}")

# Se busca la posición de una temperatura específica
posicion = temperaturas.index(28)
print(f"La temperatura 28 está en la posición: {posicion}")
```

En el ejemplo anterior se utiliza una lista de temperaturas para ilustrar estas operaciones. El método ‘pop()‘ se utiliza para eliminar la última temperatura registrada, mientras que ‘insert()‘ permite agregar una nueva temperatura en una posición específica de la lista. La ordenación de temperaturas de menor a mayor se realiza con ‘sort()‘, lo cual es útil para analizar los datos de forma organizada. Finalmente, el método ‘index()‘ ayuda a encontrar en qué posición se registró una temperatura específica.

Es importante mencionar que existen otras operaciones útiles con listas como la concatenación de dos listas mediante el operador ‘+‘, la repetición de elementos usando el operador ‘*‘, y la creación de sublistas mediante el corte

(slicing). Todas estas operaciones hacen de las listas una estructura de datos muy versátil para el manejo de colecciones ordenadas de elementos.

Las listas en Python permiten obtener múltiples elementos a la vez mediante una operación llamada corte o slicing. Esta operación utiliza la sintaxis de corchetes con dos puntos '[inicio:fin:paso]' para indicar qué porción de la lista se desea obtener. El valor de inicio indica desde qué posición se empezará a cortar, el valor de fin señala hasta qué posición (sin incluirla) se realizará el corte, y el paso indica cuántas posiciones se saltarán entre cada elemento seleccionado.

Ejemplo 1.28: Operaciones de corte en listas (corte_listas.py)

```
"""
Se ilustra cómo obtener partes de una lista mediante operaciones de corte (slicing).
El corte permite extraer varios elementos de una lista usando la sintaxis [inicio:fin:paso]
"""

# Se crea una lista con los días de la semana
dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado", "Domingo"]
print(f"Lista completa de días: {dias}")

# Se obtienen los primeros tres días
inicio_semana = dias[0:3]
print(f"Primeros tres días: {inicio_semana}")

# Se obtienen los días del medio de la semana
medio_semana = dias[2:5]
print(f"Días del medio de la semana: {medio_semana}")

# Se obtienen los días desde una posición hasta el final
fin_semana = dias[5:]
print(f"Días del fin de semana: {fin_semana}")

# Se obtienen días usando un paso de 2 (días alternados)
dias_alternados = dias[::2]
print(f"Días alternados: {dias_alternados}")

# Se obtienen los días en orden inverso
dias_inverso = dias[::-1]
print(f"Días en orden inverso: {dias_inverso}")
```

En el ejemplo anterior se muestra cómo utilizar diferentes tipos de cortes en una lista de días de la semana. Al utilizar 'dias[0:3]', se obtienen los elementos desde la posición 0 hasta la posición 2 (el 3 no se incluye). Si se omite el valor

de inicio como en ‘dias[5:]‘, Python asume que se desea comenzar desde esa posición hasta el final de la lista. De manera similar, si se omite el valor final como en ‘dias[:3]‘, se toman los elementos desde el inicio hasta la posición indicada.

El parámetro de paso resulta especialmente útil para obtener elementos con un patrón específico. Por ejemplo, ‘dias[::2]‘ selecciona elementos saltando de dos en dos, permitiendo obtener elementos alternados. Un caso particular interesante es cuando se usa un paso negativo como en ‘dias[::-1]‘, lo cual invierte el orden de los elementos en la lista.

Es importante notar que las operaciones de corte crean una nueva lista con los elementos seleccionados, dejando la lista original sin modificaciones. Esta característica es útil cuando se necesita trabajar con una porción de los datos sin alterar la información original.

Tuplas Las tuplas son estructuras de datos similares a las listas, con una diferencia fundamental: una vez creadas, no pueden modificarse. Esta característica las hace ideales para representar colecciones de datos que no deben cambiar durante la ejecución del programa, como las coordenadas de un punto en el plano o la información básica de una persona.

Para crear una tupla se utilizan paréntesis ‘()‘ en lugar de corchetes, y al igual que en las listas, los elementos se separan por comas. La sintaxis para acceder a los elementos es idéntica a la de las listas, utilizando índices que comienzan desde 0.

Ejemplo 1.29: Creación y uso de tuplas (tuplas_basicas.py)

```
"""
Se muestra la creación y uso básico de tuplas.
Una tupla es una estructura que, a diferencia de las listas, no puede modificarse
después de su creación.
"""

# Se crea una tupla con las coordenadas de un punto
coordenadas = (3, 5)
print(f"Coordenadas del punto: {coordenadas}")

# Se accede a los elementos de la tupla
x = coordenadas[0]  # Primera coordenada
y = coordenadas[1]  # Segunda coordenada
print(f"Coordenada x: {x}")
print(f"Coordenada y: {y}")

# Se crea una tupla con información de un estudiante
estudiante = ("Ana", 18, "Preparatoria")
print(f"\nDatos del estudiante: {estudiante}")

```

```
# Se accede a los datos del estudiante
nombre = estudiante[0]
edad = estudiante[1]
nivel = estudiante[2]
print(f"Nombre: {nombre}")
print(f"Edad: {edad}")
print(f"Nivel: {nivel}")
```

En el ejemplo anterior se muestran dos casos de uso comunes para las tuplas. El primero representa las coordenadas de un punto en el plano, donde es natural que los valores no cambien una vez establecidos. El segundo caso muestra cómo una tupla puede almacenar diferentes tipos de datos, en este caso la información básica de un estudiante.

Al intentar modificar cualquier elemento de una tupla, Python generará un error, lo cual es una medida de seguridad que garantiza que los datos permanecerán constantes. Esta característica hace que las tuplas sean especialmente útiles cuando se quiere asegurar que ciertos datos no sean alterados accidentalmente durante la ejecución del programa.

A pesar de su inmutabilidad, las tuplas ofrecen diversas operaciones que permiten obtener información sobre sus elementos y crear nuevas tuplas a partir de las existentes. Dos métodos fundamentales son ‘count()‘, que cuenta cuántas veces aparece un elemento específico, e ‘index()‘, que encuentra la primera posición donde aparece un valor determinado.

Ejemplo 1.30: Operaciones con tuplas (operaciones_tuplas.py)

```
"""
Se demuestran diferentes operaciones que pueden
    realizarse con tuplas:
- Contar elementos
- Encontrar posiciones
- Combinar tuplas
- Crear tuplas con elementos repetidos
"""

# Se crea una tupla de calificaciones
calificaciones = (8, 9, 7, 10, 8, 9, 8)
print(f"Calificaciones registradas:
    {calificaciones}")

# Se cuenta cuántas veces aparece un valor
num_ocho = calificaciones.count(8)
print(f"La calificación 8 aparece {num_ocho} veces")

# Se encuentra la posición de un elemento
posicion_diez = calificaciones.index(10)
print(f"La calificación 10 está en la posición
    {posicion_diez}")
```

```
20
21  # Se combinan dos tuplas
22  primer_parcial = (8, 7, 9)
23  segundo_parcial = (10, 8, 9)
24  todos_parciales = primer_parcial + segundo_parcial
25  print(f"\nTodas las calificaciones:
        {todos_parciales}")
26
27  # Se crea una tupla con elementos repetidos
28  patron = (1, 2) * 3
29  print(f"Patrón repetido: {patron}")
30
31  # Se obtiene la longitud de una tupla
32  total = len(calificaciones)
33  print(f"Total de calificaciones: {total}")
```

La demostración presentada utiliza una tupla de calificaciones para ilustrar las operaciones más comunes. La versatilidad de estas estructuras se evidencia en la facilidad para contar ocurrencias de valores específicos y localizar elementos particulares dentro de la tupla.

Las tuplas también permiten operaciones de concatenación mediante el operador ‘+‘, lo que crea una nueva tupla que combina los elementos de las tuplas originales. De manera similar, el operador ‘*‘ se utiliza para crear una nueva tupla que repite los elementos un número específico de veces. Esta característica resulta útil cuando se necesita generar patrones repetitivos de datos.

La función ‘len()‘ determina la cantidad total de elementos en una tupla, proporcionando una forma sencilla de conocer su tamaño. Todas estas operaciones comparten una característica común: ninguna modifica la tupla original, sino que generan nuevos resultados o tuplas, manteniendo así la naturaleza inmutable de esta estructura de datos.

Python ofrece una característica especial conocida como desempaquetado de tuplas, que permite asignar los elementos de una tupla a variables individuales en una sola operación. Esta funcionalidad simplifica el código y lo hace más legible, especialmente cuando se trabaja con datos que tienen un significado específico para cada posición.

Ejemplo 1.31: Desempaquetado de tuplas (desempaquetado_tuplas.py)

```
"""
Se muestra cómo desempaquetar los elementos de una tupla en variables individuales.
El desempaquetado permite asignar cada elemento de la tupla a una variable diferente
en una sola línea de código.
"""

# Se crea una tupla con información de un producto
producto = ("Laptop", 12500, "Electrónicos")

# Se desempaqueta la tupla en variables individuales
nombre, precio, categoria = producto
print(f"Nombre del producto: {nombre}")
print(f"Precio: ${precio}")
print(f"Categoría: {categoria}")

# Se crea una tupla con coordenadas
punto = (5, 8)

# Se desempaquetan las coordenadas
x, y = punto
print(f"\nCoordenada x: {x}")
print(f"Coordenada y: {y}")

# Se ignora un valor usando guión bajo
datos = ("Juan", 25, "Ciudad de México", "Estudiante")
nombre, edad, _, ocupacion = datos
print(f"\nNombre: {nombre}")
print(f"Edad: {edad}")
print(f"Ocupación: {ocupacion}")
```

La sintaxis de desempaquetado mostrada facilita la extracción de información de las tuplas de una manera clara y directa. Para realizar esta operación, el número de variables debe coincidir exactamente con la cantidad de elementos en la tupla. Si se necesita ignorar algún elemento durante el desempaquetado, se utiliza el guión bajo '_' como variable temporal, indicando que ese valor no será utilizado posteriormente.

Esta técnica de desempaquetado resulta particularmente útil cuando se manejan datos que naturalmente vienen en grupos, como coordenadas, información personal o detalles de productos. El código se vuelve más expresivo al usar nombres de variables significativos en lugar de acceder a los elementos por su índice.

Diccionarios Los diccionarios son estructuras de datos que permiten almacenar información en pares de clave-valor. Se puede pensar en un diccionario como en una agenda telefónica, donde cada nombre (clave) está asociado con un número telefónico (valor). A diferencia de las listas y tuplas, que utilizan índices numéricos, los diccionarios permiten usar cualquier valor inmutable como clave para acceder a los datos.

Ejemplo 1.32: Creación y uso de diccionarios (diccionarios_basicos.py)

```
"""
Se muestra la creación y uso básico de diccionarios.
Un diccionario almacena pares de clave-valor, donde cada valor se accede
mediante su clave única.
"""

# Se crea un diccionario con información de un estudiante
estudiante = {
    "nombre": "Carlos",
    "edad": 17,
    "promedio": 8.5,
    "materias": ["Matemáticas", "Física", "Química"]
}

# Se accede a los valores usando las claves
print(f"Nombre del estudiante: {estudiante['nombre']}")
print(f"Edad: {estudiante['edad']} años")
print(f"Promedio: {estudiante['promedio']}")
print(f"Materias: {estudiante['materias']}")

# Se modifica un valor existente
estudiante["promedio"] = 9.0
print(f"\nNuevo promedio: {estudiante['promedio']}")

# Se agrega un nuevo par clave-valor
estudiante["grupo"] = "A"
print(f"Grupo asignado: {estudiante['grupo']}")

# Se verifica si existe una clave
if "telefono" in estudiante:
    print(f"Teléfono: {estudiante['telefono']}")
else:
    print("\nNo se ha registrado número de teléfono")
```

Los diccionarios se crean utilizando llaves '{ }' y cada par clave-valor se separa por comas. La clave y el valor se vinculan mediante dos puntos ':'. Las claves en un diccionario deben ser únicas, pero los valores pueden repetirse. Esta estructura de datos permite almacenar información heterogénea de manera

organizada, donde cada elemento tiene un identificador significativo.

La flexibilidad de los diccionarios se refleja en la variedad de tipos de datos que pueden almacenar como valores. Como se observa en la demostración, un valor puede ser un número, una cadena de texto o incluso una lista. Esta característica hace que los diccionarios sean ideales para representar estructuras de datos complejas donde cada elemento tiene un significado específico.

Los diccionarios proporcionan métodos y operaciones que facilitan la manipulación de sus elementos. Estas operaciones permiten obtener información específica, modificar el contenido y combinar diferentes diccionarios de manera eficiente.

Ejemplo 1.33: Operaciones avanzadas con diccionarios (operaciones_diccionario.py)

```
"""
Se presentan diversas operaciones que pueden realizarse con diccionarios:
- Obtener claves y valores
- Eliminar elementos
- Obtener valores con un valor predeterminado
- Actualizar y combinar diccionarios
"""

# Se crea un diccionario de calificaciones por materia
calificaciones = {
    "Matemáticas": 9.0,
    "Física": 8.5,
    "Química": 9.5
}
print("Calificaciones iniciales:")
print(calificaciones)

# Se obtienen todas las materias (claves)
materias = calificaciones.keys()
print(f"\nMaterias cursadas: {list(materias)}")

# Se obtienen todas las calificaciones (valores)
notas = calificaciones.values()
print(f"Calificaciones obtenidas: {list(notas)}")

# Se obtiene un valor con get() y un valor predeterminado
nota_biologia = calificaciones.get("Biología", "No cursada")
print(f"\nCalificación de Biología: {nota_biologia}")

# Se elimina una materia
del calificaciones["Química"]
print(f"\nCalificaciones después de eliminar
```

```
        Química:")
33  print(calificaciones)
34
35  # Se agrega un nuevo conjunto de calificaciones
36  nuevas_calificaciones = {
37      "Historia": 8.5,
38      "Literatura": 9.0
39  }
40  calificaciones.update(nuevas_calificaciones)
41  print(f"\nTodas las calificaciones actualizadas:")
42  print(calificaciones)
43
44  # Se obtienen los pares clave-valor
45  for materia, nota in calificaciones.items():
46      print(f"En {materia} se obtuvo {nota}")
```

Los métodos ‘keys()‘ y ‘values()‘ devuelven vistas de las claves y valores del diccionario respectivamente. Una vista se actualiza automáticamente cuando el diccionario cambia, reflejando siempre el estado actual de la estructura. Para trabajar con estas vistas como listas, se pueden convertir usando la función ‘list()‘.

El método ‘get()‘ ofrece una forma segura de obtener valores del diccionario. A diferencia del acceso directo con corchetes, ‘get()‘ permite especificar un valor predeterminado que se devolverá si la clave no existe. Esta característica evita errores cuando se intenta acceder a claves inexistentes.

Para eliminar elementos, se utiliza la palabra clave ‘del‘ seguida del diccionario y la clave entre corchetes. El método ‘update()‘ permite combinar dos diccionarios, agregando o actualizando elementos según las claves proporcionadas.

El método ‘items()‘ resulta particularmente útil en bucles, ya que devuelve los pares clave-valor del diccionario. Esta funcionalidad permite procesar simultáneamente las claves y valores en una sola iteración, haciendo el código más eficiente y legible.

Conjuntos Un conjunto es una estructura de datos que almacena elementos únicos sin un orden específico. Esta característica lo hace ideal para eliminar duplicados de una colección de datos o para verificar la pertenencia de elementos. Los conjuntos en Python implementan el concepto matemático de conjunto, donde cada elemento aparece una sola vez.

Ejemplo 1.34: Creación y uso de conjuntos (conjuntos_basicos.py)

```
"""
Se muestra la creación y uso básico de conjuntos.
Un conjunto es una colección de elementos únicos y
    desordenados.
"""

# Se crea un conjunto con números repetidos
numeros = {1, 2, 3, 2, 1, 4, 5, 4}
print(f"Conjunto de números: {numeros}")

# Se crea un conjunto a partir de una lista
calificaciones = [8, 9, 7, 8, 9, 10, 7, 8]
calificaciones_unicas = set(calificaciones)
print(f"Calificaciones sin repetición:
    {calificaciones_unicas}")

# Se agrega un elemento al conjunto
calificaciones_unicas.add(6)
print(f"Conjunto después de agregar 6:
    {calificaciones_unicas}")

# Se elimina un elemento del conjunto
calificaciones_unicas.remove(10)
print(f"Conjunto después de eliminar 10:
    {calificaciones_unicas}")

# Se verifica si un elemento está en el conjunto
if 8 in calificaciones_unicas:
    print("La calificación 8 está en el conjunto")

# Se intenta agregar un elemento que ya existe
calificaciones_unicas.add(8)
print(f"Conjunto después de intentar agregar 8
    nuevamente: {calificaciones_unicas}")
```

Para crear un conjunto se utilizan llaves ‘‘ con elementos separados por comas, similar a los diccionarios, pero sin pares clave-valor. También se puede crear un conjunto a partir de cualquier estructura iterable utilizando la función ‘set()‘. Una característica fundamental de los conjuntos es que automáticamente eliminan los elementos duplicados durante su creación.

La modificación de conjuntos se realiza mediante métodos específicos. El método ‘add()‘ agrega un nuevo elemento si este no existe, mientras que ‘remove()‘ elimina un elemento específico. Si se intenta agregar un elemento que ya existe en el conjunto, la operación no tiene efecto, manteniendo así la propiedad de unicidad de los elementos.

Los conjuntos en Python implementan las operaciones matemáticas tradicio-

nales de teoría de conjuntos. Estas operaciones permiten combinar y comparar conjuntos de diferentes maneras, facilitando el análisis de datos y la resolución de problemas que involucran grupos de elementos.

Ejemplo 1.35: Operaciones matemáticas con conjuntos (operaciones_conjuntos.py)

```
"""
Se muestran las operaciones matemáticas que pueden realizarse con conjuntos:
- Unión (elementos que están en cualquiera de los conjuntos)
- Intersección (elementos comunes)
- Diferencia (elementos que están en un conjunto pero no en el otro)
- Diferencia simétrica (elementos que están en uno u otro conjunto, pero no en ambos)
"""

# Se crean dos conjuntos de estudiantes que toman diferentes deportes
futbol = {"Juan", "Ana", "Carlos", "María"}
basquetbol = {"María", "Pedro", "Ana", "Miguel"}

print("Estudiantes en fútbol:", futbol)
print("Estudiantes en basquetbol:", basquetbol)

# Unión: estudiantes que practican al menos un deporte
todos_deportistas = futbol | basquetbol
print(f"\nEstudiantes que practican algún deporte: {todos_deportistas}")

# Intersección: estudiantes que practican ambos deportes
ambos_deportes = futbol & basquetbol
print(f"Estudiantes que practican ambos deportes: {ambos_deportes}")

# Diferencia: estudiantes que solo juegan fútbol
solo_futbol = futbol - basquetbol
print(f"Estudiantes que solo juegan fútbol: {solo_futbol}")

# Diferencia simétrica: estudiantes que practican un solo deporte
un_deporte = futbol ^ basquetbol
print(f"Estudiantes que practican un solo deporte: {un_deporte}")

# Verificación de subconjuntos
equipo_pequeno = {"Juan", "Ana"}
```

```
34 if equipo_pequeno <= futbol:
35     print(f"\n{equipo_pequeno} es un subconjunto del
       equipo de fútbol")
```

La operación de unión, realizada con el operador '|', combina dos conjuntos incluyendo todos los elementos de ambos sin duplicados. Esta operación resulta útil cuando se necesita conocer todos los elementos únicos presentes en cualquiera de los conjuntos.

La intersección, representada por el operador '&', encuentra los elementos comunes entre dos conjuntos. Esta operación ayuda a identificar elementos que cumplen simultáneamente con dos condiciones diferentes.

La diferencia entre conjuntos, usando el operador '-', obtiene los elementos que están en el primer conjunto pero no en el segundo. Por su parte, la diferencia simétrica, con el operador '^', encuentra los elementos que están en uno u otro conjunto, pero no en ambos.

Python también permite verificar si un conjunto es subconjunto de otro mediante el operador '¡='. Esta operación confirma si todos los elementos de un conjunto están contenidos dentro de otro conjunto más grande. La capacidad de realizar estas operaciones matemáticas hace que los conjuntos sean especialmente útiles en situaciones donde se necesita analizar y comparar grupos de elementos únicos.

Estructuras de datos combinadas En situaciones reales de programación, frecuentemente se necesita combinar diferentes estructuras de datos para organizar la información de manera eficiente. Cada estructura aporta sus características particulares para resolver diferentes aspectos de un problema.

Ejemplo 1.36: Combinación de estructuras de datos (estructuras_combinadas.py)
Video: Estructuras de Datos *(Introducción)*

```
1  """
2  Se muestra cómo combinar diferentes estructuras de
       datos (listas, tuplas,
3  diccionarios y conjuntos) para organizar información
       de una escuela.
4  """
5
6  # Se crea un diccionario que contiene información de
       estudiantes
7  estudiantes = {
8      "A101": {
9          "nombre": "Ana García",
10         "materias": ["Matemáticas", "Física",
       "Química"],
11         "calificaciones": (9.0, 8.5, 9.5),
```

```
12          "actividades": {"Ajedrez", "Música"}
13      },
14      "A102": {
15          "nombre": "Carlos López",
16          "materias": ["Matemáticas", "Física",
        "Biología"],
17          "calificaciones": (8.5, 9.0, 8.0),
18          "actividades": {"Fútbol", "Música"}
19      }
20  }
21
22  # Se obtiene la información de un estudiante
23  estudiante = estudiantes["A101"]
24  print(f"Información de estudiante:")
25  print(f"Nombre: {estudiante['nombre']}")
26
27  # Se combinan las materias de ambos estudiantes sin
        repetición
28  todas_materias = set()
29  for estudiante_id in estudiantes:
30      materias_estudiante =
        set(estudiantes[estudiante_id]["materias"])
31      todas_materias = todas_materias |
        materias_estudiante
32  print(f"\nTodas las materias impartidas:
        {todas_materias}")
33
34  # Se obtienen las actividades comunes
35  actividades_ana = estudiantes["A101"]["actividades"]
36  actividades_carlos =
        estudiantes["A102"]["actividades"]
37  actividades_comunes = actividades_ana &
        actividades_carlos
38  print(f"\nActividades que comparten:
        {actividades_comunes}")
39
40  # Se calculan promedios usando tuplas de
        calificaciones
41  for estudiante_id, datos in estudiantes.items():
42      calificaciones = datos["calificaciones"]
43      promedio = sum(calificaciones) /
        len(calificaciones)
44      print(f"\nPromedio de {datos['nombre']}:
        {promedio}")
```

La demostración presentada utiliza un diccionario principal para almacenar información de estudiantes, donde cada estudiante se identifica por un código único. Para cada estudiante, se emplea otro diccionario que contiene diferentes tipos de información: una lista para las materias cursadas, una tupla

para las calificaciones (que no cambiarán) y un conjunto para las actividades extracurriculares.

Esta combinación permite aprovechar las ventajas de cada estructura: los diccionarios facilitan el acceso a la información mediante claves significativas, las listas mantienen el orden de las materias, las tuplas protegen las calificaciones de modificaciones accidentales y los conjuntos permiten realizar operaciones como encontrar actividades comunes entre estudiantes.

La capacidad de anidar estructuras de datos de esta manera proporciona una gran flexibilidad para organizar información compleja de forma lógica y accesible. Esta práctica resulta fundamental en el desarrollo de programas que manejan datos con diferentes características y requerimientos de procesamiento.

Video: Estructuras de Datos *(Introducción)*

Herramientas Esenciales

Para un desarrollo efectivo se necesita:

- Un editor de código con resaltado de sintaxis
- Un sistema de control de versiones (como Git)
- Herramientas de depuración integradas
- Acceso a la documentación oficial de Python

1.3 Funciones y Modularidad

Objetivos de Aprendizaje

Al finalizar esta sección, se estará en capacidad de:

- Diseñar funciones reutilizables que encapsulen operaciones específicas
- Implementar diferentes mecanismos de paso de parámetros para crear funciones flexibles
- Gestionar el alcance de variables para mantener la integridad de los datos
- Aplicar técnicas de recursión para resolver problemas complejos
- Crear módulos organizados que faciliten el mantenimiento del código

1.3.1 Mutabilidad de Estructuras de Datos

En Python, la mutabilidad es una característica fundamental de las estructuras de datos que determina si un objeto puede ser modificado después de su creación. Este concepto es esencial para comprender el comportamiento de las diferentes estructuras de datos y su uso apropiado en el desarrollo de aplicaciones.

Concepto de Mutabilidad La mutabilidad se refiere a la capacidad de un objeto para cambiar su estado o contenido después de ser creado. Un objeto mutable puede ser modificado sin necesidad de crear una nueva instancia, mientras que un objeto inmutable no puede cambiar una vez creado.

Ejemplo 1.37: Demostración de mutabilidad básica (mutabilidad_basica.py)

```
"""
Se demuestra el concepto básico de mutabilidad e inmutabilidad en Python
"""

def demostrar_mutabilidad():
    """
    Se ilustra la diferencia entre objetos mutables e inmutables
    mediante ejemplos prácticos
    """
    # Ejemplo con objeto inmutable (string)
    print("--- Demostración con String (Inmutable) ---")
    texto = "Python"
    id_original = id(texto)
    print(f"Texto original: {texto}")
    print(f"ID original: {id_original}")

    texto = texto + " Programming"
    print(f"Texto modificado: {texto}")
    print(f"Nuevo ID: {id(texto)}")
    print(f"¿Mismo objeto?: {id_original == id(texto)}\n")

    # Ejemplo con objeto mutable (lista)
    print("--- Demostración con Lista (Mutable) ---")
    numeros = [1, 2, 3]
    id_lista = id(numeros)
    print(f"Lista original: {numeros}")
    print(f"ID original: {id_lista}")

    numeros.append(4)
    print(f"Lista modificada: {numeros}")
```

```
31      print(f"Nuevo ID: {id(numeros)}")
32      print(f"¿Mismo objeto?: {id_lista ==
        id(numeros)}")
33
34  def demostrar_referencias():
35      """
36      Se muestra cómo las referencias afectan a
        objetos mutables e inmutables
37      """
38      # Referencias con inmutables
39      print("\n--- Referencias con Inmutables ---")
40      x = 5
41      y = x
42      x = 10
43      print(f"x = {x}, y = {y}")  # y mantiene el
        valor original
44
45      # Referencias con mutables
46      print("\n--- Referencias con Mutables ---")
47      lista_1 = [1, 2, 3]
48      lista_2 = lista_1
49      lista_1.append(4)
50      print(f"lista_1 = {lista_1}")
51      print(f"lista_2 = {lista_2}")  # lista_2 refleja
        los cambios
52
53  if __name__ == "__main__":
54      demostrar_mutabilidad()
55      demostrar_referencias()
```

Identificación de Mutabilidad

Para determinar si un objeto es mutable, se puede considerar:

- Si permite modificaciones in-place
- Si mantiene el mismo id() después de modificaciones
- Si las operaciones de modificación retornan None
- Si existe documentación que especifique su mutabilidad

Estructuras Mutables vs Inmutables Python proporciona varios tipos de estructuras de datos con diferentes características de mutabilidad:

Ejemplo 1.38: Comparación de estructuras mutables e inmutables (comparacion_mutabilidad.py)

```
"""
Se comparan diferentes estructuras de datos y su comportamiento respecto a la mutabilidad
"""

def comparar_estructuras():
    """
    Se analizan las diferencias entre estructuras mutables e inmutables
    mediante ejemplos prácticos
    """
    # Estructuras Inmutables
    print("=== Estructuras Inmutables ===")

    # Strings
    print("\n--- Strings ---")
    texto = "Hola"
    print(f"Original: {texto}, ID: {id(texto)}")
    texto += " Mundo"
    print(f"Modificado: {texto}, ID: {id(texto)}")

    # Tuplas
    print("\n--- Tuplas ---")
    tupla = (1, 2, 3)
    print(f"Original: {tupla}, ID: {id(tupla)}")
    try:
        tupla[0] = 5  # Esto generará un error
    except TypeError as e:
        print(f"Error al modificar tupla: {e}")

    # Números
    print("\n--- Números ---")
    numero = 42
    print(f"Original: {numero}, ID: {id(numero)}")
    numero += 1
    print(f"Modificado: {numero}, ID: {id(numero)}")

    # Estructuras Mutables
    print("\n=== Estructuras Mutables ===")

    # Listas
    print("\n--- Listas ---")
    lista = [1, 2, 3]
    print(f"Original: {lista}, ID: {id(lista)}")
    lista.append(4)
    print(f"Modificada: {lista}, ID: {id(lista)}")

```

```
46      # Diccionarios
47      print("\n--- Diccionarios ---")
48      diccionario = {'a': 1, 'b': 2}
49      print(f"Original: {diccionario}, ID:
        {id(diccionario)}")
50      diccionario['c'] = 3
51      print(f"Modificado: {diccionario}, ID:
        {id(diccionario)}")
52
53      # Sets
54      print("\n--- Sets ---")
55      conjunto = {1, 2, 3}
56      print(f"Original: {conjunto}, ID:
        {id(conjunto)}")
57      conjunto.add(4)
58      print(f"Modificado: {conjunto}, ID:
        {id(conjunto)}")
59
60  def demostrar_anidamiento():
61      """
62      Se muestra el comportamiento de estructuras
        anidadas
63      """
64      print("\n=== Estructuras Anidadas ===")
65
66      # Tupla con lista anidada
67      tupla_lista = (1, [2, 3], 4)
68      print(f"Tupla con lista: {tupla_lista}")
69      tupla_lista[1].append(5)  # La lista dentro de
        la tupla es mutable
70      print(f"Después de modificar lista interna:
        {tupla_lista}")
71
72  if __name__ == "__main__":
73      comparar_estructuras()
74      demostrar_anidamiento()
```

⚠ Consideraciones de Mutabilidad

Se debe tener especial cuidado con:

- Parámetros mutables en funciones
- Referencias compartidas a objetos mutables
- Modificaciones no intencionales de estructuras
- Efectos secundarios en operaciones con mutables

Casos Prácticos de Mutabilidad Las estructuras mutables más comunes en Python son las listas y los diccionarios. Estas estructuras permiten

modificaciones directas que afectan al objeto original.

Ejemplo 1.39: Operaciones con estructuras mutables (operaciones_mutables.py)

```
"""
Se demuestran operaciones comunes con estructuras de datos mutables
"""

def operaciones_listas():
    """
    Se ilustran operaciones típicas con listas y sus efectos
    """
    print("=== Operaciones con Listas ===")

    # Creación y modificación básica
    numeros = [1, 2, 3, 4, 5]
    print(f"Lista original: {numeros}")

    # Modificación por índice
    numeros[0] = 10
    print(f"Después de modificar primer elemento: {numeros}")

    # Agregar elementos
    numeros.append(6)
    print(f"Después de append: {numeros}")

    numeros.extend([7, 8])
    print(f"Después de extend: {numeros}")

    numeros.insert(2, 15)
    print(f"Después de insert en posición 2: {numeros}")

    # Eliminar elementos
    eliminado = numeros.pop()
    print(f"Elemento eliminado con pop: {eliminado}")
    print(f"Lista después de pop: {numeros}")

    numeros.remove(15)
    print(f"Después de remove(15): {numeros}")

    # Operaciones que crean nueva lista
    sublista = numeros[2:5]
    print(f"Sublista (slice): {sublista}")

def operaciones_diccionarios():
    """
    Se muestran operaciones comunes con diccionarios
```

```
    """
    print("\n=== Operaciones con Diccionarios ===")

    # Creación y modificación
    estudiante = {
        'nombre': 'Ana',
        'edad': 20,
        'cursos': ['Python', 'IA']
    }
    print(f"Diccionario original: {estudiante}")

    # Modificar valores
    estudiante['edad'] = 21
    print(f"Después de modificar edad: {estudiante}")

    # Agregar nuevas entradas
    estudiante['promedio'] = 9.5
    print(f"Después de agregar promedio:
    {estudiante}")

    # Modificar lista dentro del diccionario
    estudiante['cursos'].append('Machine Learning')
    print(f"Después de modificar lista de cursos:
    {estudiante}")

    # Eliminar entradas
    eliminado = estudiante.pop('promedio')
    print(f"Valor eliminado: {eliminado}")
    print(f"Después de pop: {estudiante}")

    # Operaciones con get y setdefault
    telefono = estudiante.get('telefono', 'No
    disponible')
    print(f"Teléfono (con get): {telefono}")

    email = estudiante.setdefault('email',
    'sin@email.com')
    print(f"Email (con setdefault): {email}")
    print(f"Diccionario final: {estudiante}")

def efectos_secundarios():
    """
    Se demuestran efectos secundarios de la
    mutabilidad
    """
    print("\n=== Efectos Secundarios de Mutabilidad
    ===")

    # Listas como valores de diccionario
    registro = {
```

```
88          'usuario1': [1, 2, 3],
89          'usuario2': [1, 2, 3]
90      }
91      print(f"Registro original: {registro}")
92
93      # Modificar lista para un usuario
94      registro['usuario1'].append(4)
95      print(f"Después de modificar usuario1:
        {registro}")
96
97      # Compartir referencias
98      usuarios = ['Ana', 'Juan']
99      grupo1 = {'miembros': usuarios}
100     grupo2 = {'miembros': usuarios}
101
102     print(f"\nGrupo 1 original: {grupo1}")
103     print(f"Grupo 2 original: {grupo2}")
104
105     usuarios.append('Carlos')
106     print(f"Después de modificar usuarios:")
107     print(f"Grupo 1: {grupo1}")
108     print(f"Grupo 2: {grupo2}")
109
110 if __name__ == "__main__":
111     operaciones_listas()
112     operaciones_diccionarios()
113     efectos_secundarios()
```

Inmutabilidad y sus Ventajas Las estructuras inmutables, como tuplas y cadenas, ofrecen garantías importantes sobre la integridad de los datos.

Ejemplo 1.40: Ejemplos de inmutabilidad (inmutabilidad_ejemplos.py)

```
1  """
2  Se ilustran conceptos y usos de estructuras
       inmutables en Python
3  """
4
5  def ejemplos_strings():
6      """
7      Se demuestra la inmutabilidad de strings y sus
       operaciones
8      """
9      print("=== Ejemplos con Strings ===")
10
11     # Operaciones básicas con strings
12     nombre = "Python"
13     print(f"String original: {nombre}")
14     print(f"ID original: {id(nombre)}")
```

```
15
16      # Intentar modificar el string
17      try:
18          nombre[0] = "p"
19      except TypeError as e:
20          print(f"Error al intentar modificar: {e}")
21
22      # Crear nuevo string
23      nombre_modificado = nombre.lower()
24      print(f"Nuevo string: {nombre_modificado}")
25      print(f"ID nuevo string:
        {id(nombre_modificado)}")
26
27      # Concatenación
28      nombre_completo = nombre + " Programming"
29      print(f"String concatenado: {nombre_completo}")
30      print(f"ID concatenación: {id(nombre_completo)}")
31
32  def ejemplos_tuplas():
33      """
34      Se exploran las características de las tuplas
35      """
36      print("\n=== Ejemplos con Tuplas ===")
37
38      # Creación de tuplas
39      coordenadas = (10, 20)
40      print(f"Tupla original: {coordenadas}")
41
42      # Intentar modificar tupla
43      try:
44          coordenadas[0] = 15
45      except TypeError as e:
46          print(f"Error al modificar tupla: {e}")
47
48      # Tupla con elementos mutables
49      tupla_mixta = (1, [2, 3], (4, 5))
50      print(f"\nTupla con elementos mixtos:
        {tupla_mixta}")
51
52      # Modificar lista dentro de tupla
53      tupla_mixta[1].append(6)
54      print(f"Después de modificar lista interna:
        {tupla_mixta}")
55
56      # Crear nueva tupla
57      nueva_tupla = tupla_mixta + (7, 8)
58      print(f"Nueva tupla concatenada: {nueva_tupla}")
59
60  def ejemplos_frozenset():
61      """
```

```
    Se demuestra el uso de frozenset como conjunto
    inmutable
    """
    print("\n=== Ejemplos con Frozenset ===")

    # Crear frozenset
    numeros = frozenset([1, 2, 3, 4, 5])
    print(f"Frozenset original: {numeros}")

    # Intentar modificar frozenset
    try:
        numeros.add(6)
    except AttributeError as e:
        print(f"Error al intentar modificar
    frozenset: {e}")

    # Operaciones válidas con frozenset
    otro_set = frozenset([4, 5, 6])

    # Unión (crea nuevo frozenset)
    union = numeros.union(otro_set)
    print(f"Unión de frozensets: {union}")

    # Intersección (crea nuevo frozenset)
    interseccion = numeros.intersection(otro_set)
    print(f"Intersección de frozensets:
    {interseccion}")

def rendimiento_inmutables():
    """
    Se demuestra el impacto en rendimiento de
    estructuras inmutables
    """
    print("\n=== Rendimiento con Inmutables ===")

    # Ejemplo con strings
    from time import time

    inicio = time()
    resultado = ""
    for i in range(1000):
        resultado += str(i)
    tiempo_string = time() - inicio
    print(f"Tiempo con concatenación de strings:
    {tiempo_string:.4f} segundos")

    # Ejemplo con lista (más eficiente)
    inicio = time()
    partes = []
    for i in range(1000):
```

```
107             partes.append(str(i))
108         resultado = "".join(partes)
109         tiempo_lista = time() - inicio
110         print(f"Tiempo con lista y join:
            {tiempo_lista:.4f} segundos")
111
112         print(f"Diferencia de rendimiento:
            {tiempo_string/tiempo_lista:.2f}x más lento con
            concatenación")
113
114 if __name__ == "__main__":
115     ejemplos_strings()
116     ejemplos_tuplas()
117     ejemplos_frozenset()
118     rendimiento_inmutables()
```

Ventajas de la Inmutabilidad

Las estructuras inmutables proporcionan:

- Mayor seguridad en el manejo de datos
- Garantía de consistencia en estructuras de datos
- Mejor rendimiento en ciertas operaciones
- Facilidad para el razonamiento sobre el código

Mejores Prácticas Para un manejo efectivo de la mutabilidad en Python:

Recomendaciones de Uso

- Utilizar estructuras inmutables para datos que no deben cambiar
- Crear copias explícitas cuando se necesite preservar el estado original
- Documentar claramente las expectativas de mutabilidad
- Considerar el uso de frozenset para conjuntos inmutables

Ejemplo 1.41: Mejores prácticas (mejores_practicas_mutabilidad.py)

```python
"""
Se implementan las mejores prácticas para el manejo de mutabilidad en Python
"""

from copy import deepcopy
from typing import List, Dict, Tuple, Any

class GestorDatos:
    """
    Se implementa una clase que demuestra las mejores prácticas
    para el manejo de datos mutables e inmutables
    """

    def __init__(self, datos_iniciales: List[Any]):
        """
        Se inicializa el gestor con una copia profunda de los datos
        para evitar modificaciones externas
        """
        self._datos = deepcopy(datos_iniciales)
        self._backup = deepcopy(datos_iniciales)

    def obtener_datos(self) -> List[Any]:
        """
        Se retorna una copia de los datos para prevenir
        modificaciones accidentales
        """
        return deepcopy(self._datos)

    def actualizar_elemento(self, indice: int, nuevo_valor: Any) -> bool:
        """
        Se actualiza un elemento de forma segura, manteniendo
        un respaldo de los datos
        """
        try:
            self._backup = deepcopy(self._datos)
            self._datos[indice] = deepcopy(nuevo_valor)
            return True
        except IndexError:
            print(f"Error: índice {indice} fuera de rango")
            return False
```

```
41
42      def restaurar_backup(self) -> None:
43          """
44          Se restauran los datos al último estado
        válido
45          """
46          self._datos = deepcopy(self._backup)
47
48  def manejo_parametros_defecto(items: List[Any] =
        None) -> List[Any]:
49      """
50      Se demuestra el manejo correcto de parámetros
        mutables por defecto
51      """
52      # Inicialización segura de parámetro mutable
53      if items is None:
54          items = []
55
56      # Se trabaja con una copia para evitar modificar
        el original
57      items_trabajo = items.copy()
58      items_trabajo.append('nuevo_item')
59      return items_trabajo
60
61  def procesar_diccionario(diccionario: Dict) ->
        Tuple[Dict, Dict]:
62      """
63      Se demuestra el manejo seguro de diccionarios
64      """
65      # Se crea una copia para modificaciones
66      dict_trabajo = diccionario.copy()
67
68      # Se mantiene un diccionario de cambios
69      cambios = {}
70
71      for clave in dict_trabajo:
72          if isinstance(dict_trabajo[clave], (list,
        dict)):
73              # Copia profunda para estructuras
        anidadas
74              cambios[clave] =
        deepcopy(dict_trabajo[clave])
75          else:
76              # Copia simple para tipos inmutables
77              cambios[clave] = dict_trabajo[clave]
78
79      return dict_trabajo, cambios
80
81  def demostrar_manejo_tuplas():
82      """
```

```
83      Se demuestra el uso correcto de tuplas para
        datos inmutables
84      """
85      # Uso de tuplas para datos que no deben cambiar
86      configuracion = ('desarrollo', 8080, True)
87      print(f"\n=== Configuración (Tupla) ===")
88      print(f"Modo: {configuracion[0]}")
89      print(f"Puerto: {configuracion[1]}")
90      print(f"Debug: {configuracion[2]}")
91
92      # Tuplas nombradas para mejor legibilidad
93      from collections import namedtuple
94      Config = namedtuple('Config', ['modo', 'puerto',
        'debug'])
95      config_nombrada = Config('desarrollo', 8080,
        True)
96      print(f"\n=== Configuración (NamedTuple) ===")
97      print(f"Modo: {config_nombrada.modo}")
98      print(f"Puerto: {config_nombrada.puerto}")
99      print(f"Debug: {config_nombrada.debug}")
100
101 def ejemplo_uso_seguro():
102     """
103     Se demuestra el uso seguro de estructuras
        mutables e inmutables
104     """
105     print("=== Demostración de Mejores Prácticas
        ===")
106
107     # Uso de GestorDatos
108     datos_originales = [1, [2, 3], {'a': 4}]
109     gestor = GestorDatos(datos_originales)
110
111     print(f"Datos originales: {datos_originales}")
112     datos_copia = gestor.obtener_datos()
113     datos_copia[1].append(5)  # No afecta los datos
        originales
114
115     print(f"Datos después de modificar copia:
        {gestor.obtener_datos()}")
116
117     # Actualización segura
118     gestor.actualizar_elemento(1, [6, 7])
119     print(f"Datos después de actualización:
        {gestor.obtener_datos()}")
120
121     # Manejo de parámetros por defecto
122     lista1 = manejo_parametros_defecto()
123     lista2 = manejo_parametros_defecto()
124     print(f"\nListas independientes:")
```

```
125     print(f"Lista 1: {lista1}")
126     print(f"Lista 2: {lista2}")
127
128     # Procesamiento seguro de diccionarios
129     dict_original = {'lista': [1, 2], 'dict': {'a':
        3}}
130     dict_procesado, cambios =
        procesar_diccionario(dict_original)
131     dict_procesado['lista'].append(4)
132
133     print(f"\nDiccionario original: {dict_original}")
134     print(f"Diccionario procesado: {dict_procesado}")
135     print(f"Registro de cambios: {cambios}")
136
137     # Demostración con tuplas
138     demostrar_manejo_tuplas()
139
140 def ejemplo_frozen_structures():
141     """
142     Se demuestra el uso de estructuras inmutables
        para datos críticos
143     """
144     print("\n=== Estructuras Inmutables para Datos
        Críticos ===")
145
146     # Uso de frozenset para conjuntos inmutables
147     permisos = frozenset(['leer', 'escribir',
        'ejecutar'])
148     print(f"Permisos (inmutables): {permisos}")
149
150     try:
151         permisos.add('administrar')
152     except AttributeError as e:
153         print(f"Intento de modificar frozenset: {e}")
154
155     # Uso de tipos inmutables en diccionarios
156     config_segura = {
157         'permisos': permisos,
158         'versión': '1.0',
159         'max_usuarios': 100
160     }
161     print(f"\nConfiguración segura: {config_segura}")
162
163 if __name__ == "__main__":
164     ejemplo_uso_seguro()
165     ejemplo_frozen_structures()
```

1.3.2 Definición y uso de funciones

En programación, una función se define como un bloque de código reutilizable que realiza una tarea específica. Las funciones permiten organizar el código en unidades lógicas y modulares, facilitando su mantenimiento y comprensión. Cuando se desarrollan soluciones basadas en inteligencia artificial, las funciones constituyen la base para estructurar el código de manera eficiente y crear componentes que puedan ser utilizados en diferentes partes del programa.

Concepto de Funciones Una función puede verse como una caja negra que recibe datos de entrada (aunque no siempre son necesarios), realiza un procesamiento específico y puede devolver un resultado. En el contexto de la inteligencia artificial, las funciones pueden encapsular desde simples cálculos hasta complejos algoritmos de procesamiento de datos o entrenamiento de modelos.

Sintaxis de una Función En Python, una función se define utilizando la palabra clave `def`, seguida del nombre de la función y paréntesis que pueden contener los parámetros. El cuerpo de la función se escribe con sangría y puede incluir una cadena de documentación (docstring) que describe su propósito.

Ejemplo 1.42: Estructura básica de una función

```
def nombre_funcion(parametro1, parametro2):
    """
    Se documenta el propósito de la función
    """
    # Se procesan los parámetros
    resultado = parametro1 + parametro2
    return resultado
```

Función Simple Se presenta a continuación un ejemplo práctico de una función que implementa un saludo personalizado, demostrando la estructura básica y el uso de parámetros.

Ejemplo 1.43: Función de saludo

```
def saludar(nombre):
"""
Se genera un saludo personalizado
"""
mensaje = f"Bienvenido al curso de IA, {nombre}"
return mensaje
```

Función con Procesamiento En el contexto de la inteligencia artificial, las funciones suelen realizar cálculos más complejos. El siguiente ejemplo muestra una función que calcula áreas de diferentes figuras geométricas, ilustrando cómo una función puede manejar diferentes casos y realizar cálculos específicos.

Ejemplo 1.44: Cálculo de área de figuras (calculo_area.py)

```
"""
Se implementa una función que calcula el área de diferentes figuras geométricas
"""

def calcular_area(figura, dimensiones):
    """
    Se calcula el área de una figura geométrica específica

    Parámetros:
    figura (str): Tipo de figura ('cuadrado', 'rectangulo', 'triangulo')
    dimensiones (list): Lista con las dimensiones necesarias para el cálculo

    Retorna:
    float: Área calculada de la figura
    """
    if figura == 'cuadrado':
        lado = dimensiones[0]
        area = lado * lado
        return area

    elif figura == 'rectangulo':
        base = dimensiones[0]
        altura = dimensiones[1]
        area = base * altura
        return area

    elif figura == 'triangulo':
        base = dimensiones[0]
```

```
29          altura = dimensiones[1]
30          area = (base * altura) / 2
31          return area
32
33      else:
34          return "Figura no reconocida"
35
36  # Se realizan ejemplos de uso
37  print("Se calculan diferentes áreas:")
38
39  # Se calcula el área de un cuadrado
40  lado_cuadrado = 5
41  area_cuadrado = calcular_area('cuadrado',
        [lado_cuadrado])
42  print(f"Área del cuadrado con lado {lado_cuadrado}:
        {area_cuadrado}")
43
44  # Se calcula el área de un rectángulo
45  dimensiones_rectangulo = [4, 6]
46  area_rectangulo = calcular_area('rectangulo',
        dimensiones_rectangulo)
47  print(f"Área del rectángulo con base
        {dimensiones_rectangulo[0]} y altura
        {dimensiones_rectangulo[1]}: {area_rectangulo}")
48
49  # Se calcula el área de un triángulo
50  dimensiones_triangulo = [3, 8]
51  area_triangulo = calcular_area('triangulo',
        dimensiones_triangulo)
52  print(f"Área del triángulo con base
        {dimensiones_triangulo[0]} y altura
        {dimensiones_triangulo[1]}: {area_triangulo}")
```

1.3.3 Parámetros y retorno de valores

En el desarrollo de sistemas de inteligencia artificial, el manejo eficiente de parámetros y valores de retorno en las funciones resulta fundamental para crear código modular y reutilizable. Los parámetros permiten que las funciones sean flexibles y adaptables a diferentes situaciones, mientras que los valores de retorno facilitan la comunicación entre diferentes componentes del sistema.

Conceptos Fundamentales de Parámetros Los parámetros son variables que se definen en la declaración de una función y actúan como contenedores para los datos que la función necesita procesar. Por otro lado, los argumentos son los valores reales que se pasan a la función cuando esta se

invoca. Esta distinción se vuelve especialmente relevante en el contexto del procesamiento de datos para modelos de IA.

Ejemplo 1.45: Distinción entre parámetros y argumentos

```
def procesar_datos(datos, factor):
    """
    Se procesan los datos aplicando un factor de escala
    """
    return [valor * factor for valor in datos]

# datos y 2.0 son argumentos que se asignan a los
parámetros
resultado = procesar_datos([1, 2, 3], 2.0)
```

Parámetros Posicionales Los parámetros posicionales se asignan según el orden en que se definen en la función. Su uso se recomienda cuando existe una secuencia lógica clara en los argumentos que se proporcionan.

Ejemplo 1.46: Función con parámetros posicionales

```
def crear_vector(x, y, z):
    """
    Se crea un vector tridimensional a partir de sus
    componentes
    """
    return [x, y, z]

vector = crear_vector(1, 2, 3)  # Se crea el vector
[1, 2, 3]
```

Parámetros por Defecto Los parámetros por defecto permiten establecer valores predeterminados que se utilizan cuando no se proporciona un argumento específico. Esta característica se emplea frecuentemente en la configuración de hiperparámetros de modelos de IA.

Ejemplo 1.47: Función con parámetros por defecto

```
def configurar_modelo(tasa_aprendizaje=0.01,
epochs=100):
    """
    Se configuran los parámetros básicos de un modelo de
    aprendizaje
    """
    configuracion = {
        'learning_rate': tasa_aprendizaje,
        'epochs': epochs
    }
    return configuracion
```

Argumentos con Nombre Los argumentos nombrados mejoran la legibilidad del código y permiten especificar valores en cualquier orden. Esta práctica se recomienda especialmente cuando se trabaja con funciones que tienen múltiples parámetros opcionales.

Ejemplo 1.48: Uso de argumentos nombrados

```
def entrenar_modelo(datos, epochs=100, batch_size=32,
verbose=True):
    """
    Se configura el entrenamiento de un modelo
    """
    configuracion = {
        'datos': datos,
        'epochs': epochs,
        'batch_size': batch_size,
        'verbose': verbose
    }
    return configuracion

    # Se utilizan argumentos nombrados para mayor claridad
    config = entrenar_modelo(
    datos=[1, 2, 3],
    verbose=False,
    epochs=50
    )
```

Número Variable de Parámetros En ocasiones, se necesita que una función acepte un número variable de argumentos. Python proporciona dos mecanismos especiales para esto: *args para argumentos posicionales y **kwargs para argumentos nombrados.

Ejemplo 1.49: Función con argumentos variables (parametros_variables.py)

```
1  # parametros_variables.py
2  """
3  Se implementa una función que acepta un número
       variable de argumentos
4  """
5
6  def combinar_caracteristicas(*args, **kwargs):
7      """
8      Se combinan múltiples características con sus
       respectivos pesos
9
10     args: valores de las características
11     kwargs: pesos asociados a cada característica
12     """
13     # Se procesan argumentos posicionales
14     valores = list(args)
```

```
15
16      # Se obtienen los pesos (por defecto 1.0)
17      pesos = []
18      for i in range(len(valores)):
19          peso_key = f'peso_{i}'
20          pesos.append(kwargs.get(peso_key, 1.0))
21
22      # Se calcula la combinación ponderada
23      resultado = sum(v * p for v, p in zip(valores,
        pesos))
24      return resultado
25
26  # Se realizan ejemplos de uso
27  print("Se combinan características con diferentes
        configuraciones:")
28
29  # Uso básico sin pesos
30  resultado1 = combinar_caracteristicas(1, 2, 3)
31  print(f"Combinación simple: {resultado1}")
32
33  # Uso con pesos personalizados
34  resultado2 = combinar_caracteristicas(1, 2, 3,
        peso_0=0.5, peso_1=1.5, peso_2=1.0)
35  print(f"Combinación ponderada: {resultado2}")
```

Sistema de Retorno El retorno de valores en Python se gestiona mediante la sentencia return, que permite a una función devolver resultados al código que la invocó. Si no se especifica un valor de retorno, la función devuelve None implícitamente.

Ejemplo 1.50: Retorno simple de valores

```
    def normalizar_dato(valor, minimo, maximo):
    """
    Se normaliza un valor en el rango [0,1]
    """
    if maximo == minimo:
    return 0
    return (valor - minimo) / (maximo - minimo)
```

Retornos Múltiples Python permite que una función retorne múltiples valores simultáneamente mediante tuplas, facilitando la devolución de varios resultados de forma organizada.

Ejemplo 1.51: Procesamiento con múltiples retornos (procesamiento_datos.py)

```python
# procesamiento_datos.py
"""
Se implementa una función que realiza múltiples operaciones de procesamiento
"""

def procesar_datos(datos):
    """
    Se realiza el procesamiento básico de una lista de datos numéricos

    Retorna:
    - Media de los datos
    - Valor mínimo
    - Valor máximo
    - Lista de datos normalizados
    """
    if not datos:
        return 0, 0, 0, []

    # Se calculan estadísticas básicas
    minimo = min(datos)
    maximo = max(datos)
    media = sum(datos) / len(datos)

    # Se normalizan los datos
    if maximo > minimo:
        normalizados = [(x - minimo) / (maximo - minimo) for x in datos]
    else:
        normalizados = [0] * len(datos)

    return media, minimo, maximo, normalizados

# Se realiza un ejemplo de uso
datos_prueba = [1, 3, 5, 7, 9]
media, min_val, max_val, norm = procesar_datos(datos_prueba)

print("Resultados del procesamiento:")
print(f"Media: {media}")
print(f"Mínimo: {min_val}")
print(f"Máximo: {max_val}")
print(f"Datos normalizados: {norm}")
```

Integración de Clases Cuando se desarrollan programas orientados a objetos en Python, las clases pueden residir en el mismo archivo del programa

principal o en archivos separados. El método de integración varía según la ubicación de la clase.

Clases en el Mismo Archivo Cuando una clase se define en el mismo archivo que el programa principal, su integración resulta directa ya que la clase está inmediatamente disponible en el espacio de nombres del script.

Ejemplo 1.52: Integración de clase en el mismo archivo (clase_mismo_archivo.py)

```
"""
Se demuestra la definición y uso de una clase en el mismo archivo del programa principal
"""

class Calculadora:
    """
    Se implementa una calculadora simple para demostrar el uso de clases
    """
    def __init__(self, valor_inicial=0):
        """
        Se inicializa la calculadora con un valor opcional
        """
        self.resultado = valor_inicial

    def sumar(self, valor):
        """
        Se suma un valor al resultado actual
        """
        self.resultado += valor
        return self.resultado

    def restar(self, valor):
        """
        Se resta un valor al resultado actual
        """
        self.resultado -= valor
        return self.resultado

    def obtener_resultado(self):
        """
        Se retorna el resultado actual
        """
        return self.resultado

def main():
    """
    Programa principal que demuestra el uso de la clase Calculadora
```

```
38      """
39      print("=== Demostración de Calculadora ===")
40
41      # Se crea una instancia de la calculadora
42      calc = Calculadora(10)
43      print(f"Valor inicial:
        {calc.obtener_resultado()}")
44
45      # Se realizan algunas operaciones
46      print(f"Después de sumar 5: {calc.sumar(5)}")
47      print(f"Después de restar 3: {calc.restar(3)}")
48
49      # Se crea otra instancia con valor por defecto
50      calc2 = Calculadora()
51      print(f"\nNueva calculadora con valor inicial:
        {calc2.obtener_resultado()}")
52      print(f"Después de sumar 7: {calc2.sumar(7)}")
53
54  if __name__ == "__main__":
55      main()
```

En este ejemplo, la clase `Calculadora` se define en el mismo archivo que el código principal. No se requiere ninguna importación especial, y la clase puede utilizarse directamente después de su definición.

Clases en Archivos Separados Cuando una clase se define en un archivo separado, se debe importar antes de poder utilizarla. Este enfoque favorece la modularidad y la reutilización del código.

Se presenta primero el archivo que contiene la definición de la clase:

Ejemplo 1.53: Definición de clase en archivo separado (calculadora.py)

```
1   """
2   Se define la clase Calculadora en un archivo
        separado para demostrar la modularización
3   """
4
5   class Calculadora:
6       """
7       Se implementa una calculadora simple que puede
        ser importada desde otros archivos
8       """
9       def __init__(self, valor_inicial=0):
10          """
11          Se inicializa la calculadora con un valor
        opcional
12          """
13          self.resultado = valor_inicial
14
```

```
15      def sumar(self, valor):
16          """
17          Se suma un valor al resultado actual
18          """
19          self.resultado += valor
20          return self.resultado
21
22      def restar(self, valor):
23          """
24          Se resta un valor al resultado actual
25          """
26          self.resultado -= valor
27          return self.resultado
28
29      def multiplicar(self, valor):
30          """
31          Se multiplica el resultado actual por un
        valor
32          """
33          self.resultado *= valor
34          return self.resultado
35
36      def dividir(self, valor):
37          """
38          Se divide el resultado actual entre un valor
39          """
40          if valor == 0:
41              raise ValueError("No se puede dividir
        entre cero")
42          self.resultado /= valor
43          return self.resultado
44
45      def obtener_resultado(self):
46          """
47          Se retorna el resultado actual
48          """
49          return self.resultado
```

Y luego el programa principal que utiliza esta clase:

Ejemplo 1.54: Uso de clase desde archivo separado (programa_principal.py)

```
"""
Se demuestra cómo importar y utilizar una clase definida en otro archivo
"""

# Se importa la clase Calculadora del módulo calculadora
from calculadora import Calculadora

def demostrar_operaciones_basicas():
    """
    Se realizan operaciones básicas con la calculadora
    """
    print("=== Operaciones Básicas ===")
    calc = Calculadora(100)

    print(f"Valor inicial: {calc.obtener_resultado()}")
    print(f"Después de sumar 50: {calc.sumar(50)}")
    print(f"Después de restar 25: {calc.restar(25)}")
    print(f"Después de multiplicar por 2: {calc.multiplicar(2)}")
    print(f"Después de dividir entre 5: {calc.dividir(5)}")

def demostrar_manejo_errores():
    """
    Se demuestra el manejo de errores con la calculadora
    """
    print("\n=== Manejo de Errores ===")
    calc = Calculadora()

    try:
        print("Intentando dividir entre cero...")
        calc.dividir(0)
    except ValueError as e:
        print(f"Error capturado: {e}")

def demostrar_multiples_instancias():
    """
    Se demuestra el uso de múltiples instancias de la calculadora
    """
    print("\n=== Múltiples Instancias ===")

    # Primera calculadora para operaciones con
```

```
        enteros
41      calc1 = Calculadora(10)
42      calc1.multiplicar(5)
43
44      # Segunda calculadora para operaciones con
        decimales
45      calc2 = Calculadora(3.14)
46      calc2.multiplicar(2)
47
48      print(f"Calculadora 1 resultado:
        {calc1.obtener_resultado()}")
49      print(f"Calculadora 2 resultado:
        {calc2.obtener_resultado()}")
50
51  def main():
52      """
53      Función principal que ejecuta todas las
        demostraciones
54      """
55      print("Demostración de uso de la clase
        Calculadora importada\n")
56
57      # Se ejecutan las diferentes demostraciones
58      demostrar_operaciones_basicas()
59      demostrar_manejo_errores()
60      demostrar_multiples_instancias()
61
62  if __name__ == "__main__":
63      main()
```

En este caso, se utiliza la sentencia `import` para hacer la clase disponible en el programa principal. Python busca el archivo en el directorio actual y en la ruta de búsqueda de módulos del sistema.

Convenciones de Importación

Al importar clases desde otros archivos:

- Colocar las importaciones al inicio del archivo
- Utilizar nombres de archivo en minúsculas
- Evitar caracteres especiales en nombres de archivo
- Mantener una estructura clara de directorios

Caso Práctico: Preprocesamiento de Datos Se presenta un ejemplo integrador que demuestra el uso combinado de diferentes tipos de parámetros y retornos en el contexto del preprocesamiento de datos para IA.

Ejemplo 1.55: Función de preprocesamiento para IA (preprocesamiento_ia.py)

```python
# preprocesamiento_ia.py
"""
Se implementa una función de preprocesamiento para datos de IA
"""

def preprocesar_datos(datos, normalizar=True, eliminar_nulos=True,
                      valor_nulo=-999, **opciones_extra):
    """
    Se realiza el preprocesamiento de datos para un modelo de IA

    Parámetros:
    datos: Lista de valores numéricos
    normalizar: Indica si se deben normalizar los datos
    eliminar_nulos: Indica si se deben eliminar valores nulos
    valor_nulo: Valor que se considera como nulo
    opciones_extra: Opciones adicionales de procesamiento

    Retorna:
    - Datos procesados
    - Estadísticas del procesamiento
    - Configuración utilizada
    """
    # Se prepara el registro de estadísticas
    estadisticas = {
        'total_inicial': len(datos),
        'nulos_encontrados': 0,
        'rango_original': 0
    }

    # Se eliminan valores nulos si se especifica
    datos_procesados = datos.copy()
    if eliminar_nulos:
        datos_procesados = [x for x in datos if x != valor_nulo]
        estadisticas['nulos_encontrados'] = len(datos) - len(datos_procesados)

    # Se realiza la normalización si se requiere
    if normalizar and datos_procesados:
        minimo = min(datos_procesados)
        maximo = max(datos_procesados)
```

```
40          estadisticas['rango_original'] = maximo -
        minimo
41
42          if maximo > minimo:
43              datos_procesados = [(x - minimo) /
        (maximo - minimo)
44                                  for x in
        datos_procesados]
45
46      # Se registra la configuración utilizada
47      configuracion = {
48          'normalizar': normalizar,
49          'eliminar_nulos': eliminar_nulos,
50          'valor_nulo': valor_nulo,
51          'opciones_extra': opciones_extra
52      }
53
54      return datos_procesados, estadisticas,
        configuracion
55
56  # Se realiza un ejemplo de uso
57  datos_ejemplo = [1, -999, 4, 7, -999, 10]
58  datos_proc, stats, config = preprocesar_datos(
59      datos_ejemplo,
60      normalizar=True,
61      eliminar_nulos=True,
62      valor_nulo=-999,
63      relleno_decimales=2
64  )
65
66  print("Resultados del preprocesamiento:")
67  print(f"Datos procesados: {datos_proc}")
68  print(f"Estadísticas: {stats}")
69  print(f"Configuración: {config}")
```

1.3.4 Alcance de variables y recursión

El alcance de una variable determina en qué partes del programa se puede acceder a dicha variable. La comprensión del alcance de las variables y el concepto de recursión son fundamentales para escribir código más organizado y resolver problemas complejos de manera elegante.

Alcance de Variables El alcance (scope) de una variable se refiere a la región del código donde la variable puede ser utilizada. Las variables locales son aquellas que se definen dentro de una función y solo pueden ser accedidas desde dentro de la misma función.

Ejemplo 1.56: Variables locales

```
    def calcular_promedio(numeros):
    """
    Se calcula el promedio de una lista de números
    """
    suma = 0  # Variable local
    for numero in numeros:
    suma += numero
    promedio = suma / len(numeros)  # Variable local
    return promedio
```

Variables Globales Las variables globales se definen fuera de cualquier función y pueden ser accedidas desde cualquier parte del programa. Para modificar una variable global dentro de una función, se debe utilizar la palabra clave `global`.

Ejemplo 1.57: Uso de variables globales

```
    contador_total = 0  # Variable global

    def incrementar_contador():
    """
    Se incrementa el contador global
    """
    global contador_total
    contador_total += 1
    return contador_total
```

Encapsulamiento El encapsulamiento ayuda a organizar el código y evitar conflictos entre variables. Las variables no locales permiten acceder a variables de un ámbito superior sin ser globales.

Ejemplo 1.58: Ejemplo de encapsulamiento (encapsulamiento.py)

```
1  # encapsulamiento.py
2  """
3  Se implementa un ejemplo de encapsulamiento usando
       variables no locales
4  """
5
6  def crear_contador():
7      """
8      Se crea un contador con funciones internas que
       mantienen su propio estado
9      """
10     cuenta = 0  # Variable en el ámbito de
       crear_contador
```

```
11
12      def incrementar():
13          """
14          Se incrementa el contador
15          """
16          nonlocal cuenta  # Se indica que cuenta no
        es local
17          cuenta += 1
18          return cuenta
19
20      def decrementar():
21          """
22          Se decrementa el contador
23          """
24          nonlocal cuenta
25          cuenta -= 1
26          return cuenta
27
28      def obtener_valor():
29          """
30          Se obtiene el valor actual
31          """
32          return cuenta
33
34      return incrementar, decrementar, obtener_valor
35
36  # Se demuestra el uso del contador
37  incrementar, decrementar, obtener = crear_contador()
38
39  print("Se realizan operaciones con el contador:")
40  print(f"Valor inicial: {obtener()}")
41  print(f"Después de incrementar: {incrementar()}")
42  print(f"Después de incrementar: {incrementar()}")
43  print(f"Después de decrementar: {decrementar()}")
44  print(f"Valor final: {obtener()}")
```

Concepto de Recursión La recursión es una técnica de programación donde una función se llama a sí misma para resolver un problema dividiéndolo en casos más simples. Toda función recursiva debe tener al menos dos elementos: un caso base que detiene la recursión y un caso recursivo que reduce el problema.

Ejemplo 1.59: Función recursiva simple

```
    def contar_regresivo(n):
    """
    Se realiza una cuenta regresiva recursiva
    """
    if n <= 0:  # Caso base
    return "¡Despegue!"
    print(n)
    return contar_regresivo(n - 1)  # Caso recursivo
```

Aplicaciones Prácticas Las funciones recursivas son especialmente útiles para resolver problemas que pueden descomponerse en subproblemas similares al original.

Ejemplo 1.60: Cálculos mediante recursión (calculo_recursivo.py)

```
1  # calculo_recursivo.py
2  """
3  Se implementan funciones recursivas para cálculos
       matemáticos básicos
4  """
5
6  def factorial(n):
7      """
8      Se calcula el factorial de un número de forma
       recursiva
9      """
10     if n <= 1:  # Caso base
11         return 1
12     return n * factorial(n - 1)  # Caso recursivo
13
14 def fibonacci(n):
15     """
16     Se calcula el n-ésimo número de Fibonacci de
       forma recursiva
17     """
18     if n <= 1:  # Casos base
19         return n
20     return fibonacci(n - 1) + fibonacci(n - 2)  #
       Caso recursivo
21
22 # Se realizan ejemplos de uso
23 print("Se calculan valores usando recursión:")
24
25 # Se calculan factoriales
26 for i in range(5):
27     print(f"Factorial de {i}: {factorial(i)}")
28
29 print("\nSe calculan números de Fibonacci:")
```

```
for i in range(7):
    print(f"Fibonacci {i}: {fibonacci(i)}")
```

Optimización Las técnicas de optimización permiten mejorar el rendimiento de las funciones recursivas, evitando cálculos redundantes y optimizando el uso de memoria.

Ejemplo 1.61: Técnicas de optimización recursiva (optimizacion_recursion.py)

```
# optimizacion_recursion.py
"""
Se implementan técnicas de optimización para funciones recursivas
"""

def fibonacci_memo(n, memo=None):
    """
    Se calcula Fibonacci usando memoización para evitar cálculos repetidos
    """
    if memo is None:
        memo = {}

    if n in memo:  # Se verifica si el valor ya fue calculado
        return memo[n]

    if n <= 1:  # Casos base
        return n

    # Se calcula y almacena el resultado
    memo[n] = fibonacci_memo(n - 1, memo) + fibonacci_memo(n - 2, memo)
    return memo[n]

def factorial_cola(n, acumulado=1):
    """
    Se calcula factorial usando recursión de cola
    """
    if n <= 1:  # Caso base
        return acumulado
    return factorial_cola(n - 1, n * acumulado)  # Llamada de cola

# Se comparan los resultados
print("Se calculan números de Fibonacci con memoización:")
for i in range(7):
    print(f"Fibonacci {i}: {fibonacci_memo(i)}")
```

```
35
36  print("\nSe calculan factoriales con recursión de
        cola:")
37  for i in range(5):
38      print(f"Factorial de {i}: {factorial_cola(i)}")
```

1.4 Manejo de Archivos y Excepciones

Objetivos de Aprendizaje

Al finalizar esta sección, se estará en capacidad de:

- Implementar operaciones de lectura y escritura de archivos en diferentes formatos
- Desarrollar sistemas robustos de manejo de errores utilizando excepciones
- Procesar archivos CSV para el análisis de datos estructurados
- Crear sistemas de registro (logging) para el seguimiento de la ejecución del programa
- Aplicar técnicas de depuración para identificar y resolver errores en el manejo de archivos

1.4.1 Lectura y escritura de archivos

En la programación, los archivos son recursos fundamentales que permiten almacenar información de manera permanente en el disco duro. Cuando un programa termina su ejecución, los datos en memoria se pierden, por lo que resulta necesario guardar la información en archivos para su uso posterior.

Apertura y cierre de archivos Para trabajar con archivos en Python, se utiliza la función `open()`. Esta función requiere dos parámetros principales:

- El nombre del archivo (incluyendo la ruta si no está en el directorio actual)
- El modo de apertura que especifica las operaciones permitidas

Es fundamental cerrar los archivos después de su uso mediante el método `close()` para liberar los recursos del sistema y asegurar que los cambios se guarden correctamente.

Ejemplo 1.62: Apertura y cierre básico de un archivo

```
archivo = open('datos.txt', 'r')
contenido = archivo.read()
archivo.close()
```

Modos de apertura Los modos de apertura determinan las operaciones que se pueden realizar en el archivo:

- `'r'`: Modo lectura. El archivo debe existir previamente. Es el modo predeterminado.
- `'w'`: Modo escritura. Crea un nuevo archivo o sobrescribe si ya existe.
- `'a'`: Modo append. Añade contenido al final del archivo existente.
- `'r+'`: Modo lectura y escritura. Permite ambas operaciones.

Adicionalmente, se puede añadir el sufijo `'b'` a cualquier modo para trabajar con archivos binarios (por ejemplo, `'rb'` para lectura binaria).

Ejemplo 1.63: Demostración de modos de apertura (modos_archivo.py)

```
"""
Se demuestra el uso de diferentes modos de apertura de archivos
"""

def demostrar_modos_archivo():
    """
    Se realizan operaciones de lectura y escritura con diferentes modos
    """
    # Modo escritura ('w') - Crea un nuevo archivo o sobrescribe si existe
    with open('ejemplo.txt', 'w') as archivo:
        archivo.write('Se escribe la primera línea\n')
        archivo.write('Se escribe la segunda línea\n')

    # Modo append ('a') - Añade contenido al final del archivo
    with open('ejemplo.txt', 'a') as archivo:
        archivo.write('Se añade una tercera línea\n')

    # Modo lectura ('r') - Lee el contenido del archivo
    with open('ejemplo.txt', 'r') as archivo:
        print("Se lee el contenido del archivo:")
        print(archivo.read())

```

```
    # Modo lectura y escritura ('r+') - Permite ambas operaciones
    with open('ejemplo.txt', 'r+') as archivo:
        contenido = archivo.read()
        archivo.seek(0)  # Se regresa al inicio del archivo
        archivo.write('Se modifica la primera línea\n')

    archivo.write(contenido[contenido.find('\n')+1:])

# Se ejecuta la demostración
if __name__ == "__main__":
    demostrar_modos_archivo()
    print("Se ha completado la demostración de modos de archivo")
```

Métodos de lectura Python proporciona varios métodos para leer el contenido de los archivos:

- `read([size])`: Lee el archivo completo o hasta el número de caracteres especificado.
- `readline()`: Lee una línea del archivo.
- `readlines()`: Lee todas las líneas y las devuelve en una lista.

El método `read()` sin argumentos lee todo el contenido del archivo como una única cadena. Si se especifica un tamaño, lee solo esa cantidad de caracteres. El método `readline()` es útil cuando se necesita procesar el archivo línea por línea, mientras que `readlines()` es conveniente cuando se necesitan todas las líneas en memoria.

Ejemplo 1.64: Métodos para leer archivos (metodos_lectura.py)

```
"""
Se demuestran los diferentes métodos para leer archivos
"""

def demostrar_metodos_lectura():
    """
    Se ejemplifican los distintos métodos de lectura de archivos
    """
    # Se crea un archivo de ejemplo
    with open('lectura.txt', 'w') as archivo:
        archivo.write('Primera línea de texto\n')
        archivo.write('Segunda línea de texto\n')
        archivo.write('Tercera línea de texto\n')
```

```
14
15      # Método read() - Lee todo el archivo
16      print("Se lee el archivo completo con read():")
17      with open('lectura.txt', 'r') as archivo:
18          contenido = archivo.read()
19          print(contenido)
20
21      # Método readline() - Lee una línea a la vez
22      print("\nSe lee línea por línea con readline():")
23      with open('lectura.txt', 'r') as archivo:
24          linea1 = archivo.readline()
25          linea2 = archivo.readline()
26          print(f"Primera línea: {linea1.strip()}")
27          print(f"Segunda línea: {linea2.strip()}")
28
29      # Método readlines() - Lee todas las líneas en
        una lista
30      print("\nSe leen todas las líneas con
        readlines():")
31      with open('lectura.txt', 'r') as archivo:
32          lineas = archivo.readlines()
33          for i, linea in enumerate(lineas, 1):
34              print(f"Línea {i}: {linea.strip()}")
35
36  # Se ejecuta la demostración
37  if __name__ == "__main__":
38      demostrar_metodos_lectura()
39      print("\nSe ha completado la demostración de
        métodos de lectura")
```

Métodos de escritura Para escribir en archivos, Python ofrece dos métodos principales:

- `write(str)`: Escribe una cadena en el archivo.
- `writelines(list)`: Escribe una lista de cadenas en el archivo.

Es importante notar que `write()` no añade automáticamente saltos de línea, por lo que estos deben incluirse explícitamente cuando sean necesarios usando el carácter `'
n'`. El método `writelines()` tampoco añade saltos de línea entre los elementos de la lista.

Ejemplo 1.65: Métodos para escribir en archivos (metodos_escritura.py)

```
"""
Se demuestran los diferentes métodos para escribir en archivos
"""

def demostrar_metodos_escritura():
    """
    Se ejemplifican los distintos métodos de escritura en archivos
    """
    # Método write() - Escribe una cadena
    with open('escritura.txt', 'w') as archivo:
        archivo.write('Se escribe una línea de texto\n')
        archivo.write('Se escribe otra línea de texto\n')

    # Método writelines() - Escribe una lista de cadenas
    lineas = [
        'Se escribe la primera línea con writelines()\n',
        'Se escribe la segunda línea con writelines()\n',
        'Se escribe la tercera línea con writelines()\n'
    ]

    with open('escritura_multiple.txt', 'w') as archivo:
        archivo.writelines(lineas)

    # Se verifica el contenido escrito
    print("Se verifica el contenido del primer archivo:")
    with open('escritura.txt', 'r') as archivo:
        print(archivo.read())

    print("\nSe verifica el contenido del segundo archivo:")
    with open('escritura_multiple.txt', 'r') as archivo:
        print(archivo.read())

# Se ejecuta la demostración
if __name__ == "__main__":
    demostrar_metodos_escritura()
    print("\nSe ha completado la demostración de
```

```
        métodos de escritura")
```

Context Manager (with) El context manager `with` proporciona una sintaxis más segura para trabajar con archivos. Este mecanismo garantiza que el archivo se cierre correctamente, incluso si ocurre un error durante su manipulación. La estructura básica es:

Ejemplo 1.66: Uso del context manager with

```
    with open('datos.txt', 'r') as archivo:
    contenido = archivo.read()
    El archivo se cierra automáticamente al salir del
   bloque with
```

Las ventajas de usar `with` incluyen:

- Cierre automático del archivo al finalizar el bloque
- Manejo implícito de excepciones
- Código más limpio y seguro

Manejo Seguro de Archivos

Prácticas recomendadas:

- Utilizar siempre el administrador de contexto (with) para archivos
- Verificar los permisos de lectura/escritura antes de operar
- Implementar cierres explícitos de archivos en casos excepcionales
- Considerar el uso de rutas absolutas para mayor robustez

1.4.2 Manejo de errores y excepciones

Durante la ejecución de un programa pueden ocurrir diversos errores que interrumpan su funcionamiento normal. Python proporciona un mecanismo llamado .ºcepciones"para manejar estas situaciones de manera controlada.

Introducción a las excepciones Una excepción es un evento que ocurre durante la ejecución de un programa e interrumpe el flujo normal de las instrucciones. Algunos ejemplos comunes incluyen:

- `ValueError`: Ocurre cuando una operación recibe un argumento con el tipo correcto pero valor inapropiado
- `TypeError`: Se produce al realizar una operación con un tipo de dato incorrecto

- `FileNotFoundError`: Se genera cuando se intenta acceder a un archivo que no existe
- `ZeroDivisionError`: Ocurre al intentar dividir por cero

Estructura try-except El manejo básico de excepciones se realiza mediante la estructura try-except:

Ejemplo 1.67: Manejo básico de excepciones (manejo_basico_excepciones.py)

```
"""
Se demuestra el manejo básico de excepciones en Python
"""

def dividir_numeros(a, b):
    """
    Se intenta realizar una división y se maneja el posible error
    """
    try:
        resultado = a / b
        print(f"El resultado de la división es: {resultado}")
    except ZeroDivisionError:
        print("Se ha intentado dividir por cero")

    print("La ejecución del programa continúa normalmente")

# Se realizan pruebas con diferentes valores
print("Se intenta dividir 10 entre 2:")
dividir_numeros(10, 2)

print("\nSe intenta dividir 10 entre 0:")
dividir_numeros(10, 0)
```

Excepciones múltiples En ocasiones, un mismo bloque de código puede generar diferentes tipos de excepciones. Python permite manejar múltiples excepciones de forma específica:

Ejemplo 1.68: Manejo de múltiples excepciones (excepciones_multiples.py)

```
"""
Se demuestra el manejo de múltiples tipos de excepciones
"""

def procesar_datos(lista, indice, divisor):
    """
    Se intenta acceder a un elemento de una lista y dividirlo
    """
    try:
        elemento = lista[indice]
        resultado = elemento / divisor
        print(f"El resultado del procesamiento es: {resultado}")
    except IndexError:
        print("Se ha intentado acceder a una posición no válida de la lista")
    except ZeroDivisionError:
        print("Se ha intentado dividir por cero")
    except TypeError:
        print("Se ha intentado realizar una operación con tipos de datos incompatibles")

# Se realizan pruebas con diferentes casos
numeros = [1, 2, 3, 4, 5]

print("Caso 1: Acceso a índice válido")
procesar_datos(numeros, 2, 2)

print("\nCaso 2: Acceso a índice inválido")
procesar_datos(numeros, 10, 2)

print("\nCaso 3: División por cero")
procesar_datos(numeros, 0, 0)

print("\nCaso 4: Tipos incompatibles")
procesar_datos(['a', 'b', 'c'], 0, 2)
```

Bloque finally El bloque finally se ejecuta siempre, independientemente de si ocurrió una excepción o no. Se utiliza comúnmente para tareas de limpieza:

Ejemplo 1.69: Uso del bloque finally (bloque_finally.py)

```
"""
Se demuestra el uso del bloque finally en el manejo de excepciones
"""

def procesar_archivo(nombre_archivo):
    """
    Se intenta abrir y procesar un archivo, asegurando su cierre
    """
    archivo = None
    try:
        archivo = open(nombre_archivo, 'r')
        contenido = archivo.read()
        print(f"Se ha leído el contenido del archivo: {contenido}")
    except FileNotFoundError:
        print("No se ha encontrado el archivo especificado")
    finally:
        if archivo:
            archivo.close()
            print("Se ha cerrado el archivo correctamente")
        print("Este mensaje se muestra siempre, haya o no errores")

# Se realizan pruebas con archivos existentes y no existentes
print("Se intenta abrir un archivo que no existe:")
procesar_archivo('archivo_inexistente.txt')

print("\nSe crea y procesa un archivo temporal:")
with open('temporal.txt', 'w') as f:
    f.write('Contenido de prueba')
procesar_archivo('temporal.txt')
```

Gestión de Excepciones

Un manejo efectivo de excepciones busca:

- Identificar y capturar tipos específicos de errores
- Proporcionar mensajes de error informativos
- Mantener la integridad de los datos ante fallos
- Implementar estrategias de recuperación apropiadas

Creación de excepciones personalizadas Python permite crear excepciones propias heredando de la clase Exception. Esto es útil cuando se necesitan manejar errores específicos de la aplicación:

Ejemplo 1.70: Excepciones personalizadas (excepciones_personalizadas.py)

```
"""
Se demuestra la creación y uso de excepciones personalizadas
"""

class ValorNegativoError(Exception):
    """
    Se define una excepción personalizada para valores negativos
    """
    def __init__(self, valor):
        self.valor = valor
        self.mensaje = f"No se permiten valores negativos: {valor}"
        super().__init__(self.mensaje)

def calcular_raiz_cuadrada(numero):
    """
    Se calcula la raíz cuadrada verificando que el número sea positivo
    """
    try:
        if numero < 0:
            raise ValorNegativoError(numero)
        resultado = numero ** 0.5
        print(f"La raíz cuadrada de {numero} es: {resultado}")
    except ValorNegativoError as e:
        print(f"Error: {e.mensaje}")

# Se realizan pruebas con diferentes valores
print("Se calcula la raíz cuadrada de un número positivo:")
calcular_raiz_cuadrada(16)

print("\nSe intenta calcular la raíz cuadrada de un número negativo:")
calcular_raiz_cuadrada(-9)
```

Excepciones Personalizadas

Consideraciones importantes:

- Crear jerarquías lógicas de excepciones
- Proporcionar información contextual útil
- Evitar capturar Exception genérica sin un propósito específico
- Documentar las condiciones que lanzan cada excepción

Backup y Recuperación

Prácticas esenciales:

- Implementar copias de seguridad antes de modificaciones
- Verificar la integridad de los archivos después de operaciones
- Mantener un registro de cambios realizados
- Establecer estrategias de recuperación ante fallos

Operaciones con Archivos

Se debe tener precaución con:

- No cerrar los archivos puede causar pérdida de datos
- Los errores de codificación al leer archivos de texto
- El manejo de rutas en diferentes sistemas operativos
- La modificación de archivos que están siendo utilizados por otros procesos

Conceptos Clave: Manejo de Archivos y Excepciones

- **Operaciones con Archivos:**
  - Uso del administrador de contexto (with) para garantizar el cierre de archivos
  - Diferentes modos de apertura: lectura ('r'), escritura ('w'), anexar ('a')
  - Métodos principales: read(), write(), readline(), readlines()
  - Manejo de archivos binarios y de texto
- **Manejo de Excepciones:**
  - Estructura try-except para capturar y manejar errores
  - Uso de múltiples bloques except para diferentes tipos de excepciones
  - Bloque finally para código que debe ejecutarse siempre
  - Creación de excepciones personalizadas para casos específicos
- **Procesamiento de CSV:**

- Uso del módulo csv para lectura y escritura estructurada
- Manejo de diferentes formatos de delimitadores
- Procesamiento de encabezados y tipos de datos
- Validación de estructura y contenido

- **Sistema de Logging:**
  - Configuración de registros con diferentes niveles de severidad
  - Formato personalizado de mensajes de log
  - Rotación y gestión de archivos de registro
  - Seguimiento de errores y eventos del programa
- **Depuración:**
  - Técnicas de depuración con puntos de interrupción
  - Manejo de errores en tiempo de ejecución
  - Validación de datos y estados del programa
  - Estrategias de recuperación ante fallos

1.5 Ejercicios

1.5.1 Ejercicios de la sección 1.3 Funciones y Modularidad.

Ejercicio 1: Función Simple Implementar una función `convTemperatura` que reciba un valor de temperatura en grados Celsius y retorne su equivalente en grados Fahrenheit. Se debe utilizar la fórmula de conversión:

$$F = \left(C \times \frac{9}{5}\right) + 32$$

Ejercicio 2: Parámetros por Defecto Desarrollar una función llamada `crearListaNumeros` que genere una lista de números con tres parámetros: `inicio` (por defecto 0), `fin` (por defecto 10) y `paso` (por defecto 1). La función debe devolver una lista con los valores generados en ese rango.

Ejercicio 3: Alcance de Variables Crear dos funciones:

- `inicializarContador`, que establezca una variable global.

- `incrementarContador`, que modifique dicha variable.

Se debe demostrar el uso correcto de la palabra clave `global` para modificar variables globales dentro de una función.

Ejercicio 4: Encapsulamiento de Datos Escribir una función `crearBanco` que genere tres funciones internas: - `depositar`, que agregue dinero a una cuenta. - `retirar`, que reste dinero si hay saldo suficiente. - `consultarSaldo`, que devuelva el saldo actual. El saldo debe estar encapsulado dentro de la función principal para que no pueda ser modificado directamente desde fuera.

Ejercicio 5: Recursión Simple Definir una función recursiva `sumarLista` que reciba una lista de números y retorne la suma de todos sus elementos. No se permite usar la función `sum()`.

Ejercicio 6: Argumentos Variables Crear una función `calcEstadisticas` que acepte un número variable de valores numéricos y devuelva tres resultados: el promedio, el valor mínimo y el valor máximo.

Ejercicio 7: Recursión con Memoria Implementar la función llamada `caminosPosibles` que calcule cuántas formas hay de moverse desde la posición $(0, 0)$ hasta (n, m) en una cuadrícula, desplazándose solo hacia la derecha o hacia abajo. Se debe utilizar **memoización** para optimizar el cálculo.

Ejercicio 8: Sistema de Transformación Diseñar un sistema de funciones que permita transformar datos mediante una cadena de procesamiento.

Ejercicio 9: Gestor de Archivos Seguro Crear una clase `GestorArchivos` que implemente el protocolo de **context manager** para garantizar el manejo seguro de archivos.

1.5.2 Ejercicios de la sección 1.4 Manejo de Archivos y Excepciones.

Ejercicio 10: Contador de Líneas Implementar una función que lea un archivo de texto y retorne el número total de líneas, palabras y caracteres. La función debe manejar apropiadamente las excepciones en caso de que el archivo no exista.

Ejercicio 11: Registro de Logs Crear una función que escriba mensajes de log en un archivo, incluyendo la fecha y hora de cada registro. Debe implementar un manejo de excepciones para casos donde el archivo no tenga permisos de escritura.

Ejercicio 12: Fusión de Archivos Desarrollar una función que combine el contenido de dos archivos de texto en un tercero, alternando las líneas de cada archivo. Manejar adecuadamente los casos donde los archivos tengan diferente número de líneas.

Ejercicio 13: Buscador de Palabras Implementar una función que busque una palabra específica en todos los archivos .txt de un directorio y sus subdirectorios, retornando la lista de archivos y las líneas donde aparece la palabra.

Ejercicio 14: Respaldo Automático Crear una función que realice una copia de seguridad de un archivo, añadiendo la fecha actual al nombre del archivo de respaldo. Debe manejar excepciones para casos de archivo corrupto o sin espacio en disco.

Ejercicio 15: Validador CSV Desarrollar una función que valide la estructura de un archivo CSV, verificando que todas las filas tengan el mismo número de columnas y que los tipos de datos sean consistentes. Debe lanzar excepciones personalizadas para cada tipo de error encontrado.

Ejercicio 16: Monitor de Archivos Implementar una clase que monitoree cambios en un archivo específico y registre en un log cualquier modificación detectada. Debe manejar excepciones relacionadas con permisos y acceso al sistema de archivos.

Ejercicio 17: Convertidor de Formatos Crear una función que convierta un archivo de texto a diferentes formatos (HTML, JSON, XML) manejando apropiadamente los caracteres especiales y posibles errores de codificación.

Ejercicio 18: Archivador Seguro Implementar una clase que permita comprimir y descomprimir archivos de manera segura, verificando la integridad del archivo resultante. Debe incluir manejo de excepciones para errores de compresión y corrupción de archivos.

Ejercicio 19: Importador de Datos Desarrollar una función que lea datos de diferentes tipos de archivos (CSV, JSON, XML) y los unifique en una estructura de datos común, manejando las diferentes excepciones que puedan surgir en cada formato.

Ejercicio 20: Sistema de Logs con Respaldo Crear un script que utilice la clase creada en el ejercicio 11 para gestionar logs y la clase del ejercicio 14 para implementar un sistema de logs que automáticamente genere respaldos cuando el archivo de log alcance cierto tamaño.

Ejercicio 21: Conversor de Datos Multi-formato Desarrollar un script que utilice la clase del ejercicio 19 y la clase del ejercicio 17 para leer datos de múltiples archivos (CSV, JSON, XML) y exportarlos todos a un formato común seleccionado por el usuario.

Ejercicio 22: Monitor de Directorio Seguro Implementar un script que use la clase del ejercicio 16 y la clase del ejercicio 18 para vigilar un directorio y automáticamente comprimir de forma segura cualquier archivo nuevo que aparezca.

Ejercicio 23: Validador de Datos con Logs Crear un script que utilice la clase del ejercicio 15 y la clase del ejercicio 11 para validar múltiples archivos CSV y generar un log detallado de los errores encontrados.

Ejercicio 24: Sistema de Respaldo Simple Crear un script que utilice la clase del ejercicio 14 y la clase del ejercicio 11 para implementar un sistema simple que: 1. Respalde un archivo específico 2. Registre en un log cada vez que se hace un respaldo 3. Incluya una función para verificar si el archivo ha cambiado comparando su tamaño actual

1.5.3 Respuestas de los ejercicios de la sección 1.3 Funciones y Modularidad.

Respuesta 1: (convertir_temperatura.py)

Para convertir una temperatura de grados Celsius a Fahrenheit, se utiliza la siguiente fórmula:

$$F = C \times \frac{9}{5} + 32$$

La función implementada en Python realiza esta conversión de manera sencilla:

```
1  def convertir_temperatura(celsius):
2          return (celsius * 9/5) + 32
```

Respuesta 2: (crear_lista_numeros.py)

En este ejercicio se define una función para generar una lista de números. Se utilizan parámetros por defecto para permitir que el usuario personalice el intervalo de la lista:

```
def crear_lista_numeros(inicio=0, fin=10, paso=1):
return list(range(inicio, fin, paso))
```

Esta función devuelve una lista de números desde 'inicio' hasta 'fin' (sin incluirlo), con incrementos de 'paso'.

Respuesta 3: (inicializar_contador.py)

Este ejercicio ilustra cómo manipular variables globales en Python. La función 'inicializar_contador' establece una variable global, mientras que 'incrementar_contador' la modifica correctamente usando la palabra clave 'global'.

```
contador = 0

def inicializar_contador():
global contador
contador = 0

def incrementar_contador():
global contador
contador += 1
```

Respuesta 4: (crear_banco.py)

En este ejercicio se implementa un sistema de gestión de cuentas bancarias mediante el uso de funciones anidadas. La variable 'saldo' se encapsula dentro de la función 'crear_banco', evitando modificaciones externas y permitiendo un control adecuado mediante 'depositar', 'retirar' y 'consultar_saldo'.

```
    def crear_banco():
    saldo = 0

    def depositar(cantidad):
    nonlocal saldo
    saldo += cantidad

    def retirar(cantidad):
    nonlocal saldo
    if cantidad <= saldo:
    saldo -= cantidad

    def consultar_saldo():
    return saldo

    return depositar, retirar, consultar_saldo
```

Respuesta 5: (sumar_lista.py)

Para calcular la suma de todos los números de una lista sin utilizar la función 'sum()', se puede usar una función recursiva. La implementación recorre la lista sumando el primer elemento y llamando a la función nuevamente con el resto de la lista:

```
    def sumar_lista(lista):
    if not lista:
    return 0
    return lista[0] + sumar_lista(lista[1:])
```

Respuesta 6: (calcular_estadisticas.py)

Para calcular estadísticas a partir de una cantidad variable de argumentos, la función debe recibir '*args' y calcular el promedio, el mínimo y el máximo:

```
    def calcular_estadisticas(*numeros):
    if not numeros:
    return None, None, None
    return sum(numeros) / len(numeros), min(numeros),
   max(numeros)
```

Respuesta 7: (caminos_posibles.py)

Para calcular cuántas formas diferentes hay de llegar desde la posición (0,0) hasta (n,m) en una cuadrícula, moviéndose solo hacia la derecha o hacia abajo, se utiliza un enfoque recursivo optimizado con memoización. La función 'caminos_posibles' almacena resultados previos para evitar cálculos redundantes:

```
    from functools import lru_cache

    @lru_cache(None)
```

```
    def caminos_posibles(n, m):
    # Si llegamos a una fila o columna vacía, hay solo un
    camino posible
    if n == 0 or m == 0:
    return 1
    # Se suman los caminos posibles desde la izquierda y
    desde arriba
    return caminos_posibles(n-1, m) + caminos_posibles(n,
    m-1)
```

Respuesta 8: (sistema_transformacion.py)

En este ejercicio se crea un sistema de transformación de datos, permitiendo aplicar una serie de funciones en una cadena de procesamiento.

```
1  def transformarDato(dato, *funciones):
2      """
3      Aplica una serie de funciones al dato de manera
       secuencial.
4
5      :param dato: El dato inicial a transformar.
6      :param funciones: Una secuencia de funciones a
       aplicar en orden.
7      :return: El resultado de aplicar todas las
       funciones en cadena.
8      """
9      for funcion in funciones:
10         dato = funcion(dato)
11     return dato
12
13
14 # Funciones de ejemplo para transformación
15 def convertirAMayusculas(texto):
16     """Convierte una cadena a mayúsculas."""
17     return texto.upper()
18
19 def eliminarEspacios(texto):
20     """Elimina los espacios de una cadena."""
21     return texto.replace(" ", "")
22
23 def invertirTexto(texto):
24     """Invierte el orden de los caracteres en la
       cadena."""
25     return texto[::-1]
26
27 def agregarPrefijo(texto, prefijo="PRE-"):
28     """Agrega un prefijo a la cadena."""
29     return prefijo + texto
30
31
32 if __name__ == "__main__":
```

```
33      # Ejemplo de uso del sistema de transformación
34      texto_original = "Ejemplo de Texto"
35
36      resultado = transformarDato(
37          texto_original,
38          convertirAMayusculas,
39          eliminarEspacios,
40          invertirTexto
41      )
42
43      print("Texto original:", texto_original)
44      print("Texto transformado:", resultado)
```

Respuesta 9: (gestor_archivos.py)

Se implementa un gestor de archivos utilizando ‘with‘ para garantizar un manejo seguro y adecuado de los recursos.

```
1   """
2   Se implementa un gestor de archivos seguro
        utilizando el protocolo de context manager
3   """
4
5   class GestorArchivos:
6       """
7       Se implementa un gestor de contexto para el
        manejo seguro de archivos
8       """
9       def __init__(self, nombre_archivo, modo='r',
        encoding='utf-8'):
10          """
11          Se inicializa el gestor de archivos
12
13          Args:
14              nombre_archivo (str): Nombre del archivo
        a gestionar
15              modo (str): Modo de apertura del archivo
16              encoding (str): Codificación del archivo
17          """
18          self.nombre_archivo = nombre_archivo
19          self.modo = modo
20          self.encoding = encoding
21          self.archivo = None
22
23      def __enter__(self):
24          """
25          Se ejecuta al entrar al contexto del with
26
27          Returns:
28              file: El objeto archivo abierto
```

```
29          """
30          try:
31              self.archivo = open(
32                  self.nombre_archivo,
33                  self.modo,
34                  encoding=self.encoding
35              )
36              return self.archivo
37          except Exception as e:
38              print(f"Error al abrir el archivo: {e}")
39              raise
40
41      def __exit__(self, exc_type, exc_val, exc_tb):
42          """
43          Se ejecuta al salir del contexto del with
44
45          Args:
46              exc_type: Tipo de excepción si ocurrió
    alguna
47              exc_val: Valor de la excepción
48              exc_tb: Traceback de la excepción
49          """
50          try:
51              if self.archivo:
52                  self.archivo.close()
53                  print(f"Se ha cerrado el archivo:
    {self.nombre_archivo}")
54          except Exception as e:
55              print(f"Error al cerrar el archivo: {e}")
56              return False
57
58          # Se registra si ocurrió alguna excepción
59          if exc_type is not None:
60              print(f"Se produjo una excepción:
    {exc_type.__name__}: {exc_val}")
61          return False  # Se propagan las excepciones
62
63  # Ejemplo de uso
64  if __name__ == "__main__":
65      # Ejemplo de escritura
66      print("Ejemplo de escritura:")
67      try:
68          with GestorArchivos("ejemplo.txt", "w") as
    archivo:
69              archivo.write("Línea 1: Prueba de
    escritura\n")
70              archivo.write("Línea 2: Segunda
    prueba\n")
71      except Exception as e:
72          print(f"Error en la escritura: {e}")
```

```
73
74      # Ejemplo de lectura
75      print("\nEjemplo de lectura:")
76      try:
77          with GestorArchivos("ejemplo.txt", "r") as
        archivo:
78              contenido = archivo.read()
79              print("Contenido del archivo:")
80              print(contenido)
81      except Exception as e:
82          print(f"Error en la lectura: {e}")
83
84      # Ejemplo de manejo de errores
85      print("\nEjemplo de manejo de errores:")
86      try:
87          with
        GestorArchivos("archivo_inexistente.txt", "r") as
        archivo:
88              contenido = archivo.read()
89      except FileNotFoundError:
90          print("Se ha manejado correctamente el error
        de archivo no encontrado")
```

Ejercicio 25: Sistema de Logs con Respaldo Crear un script que utilice la clase creada en el ejercicio 11 para gestionar logs y la clase del ejercicio 14 para implementar un sistema de logs que automáticamente genere respaldos cuando el archivo de log alcance cierto tamaño.

Ejercicio 26: Conversor de Datos Multi-formato Desarrollar un script que utilice la clase del ejercicio 19 y la clase del ejercicio 17 para leer datos de múltiples archivos (CSV, JSON, XML) y exportarlos todos a un formato común seleccionado por el usuario.

Ejercicio 27: Monitor de Directorio Seguro Implementar un script que use la clase del ejercicio 16 y la clase del ejercicio 18 para vigilar un directorio y automáticamente comprimir de forma segura cualquier archivo nuevo que aparezca.

Ejercicio 28: Validador de Datos con Logs Crear un script que utilice la clase del ejercicio 15 y la clase del ejercicio 11 para validar múltiples archivos CSV y generar un log detallado de los errores encontrados.

Ejercicio 29: Sistema de Respaldo Simple Crear un script que utilice la clase del ejercicio 14 y la clase del ejercicio 11 para implementar un sistema simple que: 1. Respalde un archivo específico 2. Registre en un log cada vez que se hace un respaldo 3. Incluya una función para verificar si el archivo ha cambiado comparando su tamaño actual

1.5.4 Respuestas a los Ejercicios de la sección 1.4 Manejo de Archivos y Excepciones.

Respuesta 10: (contador_lineas.py) Para contar líneas, palabras y caracteres de un archivo, se implementa una función que utiliza el manejo de contexto (with) para garantizar el cierre adecuado del archivo. Se incorpora el manejo de excepciones para tratar casos donde el archivo no exista o no se pueda leer.

```
def contar_elementos_archivo(ruta_archivo):
    """
    Cuenta el número de líneas, palabras y
    caracteres en un archivo.

    Args:
        ruta_archivo (str): Ruta al archivo a
    analizar

    Returns:
        tuple: (num_lineas, num_palabras,
    num_caracteres)

    Raises:
        FileNotFoundError: Si el archivo no existe
        PermissionError: Si no hay permisos para
    leer el archivo
    """
    try:
        with open(ruta_archivo, 'r',
    encoding='utf-8') as archivo:
            contenido = archivo.read()
            lineas = contenido.split('\n')
            palabras = contenido.split()

            return (
                len(lineas),
                len(palabras),
                len(contenido)
            )

    except FileNotFoundError:
        raise FileNotFoundError(f"El archivo
    {ruta_archivo} no fue encontrado")
```

```
29      except PermissionError:
30          raise PermissionError(f"No hay permisos para
        leer el archivo {ruta_archivo}")
31      except Exception as e:
32          raise Exception(f"Error inesperado al leer
        el archivo: {str(e)}")
```

Respuesta 11: (registro_logs.py) Para el registro de logs, se implementa una función que utiliza la biblioteca logging de Python para gestionar los registros de manera profesional. Se incluye el manejo de excepciones para problemas de permisos y se formatea la salida incluyendo timestamp y nivel de severidad del mensaje.

```
1   import logging
2   from datetime import datetime
3   import os
4
5   class RegistroLogger:
6       """
7       Clase para manejar el registro de logs en
        archivos.
8       """
9       def __init__(self, archivo_log):
10          """
11          Inicializa el logger con un archivo
        específico.
12
13          Args:
14              archivo_log (str): Ruta del archivo
        donde se guardarán los logs
15          """
16          self.archivo_log = archivo_log
17          self.configurar_logger()
18
19      def configurar_logger(self):
20          """Configura el formato y nivel del logger"""
21          try:
22              logging.basicConfig(
23                  filename=self.archivo_log,
24                  level=logging.INFO,
25                  format='%(asctime)s - %(levelname)s
        - %(message)s',
26                  datefmt='%Y-%m-%d %H:%M:%S'
27              )
28          except PermissionError:
29              raise PermissionError(f"No hay permisos
        para escribir en {self.archivo_log}")
30          except Exception as e:
31              raise Exception(f"Error al configurar el
        logger: {str(e)}")
```

```
32
33      def registrar_mensaje(self, mensaje,
        nivel='INFO'):
34          """
35          Registra un mensaje en el archivo de log.
36
37          Args:
38              mensaje (str): Mensaje a registrar
39              nivel (str): Nivel del mensaje (INFO,
        WARNING, ERROR, CRITICAL)
40          """
41          try:
42              nivel = nivel.upper()
43              if nivel == 'INFO':
44                  logging.info(mensaje)
45              elif nivel == 'WARNING':
46                  logging.warning(mensaje)
47              elif nivel == 'ERROR':
48                  logging.error(mensaje)
49              elif nivel == 'CRITICAL':
50                  logging.critical(mensaje)
51              else:
52                  raise ValueError(f"Nivel de log no
        válido: {nivel}")
53
54          except Exception as e:
55              raise Exception(f"Error al registrar el
        mensaje: {str(e)}")
```

Respuesta 12: (fusion_archivos.py) Para fusionar dos archivos alternando sus líneas, se implementa una función que lee ambos archivos simultáneamente y escribe en un tercer archivo. Se maneja el caso donde los archivos tienen diferente longitud continuando con el archivo más largo una vez que el más corto se haya terminado.

```
1  def fusionar_archivos(archivo1, archivo2,
       archivo_salida):
2       """
3       Fusiona dos archivos alternando sus líneas en un
       tercer archivo.
4
5       Args:
6           archivo1 (str): Ruta del primer archivo
7           archivo2 (str): Ruta del segundo archivo
8           archivo_salida (str): Ruta del archivo de
       salida
9
10      Raises:
11          FileNotFoundError: Si alguno de los archivos
       de entrada no existe
```

```
        PermissionError: Si hay problemas de permisos
    """
    try:
        with open(archivo1, 'r', encoding='utf-8')
as f1, \
             open(archivo2, 'r', encoding='utf-8')
as f2, \
             open(archivo_salida, 'w',
encoding='utf-8') as fout:

            # Convertir los file handlers a
iteradores
            lineas1 = iter(f1)
            lineas2 = iter(f2)

            while True:
                try:
                    # Intentar obtener línea del
primer archivo
                    linea1 = next(lineas1)
                    fout.write(linea1)
                except StopIteration:
                    # Si no hay más líneas en
archivo1, escribir el resto de archivo2
                    for linea in lineas2:
```

Respuesta 13: (buscador_palabras.py) Para buscar palabras en archivos de texto, se implementa una función que utiliza el módulo pathlib para recorrer recursivamente directorios y expresiones regulares para la búsqueda precisa de palabras. Se incluye manejo de excepciones para archivos inaccesibles o corruptos.

```
# buscador_palabras.py
from pathlib import Path
import re
from typing import List, Tuple, Dict
import logging

class BuscadorPalabras:
    """
    Clase para buscar palabras en archivos de texto
    dentro de un directorio y sus subdirectorios.
    """
    def __init__(self, directorio_base: str):
        """
        Inicializa el buscador con un directorio
    base.

        Args:
            directorio_base (str): Ruta del
```

```
        directorio donde buscar
17          """
18          self.directorio_base = Path(directorio_base)
19          self.configurar_logger()
20
21      def configurar_logger(self):
22          """Configura el logging para la clase"""
23          logging.basicConfig(
24              level=logging.INFO,
25              format='%(asctime)s - %(levelname)s -
        %(message)s'
26          )
27
28      def buscar_palabra(self, palabra: str) ->
        Dict[str, List[Tuple[int, str]]]:
29          """
30          Busca una palabra en todos los archivos .txt
        del directorio y subdirectorios.
31
32          Args:
33              palabra (str): Palabra a buscar
34
35          Returns:
36              Dict[str, List[Tuple[int, str]]]:
        Diccionario con archivos y líneas donde se
        encontró la palabra
37
38          Raises:
39              FileNotFoundError: Si el directorio base
        no existe
40          """
41          if not self.directorio_base.exists():
42              raise FileNotFoundError(f"El directorio
        {self.directorio_base} no existe")
43
44          resultados = {}
45          patron = re.compile(r'\b' +
        re.escape(palabra) + r'\b', re.IGNORECASE)
46
47          try:
48              for archivo in
        self.directorio_base.rglob('*.txt'):
49                  try:
50                      coincidencias =
        self._buscar_en_archivo(archivo, patron)
51                      if coincidencias:
52                          resultados[str(archivo)] =
        coincidencias
53                  except Exception as e:
54                      logging.error(f"Error al
```

```
        procesar {archivo}: {str(e)}")
55                      continue
56
57              return resultados
58
59          except Exception as e:
60              raise Exception(f"Error al buscar en los
        archivos: {str(e)}")
61
62      def _buscar_en_archivo(self, archivo: Path,
        patron: re.Pattern) -> List[Tuple[int, str]]:
63          """
64          Busca coincidencias de un patrón en un
        archivo específico.
65
66          Args:
67              archivo (Path): Archivo donde buscar
68              patron (re.Pattern): Patrón compilado de
        búsqueda
69
70          Returns:
71              List[Tuple[int, str]]: Lista de tuplas
        (número de línea, contenido)
72          """
73          coincidencias = []
74          try:
75              with archivo.open('r', encoding='utf-8')
        as f:
76                  for num_linea, linea in enumerate(f,
        1):
77                      if patron.search(linea):
78
        coincidencias.append((num_linea, linea.strip()))
79              return coincidencias
80          except Exception as e:
81              logging.warning(f"No se pudo leer
        {archivo}: {str(e)}")
82              return []
```

Respuesta 14: (respaldo_automatico.py) Para realizar copias de seguridad automatizadas, se implementa una función que utiliza la biblioteca shutil para copiar archivos y datetime para generar nombres de archivo únicos con la fecha. Se incluye verificación de integridad del archivo y manejo de excepciones para casos de espacio insuficiente.

```
1  import shutil
2  from datetime import datetime
3  import os
4  import hashlib
5  from typing import Tuple
```

```
6
7  class RespaldoArchivo:
8      """
9      Clase para realizar copias de seguridad de
       archivos con verificación de integridad.
10     """
11     def __init__(self, directorio_respaldo: str):
12         """
13         Inicializa el sistema de respaldo.
14
15         Args:
16             directorio_respaldo (str): Directorio
       donde se guardarán los respaldos
17         """
18         self.directorio_respaldo =
       directorio_respaldo
19         self._crear_directorio_respaldo()
20
21     def _crear_directorio_respaldo(self):
22         """Crea el directorio de respaldo si no
       existe"""
23         try:
24             os.makedirs(self.directorio_respaldo,
       exist_ok=True)
25         except Exception as e:
26             raise Exception(f"No se pudo crear el
       directorio de respaldo: {str(e)}")
27
28     def _calcular_hash(self, ruta_archivo: str) ->
       str:
29         """
30         Calcula el hash SHA-256 de un archivo.
31
32         Args:
33             ruta_archivo (str): Ruta del archivo
34
35         Returns:
36             str: Hash SHA-256 del archivo
37         """
38         sha256_hash = hashlib.sha256()
39         try:
40             with open(ruta_archivo, "rb") as f:
41                 for byte_block in iter(lambda:
       f.read(4096), b""):
42                     sha256_hash.update(byte_block)
43             return sha256_hash.hexdigest()
44         except Exception as e:
45             raise Exception(f"Error al calcular el
       hash: {str(e)}")
46
```

```
47      def crear_respaldo(self, archivo_origen: str) ->
        Tuple[str, str]:
48          """
49          Crea una copia de seguridad del archivo
        especificado.
50
51          Args:
52              archivo_origen (str): Ruta del archivo a
        respaldar
53
54          Returns:
55              Tuple[str, str]: (Ruta del archivo de
        respaldo, hash del archivo)
56
57          Raises:
58              FileNotFoundError: Si el archivo origen
        no existe
59              OSError: Si no hay espacio suficiente en
        disco
60          """
61          if not os.path.exists(archivo_origen):
62              raise FileNotFoundError(f"El archivo
        {archivo_origen} no existe")
63
64          try:
65              # Generar nombre del archivo de respaldo
66              nombre_base =
        os.path.basename(archivo_origen)
67              nombre, extension =
        os.path.splitext(nombre_base)
68              fecha_actual =
        datetime.now().strftime("%Y%m%d_%H%M%S")
69              nombre_respaldo =
        f"{nombre}_{fecha_actual}{extension}"
70              ruta_respaldo =
        os.path.join(self.directorio_respaldo,
        nombre_respaldo)
71
72              # Verificar espacio en disco
73              espacio_necesario =
        os.path.getsize(archivo_origen)
74              espacio_disponible =
        shutil.disk_usage(self.directorio_respaldo).free
75              if espacio_disponible <
        espacio_necesario:
76                  raise OSError("Espacio insuficiente
        en disco para crear el respaldo")
77
78              # Crear respaldo
79              shutil.copy2(archivo_origen,
```

```
        ruta_respaldo)
80
81              # Verificar integridad
82              hash_original =
        self._calcular_hash(archivo_origen)
83              hash_respaldo =
        self._calcular_hash(ruta_respaldo)
84
85              if hash_original != hash_respaldo:
86                  os.remove(ruta_respaldo)
87                  raise Exception("Error de integridad
        en el respaldo")
88
89              return ruta_respaldo, hash_respaldo
90
91          except Exception as e:
92              raise Exception(f"Error al crear el
        respaldo: {str(e)}")
```

Respuesta 15: (validador_csv.py) Para validar archivos CSV, se implementa una clase que verifica la estructura y consistencia de los datos utilizando el módulo csv de Python. Se crean excepciones personalizadas para diferentes tipos de errores y se incluye validación de tipos de datos por columna.

```
1  # validador_csv.py
2  import csv
3  from typing import List, Dict, Any
4  import logging
5
6  class ErrorCSV(Exception):
7      """Clase base para excepciones de validación
       CSV"""
8      pass
9
10 class ErrorEstructuraCSV(ErrorCSV):
11     """Error en la estructura del CSV (número de
       columnas)"""
12     pass
13
14 class ErrorTipoDatoCSV(ErrorCSV):
15     """Error en el tipo de dato de una columna"""
16     pass
17
18 class ValidadorCSV:
19     """
20     Clase para validar la estructura y consistencia
       de archivos CSV.
21     """
22     def __init__(self, tipos_columnas: Dict[str,
       type]):
```

```
23          """
24          Inicializa el validador con los tipos
    esperados por columna.
25
26          Args:
27              tipos_columnas (Dict[str, type]):
    Diccionario con nombres de columnas y sus tipos
28          """
29          self.tipos_columnas = tipos_columnas
30          self._configurar_logger()
31
32      def _configurar_logger(self):
33          """Configura el sistema de logging"""
34          logging.basicConfig(
35              level=logging.INFO,
36              format='%(asctime)s - %(levelname)s -
    %(message)s'
37          )
38
39      def _validar_tipo_dato(self, valor: str, tipo:
    type, columna: str) -> bool:
40          """
41          Valida si un valor puede ser convertido al
    tipo especificado.
42
43          Args:
44              valor (str): Valor a validar
45              tipo (type): Tipo esperado
46              columna (str): Nombre de la columna
47
48          Returns:
49              bool: True si el valor es válido
50
51          Raises:
52              ErrorTipoDatoCSV: Si el valor no puede
    ser convertido al tipo esperado
53          """
54          try:
55              if tipo == str:
56                  return True
57              elif tipo == int:
58                  int(valor)
59              elif tipo == float:
60                  float(valor)
61              elif tipo == bool:
62                  valor = valor.lower()
63                  if valor not in ('true', 'false',
    '1', '0'):
64                      raise ValueError
65              return True
```

```
66          except ValueError:
67              raise ErrorTipoDatoCSV(
68                  f"Valor '{valor}' en columna
    '{columna}' no es de tipo {tipo.__name__}"
69              )
70
71      def validar_archivo(self, ruta_archivo: str) ->
    bool:
72          """
73          Valida la estructura y tipos de datos de un
    archivo CSV.
74
75          Args:
76              ruta_archivo (str): Ruta al archivo CSV
    a validar
77
78          Returns:
79              bool: True si el archivo es válido
80
81          Raises:
82              ErrorCSV: Si se encuentra algún error en
    el archivo
83          """
84          try:
85              with open(ruta_archivo, 'r',
    encoding='utf-8') as archivo:
86                  lector = csv.DictReader(archivo)
87
88                  # Validar nombres de columnas
89                  columnas_esperadas =
    set(self.tipos_columnas.keys())
90                  columnas_archivo =
    set(lector.fieldnames)
91
92                  if columnas_esperadas !=
    columnas_archivo:
93                      raise ErrorEstructuraCSV(
94                          f"Columnas no coinciden.
    Esperadas: {columnas_esperadas}, "
95                          f"Encontradas:
    {columnas_archivo}"
96                      )
97
98                  # Validar cada fila
99                  for num_fila, fila in
    enumerate(lector, start=2):
100                     for columna, valor in
    fila.items():
101                         tipo_esperado =
    self.tipos_columnas[columna]
```

```
102                         try:
103
        self._validar_tipo_dato(valor, tipo_esperado,
        columna)
104                         except ErrorTipoDatoCSV as e:
105                             raise ErrorTipoDatoCSV(
106                                 f"Error en fila
        {num_fila}: {str(e)}"
107                             )
108
109                 return True
110
111         except FileNotFoundError:
112             raise ErrorCSV(f"No se encontró el
        archivo {ruta_archivo}")
113         except csv.Error as e:
114             raise ErrorCSV(f"Error al leer el CSV:
        {str(e)}")
115         except Exception as e:
116             raise ErrorCSV(f"Error inesperado:
        {str(e)}")
```

Respuesta 16: (monitor_archivos.py) Para monitorear cambios en archivos, se implementa una clase que utiliza la biblioteca watchdog para detectar modificaciones en tiempo real. Se incluye un sistema de logging para registrar los cambios y manejo de excepciones para problemas de permisos del sistema de archivos.

```
1  import os
2  import time
3  import logging
4  from datetime import datetime
5  from typing import Optional, Callable
6  import threading
7
8  class MonitorArchivos:
9      """
10     Clase para monitorear cambios en archivos
       específicos.
11     """
12     def __init__(self, archivo_monitorear: str,
       archivo_log: str):
13         """
14         Inicializa el monitor de archivos.
15
16         Args:
17             archivo_monitorear (str): Ruta del
       archivo a monitorear
18             archivo_log (str): Ruta del archivo
       donde se guardarán los logs
```

```
19          """
20          self.archivo_monitorear =
    os.path.abspath(archivo_monitorear)
21          self.archivo_log = archivo_log
22          self.ultima_modificacion = None
23          self.detener = False
24          self.hilo_monitor:
    Optional[threading.Thread] = None
25          self._configurar_logger()
26
27      def _configurar_logger(self):
28          """Configura el sistema de logging"""
29          logging.basicConfig(
30              filename=self.archivo_log,
31              level=logging.INFO,
32              format='%(asctime)s - %(levelname)s -
    %(message)s',
33              datefmt='%Y-%m-%d %H:%M:%S'
34          )
35
36      def _verificar_cambios(self, callback:
    Optional[Callable] = None):
37          """
38          Verifica cambios en el archivo monitoreado.
39
40          Args:
41              callback (Callable, optional): Función a
    llamar cuando se detecta un cambio
42          """
43          while not self.detener:
44              try:
45                  if
    os.path.exists(self.archivo_monitorear):
46                      estado_actual =
    os.stat(self.archivo_monitorear)
47                      tiempo_mod_actual =
    estado_actual.st_mtime
48
49                      if self.ultima_modificacion is
    None:
50                          self.ultima_modificacion =
    tiempo_mod_actual
51                      elif tiempo_mod_actual !=
    self.ultima_modificacion:
52                          # Se detectó un cambio
53                          mensaje = f"Archivo
    modificado: {self.archivo_monitorear}"
54                          logging.info(mensaje)
55
56                          if callback:
```

```
57                                  # Crear un objeto evento
    simple para mantener compatibilidad
58                                  class EventoSimple:
59                                      def __init__(self,
    ruta):
60                                          self.src_path =
    ruta
61
    self.is_directory = False
62
63
    callback(EventoSimple(self.archivo_monitorear))
64
65                              self.ultima_modificacion =
    tiempo_mod_actual
66
    self._registrar_detalles_modificacion()
67                      else:
68                          if self.ultima_modificacion is
    not None:
69                              # El archivo fue eliminado
70                              mensaje = f"Archivo
    eliminado: {self.archivo_monitorear}"
71                              logging.warning(mensaje)
72                              self.ultima_modificacion =
    None
73
74                  except Exception as e:
75                      logging.error(f"Error al monitorear
    archivo: {str(e)}")
76
77                  time.sleep(1)  # Esperar 1 segundo entre
    verificaciones
78
79          def _registrar_detalles_modificacion(self):
80              """Registra detalles adicionales sobre la
    modificación"""
81              try:
82                  stats = os.stat(self.archivo_monitorear)
83                  detalles = {
84                      'tamaño': stats.st_size,
85                      'última_modificación':
    datetime.fromtimestamp(stats.st_mtime),
86                      'permisos': oct(stats.st_mode)[-3:]
87                  }
88                  logging.info(f"Detalles de modificación:
    {detalles}")
89              except Exception as e:
90                  logging.error(f"Error al obtener
    detalles del archivo: {str(e)}")
```

```
91
92      def iniciar_monitoreo(self, callback:
        Optional[Callable] = None):
93          """
94          Inicia el monitoreo del archivo.
95
96          Args:
97              callback (Callable, optional): Función a
        llamar cuando se detecta un cambio
98
99          Raises:
100             FileNotFoundError: Si el archivo a
        monitorear no existe
101             PermissionError: Si no hay permisos
        suficientes
102         """
103         if not
        os.path.exists(self.archivo_monitorear):
104             raise FileNotFoundError(f"El archivo
        {self.archivo_monitorear} no existe")
105
106         try:
107             self.detener = False
108             self.hilo_monitor = threading.Thread(
109                 target=self._verificar_cambios,
110                 args=(callback,),
111                 daemon=True
112             )
113             self.hilo_monitor.start()
114             logging.info(f"Iniciando monitoreo de
        {self.archivo_monitorear}")
115
116         except Exception as e:
117             raise Exception(f"Error al iniciar el
        monitoreo: {str(e)}")
118
119     def detener_monitoreo(self):
120         """Detiene el monitoreo del archivo"""
121         self.detener = True
122         if self.hilo_monitor:
123             self.hilo_monitor.join(timeout=2)
124             logging.info("Monitoreo detenido")
```

Respuesta 17: (convertidor_formatos.py) Para convertir archivos entre diferentes formatos, se implementa una clase que utiliza las bibliotecas json y xml.etree.ElementTree de Python. Se incluye manejo de codificación de caracteres y validación de formatos de entrada/salida.

```
1  # convertidor_formatos.py
2  import json
```

```
3  import xml.etree.ElementTree as ET
4  from typing import Dict, Any, Union
5  import html
6  import logging
7
8  class ErrorConversion(Exception):
9      """Excepción base para errores de conversión"""
10     pass
11
12 class ConvertidorFormatos:
13     """
14     Clase para convertir archivos entre diferentes
       formatos (texto, JSON, XML, HTML).
15     """
16     def __init__(self):
17         """Inicializa el convertidor y configura el
       logging"""
18         self._configurar_logger()
19
20     def _configurar_logger(self):
21         """Configura el sistema de logging"""
22         logging.basicConfig(
23             level=logging.INFO,
24             format='%(asctime)s - %(levelname)s -
       %(message)s'
25         )
26
27     def _leer_archivo(self, ruta_archivo: str,
       encoding: str = 'utf-8') -> str:
28         """
29         Lee el contenido de un archivo.
30
31         Args:
32             ruta_archivo (str): Ruta del archivo a
       leer
33             encoding (str): Codificación del archivo
34
35         Returns:
36             str: Contenido del archivo
37         """
38         try:
39             with open(ruta_archivo, 'r',
       encoding=encoding) as f:
40                 return f.read()
41         except UnicodeDecodeError:
42             raise ErrorConversion(f"Error de
       codificación al leer {ruta_archivo}")
43         except Exception as e:
44             raise ErrorConversion(f"Error al leer el
       archivo: {str(e)}")
```

```
45
46      def _escribir_archivo(self, ruta_archivo: str,
        contenido: str, encoding: str = 'utf-8'):
47          """
48          Escribe contenido en un archivo.
49
50          Args:
51              ruta_archivo (str): Ruta donde escribir
52              contenido (str): Contenido a escribir
53              encoding (str): Codificación a usar
54          """
55          try:
56              with open(ruta_archivo, 'w',
        encoding=encoding) as f:
57                  f.write(contenido)
58          except Exception as e:
59              raise ErrorConversion(f"Error al
        escribir el archivo: {str(e)}")
60
61      def texto_a_json(self, texto: str) -> Dict[str,
        Any]:
62          """
63          Convierte texto a formato JSON.
64
65          Args:
66              texto (str): Texto a convertir
67
68          Returns:
69              Dict[str, Any]: Diccionario con el
        contenido
70          """
71          try:
72              return {
73                  "contenido": texto,
74                  "longitud": len(texto),
75                  "lineas": texto.count('\n') + 1
76              }
77          except Exception as e:
78              raise ErrorConversion(f"Error al
        convertir a JSON: {str(e)}")
79
80      def texto_a_xml(self, texto: str) -> str:
81          """
82          Convierte texto a formato XML.
83
84          Args:
85              texto (str): Texto a convertir
86
87          Returns:
88              str: String con formato XML
```

```
89              """
90              try:
91                  root = ET.Element("documento")
92                  contenido = ET.SubElement(root,
        "contenido")
93                  contenido.text = texto
94
95                  metadata = ET.SubElement(root,
        "metadata")
96                  longitud = ET.SubElement(metadata,
        "longitud")
97                  longitud.text = str(len(texto))
98
99                  return ET.tostring(root,
        encoding='unicode', method='xml')
100             except Exception as e:
101                 raise ErrorConversion(f"Error al
        convertir a XML: {str(e)}")
102
103     def texto_a_html(self, texto: str) -> str:
104         """
105         Convierte texto a formato HTML.
106
107         Args:
108             texto (str): Texto a convertir
109
110         Returns:
111             str: String con formato HTML
112         """
113         try:
114             texto_escapado = html.escape(texto)
115             parrafos = [f"<p>{p}</p>" for p in
        texto_escapado.split('\n\n')]
116             return f"""
117             <!DOCTYPE html>
118             <html>
119             <head>
120                 <meta charset="utf-8">
121                 <title>Documento Convertido</title>
122                 <style>
123                     body {{ font-family: Arial,
        sans-serif; margin: 2em; }}
124                     p {{ line-height: 1.5; }}
125                 </style>
126             </head>
127             <body>
128                 {''.join(parrafos)}
129             </body>
130             </html>
131             """
```

```
132             except Exception as e:
133                 raise ErrorConversion(f"Error al
        convertir a HTML: {str(e)}")
134
135         def convertir_archivo(self, archivo_entrada:
        str, archivo_salida: str,
136                               formato_salida: str,
        encoding: str = 'utf-8'):
137             """
138             Convierte un archivo a otro formato.
139
140             Args:
141                 archivo_entrada (str): Ruta del archivo
        de entrada
142                 archivo_salida (str): Ruta del archivo
        de salida
143                 formato_salida (str): Formato deseado
        ('json', 'xml', 'html')
144                 encoding (str): Codificación a usar
145             """
146             formato_salida = formato_salida.lower()
147             if formato_salida not in ['json', 'xml',
        'html']:
148                 raise ValueError(f"Formato de salida no
        soportado: {formato_salida}")
149
150             try:
151                 contenido =
        self._leer_archivo(archivo_entrada, encoding)
152
153                 if formato_salida == 'json':
154                     resultado =
        json.dumps(self.texto_a_json(contenido), indent=2)
155                 elif formato_salida == 'xml':
156                     resultado =
        self.texto_a_xml(contenido)
157                 else:  # html
158                     resultado =
        self.texto_a_html(contenido)
159
160                 self._escribir_archivo(archivo_salida,
        resultado, encoding)
161                 logging.info(f"Archivo convertido
        exitosamente a {formato_salida}")
162
163             except Exception as e:
164                 raise ErrorConversion(f"Error en la
        conversión: {str(e)}")
```

Respuesta 18: (archivador_seguro.py) Para comprimir y descomprimir ar-

chivos de manera segura, se implementa una clase que utiliza el módulo zipfile con verificación de integridad mediante checksums. Se incluye encriptación básica y manejo de excepciones para casos de corrupción.

```
import zipfile
import os
import hashlib
from typing import List, Tuple
import logging
from pathlib import Path

class ArchivadorSeguro:
    """
    Clase para comprimir y descomprimir archivos de manera segura.
    """
    def __init__(self, directorio_trabajo: str):
        """
        Inicializa el archivador con un directorio de trabajo.

        Args:
            directorio_trabajo (str): Directorio base para las operaciones
        """
        self.directorio_trabajo = Path(directorio_trabajo)
        self._configurar_logger()

    def _configurar_logger(self):
        """Configura el sistema de logging"""
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(levelname)s - %(message)s'
        )

    def _calcular_checksum(self, ruta_archivo: str) -> str:
        """
        Calcula el checksum SHA-256 de un archivo.

        Args:
            ruta_archivo (str): Ruta del archivo

        Returns:
            str: Checksum del archivo
        """
        sha256_hash = hashlib.sha256()
        try:
```

```
41              with open(ruta_archivo, "rb") as f:
42                  for byte_block in iter(lambda:
    f.read(4096), b""):
43                      sha256_hash.update(byte_block)
44              return sha256_hash.hexdigest()
45          except Exception as e:
46              raise Exception(f"Error al calcular
    checksum: {str(e)}")
47
48      def comprimir(self, archivos: List[str],
    archivo_zip: str,
49                    password: str = None) -> Tuple[str,
    List[str]]:
50          """
51          Comprime archivos en un ZIP con verificación
    de integridad.
52
53          Args:
54              archivos (List[str]): Lista de archivos
    a comprimir
55              archivo_zip (str): Nombre del archivo
    ZIP resultante
56              password (str, optional): Contraseña
    para el ZIP
57
58          Returns:
59              Tuple[str, List[str]]: (Ruta del ZIP,
    Lista de checksums)
60          """
61          try:
62              ruta_zip = self.directorio_trabajo /
    archivo_zip
63              checksums = []
64
65              # Calcular checksums antes de comprimir
66              for archivo in archivos:
67                  if not os.path.exists(archivo):
68                      raise
    FileNotFoundError(f"Archivo no encontrado:
    {archivo}")
69
    checksums.append(self._calcular_checksum(archivo))
70
71              # Crear archivo ZIP
72              with zipfile.ZipFile(ruta_zip, 'w',
    zipfile.ZIP_DEFLATED) as zf:
73                  if password:
74                      zf.setpassword(password.encode())
75
76                  # Agregar archivos al ZIP
```

```
77                  for archivo in archivos:
78                      zf.write(archivo,
    os.path.basename(archivo))
79
80                  # Agregar archivo de verificación
81                  info_verificacion = "Checksums:\n"
82                  for archivo, checksum in
    zip(archivos, checksums):
83                      info_verificacion +=
    f"{os.path.basename(archivo)}: {checksum}\n"
84                  zf.writestr("verificacion.txt",
    info_verificacion)
85
86              return str(ruta_zip), checksums
87
88          except Exception as e:
89              if os.path.exists(ruta_zip):
90                  os.remove(ruta_zip)
91              raise Exception(f"Error en la
    compresión: {str(e)}")
92
93      def descomprimir(self, archivo_zip: str,
    directorio_destino: str,
94                       password: str = None) -> bool:
95          """
96          Descomprime un archivo ZIP y verifica su
    integridad.
97
98          Args:
99              archivo_zip (str): Ruta del archivo ZIP
100             directorio_destino (str): Directorio
    donde descomprimir
101             password (str, optional): Contraseña del
    ZIP
102
103         Returns:
104             bool: True si la descompresión fue
    exitosa y verificada
105         """
106         try:
107             ruta_zip = Path(archivo_zip)
108             dir_destino = Path(directorio_destino)
109             dir_destino.mkdir(parents=True,
    exist_ok=True)
110
111             with zipfile.ZipFile(ruta_zip, 'r') as
    zf:
112                 if password:
113                     zf.setpassword(password.encode())
114
```

```
115                 # Verificar integridad del ZIP
116                 if zf.testzip() is not None:
117                     raise Exception("El archivo ZIP
    está corrupto")
118 
119                 # Extraer archivos
120                 zf.extractall(dir_destino)
121 
122                 # Verificar checksums si existe
    archivo de verificación
123                 if "verificacion.txt" in
    zf.namelist():
124 
    self._verificar_checksums(dir_destino)
125 
126             return True
127 
128         except Exception as e:
129             raise Exception(f"Error en la
    descompresión: {str(e)}")
130 
131     def _verificar_checksums(self, directorio: Path)
    -> bool:
132         """
133         Verifica los checksums de los archivos
    descomprimidos.
134 
135         Args:
136             directorio (Path): Directorio con los
    archivos descomprimidos
137 
138         Returns:
139             bool: True si todos los checksums
    coinciden
140         """
141         try:
142             verificacion = directorio /
    "verificacion.txt"
143             checksums_originales = {}
144 
145             # Leer checksums originales
146             with open(verificacion, 'r') as f:
147                 for linea in f.readlines()[1:]:  #
    Saltar la primera línea
148                     archivo, checksum =
    linea.strip().split(': ')
149                     checksums_originales[archivo] =
    checksum
150 
151             # Verificar cada archivo
```

```
152             for archivo, checksum_original in
    checksums_originales.items():
153                 if archivo != "verificacion.txt":
154                     ruta_archivo = directorio /
    archivo
155                     checksum_actual =
    self._calcular_checksum(str(ruta_archivo))
156                     if checksum_actual !=
    checksum_original:
157                         raise Exception(f"Checksum
    no coincide para {archivo}")
158
159             return True
160
161         except Exception as e:
162             raise Exception(f"Error en la
    verificación de checksums: {str(e)}")
```

Respuesta 19: (importador_datos.py) Para importar datos de diferentes formatos, se implementa una clase que utiliza múltiples bibliotecas (csv, json, xml.etree) para procesar cada tipo de archivo. Se incluye validación de estructura y manejo de excepciones específicas por formato.

```
1  import csv
2  import json
3  import xml.etree.ElementTree as ET
4  from typing import Dict, List, Any, Union
5  import logging
6  from pathlib import Path
7
8  class ErrorImportacion(Exception):
9      """Excepción base para errores de importación"""
10     pass
11
12 class ImportadorDatos:
13     """
14     Clase para importar datos desde diferentes
       formatos de archivo.
15     """
16     def __init__(self):
17         """Inicializa el importador y configura el
       logging"""
18         self._configurar_logger()
19         self.formatos_soportados = {
20             '.csv': self._importar_csv,
21             '.json': self._importar_json,
22             '.xml': self._importar_xml
23         }
24
25     def _configurar_logger(self):
```

```
26          """Configura el sistema de logging"""
27          logging.basicConfig(
28              level=logging.INFO,
29              format='%(asctime)s - %(levelname)s -
    %(message)s'
30          )
31
32      def _detectar_formato(self, ruta_archivo: str)
    -> str:
33          """
34          Detecta el formato del archivo por su
    extensión.
35
36          Args:
37              ruta_archivo (str): Ruta del archivo
38
39          Returns:
40              str: Extensión del archivo
41          """
42          extension = Path(ruta_archivo).suffix.lower()
43          if extension not in self.formatos_soportados:
44              raise ErrorImportacion(f"Formato no
    soportado: {extension}")
45          return extension
46
47      def _importar_csv(self, ruta_archivo: str) ->
    List[Dict[str, Any]]:
48          """
49          Importa datos desde un archivo CSV.
50
51          Args:
52              ruta_archivo (str): Ruta del archivo CSV
53
54          Returns:
55              List[Dict[str, Any]]: Lista de
    diccionarios con los datos
56          """
57          try:
58              datos = []
59              with open(ruta_archivo, 'r',
    encoding='utf-8') as f:
60                  lector = csv.DictReader(f)
61                  for fila in lector:
62                      datos.append(dict(fila))
63              return datos
64          except Exception as e:
65              raise ErrorImportacion(f"Error al
    importar CSV: {str(e)}")
66
67      def _importar_json(self, ruta_archivo: str) ->
```

```
        Union[Dict[str, Any], List[Any]]:
68          """
69          Importa datos desde un archivo JSON.
70
71          Args:
72              ruta_archivo (str): Ruta del archivo JSON
73
74          Returns:
75              Union[Dict[str, Any], List[Any]]: Datos
        del JSON
76          """
77          try:
78              with open(ruta_archivo, 'r',
        encoding='utf-8') as f:
79                  return json.load(f)
80          except json.JSONDecodeError as e:
81              raise ErrorImportacion(f"Error de
        formato JSON: {str(e)}")
82          except Exception as e:
83              raise ErrorImportacion(f"Error al
        importar JSON: {str(e)}")
84
85      def _importar_xml(self, ruta_archivo: str) ->
        Dict[str, Any]:
86          """
87          Importa datos desde un archivo XML.
88
89          Args:
90              ruta_archivo (str): Ruta del archivo XML
91
92          Returns:
93              Dict[str, Any]: Diccionario con los
        datos del XML
94          """
95          try:
96              tree = ET.parse(ruta_archivo)
97              root = tree.getroot()
98              return self._xml_a_dict(root)
99          except ET.ParseError as e:
100             raise ErrorImportacion(f"Error de
        formato XML: {str(e)}")
101         except Exception as e:
102             raise ErrorImportacion(f"Error al
        importar XML: {str(e)}")
103
104     def _xml_a_dict(self, elemento: ET.Element) ->
        Union[Dict[str, Any], str]:
105         """
106         Convierte un elemento XML a diccionario.
107
```

```
108             Args:
109                 elemento (ET.Element): Elemento XML
110 
111             Returns:
112                 Union[Dict[str, Any], str]: Diccionario
        o string con los datos
113             """
114             resultado = {}
115 
116             # Procesar atributos
117             if elemento.attrib:
118                 resultado.update(elemento.attrib)
119 
120             # Procesar subelementos
121             for subelemento in elemento:
122                 if len(subelemento) == 0 and not
        subelemento.attrib:
123                     # Elemento simple con solo texto
124                     resultado[subelemento.tag] =
        subelemento.text
125                 else:
126                     # Elemento complejo
127                     if subelemento.tag not in resultado:
128                         resultado[subelemento.tag] =
        self._xml_a_dict(subelemento)
129                     else:
130                         # Si ya existe, convertir a lista
131                         if not
        isinstance(resultado[subelemento.tag], list):
132                             resultado[subelemento.tag] =
        [resultado[subelemento.tag]]
133 
        resultado[subelemento.tag].append(
134 
        self._xml_a_dict(subelemento))
135 
136             return resultado if resultado else
        (elemento.text or "")
137 
138         def importar_archivo(self, ruta_archivo: str) ->
        Union[Dict[str, Any], List[Any]]:
139             """
140             Importa datos desde un archivo en cualquier
        formato soportado.
141 
142             Args:
143                 ruta_archivo (str): Ruta del archivo a
        importar
144 
145             Returns:
```

```
146             Union[Dict[str, Any], List[Any]]: Datos
    importados
147         """
148         try:
149             formato =
    self._detectar_formato(ruta_archivo)
150             funcion_importacion =
    self.formatos_soportados[formato]
151             datos = funcion_importacion(ruta_archivo)
152
153             logging.info(f"Archivo importado
    exitosamente: {ruta_archivo}")
154             return datos
155
156         except Exception as e:
157             raise ErrorImportacion(f"Error al
    importar {ruta_archivo}: {str(e)}")
158
159     def importar_multiples(self, rutas_archivos:
    List[str]) -> Dict[str, Any]:
160         """
161         Importa datos desde múltiples archivos.
162
163         Args:
164             rutas_archivos (List[str]): Lista de
    rutas de archivos
165
166         Returns:
167             Dict[str, Any]: Diccionario con los
    datos de todos los archivos
168         """
169         resultados = {}
170         errores = []
171
172         for ruta in rutas_archivos:
173             try:
174                 nombre_archivo = Path(ruta).stem
175                 resultados[nombre_archivo] =
    self.importar_archivo(ruta)
176             except Exception as e:
177                 errores.append(f"{ruta}: {str(e)}")
178
179         if errores:
180             logging.warning("Errores durante la
    importación múltiple:\n" +
181                             "\n".join(errores))
182
183         return resultados
```

Respuesta 25: (sistema_logs_respaldo.py) Se implementa un sistema que

combina el registro de logs con respaldo automático utilizando las clases RegistroLogger y RespaldoArchivo. El sistema monitorea el tamaño del archivo de log y crea respaldos automáticamente cuando se excede un límite especificado.

```
from registro_logs import RegistroLogger
from respaldo_automatico import RespaldoArchivo
import os
from datetime import datetime
import time

class SistemaLogsRespaldo:
    """
    Sistema que combina logging con respaldo automático.
    """
    def __init__(self, archivo_log: str, dir_respaldo: str, tamano_max_mb: float = 10.0):
        """
        Inicializa el sistema de logs con respaldo.

        Args:
            archivo_log (str): Ruta del archivo de log
            dir_respaldo (str): Directorio para respaldos
            tamano_max_mb (float): Tamaño máximo del log en MB
        """
        self.archivo_log = archivo_log
        self.tamano_max_bytes = tamano_max_mb * 1024 * 1024
        self.logger = RegistroLogger(archivo_log)
        self.respaldador = RespaldoArchivo(dir_respaldo)

    def verificar_tamano_log(self) -> bool:
        """Verifica si el archivo de log excede el tamaño máximo"""
        return os.path.getsize(self.archivo_log) > self.tamano_max_bytes

    def respaldar_log(self):
        """Crea un respaldo del archivo de log actual"""
        try:
            ruta_respaldo, hash_respaldo = self.respaldador.crear_respaldo(self.archivo_log)
            # Limpiar archivo de log original
            open(self.archivo_log, 'w').close()
```

```
35              return ruta_respaldo
36          except Exception as e:
37              print(f"Error al respaldar log: {e}")
38              return None
39
40  def ejemplo_uso():
41      """Ejemplo de uso del sistema de logs con
        respaldo"""
42      sistema = SistemaLogsRespaldo(
43          archivo_log="app.log",
44          dir_respaldo="respaldos/logs",
45          tamano_max_mb=0.1  # 100 KB para prueba
46      )
47
48      # Simular actividad del sistema
49      for i in range(1000):
50          try:
51              # Simular diferentes tipos de mensajes
52              if i % 3 == 0:
53                  sistema.logger.registrar_mensaje(
54                      f"Operación normal #{i}", "INFO"
55                  )
56              elif i % 3 == 1:
57                  sistema.logger.registrar_mensaje(
58                      f"Advertencia en proceso #{i}",
        "WARNING"
59                  )
60              else:
61                  sistema.logger.registrar_mensaje(
62                      f"Error en operación #{i}",
        "ERROR"
63                  )
64
65              # Verificar tamaño y respaldar si es
        necesario
66              if sistema.verificar_tamano_log():
67                  ruta_respaldo =
        sistema.respaldar_log()
68                  print(f"Log respaldado en:
        {ruta_respaldo}")
69
70              time.sleep(0.1)  # Simular tiempo entre
        operaciones
71
72          except Exception as e:
73              print(f"Error en la simulación: {e}")
74              break
75
76  if __name__ == "__main__":
77      ejemplo_uso()
```

Respuesta 26: (conversor_multiformato.py) Se desarrolla un sistema que utiliza las clases ImportadorDatos y ConvertidorFormatos para procesar archivos en diferentes formatos y convertirlos a un formato común. El sistema maneja errores de forma robusta y registra el progreso de las conversiones.

```
from importador_datos import ImportadorDatos
from convertidor_formatos import ConvertidorFormatos
from pathlib import Path
from typing import List, Dict, Any
import logging

class ConversorMultiformato:
    """
    Clase para convertir múltiples archivos a un formato común.
    """
    def __init__(self, dir_entrada: str, dir_salida: str):
        """
        Inicializa el conversor multiformato.

        Args:
            dir_entrada (str): Directorio con archivos a convertir
            dir_salida (str): Directorio para archivos convertidos
        """
        self.dir_entrada = Path(dir_entrada)
        self.dir_salida = Path(dir_salida)
        self.importador = ImportadorDatos()
        self.convertidor = ConvertidorFormatos()
        self._configurar_logger()

    def _configurar_logger(self):
        """Configura el sistema de logging"""
        logging.basicConfig(
            level=logging.INFO,
            format='%(asctime)s - %(levelname)s - %(message)s'
        )

    def procesar_archivos(self, formato_salida: str):
        """
        Procesa todos los archivos soportados en el directorio.

        Args:
            formato_salida (str): Formato de salida deseado
        """
```

```
39          self.dir_salida.mkdir(parents=True,
    exist_ok=True)
40          archivos_procesados = []
41          errores = []
42
43          # Buscar archivos en formatos soportados
44          archivos = []
45          for extension in
    self.importador.formatos_soportados.keys():
46              archivos.extend(
47
    self.dir_entrada.glob(f"*{extension}"))
48
49          for archivo in archivos:
50              try:
51                  # Importar datos
52                  datos =
    self.importador.importar_archivo(str(archivo))
53
54                  # Crear archivo temporal con los
    datos
55                  archivo_temp = self.dir_salida /
    f"temp_{archivo.stem}.txt"
56                  with open(archivo_temp, 'w',
    encoding='utf-8') as f:
57                      f.write(str(datos))
58
59                  # Convertir al formato deseado
60                  archivo_salida = self.dir_salida /
    f"{archivo.stem}.{formato_salida}"
61                  self.convertidor.convertir_archivo(
62                      str(archivo_temp),
63                      str(archivo_salida),
64                      formato_salida
65                  )
66
67                  # Eliminar archivo temporal
68                  archivo_temp.unlink()
69
70
    archivos_procesados.append(archivo.name)
71                  logging.info(f"Archivo procesado:
    {archivo.name}")
72
73              except Exception as e:
74                  errores.append((archivo.name,
    str(e)))
75                  logging.error(f"Error procesando
    {archivo.name}: {e}")
76
```

```
77          return archivos_procesados, errores
78
79  def ejemplo_uso():
80      """Ejemplo de uso del conversor multiformato"""
81      # Configurar directorios
82      dir_entrada = "datos/entrada"
83      dir_salida = "datos/salida"
84
85      # Crear instancia del conversor
86      conversor = ConversorMultiformato(dir_entrada,
        dir_salida)
87
88      # Procesar archivos a formato HTML
89      print("Convirtiendo archivos a HTML...")
90      procesados, errores =
        conversor.procesar_archivos("html")
91
92      # Mostrar resultados
93      print("\nArchivos procesados exitosamente:")
94      for archivo in procesados:
95          print(f"OK {archivo}")
96
97      if errores:
98          print("\nErrores encontrados:")
99          for archivo, error in errores:
100             print(f"ERROR en {archivo}: {error}")
101
102 if __name__ == "__main__":
103     ejemplo_uso()
```

Respuesta 27: (monitor_directorio_seguro.py) Se implementa un sistema de monitoreo que combina las clases MonitorArchivos y ArchivadorSeguro para vigilar un directorio y comprimir automáticamente los nuevos archivos que aparezcan, incluyendo verificación de integridad y manejo de concurrencia.

```
1   """
2   Se implementa un sistema de monitoreo que vigila un
        directorio y comprime
3   automáticamente los archivos nuevos de forma segura,
        sin depender de watchdog
4   """
5
6   import os
7   import time
8   import hashlib
9   import logging
10  from datetime import datetime
11  import zipfile
12  from typing import Set, Optional
13
```

```
14  class ArchivadorSeguro:
15      """
16      Se implementa la compresión segura de archivos
        con verificación de integridad
17      """
18      def __init__(self, directorio_destino: str):
19          self.directorio_destino = directorio_destino
20          if not os.path.exists(directorio_destino):
21              os.makedirs(directorio_destino)
22
23      def _calcular_checksum(self, archivo: str) ->
        str:
24          """
25          Se calcula el checksum SHA-256 de un archivo
26          """
27          hasher = hashlib.sha256()
28          with open(archivo, 'rb') as f:
29              for chunk in iter(lambda: f.read(4096),
        b''):
30                  hasher.update(chunk)
31          return hasher.hexdigest()
32
33      def comprimir_archivo(self, archivo_origen: str)
        -> Optional[str]:
34          """
35          Se comprime un archivo de forma segura y
        verifica su integridad
36          """
37          if not os.path.exists(archivo_origen):
38              logging.error(f"El archivo
        {archivo_origen} no existe")
39              return None
40
41          try:
42              # Se calcula el checksum original
43              checksum_original =
        self._calcular_checksum(archivo_origen)
44
45              # Se genera el nombre del archivo
        comprimido
46              nombre_base =
        os.path.basename(archivo_origen)
47              timestamp =
        datetime.now().strftime("%Y%m%d_%H%M%S")
48              archivo_zip = os.path.join(
49                  self.directorio_destino,
50                  f"{nombre_base}_{timestamp}.zip"
51              )
52
53              # Se comprime el archivo
```

```
54              with zipfile.ZipFile(archivo_zip, 'w',
    zipfile.ZIP_DEFLATED) as zf:
55                  zf.write(archivo_origen, nombre_base)
56                  # Se agrega el checksum al archivo
    ZIP
57                  zf.writestr('checksum.txt',
    checksum_original)
58
59              if
    self.verificar_integridad(archivo_zip):
60                  logging.info(f"Archivo comprimido
    exitosamente: {archivo_zip}")
61                  return archivo_zip
62              else:
63                  os.remove(archivo_zip)  # Se elimina
    el archivo corrupto
64                  logging.error("Fallo en la
    verificación de integridad")
65                  return None
66
67          except Exception as e:
68              logging.error(f"Error al comprimir
    archivo: {e}")
69              return None
70
71      def verificar_integridad(self, archivo_zip: str)
    -> bool:
72          """
73          Se verifica la integridad del archivo
    comprimido
74          """
75          try:
76              with zipfile.ZipFile(archivo_zip, 'r')
    as zf:
77                  # Se obtiene el checksum original
78                  checksum_original =
    zf.read('checksum.txt').decode('utf-8')
79
80                  # Se extrae el archivo temporalmente
    para verificar
81                  archivo_original = None
82                  for nombre in zf.namelist():
83                      if nombre != 'checksum.txt':
84                          archivo_original = nombre
85                          zf.extract(nombre,
    path=self.directorio_destino)
86
87                  if archivo_original:
88                      archivo_temp =
    os.path.join(self.directorio_destino,
```

```
        archivo_original)
89                      checksum_actual =
        self._calcular_checksum(archivo_temp)
90                      os.remove(archivo_temp)  # Se
        limpia el archivo temporal
91                      return checksum_original ==
        checksum_actual
92
93              return False
94
95          except Exception as e:
96              logging.error(f"Error al verificar
        integridad: {e}")
97              return False
98
99  class MonitorDirectorioSeguro:
100     """
101     Se implementa el sistema de monitoreo y
        compresión segura sin watchdog
102     """
103     def __init__(self, directorio_observado: str,
        directorio_destino: str):
104         self.directorio_observado =
        directorio_observado
105         self.archivador =
        ArchivadorSeguro(directorio_destino)
106         self.archivos_conocidos: Set[str] = set()
107         self.activo = True
108
109         # Se configura el logging
110         logging.basicConfig(
111             level=logging.INFO,
112             format='%(asctime)s - %(levelname)s -
        %(message)s'
113         )
114
115         # Se asegura que los directorios existan
116         if not os.path.exists(directorio_observado):
117             os.makedirs(directorio_observado)
118
119     def _obtener_archivos_actuales(self) -> Set[str]:
120         """
121         Se obtiene el conjunto de archivos actuales
        en el directorio
122         """
123         try:
124             archivos = set()
125             for archivo in
        os.listdir(self.directorio_observado):
126                 ruta_completa =
```

```
        os.path.join(self.directorio_observado, archivo)
127                 if os.path.isfile(ruta_completa):
128                     archivos.add(ruta_completa)
129             return archivos
130         except Exception as e:
131             logging.error(f"Error al leer
        directorio: {e}")
132             return set()
133
134     def _procesar_archivo_nuevo(self, archivo: str)
        -> None:
135         """
136         Se procesa un archivo nuevo detectado en el
        directorio
137         """
138         try:
139             # Se espera un momento para asegurar que
        el archivo esté completo
140             time.sleep(1)
141
142             if os.path.exists(archivo):
143                 archivo_zip =
        self.archivador.comprimir_archivo(archivo)
144                 if archivo_zip:
145                     logging.info(f"Archivo procesado
        exitosamente: {archivo}")
146                 else:
147                     logging.error(f"Error al
        procesar archivo: {archivo}")
148
149         except Exception as e:
150             logging.error(f"Error al procesar
        archivo nuevo: {e}")
151
152     def monitorear(self, intervalo: int = 5) -> None:
153         """
154         Se inicia el monitoreo del directorio
155         """
156         logging.info(f"Iniciando monitoreo del
        directorio: {self.directorio_observado}")
157
158         # Se obtiene el estado inicial del directorio
159         self.archivos_conocidos =
        self._obtener_archivos_actuales()
160
161         try:
162             while self.activo:
163                 # Se obtienen los archivos actuales
164                 archivos_actuales =
        self._obtener_archivos_actuales()
```

```
165
166                     # Se identifican archivos nuevos
167                     archivos_nuevos = archivos_actuales
    - self.archivos_conocidos
168
169                     # Se procesan los archivos nuevos
170                     for archivo in archivos_nuevos:
171
    self._procesar_archivo_nuevo(archivo)
172
173                     # Se actualiza el conjunto de
    archivos conocidos
174                     self.archivos_conocidos =
    archivos_actuales
175
176                     # Se espera antes de la siguiente
    verificación
177                     time.sleep(intervalo)
178
179             except KeyboardInterrupt:
180                 logging.info("Monitoreo interrumpido por
    el usuario")
181             except Exception as e:
182                 logging.error(f"Error en el monitoreo:
    {e}")
183             finally:
184                 self.detener()
185
186         def detener(self) -> None:
187             """
188             Se detiene el sistema de monitoreo
189             """
190             self.activo = False
191             logging.info("Sistema de monitoreo detenido")
192
193 # Ejemplo de uso del sistema
194 if __name__ == "__main__":
195     monitor = MonitorDirectorioSeguro(
196
    directorio_observado="directorio_a_monitorear",
197         directorio_destino="archivos_comprimidos"
198     )
199     # Se inicia el monitoreo con verificación cada 5
    segundos
200     monitor.monitorear(intervalo=5)
```

Respuesta 28: (validador_datos_logs.py) Se implementa un sistema que combina el ValidadorCSV con RegistroLogger para procesar múltiples archivos CSV y mantener un registro detallado de la validación. El sistema maneja

la validación de tipos de datos y estructura, registrando todos los errores encontrados.

```python
# validador_datos_logs.py
from validador_csv import ValidadorCSV, ErrorCSV
from registro_logs import RegistroLogger
from pathlib import Path
from typing import Dict, List, Tuple
import time

class ValidadorDatosLogs:
    """
    Sistema que valida múltiples archivos CSV y registra los resultados.
    """
    def __init__(self, tipos_columnas: Dict[str, type], archivo_log: str):
        """
        Inicializa el validador con logging.

        Args:
            tipos_columnas (Dict[str, type]): Tipos esperados por columna
            archivo_log (str): Ruta del archivo de log
        """
        self.validador = ValidadorCSV(tipos_columnas)
        self.logger = RegistroLogger(archivo_log)

    def validar_directorio(self, directorio: str) -> Tuple[List[str], List[Tuple[str, str]]]:
        """
        Valida todos los archivos CSV en un directorio.

        Args:
            directorio (str): Directorio a procesar

        Returns:
            Tuple[List[str], List[Tuple[str, str]]]: Archivos válidos y errores
        """
        dir_path = Path(directorio)
        archivos_validos = []
        errores = []

        # Buscar todos los archivos CSV
        archivos_csv = list(dir_path.glob("*.csv"))
        total_archivos = len(archivos_csv)

```

```
41          self.logger.registrar_mensaje(
42              f"Iniciando validación de
    {total_archivos} archivos CSV en {directorio}",
43              "INFO"
44          )
45
46          for i, archivo in enumerate(archivos_csv, 1):
47              try:
48                  self.logger.registrar_mensaje(
49                      f"Validando archivo
    {i}/{total_archivos}: {archivo.name}",
50                      "INFO"
51                  )
52
53                  # Intentar validar el archivo
54
    self.validador.validar_archivo(str(archivo))
55                  archivos_validos.append(archivo.name)
56
57                  self.logger.registrar_mensaje(
58                      f"Archivo {archivo.name}
    validado correctamente",
59                      "INFO"
60                  )
61
62              except ErrorCSV as e:
63                  mensaje_error = f"Error en
    {archivo.name}: {str(e)}"
64                  errores.append((archivo.name,
    str(e)))
65
    self.logger.registrar_mensaje(mensaje_error,
    "ERROR")
66
67              except Exception as e:
68                  mensaje_error = f"Error inesperado
    en {archivo.name}: {str(e)}"
69                  errores.append((archivo.name,
    str(e)))
70
    self.logger.registrar_mensaje(mensaje_error,
    "CRITICAL")
71
72          # Registrar resumen
73          self.logger.registrar_mensaje(
74              f"Validación completada.
    {len(archivos_validos)} archivos válidos, "
75              f"{len(errores)} con errores",
76              "INFO"
77          )
```

```
78
79          return archivos_validos, errores
80
81  def ejemplo_uso():
82      """Ejemplo de uso del validador con logs"""
83      # Definir tipos esperados para las columnas
84      tipos_columnas = {
85          "id": int,
86          "nombre": str,
87          "edad": int,
88          "salario": float,
89          "activo": bool
90      }
91
92      # Crear instancia del validador
93      validador = ValidadorDatosLogs(
94          tipos_columnas=tipos_columnas,
95          archivo_log="validacion_datos.log"
96      )
97
98      # Validar archivos en un directorio
99      print("Iniciando validación de archivos CSV...")
100     validos, errores =
        validador.validar_directorio("datos/empleados")
101
102     # Mostrar resultados
103     print("\nArchivos válidos:")
104     for archivo in validos:
105         print(f"OK {archivo}")
106
107     if errores:
108         print("\nErrores encontrados:")
109         for archivo, error in errores:
110             print(f"Error en archivo: {archivo}:
        {error}")
111
112     print("\nConsulte validacion_datos.log para más
        detalles.")
113
114 if __name__ == "__main__":
115     ejemplo_uso()
```

Respuesta 29: (sistema_respaldo_simple.py) Se implementa un sistema básico que combina las clases RespaldoArchivo y RegistroLogger para crear una solución de respaldo sencilla. El sistema verifica cambios en el archivo monitoreando su tamaño y crea respaldos cuando detecta modificaciones. Cada operación de respaldo se registra en un archivo de log.

La implementación utiliza conceptos básicos como: - Manejo de archivos y directorios - Verificación de cambios mediante tamaño de archivo - Registro

de operaciones en un log - Manejo básico de excepciones

El ejemplo de uso muestra cómo configurar el sistema para un archivo específico y ejecutar un ciclo simple de monitoreo.

```
from respaldo_automatico import RespaldoArchivo
from registro_logs import RegistroLogger
import os
import time

class SistemaRespaldoSimple:
    """
    Sistema simple para respaldar un archivo y registrar la actividad.
    """
    def __init__(self, archivo_origen: str, directorio_respaldo: str):
        """
        Inicializa el sistema de respaldo.

        Args:
            archivo_origen (str): Archivo que se va a respaldar
            directorio_respaldo (str): Directorio donde se guardarán los respaldos
        """
        self.archivo_origen = archivo_origen
        self.respaldador = RespaldoArchivo(directorio_respaldo)
        self.logger = RegistroLogger("respaldo_simple.log")
        self.ultimo_tamano = self._obtener_tamano_archivo()

    def _obtener_tamano_archivo(self) -> int:
        """
        Obtiene el tamaño actual del archivo.

        Returns:
            int: Tamaño del archivo en bytes
        """
        try:
            return os.path.getsize(self.archivo_origen)
        except OSError:
            return 0

    def archivo_modificado(self) -> bool:
        """
        Verifica si el archivo ha sido modificado comparando su tamaño.
```

```
38
39          Returns:
40              bool: True si el archivo cambió de tamaño
41          """
42          tamano_actual =
        self._obtener_tamano_archivo()
43          return tamano_actual != self.ultimo_tamano
44
45      def crear_respaldo(self):
46          """Crea un respaldo del archivo y lo
        registra en el log."""
47          try:
48              # Crear el respaldo
49              ruta_respaldo, _ =
        self.respaldador.crear_respaldo(self.archivo_origen)
50
51              # Registrar la operación exitosa
52              self.logger.registrar_mensaje(
53                  f"Respaldo creado exitosamente en:
        {ruta_respaldo}",
54                  "INFO"
55              )
56
57              # Actualizar el tamaño registrado
58              self.ultimo_tamano =
        self._obtener_tamano_archivo()
59
60          except Exception as e:
61              # Registrar si ocurre algún error
62              self.logger.registrar_mensaje(
63                  f"Error al crear respaldo: {str(e)}",
64                  "ERROR"
65              )
66
67  def ejemplo_uso():
68      """Ejemplo de uso del sistema de respaldo
        simple"""
69      # Crear una instancia del sistema
70      sistema = SistemaRespaldoSimple(
71          archivo_origen="datos/documento.txt",
72          directorio_respaldo="respaldos"
73      )
74
75      # Simular un programa que revisa cambios cada 5
        segundos
76      print("Sistema de respaldo iniciado. Presione
        Ctrl+C para detener.")
77
78      try:
79          while True:
```

```
80                if sistema.archivo_modificado():
81                    print("Archivo modificado, creando
      respaldo...")
82                    sistema.crear_respaldo()
83                time.sleep(5)  # Esperar 5 segundos
      antes de la siguiente revisión
84
85            except KeyboardInterrupt:
86                print("\nSistema de respaldo detenido.")
87
88    if __name__ == "__main__":
89        ejemplo_uso()
```

1.6 Próximamente: Volumen 2 - Introducción a la IA y Aprendizaje Automático

Lo que nos espera en el próximo volumen

Tras dominar los fundamentos de la programación con Python, el Volumen 2 nos introducirá en el fascinante mundo de la Inteligencia Artificial y el Aprendizaje Automático, donde se aplicarán las habilidades de programación adquiridas para desarrollar sistemas inteligentes.

En el segundo volumen de esta serie, se exploran los conceptos y técnicas fundamentales que transforman el código en sistemas capaces de aprender y adaptarse. El contenido está diseñado para construir sobre las bases de programación establecidas en este primer volumen, aplicando las estructuras de datos, funciones y manejo de archivos en contextos prácticos de IA.

Contenido destacado del Volumen 2

Fundamentos de Machine Learning Se explorarán los diferentes tipos de aprendizaje automático, comprendiendo cómo los datos se transforman en conocimiento a través de algoritmos. Se analizará el proceso completo desde la recolección de datos hasta la evaluación de modelos, haciendo especial énfasis en la preparación y limpieza de datos.

Algoritmos esenciales Se estudiarán en profundidad los modelos de regresión y clasificación, implementando desde cero algoritmos fundamentales que constituyen la base de sistemas más complejos. Las técnicas de regularización y evaluación de modelos se abordarán con ejemplos prácticos.

Aplicaciones por especialidad Cada concepto teórico se complementará con implementaciones específicas para diferentes campos:

- **TIC:** Sistemas predictivos para análisis de datos de red
- **Electrónica:** Procesamiento inteligente de señales de sensores
- **Energía:** Optimización de consumo energético
- **Química:** Control de procesos industriales

Conexión con el Volumen 1

Los conceptos de funciones, estructuras de datos y manejo de archivos presentados en el Volumen 1 son fundamentales para implementar algoritmos de aprendizaje automático. Por ejemplo, las funciones recursivas estudiadas serán aplicadas en algoritmos de agrupamiento, mientras que el manejo eficiente de archivos CSV resultará esencial para el procesamiento de conjuntos de datos.

¿Por qué continuar con el Volumen 2?

El segundo volumen transforma el conocimiento de programación en capacidades prácticas para crear sistemas inteligentes. A través de ejemplos progresivamente más complejos, se desarrollará la intuición necesaria para seleccionar, implementar y evaluar algoritmos de IA adecuados para diferentes problemas.

Los ejercicios propuestos permitirán consolidar conocimientos mediante la implementación de:

- Modelos predictivos con datos reales
- Sistemas de clasificación para toma de decisiones
- Técnicas de validación y optimización de modelos
- Aplicaciones específicas para cada especialidad

⚠ Preparación necesaria

Para aprovechar al máximo el contenido del Volumen 2, se recomienda dominar los conceptos presentados en este primer volumen, especialmente las secciones sobre manejo de datos, funciones y estructuras de control.

El Volumen 2 estará disponible próximamente, continuando

nuestro viaje desde los fundamentos de programación hacia el desarrollo de sistemas de inteligencia artificial.