PYTHON PARA DESARROLLADORES: UN LENGUAJE DE ALTO NIVEL

Book · January 2023

CITATIONS
0

READS
149

1 author:

Carlos Polanco
National Institute of Cardiology
356 PUBLICATIONS **664** CITATIONS

SEE PROFILE

PYTHON PARA DESARROLLADORES

UN LENGUAJE DE ALTO NIVEL

Carlos Polanco

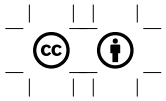
PYTHON PARA DESARROLLADORES

UN LENGUAJE DE ALTO NIVEL

Carlos Polanco

28 de noviembre de 2023

Derechos de Autor



Word processor: LaTeX ©2023; Operating system: Linux Fedora-39 ©2023.

Software: QuillBot (Course Hero), LLC. ©2023, & `ChatGPT` 4.0 OpenAI ©2023.

Nota al Lector

Este libro introduce la programación en Python mediante casos prácticos enfocados en Inteligencia Artificial, utilizando `ChatGPT` [2] para una comprensión más profunda de la programación en este lenguaje.

Se asume que el lector posee conocimientos previos y experiencia práctica en programación con GNU Linux Fedora, ya que la codificación de los programas Python se realizará en esta plataforma, utilizando la terminal de Linux Fedora.

El libro está enriquecido con hipervínculos que facilitan la consulta de las distintas secciones del índice de contenidos. Además, incluye accesos directos a programas ejemplificados para una mejor comprensión de los conceptos, complementados con videos ilustrativos.

El curso, que se llevará a cabo de junio a octubre de 2024, utilizará este libro como material de texto principal. Los participantes que completen todas las tutorías impartidas por el Dr. Brayans Becerra recibirán una constancia oficial. Para consultas, contactar al Dr. Becerra en el siguiente correo electrónico: `pacorro28144@hotmail.com`.

El autor desea expresar su agradecimiento al Instituto de Ciencias Nucleares y a la Facultad de Ciencias de la Universidad Nacional Autónoma de México, así como al Instituto Nacional de Cardiología “Ignacio Chávez”, por su apoyo en la elaboración de este libro.

CONFLICTO DE INTERÉS

El autor declara que no existe ningún conflicto de interés respecto al contenido de este libro.

Carlos Polanco

Correo: `polanco@unam.mx`
Departamento de Nuevas Tecnologías y Protección Intelectual
Instituto Nacional de Cardiología “Ignacio Chávez”
México

Departamento de Matemáticas, Facultad de Ciencias
Universidad Nacional Autónoma de México
México

Agradecimientos

Quiero agradecer a la Maestra en Ciencias Martha Rios Castro por la revisión de este libro, lo cual imprimió claridad a las secciones, al Doctor en Ciencias Brayans Becerra Luna por estar a cargo de las tutorías, y a la Licenciada Maritza Rosas por el diseño de la portada.

Índice general

Casos Prácticos

Capítulo 1

Introducción

La enseñanza y el aprendizaje de Python en el contexto de la inteligencia artificial (IA) se potencian enormemente cuando se enriquecen con casos de estudio prácticos. Este curso está diseñado para proporcionar una comprensión profunda de Python no solo como un lenguaje de programación, sino también como una herramienta esencial en el desarrollo de soluciones de IA. A través de una serie de casos de estudio cuidadosamente seleccionados, los estudiantes tendrán la oportunidad de aplicar teorías y conceptos en situaciones del mundo real, abordando problemas y desafíos típicos en el campo de la IA.

El enfoque basado en casos de estudio permite una experiencia de aprendizaje más interactiva y aplicada. Cada caso de estudio ha sido diseñado para ilustrar y reforzar los conceptos de Python y de inteligencia artificial. Los estudiantes no solo aprenderán la sintaxis y las estructuras de Python, sino que también verán cómo se pueden utilizar estas herramientas para construir modelos de aprendizaje automático, procesar y analizar datos, y desarrollar sistemas inteligentes que pueden aprender y adaptarse.

A lo largo del curso, exploraremos una variedad de aplicaciones de IA, desde el análisis de datos hasta el aprendizaje profundo, pasando por el procesamiento del lenguaje natural y las redes neuronales. Cada módulo combina teoría con práctica, asegurando que los estudiantes obtengan una comprensión práctica de cómo se puede utilizar Python para resolver problemas complejos en el mundo real. Al final del curso, los estudiantes no solo serán competentes en programación en Python, sino que también tendrán una sólida comprensión de cómo aplicar estas habilidades en el emocionante y en constante evolución campo de la inteligencia artificial.

1.1. sUn Lenguaje con Enfoque Matemático-Computacional

Python se ha establecido como un lenguaje de programación predominante en el mundo de la ciencia computacional y la matemática aplicada. Su diseño, que enfatiza la legibilidad y la simplicidad, lo convierte en una herramienta ideal para profesionales y académicos que se adentran en la programación computacional. La sintaxis clara y concisa de Python permite a los usuarios centrarse en los conceptos matemáticos y algorítmicos subyacentes, minimizando la distracción causada por la complejidad del lenguaje de programación.

Desde el punto de vista matemático, Python ofrece una extensa biblioteca de herramientas que facilitan la implementación de complejos cálculos numéricos y algoritmos. Con bibliotecas como `NumPy` y `SciPy`, Python proporciona una plataforma robusta para operaciones matemáticas que van desde el álgebra lineal básica hasta la optimización y la resolución de ecuaciones diferenciales. Estas herramientas no solo son poderosas sino también accesibles, lo que permite a los científicos e ingenieros aplicar conceptos matemáticos a problemas prácticos de manera eficiente.

En el ámbito de la computación, Python destaca por su flexibilidad y su amplio rango de aplicaciones. Desde la escritura de programas simples hasta el desarrollo de sistemas complejos, Python se adapta a una variedad de necesidades de programación. Su capacidad para integrarse con otros lenguajes y herramientas, como C y Fortran, lo convierte en una elección ideal para proyectos que requieren un alto rendimiento computacional. Además, la amplia comunidad de Python contribuye activamente al desarrollo y mejora de una gran cantidad

de bibliotecas y frameworks, asegurando que el lenguaje siga siendo relevante y efectivo para las necesidades computacionales modernas.

1.2. Enfoque y Ventajas

Python se ha establecido como uno de los lenguajes de programación más versátiles y demandados en el mundo moderno. Su simplicidad y legibilidad lo hacen ideal para principiantes, mientras que su poderosa colección de bibliotecas y frameworks lo convierten en una elección preferida para los desarrolladores web y de software.

En el ámbito de la programación web, Python se destaca con frameworks como Django y Flask, que facilitan el desarrollo de aplicaciones web robustas y escalables. Django, conocido por su 'baterías incluidas' enfoque, proporciona herramientas integradas para casi todas las necesidades del desarrollo web, mientras que Flask ofrece flexibilidad y ligereza para proyectos más pequeños o personalizados.

Además, Python juega un papel crucial en el desarrollo de tecnologías modernas como la Inteligencia Artificial (IA), el Aprendizaje Automático (Machine Learning), y la Ciencia de Datos. Bibliotecas como TensorFlow, PyTorch, y Pandas, han hecho que Python sea indispensable para los profesionales que trabajan en estos campos emergentes.

La combinación de versatilidad, facilidad de uso y la fuerte comunidad de soporte, continúa asegurando que Python sea un pilar fundamental en el desarrollo tecnológico contemporáneo.

En el campo de la inteligencia artificial, Python se ha convertido en el lenguaje de programación preferido por varias razones. Su simplicidad y facilidad de aprendizaje lo hacen accesible para profesionales de diferentes disciplinas, permitiendo que un público más amplio participe en el desarrollo de soluciones de IA. Esta accesibilidad es crucial, ya que la IA es un campo interdisciplinario que beneficia de la integración de conocimientos provenientes de la matemática, la estadística, la ciencia de datos, la ingeniería, y más.

Python también es elogiado por su extenso ecosistema de bibliotecas de IA y aprendizaje automático. Herramientas como `Keras`, y `PyTorch` ofrecen capacidades avanzadas para la construcción y entrenamiento de modelos de redes neuronales, mientras que bibliotecas como `Scikit-learn` brindan implementaciones eficientes de una variedad de algoritmos de aprendizaje automático. Estas bibliotecas no solo son potentes sino también bien documentadas y respaldadas por una comunidad activa, lo que facilita su adopción y uso en proyectos de IA.

Además, la versatilidad de Python para manejar y procesar grandes conjuntos de datos es fundamental en la IA. La capacidad de integrarse con sistemas de bases de datos y realizar análisis de datos complejos con `Pandas`, por ejemplo, hace de Python una herramienta completa para el flujo de trabajo en proyectos de IA. Desde la recopilación y limpieza de datos hasta el modelado y la visualización, Python ofrece soluciones coherentes y eficientes.

Python no solo facilita la implementación técnica de algoritmos y modelos de IA, sino que también sirve como un puente que conecta las ideas matemáticas y computacionales con aplicaciones prácticas en la inteligencia artificial. Esta combinación de accesibilidad, poder, y flexibilidad es lo que establece a Python como un pilar fundamental en el desarrollo actual y futuro de la IA.

Capítulo 2

Fundamentos de Programación

Python se destaca en el mundo de la programación por su enfoque en la legibilidad y eficiencia del código. La filosofía de diseño del lenguaje enfatiza la importancia de un código que sea fácil de leer y escribir, lo que no solo facilita el aprendizaje para los principiantes sino que también permite a los desarrolladores experimentados implementar soluciones complejas de manera eficiente. En esta sección, nos sumergiremos en las características que hacen de Python un lenguaje sencillo y accesible, y cómo estas características lo han convertido en una herramienta valiosa para programadores de todos los niveles.

2.1. Conceptos Básicos

Este documento presenta la documentación del programa `hola_mundo.py`, escrito en Python. Este programa es un ejemplo clásico de introducción a la programación, mostrando la simplicidad y la sintaxis básica del lenguaje Python.

El código fuente del programa `hola_mundo.py` se muestra a continuación:

```
#!/usr/bin/env python3
# hola_mundo.py
# Este programa imprime "Hola Mundo" en la pantalla.

print("Hola Mundo")
```

Para escribir este programa, debe abrirse una terminal en un sistema operativo Linux Fedora 39 y seguir los siguientes pasos:

(1) Abrir el editor vi con el comando `vi hola_mundo.py`.

(2) Ingresar al modo de inserción presionando `i`.

(3) Escribir el código del programa.

(4) Salir del modo de inserción con `Esc`, guardar con `:w` y salir con `:q`.

Sobre el programa Python

```
#!/usr/bin/env python3
```

Esta línea es conocida como `shebang` y se utiliza en scripts ejecutables en sistemas Unix y Linux. Indica al sistema que el script debe ser ejecutado con el intérprete de Python 3. `env` es utilizado para encontrar el intérprete de Python en el entorno del usuario, lo que hace que el script sea más portátil.

Los comentarios son líneas en el código que no son ejecutadas por el intérprete. En Python, los comentarios comienzan con #. Este comentario describe el propósito del script:

```
# Este programa imprime "Hola Mundo" en la pantalla.
```

```
print("Hola Mundo")
```

Esta es la única línea de código que se ejecuta en el programa. La función `print()` es una función incorporada en Python que imprime el argumento dado en la pantalla. En este caso, imprime el string "Hola Mundo".

Para ejecutar este programa:

(1) Haga el script ejecutable con el comando: `chmod +x hola_mundo.py`.

(2) Ejecute el script con: `./hola_mundo.py`.

Al ejecutar este programa, se espera que imprima el siguiente mensaje en la terminal:

```
Hola Mundo
```

El programa `hola_mundo.py` es un ejemplo básico de un programa Python que demuestra la impresión de texto en la pantalla, sirviendo como una introducción a los conceptos fundamentales de la programación en Python.

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

Práctica 1.

Es conveniente que el lector pida a `ChatGPT` otros ejemplos que incluyan comentarios largos al código.

2.2. Tipos de datos y variables

Python es un lenguaje de programación de alto nivel que soporta una amplia variedad de tipos de datos. Entender estos tipos de datos y cómo se manejan las variables es fundamental para la programación en Python.

A continuación se presenta una tabla resumen de los tipos de datos en Python:

Tipo de Dato	Descripción	Ejemplo
`int`	Número entero	`5`
`float`	Número con coma flotante	`5.0`
`str`	Cadena de caracteres	`"Hola"`
`bool`	Valor booleano (Verdadero o Falso)	`True`
`list`	Colección ordenada y modificable	`[1, 2, 3]`
`tuple`	Colección ordenada e inmutable	`(1, 2, 3)`
`set`	Colección no ordenada y sin elementos duplicados	`{1, 2, 3}`
`dict`	Colección de pares clave-valor	`{."a": 1, "b": 2}`

Cuadro 2.1: Resumen de Tipos de Datos en Python

Los tipos de datos y las variables en Python son fundamentales para el manejo de la información en la programación. Una comprensión clara de estos conceptos es esencial para cualquier programador de Python.

Ejemplo 1.

```python
#!/usr/bin/env python3

# Programa Python para ilustrar diferentes tipos de datos y variables

# Entero (int)
entero = 10
print("Entero:", entero)

# Flotante (float)
flotante = 10.5
print("Flotante:", flotante)

# Cadena de texto (str)
cadena = "Hola Mundo"
print("Cadena:", cadena)

# Booleano (bool)
verdadero = True
falso = False
print("Booleano Verdadero:", verdadero, "y Falso:", falso)

# Lista (list)
lista = [1, 2, 3, "cuatro", 5.0]
print("Lista:", lista)

# Tupla (tuple)
tupla = (1, 2, 3, "cuatro", 5.0)
print("Tupla:", tupla)

# Conjunto (set)
conjunto = {1, 2, 3, 4, 5}
print("Conjunto:", conjunto)

# Diccionario (dict)
diccionario = {"uno": 1, "dos": 2, "tres": 3}
print("Diccionario:", diccionario)
```

```
chmod +x tipo_datos.py
./tipo_datos.py

Entero: 10
Flotante: 10.5
Cadena: Hola Mundo
Booleano Verdadero: True y Falso: False
Lista: [1, 2, 3, 'cuatro', 5.0]
Tupla: (1, 2, 3, 'cuatro', 5.0)
Conjunto: {1, 2, 3, 4, 5}
Diccionario: {'uno': 1, 'dos': 2, 'tres': 3}
```

El lector puede bajar este programa desde el siguiente vínculo:

Práctica 2.

Es conveniente que el lector pida a `ChatGPT` otros ejemplos que incluyan variantes a los resultados de este programa.

2.3. Operadores y Expresiones

Este documento proporciona una descripción detallada de los operadores y expresiones en Python, incluyendo su sintaxis y uso común. Python soporta una variedad de operadores, que pueden ser clasificados en varios tipos como aritméticos, de comparación, lógicos, entre otros.

Los operadores aritméticos se utilizan para realizar operaciones matemáticas básicas.

Operador	Descripción	Ejemplo
+	Suma	a + b
-	Resta	a - b
*	Multiplicación	a * b
/	División	a / b
%	Módulo	a% b
**	Exponente	a ** b
//	División entera	a // b

Cuadro 2.2: Resumen de los Operadores Aritméticos en Python

Se utilizan para comparar valores. El resultado es un booleano.

Operador	Descripción	Ejemplo
==	Igual a	a == b
!=	No igual a	a != b
>	Mayor que	a >b
<	Menor que	a <b
>=	Mayor o igual que	a >= b
<=	Menor o igual que	a <= b

Cuadro 2.3: Resumen de los Operadores Lógicos en Python

Se usan para combinar declaraciones condicionales.

Operador	Descripción	Ejemplo
and	Verdadero si ambos operandos son verdaderos	a and b
or	Verdadero si al menos un operando es verdadero	a or b
not	Invierte el estado lógico del operando	not a

Cuadro 2.4: Resumen de los Operadores Condicionales en Python

Ejemplo 2.

```
#!/usr/bin/env python3

# Ejemplo de Operadores en Python

# Operadores Aritméticos
suma = 5 + 3          # Suma
resta = 5 - 3         # Resta
multiplicacion = 5 * 3  # Multiplicación
division = 10 / 2     # División
modulo = 10 % 3       # Módulo
exponente = 2 ** 3    # Exponente
division_entera = 10 // 3 # División Entera

# Operadores de Comparación
igual = (5 == 3)      # Igual a
no_igual = (5 != 3)   # No igual a
mayor_que = (5 > 3)   # Mayor que
menor_que = (5 < 3)   # Menor que
mayor_o_igual = (5 >= 3) # Mayor o igual que
menor_o_igual = (5 <= 3) # Menor o igual que

# Operadores Lógicos
y = (True and False) # Operador AND
```

```
o = (True or False)  # Operador OR
no = (not True)      # Operador NOT

# Operadores de Asignación
a = 10
a += 5  # Equivalente a a = a + 5
a -= 5  # Equivalente a a = a - 5
a *= 2  # Equivalente a a = a * 2
a /= 2  # Equivalente a a = a / 2

# Operadores de Identidad
a = [1, 2, 3]
b = [1, 2, 3]
c = a

es_mismo_objeto = (a is c)      # True porque a y c son el mismo objeto
no_es_mismo_objeto = (a is not b) # True porque a y b no son el mismo objeto

# Operadores de Pertenencia
lista = [1, 2, 3, 4, 5]
contiene = 3 in lista           # True porque 3 está en lista
no_contiene = 6 not in lista    # True porque 6 no está en lista

# Imprimir resultados
print("Suma:", suma)
print("Resta:", resta)
print("Multiplicación:", multiplicacion)
print("División:", division)
print("Módulo:", modulo)
print("Exponente:", exponente)
print("División Entera:", division_entera)
print("Igual:", igual)
print("No Igual:", no_igual)
print("Mayor Que:", mayor_que)
print("Menor Que:", menor_que)
print("Mayor o Igual Que:", mayor_o_igual)
print("Menor o Igual Que:", menor_o_igual)
print("Operador AND:", y)
print("Operador OR:", o)
print("Operador NOT:", no)
print("Valor de a después de operaciones de asignación:", a)
print("Es el mismo objeto:", es_mismo_objeto)
print("No es el mismo objeto:", no_es_mismo_objeto)
print("Contiene 3:", contiene)
print("No contiene 6:", no_contiene)
```

```
chmod +x operadores.py
./operadores.py

Suma: 8
Resta: 2
Multiplicación: 15
División: 5.0
Módulo: 1
Exponente: 8
División Entera: 3
Igual: False
No Igual: True
Mayor Que: True
Menor Que: False
Mayor o Igual Que: True
Menor o Igual Que: False
Operador AND: False
Operador OR: True
Operador NOT: False
Valor de a después de operaciones de asignación: [1, 2, 3]
Es el mismo objeto: True
No es el mismo objeto: True
```

```
Contiene 3: True
No contiene 6: True
```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

Práctica 3.

Es conveniente que el lector pida a `ChatGPT` otros ejemplos que incluyan variantes a este programa.

2.4. Estructuras de Control

Las estructuras de control en Python permiten dirigir el flujo de ejecución del programa.

(1) Los *bucles* permiten repetir una acción múltiples veces. Python ofrece principalmente dos tipos: `for` y `while`.

   (a) El bucle `for` se utiliza para iterar sobre una secuencia (como una lista, una tupla, un diccionario, un conjunto o una cadena).

   (b) El bucle `while` se ejecuta mientras una condición especificada sea verdadera.

(2) Los *condicionales* permiten ejecutar diferentes acciones dependiendo de si una condición es verdadera o falsa. Las estructuras condicionales en Python se manejan con `if`, `elif`, y `else`.

Ejemplo 3.

```python
#!/usr/bin/env python3

# Programa Python para demostrar bucles, condicionales

# Bucle For
print("Ejemplo de Bucle For:")
for i in range(5):
    print(i, end=' ')
print("\n")

# Bucle While
print("Ejemplo de Bucle While:")
contador = 0
while contador < 5:
    print(contador, end=' ')
    contador += 1
print("\n")

# Condicionales
print("Ejemplo de Condicionales:")
numero = 4
if numero > 5:
    print("El número es mayor que 5")
elif numero < 5:
    print("El número es menor que 5")
else:
    print("El número es 5")
print("\n")
```

```
chmod +x estructuras_control.py
./estructuras_control.py
Ejemplo de Bucle For:
0 1 2 3 4

Ejemplo de Bucle While:
```

```
0 1 2 3 4

Ejemplo de Condicionales:
El número es menor que 5
```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

Práctica 4.

Es conveniente que el lector pida a `ChatGPT` variantes a este programa.

2.5. Funciones y Módulos

En Python, las funciones y los módulos son fundamentales para la organización y reutilización del código.

(1) **Definición y Uso de Funciones**

(a) Una *función* en Python es una agrupación de declaraciones relacionadas que realizan una tarea específica.

(b) Las funciones se definen usando la palabra clave `def`, seguida de un nombre de función, paréntesis y dos puntos.

(c) Las funciones pueden recibir argumentos y devolver un resultado.

(d) Ejemplo:

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

```

(e) Las funciones permiten reutilizar código, mejorar la claridad y facilitar el mantenimiento.

Ejemplo 4.

```python
# Programa Python para demostrar el uso de funciones

# Función para sumar dos números
def sumar(a, b):
    return a + b

# Función para calcular la media de una lista de números
def calcular_media(numeros):
    total = sum(numeros)
    cantidad = len(numeros)
    return total / cantidad

# Usando la función sumar
resultado_suma = sumar(5, 3)
print("La suma de 5 y 3 es:", resultado_suma)

# Usando la función calcular_media
lista_numeros = [10, 20, 30, 40, 50]
media = calcular_media(lista_numeros)
print("La media de", lista_numeros, "es:", media)
  \endlstlisting{}

  \begin{lstlisting}[language=bash]
chmod +x funciones.py
./funciones.py
La suma de 5 y 3 es: 8
La media de [10, 20, 30, 40, 50] es: 30.0

```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

Práctica 5.

Es conveniente que el lector pida a `ChatGPT` variantes a este programa, particularmente añadiendo dos o mas funciones.

Nota importante: La identación, o sangría, en Python no es solo una cuestión de estilo, sino una parte fundamental de la sintaxis del lenguaje. A continuación, se detalla su importancia y cómo se utiliza en la programación Python.

En Python, la identación se utiliza para delinear bloques de código. Esto es fundamental para la estructura del programa. Por ejemplo:

```
for i in range(5):
    print(i)
```

En este bucle `for`, la línea `print(i)` está indentada y forma parte del bucle.

Python no permite la mezcla de espacios y tabulaciones para la indentación. Es crucial mantener la consistencia. Por ejemplo, si se inicia un bloque con espacios, todo el bloque debe usar espacios.

Una identación incorrecta puede llevar a errores de sintaxis en Python, lo que resulta en `IndentationError`. Esto ocurre cuando las líneas de código no están correctamente alineadas a la misma indentación.

Se recomienda usar siempre cuatro espacios para cada nivel de identación, que es la convención en la comunidad Python.

2.6. Caso Práctico

Python soporta varios tipos de datos, como enteros (int), números de punto flotante (float), cadenas (str), y booleanos (bool). Las variables en Python no necesitan ser declaradas y su tipo se infiere en tiempo de ejecución.

```
numero = 10           # int
decimal = 3.14        # float
texto = "Hola Mundo" # str
verdadero = True      # bool

print(numero, decimal, texto, verdadero)
```

Los operadores en Python incluyen operadores aritméticos, de comparación y lógicos. Las expresiones usan estos operadores para evaluar valores o realizar cálculos.

```
suma = 5 + 3          # Aritmético
igual = (5 == 3)      # Comparación
y = (True and False) # Lógico

print(suma, igual, y)
```

Python utiliza estructuras de control como bucles y condicionales para dirigir el flujo de ejecución del programa. Ejemplo:

```
# Bucle For
for i in range(5):
    print(i)

# Condicional If
numero = 4
if numero > 5:
    print("Mayor que 5")
elif numero < 5:
    print("Menor que 5")
else:
    print("El número es 5")
```

Las funciones en Python son bloques de código reutilizables que realizan una tarea específica.

```python
# Definición de una función
def sumar(a, b):
    return a + b

# Uso de la función
resultado = sumar(5, 3)
print("La suma de 5 y 3 es:", resultado)
```

⚠ El lector puede bajar estos programas desde el siguiente vínculo: 🖵

Práctica 6.

Es conveniente que el lector ejecute cada uno de estos programas.

Capítulo 3

Listas

Las listas en Python son una de las estructuras de datos más versátiles y comúnmente utilizadas. Son colecciones ordenadas y mutables, lo que significa que los elementos dentro de una lista pueden ser cambiados después de su creación. Esto las distingue de las tuplas, que son colecciones inmutables. Las listas en Python se pueden identificar por estar encerradas entre corchetes `[]` y pueden contener elementos de diferentes tipos, incluyendo números, cadenas y otras listas.

El uso de las listas en Python es muy extenso. Algunos ejemplos comunes incluyen el almacenamiento de series de datos, la realización de operaciones de secuencia como iteraciones, y la manipulación de elementos a través de métodos como `append()`, `remove()`, `sort()`, entre otros. La habilidad de las listas para almacenar diferentes tipos de datos hace que sean particularmente útiles en una variedad de aplicaciones prácticas en programación.

Las principales ventajas de usar listas en Python son:

(1) **Flexibilidad**: Las listas pueden contener elementos de diferentes tipos, incluyendo otras listas, lo que las hace extremadamente flexibles.

(2) **Mutabilidad**: Los elementos de una lista pueden modificarse, lo que permite una manipulación dinámica de los datos.

(3) **Métodos Integrados**: Python proporciona una amplia gama de métodos integrados para realizar operaciones comunes en listas, como añadir, eliminar y ordenar elementos.

(4) **Slice y Acceso por Índice**: Las listas permiten el acceso y modificación de elementos específicos mediante índices, lo que facilita la gestión de subconjuntos de datos.

(5) **Iterabilidad**: Las listas son iterables, lo que significa que pueden ser usadas en bucles y comprehensions de listas para operaciones repetitivas y eficientes.

Las listas son una herramienta poderosa y flexible en Python, adecuadas para una amplia gama de aplicaciones en programación. Su facilidad de uso y la amplia gama de funcionalidades que ofrecen las hacen una opción popular entre los desarrolladores.

3.1. Listas Unidimensionales en Python

En Python, un lista unidimensional se refiere a una secuencia lineal de elementos. Aunque Python no tiene un tipo de datos específico para listas, la funcionalidad de los listas se logra a través del tipo de datos `list`. Una lista en Python es una colección ordenada y mutable que se puede utilizar para almacenar una serie de elementos.

Para crear un lista unidimensional en Python, se declaran los elementos dentro de corchetes `[]` separados por comas. Por ejemplo:

```
miLista = [10, 20, 30, 40, 50]
```

Esta línea de código crea un lista unidimensional llamado `miLista` que contiene cinco enteros.

Los listas unidimensionales en Python tienen las siguientes características:

(1) **Ordenados**: Los elementos en un lista mantienen el orden en que se añaden.

(2) **Accesibles por Índice**: Cada elemento en el lista se puede acceder mediante un índice. Por ejemplo, `miLista[0]` devuelve el primer elemento.

(3) **Mutables**: Los valores almacenados en el lista pueden modificarse.

(4) **Heterogéneos**: Un lista puede contener elementos de diferentes tipos, como enteros, cadenas y objetos.

Las operaciones básicas que se pueden realizar en listas unidimensionales incluyen:

(1) **Acceso a elementos**: Usando índices para obtener el valor de un elemento específico.

(2) **Modificación de elementos**: Cambiar el valor de uno o más elementos.

(3) **Adición de elementos**: Utilizar métodos como `append()` o `insert()` para añadir elementos.

(4) **Eliminación de elementos**: Utilizar métodos como `remove()` o `pop()` para eliminar elementos.

Aquí se muestra un ejemplo simple que ilustra la creación y manipulación de un lista unidimensional en Python:

```
# Creación del lista
miLista = [10, 20, 30, 40, 50]

# Acceso a elementos
print(miLista[0])  # Salida: 10
print(miLista[3])  # Salida: 40

# Modificación de elementos
miLista[2] = 35
print(miLista)  # Salida: [10, 20, 35, 40, 50]

# Adición de elementos
miLista.append(60)
print(miLista)  # Salida: [10, 20, 35, 40, 50, 60]

# Eliminación de elementos
miLista.remove(20)
print(miLista)  # Salida: [10, 35, 40, 50, 60]
```

3.2. Listas Bidimensionales en Python

Una lista bidimensional en Python es una lista de listas. Cada elemento de la lista principal puede ser otra lista que contiene múltiples elementos.

(1) Índices Positivos: Se accede a los elementos utilizando dos índices: `[i][j]`, donde `i` es el índice de la sublista, y `j` es el índice del elemento dentro de esa sublista.

(2) Índices Negativos: Funcionan similar a los índices positivos, pero comienzan desde el final.

```
# Añadir un elemento
matriz[0].append(10)
# Borrar un elemento
del matriz[1][2]
```

```
# Imprimir toda la lista
print(matriz)
# Imprimir elementos específicos
print(matriz[0][0], matriz[1][1])
```

```
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
# Acceso al primer elemento de la primera sublista
print(matriz[0][0])  # Salida: 1
# Acceso al último elemento de la última sublista
print(matriz[-1][-1]) # Salida: 9
```

3.3. Listas Tridimensionales en Python

Una lista tridimensional es una lista de listas de listas. Se puede pensar como un cubo, donde cada elemento es accesible mediante tres índices.

Similar a las listas bidimensionales, pero con un índice adicional.

Añadir y Borrar Elementos.

```
# Añadir un subgrupo
cubo[0].append([9, 10])
# Borrar un subgrupo
del cubo[1][1]
```

```
# Imprimir toda la lista
print(cubo)
# Imprimir elementos específicos
print(cubo[0][0][0], cubo[1][0][1])
```

```
cubo = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
# Acceso al primer elemento del primer subgrupo de la primera sublista
print(cubo[0][0][0])  # Salida: 1
# Acceso al último elemento del último subgrupo de la última sublista
print(cubo[-1][-1][-1]) # Salida: 8
```

3.4. Caso Práctico

El siguiente programa es un sistema de gestión de inventario para una tienda de electrónica. Utiliza listas bidimensionales para almacenar información sobre los productos y listas unidimensionales para operaciones específicas.

```
#!/usr/bin/env python3

# Suponemos que inventario es una lista de tuplas (nombre, cantidad)
inventario = []

def anadir_inventario():
    print("\nAñadir elemento al inventario")
    print("Ejemplo: Para añadir un producto, ingresa el nombre y la cantidad.")
    print("Formato del ejemplo: 'Nombre del producto, Cantidad'")
    print("Por ejemplo: 'Lápices, 20'")

    entrada_usuario = input("Ingrese el nombre y la cantidad del producto separados por una
    coma: ")
    try:
        nombre, cantidad = entrada_usuario.split(',')
        cantidad = int(cantidad.strip())  # Convierte la cantidad a entero
```

```
16          inventario.append((nombre.strip(), cantidad))
17          print(f"Producto '{nombre.strip()}' con cantidad {cantidad} añadido al inventario.")
18      except ValueError:
19          print("Formato incorrecto. Por favor, siga el ejemplo.")
20
21  def borrar_inventario():
22      print("\nBorrar elemento del inventario")
23      listar_inventario()
24      if inventario:
25          try:
26              item_id = int(input("Ingrese el ID del producto a borrar: "))
27              if 0 <= item_id < len(inventario):
28                  elemento_borrado = inventario.pop(item_id)
29                  print(f"Producto eliminado: {elemento_borrado}")
30              else:
31                  print("ID no válido.")
32          except ValueError:
33              print("Por favor, ingrese un número válido.")
34
35  def listar_inventario():
36      print("\nListar elementos del inventario")
37      for idx, item in enumerate(inventario):
38          print(f"{idx}: {item}")
39      if not inventario:
40          print("El inventario está vacío.")
41
42  def mostrar_menu():
43      while True:
44          print("\nMenú de Inventario")
45          print("1. Añadir al inventario")
46          print("2. Borrar del inventario")
47          print("3. Listar inventario")
48          print("4. Salir")
49
50          opcion = input("Seleccione una opción: ")
51
52          if opcion == '1':
53              anadir_inventario()
54          elif opcion == '2':
55              borrar_inventario()
56          elif opcion == '3':
57              listar_inventario()
58          elif opcion == '4':
59              print("Saliendo del programa.")
60              break
61          else:
62              print("Opción no válida, intente de nuevo.")
63
64  # Llamada a la función para mostrar el menú
65  mostrar_menu()
```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

Práctica 7.

Es conveniente que el lector ensaye este programa y en su caso lo modifique para dominarlo.

Capítulo 4

Bibliotecas Científicas

En esta sección, nos enfocaremos en la instalación de bibliotecas esenciales para la inteligencia artificial y la ciencia de datos en Fedora 39. Este sistema operativo, conocido por su estabilidad y su impulso hacia la innovación, es ideal para desarrolladores y aficionados a la IA. Cubriremos bibliotecas clave como `PyTorch`, `Scikit-learn`, `Keras` para el diseño de redes neuronales, `NumPy` para operaciones matemáticas avanzadas, la biblioteca `Math` de Python para cálculos esenciales, y `Pandas` para la manipulación y análisis de datos. Proporcionaremos instrucciones detalladas para una instalación exitosa y consejos para optimizar el rendimiento de estas bibliotecas en Fedora 39, asegurando que los usuarios puedan aprovechar al máximo sus capacidades en este entorno de desarrollo dinámico.

4.1. Biblioteca `NumPy` para Cálculo Numérico

`NumPy`, una abreviatura de "Numerical Python", es una biblioteca fundamental en el ecosistema de Python, especialmente en los campos de la ciencia de datos, la ingeniería y la investigación científica. Este preámbulo pretende ilustrar las múltiples ventajas que hacen de `NumPy` una herramienta indispensable para los profesionales y entusiastas de estas áreas. Entre sus principales beneficios, destacan su eficiencia en el manejo y operación de grandes conjuntos de datos, gracias a su estructura interna optimizada que permite realizar cálculos numéricos de alta velocidad. Además, `NumPy` ofrece una extensa biblioteca de funciones matemáticas que facilitan desde operaciones básicas hasta cálculos complejos, soportando una amplia gama de aplicaciones en diferentes disciplinas. Su compatibilidad e integración con otras bibliotecas importantes de Python, como `Pandas`, `Matplotlib` y `SciPy`, amplían aún más su utilidad, convirtiéndola en una base para el análisis y la visualización de datos. En este preámbulo, exploraremos estas ventajas en detalle, proporcionando una comprensión clara de por qué `NumPy` es tan valorado en la comunidad científica y tecnológica.

La instalación de la biblioteca `NumPy` supone la ejecución de los siguientes comandos:

```
# Actualizar el sistema
sudo dnf update -y

# Instalar Pip para Python 3
sudo dnf install python3-pip -y

# Instalar NumPy usando Pip
pip3 install numpy
```

4.2. Principales Funciones de `NumPy`

`NumPy` es una biblioteca esencial en Python para la ciencia de datos y la computación científica, proporcionando una amplia gama de funciones. Algunas de sus funciones clave incluyen:

Función	**Descripción**

`numpy.array`	Crea un array N-dimensional.
`numpy.zeros`	Genera un array lleno de ceros.
`numpy.ones`	Crea un array lleno de unos.
`numpy.empty`	Crea un array sin inicializar.
`numpy.arange`	Genera valores espaciados uniformemente en un intervalo.
`numpy.linspace`	Crea un array de valores linealmente espaciados.
`numpy.reshape`	Cambia la forma de un array.
`numpy.ravel`	Aplana un array a una dimensión.
`numpy.transpose`	Transpone un array.
Operaciones básicas (+, -, *, /)	Realiza cálculos elementales en arrays.
`numpy.dot`	Realiza el producto punto entre arrays.
Funciones universales (`ufunc`)	Aplica funciones matemáticas a cada elemento del array (p.ej. `sin`, `cos`).
`numpy.linalg.inv`	Calcula la inversa de una matriz.
`numpy.linalg.eig`	Encuentra los valores y vectores propios de una matriz.
`numpy.linalg.svd`	Realiza la descomposición en valores singulares.
`numpy.mean`	Calcula la media de los elementos de un array.
`numpy.median`	Encuentra la mediana de un array.
`numpy.std`	Calcula la desviación estándar de un array.
`numpy.var`	Encuentra la varianza de un array.
`numpy.corrcoef`	Calcula los coeficientes de correlación.
`numpy.random.rand`	Genera números aleatorios en un array.
`numpy.random.randint`	Retorna enteros aleatorios de un rango.
`numpy.loadtxt`	Carga datos desde un archivo de texto.
`numpy.savetxt`	Guarda un array a un archivo de texto.

Cuadro 4.1: Principales funciones de la biblioteca `NumPy`.

Nota importante: Para más información, documentación detallada y tutoriales sobre `NumPy`, visite el sitio web oficial: https://numpy.org/.

Ejemplo 7.

```
#!/usr/bin/env python3

import numpy as np

# Crear un array 2D de números aleatorios
array_2d = np.random.rand(5, 5)

# Realizar operaciones matemáticas elementales
array_cuadrado = np.square(array_2d)
array_raiz = np.sqrt(array_2d)

# Crear un array 1D y redimensionarlo a 2D
array_1d = np.arange(25)
array_1d_reshape = array_1d.reshape(5, 5)

# Multiplicación de matrices
multiplicacion_matrices = np.dot(array_2d, array_1d_reshape)

# Encontrar valores máximos, mínimos y la media
maximo = np.max(array_2d)
minimo = np.min(array_2d)
media = np.mean(array_2d)

# Aplicar una función a todos los elementos
funcion_aplicada = np.sin(array_2d) + np.cos(array_2d)

```

```
print("Array 2D original:\n", array_2d)
print("\nArray 2D al cuadrado:\n", array_cuadrado)
print("\nRaíz cuadrada del array 2D:\n", array_raiz)
print("\nArray 1D redimensionado a 2D:\n", array_1d_reshape)
print("\nResultado de la multiplicación de matrices:\n", multiplicacion_matrices)
print("\nMáximo, mínimo y media del array 2D:", maximo, minimo, media)
print("\nArray 2D después de aplicar la función sin + cos:\n", funcion_aplicada)
```

Nota importante: El comando es un alias, establece `np` como un nombre alternativo para referirse a NumPy en el código. Esto se hace por conveniencia y para mantener el código más conciso. Por lo tanto, en lugar de escribir `numpy.funcion()` para cada llamada a una función de NumPy, simplemente se puede escribir `np.funcion()`.

```
./calculo_numerico.py
Array 2D original:
 [[0.38184212 0.37133481 0.14481153 0.10695061 0.07824697]
 [0.24553256 0.92266654 0.47578315 0.53080238 0.25459659]
 [0.48950207 0.72646162 0.79469694 0.5267596  0.9123638 ]
 [0.17367246 0.4101474  0.34811659 0.3809394  0.11953068]
 [0.68360381 0.46146527 0.57394428 0.02992457 0.01870444]]

Array 2D al cuadrado:
 [[1.45803406e-01 1.37889541e-01 2.09703778e-02 1.14384324e-02
  6.12258870e-03]
 [6.02862391e-02 8.51313535e-01 2.26369601e-01 2.81751169e-01
  6.48194259e-02]
 [2.39612274e-01 5.27746491e-01 6.31543233e-01 2.77475678e-01
  8.32407710e-01]
 [3.01621223e-02 1.68220886e-01 1.21185161e-01 1.45114823e-01
  1.42875840e-02]
 [4.67314174e-01 2.12950195e-01 3.29412031e-01 8.95480050e-04
  3.49856039e-04]]

Raíz cuadrada del array 2D:
 [[0.61793375 0.60937247 0.3805411  0.32703304 0.2797266 ]
 [0.49551242 0.96055533 0.68977036 0.72856186 0.50457566]
 [0.69964424 0.85232718 0.89145776 0.72578206 0.95517737]
 [0.41674028 0.64042751 0.59001406 0.61720288 0.3457321 ]
 [0.82680337 0.67931235 0.7575911  0.1729872  0.13676417]]

Array 1D redimensionado a 2D:
 [[ 0  1  2  3  4]
 [ 5  6  7  8  9]
 [10 11 12 13 14]
 [15 16 17 18 19]
 [20 21 22 23 24]]

Resultado de la multiplicación de matrices:
 [[ 6.47398786  7.5571739   8.64035994  9.72354597 10.80673201]
 [22.42513176 24.85451297 27.28389419 29.71327541 32.14265663]
 [37.72794766 41.1777317  44.62751574 48.07729979 51.52708383]
 [13.63660747 15.06901399 16.50142051 17.93382703 19.36623355]
 [ 8.86972647 10.63736884 12.40501121 14.17265359 15.94029596]]

Máximo, mínimo y media del array 2D: 0.9226665352902551 0.018704439014055607
    0.4064960075910038

Array 2D después de aplicar la función sin + cos:
 [[1.30061033 1.29470342 1.13383905 1.10103307 1.07510742]
 [1.21308096 1.40091074 1.34696904 1.36862664 1.21962   ]
 [1.35275358 1.41175813 1.41415242 1.36717552 1.40283011]
 [1.15775752 1.31580656 1.28114484 1.30010847 1.11211096]
 [1.40689278 1.34066162 1.38271451 1.0294724  1.01852843]]
```

Este programa realiza las siguientes operaciones con `NumPy`:

(1) Creación de un array 2D con números aleatorios.

(2) Aplicación de operaciones matemáticas elementales como el cuadrado y la raíz cuadrada.

(3) Redimensionamiento de un array 1D a un array 2D.

(4) Realización de la multiplicación de matrices.

(5) Cálculo de valores máximos, mínimos y la media del array.

(6) Aplicación de funciones trigonométricas a todos los elementos del array.

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

Práctica 8.

Es conveniente que el lector pida a `ChatGPT` ejemplos de uso de cada una de las funciones en (Sección 4.2), y los compile y ejecute individualmente como en (Ejemplo 7).

Ejemplo 8.

```
#!/usr/bin/env python3
import numpy as np
# Crear una matriz cuadrada
A = np.array([[1, 2], [3, 4]])

# Calcular la inversa de la matriz
A_inv = np.linalg.inv(A)

print("Matriz Original:\n", A)
print("Inversa de la Matriz:\n", A_inv)

```

```
chmod +x uso_comando.py
./uso_comando.py
Matriz Original:
 [[1 2]
 [3 4]]
Inversa de la Matriz:
 [[-2.   1. ]
 [ 1.5 -0.5]]

```

4.3. Biblioteca `Pandas` para Manipulación de Datos

`Pandas` es una biblioteca de software escrita como extensión de `NumPy` para manipulación y análisis de datos en Python. Ofrece estructuras de datos y operaciones para manipular tablas numéricas y series temporales, siendo una herramienta indispensable en el análisis de datos. Entre sus principales ventajas, destacamos:

(1) **Facilidad de Uso:** Pandas simplifica el proceso de carga, manipulación y análisis de datos con su poderosa estructura de datos DataFrame.

(2) **Alto Rendimiento:** Para operaciones de datos de gran tamaño, Pandas está optimizado con técnicas de rendimiento crítico.

(3) **Integración con Otras Herramientas:** Se integra perfectamente con otras bibliotecas utilizadas en la ciencia de datos como NumPy, SciPy, y Matplotlib.

(4) **Funcionalidad Versátil:** Incluye funciones para leer y escribir datos entre diferentes formatos, manipulación de datos faltantes, agrupación y remuestreo de datos, entre otros.

(5) **Amplio Uso en la Industria:** Es ampliamente utilizada en el ámbito académico y comercial, lo que garantiza un continuo desarrollo y mantenimiento.

Estas características hacen de `Pandas` una herramienta esencial para cualquier científico de datos o analista que trabaje con Python.

La instalación de la biblioteca `Pandas` supone la ejecución de los siguientes comandos:

```
# Actualizar el sistema
sudo dnf update -y

# Instalar Pip para Python 3
sudo dnf install python3-pip -y

# Instalar NumPy usando Pip
pip3 install pandas
```

Nota importante: De haber instalado la biblioteca `NumPy` los primeros dos pasos de este procedimiento no es necesario ejecutarlos.

4.4. Principales Funciones de `Pandas`

A continuación se presenta una tabla con algunas de las principales funciones de Pandas organizadas por categorías:

Categoría	**Funciones**
Creación de DataFrames	`pandas.DataFrame()`, `pandas.read_csv()`, `pandas.read_excel()`, `pandas.read_json()`
Manipulación de Datos	`DataFrame.head()`, `DataFrame.tail()`, `DataFrame.drop()`, `DataFrame.rename()`
Selección y Filtrado	`DataFrame.loc[]`, `DataFrame.iloc[]`, `DataFrame.query()`
Operaciones con Datos	`DataFrame.groupby()`, `DataFrame.merge()`, `DataFrame.join()`, `DataFrame.concat()`
Funciones Estadísticas	`DataFrame.describe()`, `DataFrame.mean()`, `DataFrame.median()`, `DataFrame.std()`
Manejo de Datos Faltantes	`DataFrame.dropna()`, `DataFrame.fillna()`, `DataFrame.isna()`
Conversión y Exportación	`DataFrame.to_csv()`, `DataFrame.to_excel()`, `DataFrame.to_json()`

Cuadro 4.2: Principales funciones de la biblioteca `Pandas`.

Ejemplo 8.

```
  #!/usr/bin/env python3

import pandas as pd

# Creación de un DataFrame a partir de un diccionario
data = {'Name': ['Alice', 'Bob', 'Charlie'],
        'Age': [25, 30, 35],
        'City': ['New York', 'Los Angeles', 'Chicago']}
df = pd.DataFrame(data)

# Mostrar las primeras filas del DataFrame
```

```
print("Primeras filas del DataFrame:")
print(df.head())

# Manipulación de datos: añadir una nueva columna
df['Salary'] = [70000, 80000, 90000]
print("\nDataFrame después de añadir la columna 'Salary':")
print(df)

# Selección y filtrado: seleccionar filas basadas en una condición
filtered_df = df[df['Age'] > 28]
print("\nFilas donde la edad es mayor que 28:")
print(filtered_df)

# Operaciones con datos: calcular la media de una columna
average_age = df['Age'].mean()
print("\nEdad media de las personas en el DataFrame:", average_age)

# Funciones estadísticas: descripción estadística del DataFrame
print("\nDescripción estadística del DataFrame:")
print(df.describe())

# Manejo de datos faltantes: añadir filas con datos faltantes y manejarlos
df.loc[3] = ['David', None, 'Miami', 85000]
print("\nDataFrame con una fila con datos faltantes:")
print(df)
df.fillna({'Age': df['Age'].mean()}, inplace=True)
print("\nDataFrame después de manejar los datos faltantes en 'Age':")
print(df)

# Conversión y exportación: convertir el DataFrame a CSV
df.to_csv('data.csv', index=False)
print("\nDataFrame guardado como 'data.csv'.")

# Lectura de datos: cargar un DataFrame desde un archivo CSV
loaded_df = pd.read_csv('data.csv')
print("\nDataFrame cargado desde 'data.csv':")
print(loaded_df)
```

```
Primeras filas del DataFrame:
      Name  Age         City
0    Alice   25     New York
1      Bob   30  Los Angeles
2  Charlie   35      Chicago

DataFrame después de añadir la columna 'Salary':
      Name  Age         City  Salary
0    Alice   25     New York   70000
1      Bob   30  Los Angeles   80000
2  Charlie   35      Chicago   90000

Filas donde la edad es mayor que 28:
      Name  Age         City  Salary
1      Bob   30  Los Angeles   80000
2  Charlie   35      Chicago   90000

Edad media de las personas en el DataFrame: 30.0

Descripción estadística del DataFrame:
        Age   Salary
count   3.0      3.0
mean   30.0  80000.0
std     5.0  10000.0
min    25.0  70000.0
25%    27.5  75000.0
50%    30.0  80000.0
75%    32.5  85000.0
max    35.0  90000.0

```

```
DataFrame con una fila con datos faltantes:
      Name    Age         City  Salary
0    Alice     25     New York   70000
1      Bob     30  Los Angeles   80000
2  Charlie     35      Chicago   90000
3    David   None        Miami   85000

DataFrame después de manejar los datos faltantes en 'Age':
      Name    Age         City  Salary
0    Alice   25.0     New York   70000
1      Bob   30.0  Los Angeles   80000
2  Charlie   35.0      Chicago   90000
3    David   30.0        Miami   85000

DataFrame guardado como 'data.csv'.

DataFrame cargado desde 'data.csv':
      Name    Age         City  Salary
0    Alice   25.0     New York   70000
1      Bob   30.0  Los Angeles   80000
2  Charlie   35.0      Chicago   90000
3    David   30.0        Miami   85000

```

Este programa realiza las siguientes operaciones con Pandas:

(1) Creación de un `DataFrame` a partir de un diccionario.

(2) Mostrar las primeras filas del `DataFrame`.

(3) Manipulación de datos añadiendo una nueva columna.

(4) Selección y filtrado basado en condiciones.

(5) Cálculo de la media de una columna.

(6) Obtención de una descripción estadística del `DataFrame`.

(7) Manejo de datos faltantes.

(8) Exportación del `DataFrame` a un archivo CSV.

(9) Carga de un `DataFrame` desde un archivo CSV.

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

Práctica 9.

Es conveniente que el lector pida a `ChatGPT` ejemplos de uso de cada una de las funciones en (Sección 4.4), y los compile y ejecute individualmente.

4.5. Visualización con `Matplotlib`

`Matplotlib` es una biblioteca de gráficos para el lenguaje de programación Python y su extensión numérica matemática `NumPy`. Fue creada originalmente por John D. Hunter en 2003 [1] y se ha convertido en una herramienta esencial para la visualización de datos en Python. Proporciona una interfaz orientada a objetos para integrar gráficos en aplicaciones con diversos kits de herramientas de interfaz de usuario.

4.6. Ventajas de `Matplotlib`

Numerosas son las ventajas de usar esta biblioteca, entre las que se destacan su facilidad de uso y compatibilidad con bibliotecas.

(1) **Facilidad de Uso**: `Matplotlib` es conocida por su simplicidad, permitiendo a los usuarios crear gráficos de alta calidad de manera rápida y eficiente.

(2) **Personalización**: Ofrece amplias opciones de personalización, permitiendo ajustar casi todos los aspectos de los gráficos.

(3) **Variedad de Gráficos**: Soporta una amplia gama de tipos de gráficos, adecuada para diversas aplicaciones y campos.

(4) **Compatibilidad con `NumPy` y `Pandas`**: Se integra perfectamente con `NumPy` y `Pandas`, facilitando la visualización de datos complejos y el análisis estadístico.

(5) **Gráficos Interactivos y Animaciones**: Admite la creación de visualizaciones interactivas y animaciones, ampliando su funcionalidad.

(6) **Exportación en Múltiples Formatos**: Permite exportar gráficos en diversos formatos, facilitando su inclusión en publicaciones o presentaciones.

(7) **Comunidad y Documentación**: Cuenta con una amplia comunidad y una extensa documentación, proporcionando un gran soporte para los usuarios.

4.7. Tabla de Funciones de Matplotlib

Función	Descripción	Ejemplo de Uso
`plot`	Gráfico de líneas o marcadores.	`plt.plot(x, y)`
`scatter`	Gráfico de dispersión.	`plt.scatter(x, y)`
`bar`	Gráfico de barras.	`plt.bar(x, y)`
`hist`	Histograma.	`plt.hist(data)`
`imshow`	Mostrar imágenes.	`plt.imshow(image)`
`subplot`	Crear subgráficos.	`plt.subplot(1, 2, 1)`
`xlabel`	Etiqueta del eje X.	`plt.xlabel('Eje X')`
`ylabel`	Etiqueta del eje Y.	`plt.ylabel('Eje Y')`
`title`	Título del gráfico.	`plt.title('Título')`
`legend`	Añadir leyenda.	`plt.legend()`
`show`	Mostrar el gráfico.	`plt.show()`

Cuadro 4.3: Resumen de las Funciones Principales de `Matplotlib`.

Ejemplo 9.

```
#!/usr/bin/env python3
import matplotlib.pyplot as plt
import numpy as np

# Crear los valores para el eje x
x = np.linspace(-np.pi, np.pi, 300)

# Calcular los valores de la función seno para cada punto en x
y = np.sin(x)
```

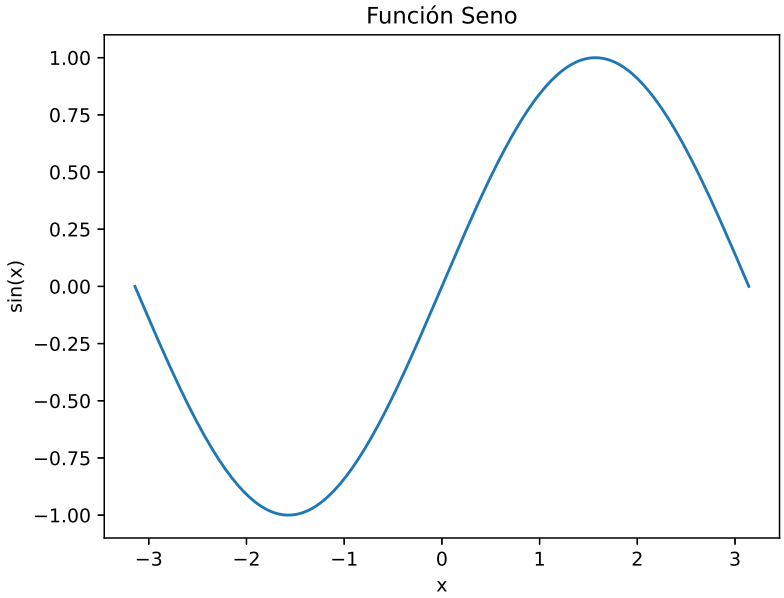


Figura 4.1: Gráfico de la función $\sin x$ a partir de `Matplotlib`.

```

# Crear el gráfico
plt.plot(x, y)

# Agregar título y etiquetas
plt.title("Función Seno")
plt.xlabel("x")
plt.ylabel("sin(x)")

# Guardar el gráfico en formato EPS
plt.savefig("seno1.eps")
```

```
chmod +x seno.py
./seno.py
```

El lector puede bajar este programa desde el siguiente vínculo:

```
#!/usr/bin/env python3
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np

# Crear una malla de puntos en el espacio x-y
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
x, y = np.meshgrid(x, y)

# Calcular z = x^2 + y^2
z = x**2 + y**2

# Crear una figura y un eje 3D
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

# Dibujar la superficie
ax.plot_surface(x, y, z, cmap='viridis')

```

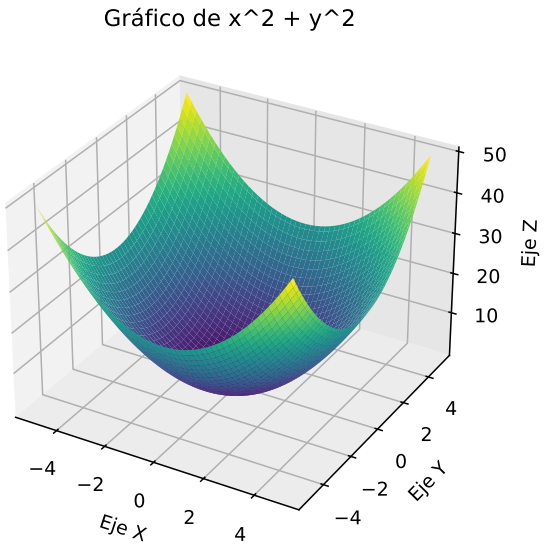


Figura 4.2: Gráfico de la función $x^2 + y^2$ a partir de `Matplotlib`.

```
# Agregar títulos y etiquetas
ax.set_title('Gráfico de x^2 + y^2')
ax.set_xlabel('Eje X')
ax.set_ylabel('Eje Y')
ax.set_zlabel('Eje Z')

# Guardar el gráfico en formato EPS
plt.savefig("several_variables.eps")
```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

4.8. Caso Práctico

El álgebra vectorial es una rama fundamental de las matemáticas que trata con vectores, entidades que tienen tanto magnitud como dirección. Se utiliza en diversas áreas como la física, la ingeniería, y las ciencias de la computación para representar fenómenos físicos y abstracciones matemáticas.

Un **vector** en el espacio bidimensional se representa comúnmente como $\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$, donde v_1 y v_2 son las componentes del vector en los ejes x e y, respectivamente.

Las operaciones fundamentales en el álgebra vectorial incluyen:

La suma de dos vectores $\mathbf{a}$ y $\mathbf{b}$ se define como:

$$\mathbf{a} + \mathbf{b} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \end{pmatrix}$$

La resta de dos vectores $\mathbf{a}$ y $\mathbf{b}$ se define como:

$$\mathbf{a} - \mathbf{b} = \begin{pmatrix} a_1 - b_1 \\ a_2 - b_2 \end{pmatrix}$$

El producto punto de dos vectores **a** y **b**, también conocido como producto escalar, se define como:

$$\mathbf{a} \cdot \mathbf{b} = a_1 b_1 + a_2 b_2$$

El álgebra vectorial es esencial para entender y trabajar con magnitudes físicas y conceptos matemáticos que tienen tanto dirección como magnitud. En este documento, exploraremos estas operaciones básicas utilizando herramientas computacionales.

Ejemplo 9.

```python
#!/usr/bin/env python3

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import normalize

# Creación de vectores usando Numpy
vector_a = np.array([2, 3])
vector_b = np.array([1, 4])

# Operaciones de álgebra vectorial
suma = vector_a + vector_b
resta = vector_a - vector_b
producto_punto = np.dot(vector_a, vector_b)

# Normalización de vectores usando Scikit-learn
vector_a_normalizado = normalize(vector_a.reshape(1, -1))
vector_b_normalizado = normalize(vector_b.reshape(1, -1))

# Creación de DataFrames usando Pandas
df_vectores = pd.DataFrame({
    'Vector A': vector_a,
    'Vector B': vector_b
})

df_resultados = pd.DataFrame({
    'Operación': ['Suma', 'Resta', 'Producto Punto', 'Vector A Normalizado', 'Vector B Normalizado'],
    'Resultado': [suma, resta, producto_punto, vector_a_normalizado[0], vector_b_normalizado[0]]
})

print("Vectores:")
print(df_vectores)
print("\nResultados de Operaciones:")
print(df_resultados)

# Gráficos usando Matplotlib
plt.quiver(0, 0, vector_a[0], vector_a[1], angles='xy', scale_units='xy', scale=1, color='r')
plt.quiver(0, 0, vector_b[0], vector_b[1], angles='xy', scale_units='xy', scale=1, color='b')
plt.xlim(-5, 5)
plt.ylim(-5, 5)
plt.grid()
plt.title('Vectores y sus operaciones')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend(['Vector A', 'Vector B'])

# Guardar la figura como archivo .eps
plt.savefig("vectores_operaciones.eps", format='eps')
```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

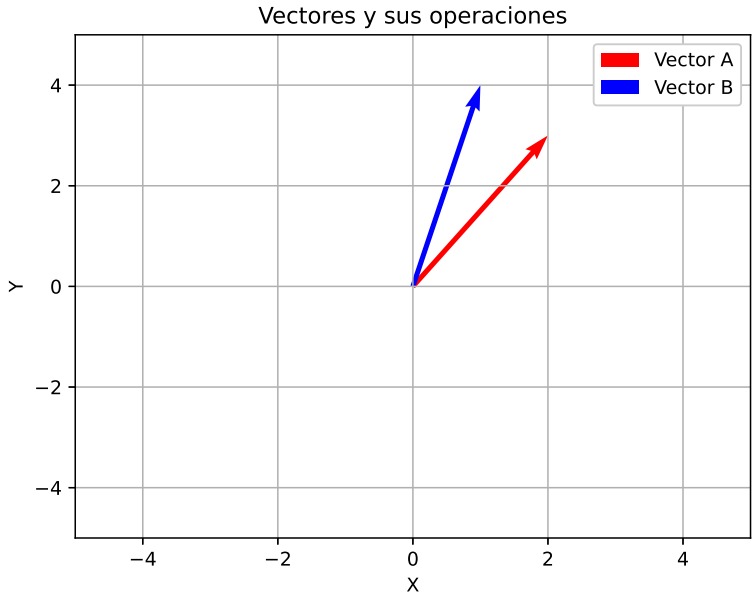


Figura 4.3: Gráfico de los vectores involucrados en las operaciones a partir de `Matplotlib`.

```
./vector.py
Vectores:
   Vector A  Vector B
0         2         1
1         3         4

Resultados de Operaciones:
               Operación                                      Resultado
0                  Suma                                         [3, 7]
1                 Resta                                        [1, -1]
2        Producto Punto                                             14
3  Vector A Normalizado   [0.5547001962252291, 0.8320502943378437]
4  Vector B Normalizado   [0.24253562503633297, 0.9701425001453319]
```

El programa implementa las siguientes funcionalidades:

(1) **Creación de Vectores**: Utiliza Numpy para definir vectores en un espacio bidimensional.

(2) **Operaciones Vectoriales**: Realiza suma, resta y producto punto entre vectores.

(3) **Normalización de Vectores**: Emplea Scikit-learn para normalizar los vectores.

(4) **Visualización de Resultados**: Usa Matplotlib para graficar los vectores y sus operaciones.

(5) **Exportación de Gráficos**: Guarda los gráficos en formato .eps.

El programa muestra efectivamente cómo se pueden realizar y visualizar operaciones básicas de álgebra vectorial en Python. Los vectores y sus operaciones se presentan tanto en formato tabular como gráfico, facilitando la comprensión de estas operaciones fundamentales. La exportación del gráfico en formato .eps permite una visualización de alta calidad, adecuada para la inclusión en reportes o publicaciones científicas. Los resultados demuestran la utilidad de Python y sus bibliotecas en el manejo y análisis de datos vectoriales, siendo una herramienta valiosa en campos como la ingeniería, la física y la informática.

Práctica 10.

Es conveniente que el lector pida a `ChatGPT` variantes a este ejemplo que incluya cambios en los valores y nuevos operadores.

Capítulo 5

Manejo de Excepciones

Los errores en programación pueden ser divididos de acuerdo en que momento ocurren, en tiempo de **ejecución** o en tiempo de **compilación**.

5.1. Errores Comunes

En programación, los errores son inevitables. En Python, estos errores se conocen como "excepciones". Algunos de los errores comunes incluyen:

(1) `SyntaxError`: Ocurre cuando Python encuentra un error en la sintaxis del código.

(2) `NameError`: Se produce cuando una variable no ha sido definida.

(3) `TypeError`: Aparece cuando se realiza una operación con tipos de datos incompatibles.

(4) `IndexError` y `KeyError`: Ocurren al intentar acceder a un índice o clave inexistente en una lista o diccionario, respectivamente.

5.1.1. Bloques `try` y `except`

Para manejar las excepciones, Python ofrece los bloques `try` y `except`. La estructura básica es la siguiente:

```
try:
    # Intenta ejecutar este código
    ...
except ExcepcionComoOcurre:
    # Ejecuta este código si se produce la excepción especificada
    ...
```

El código dentro del bloque `try` se ejecuta primero. Si se encuentra una excepción, se interrumpe el flujo normal y se ejecuta el código dentro del bloque `except` correspondiente.

5.2. Manejo de Excepciones Personalizadas

Las instrucciones `try` y `except` permiten manejar errores comunes como la división por cero. Ello hace robusto el programa, y evita que el programa en tiempo de ejecución falle.

Además de las excepciones incorporadas, Python permite crear excepciones personalizadas. Esto se hace extendiendo la clase `Exception` de Python.

Por ejemplo:

```
class MiExcepcion(Exception):
    pass
```

Luego, puedes lanzar tu excepción personalizada usando `raise`:

```
raise MiExcepcion("Mensaje de error personalizado")
```

Estas excepciones personalizadas son útiles para casos específicos en tus programas y ayudan a hacer tu código más legible y fácil de depurar.

Ejemplo 10.

```
def dividir(a, b):
    try:
        # Intenta realizar la operación de división.
        # Si 'b' es cero, se producirá un ZeroDivisionError.
        resultado = a / b
    except ZeroDivisionError:
        # Este bloque se ejecuta si se produce un ZeroDivisionError.
        # En este caso, se imprime un mensaje y se establece 'resultado' en None.
        print("No se puede dividir por cero.")
        resultado = None
    return resultado

# Ejemplo de uso
numero1 = 10
numero2 = 0

# Llamada a la función 'dividir'. Si 'numero2' es cero, se activará el except.
resultado = dividir(numero1, numero2)

# Comprobamos si el resultado es válido (no None) antes de imprimirlo.
if resultado is not None:
    print("El resultado es:", resultado)
else:
    print("La operación no se pudo completar.")

```

```
./excepción.py
No se puede dividir por cero.
La operación no se pudo completar.

```

El lector puede bajar este programa desde el siguiente vínculo:

En este código:

(1) La línea `resultado = a / b` dentro del bloque `try` es donde puede ocurrir un `ZeroDivisionError`.

(2) Si `b` es 0, Python lanza automáticamente la excepción `ZeroDivisionError`.

(3) El bloque `except ZeroDivisionError` atrapa esta excepción y ejecuta el código dentro de él, evitando que el programa se detenga abruptamente.

(4) Este manejo de excepciones permite que el programa continúe ejecutándose de manera controlada, incluso después de un error.

5.3. Caso Práctico

En este ejemplo, discutiremos el manejo de excepciones en Python. Las excepciones son errores que pueden ocurrir durante la ejecución de un programa y pueden interrumpir su flujo normal. El manejo de excepciones nos permite controlar estos errores de manera efectiva, evitando que el programa se bloquee y proporcionando una forma de gestionar las situaciones inesperadas.

5.3.1. Sin excepciones (sin manejo de errores)

En este ejemplo, intentaremos abrir un archivo que no existe y luego intentaremos dividir un número entre cero, lo que provocará errores y terminará el programa abruptamente.

```
#!/usr/bin/env python3

# Sin el uso de excepciones
print("Ejemplo sin excepciones:")

# Intentar abrir un archivo que no existe
archivo = open("archivo_que_no_existe.txt", "r")
contenido = archivo.read()
archivo.close()

# Intentar dividir por cero
resultado = 10 / 0

print("Este mensaje no se imprimirá debido a los errores anteriores.")
```

Cuando ejecutes este código, verás que se producirán errores y el programa se detendrá sin mostrar el último mensaje.

```
./sinexcepciones.py

Ejemplo sin excepciones:
Traceback (most recent call last):
  File "sinexcepcion.py", line 5, in <module>
    archivo = open("archivo_que_no_existe.txt", "r")
FileNotFoundError: [Errno 2] No such file or directory: 'archivo_que_no_existe.txt'
```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

5.3.2. Con excepciones (manejo de errores)

En este ejemplo, utilizaremos las excepciones (`try` y `except`) para manejar errores de manera controlada y asegurarnos de que el programa no se detenga abruptamente.

```
#!/usr/bin/env python3

# Con el uso de excepciones
print("Ejemplo con excepciones:")

try:
    # Intentar abrir un archivo que no existe
    archivo = open("archivo_que_no_existe.txt", "r")
    contenido = archivo.read()
    archivo.close()
except FileNotFoundError as e:
    print(f"Error al abrir el archivo: {e}")

try:
    # Intentar dividir por cero
    resultado = 10 / 0
except ZeroDivisionError as e:
    print(f"Error al dividir por cero: {e}")

print("Este mensaje se imprimirá incluso después de los errores anteriores.")
```

Cuando ejecutes este código, verás que, aunque se produzcan errores al intentar abrir el archivo inexistente y al dividir por cero, el programa no se detendrá abruptamente. En su lugar, mostrará mensajes de error personalizados y continuará ejecutando el código restante.

```
./conexcepciones.py

Ejemplo con excepciones:
```

```
Error al abrir el archivo: [Errno 2] No such file or directory: 'archivo_que_no_existe.txt'
Error al dividir por cero: division by zero
Este mensaje se imprimirá incluso después de los errores anteriores.
```

El lector puede bajar este programa desde el siguiente vínculo:

El uso de excepciones permite manejar errores de manera más controlada, lo que puede ser crucial en programas más grandes y complejos para garantizar que el programa no falle por completo debido a un error inesperado.

Capítulo 6

Manipulación de Archivos

En la programación y el procesamiento de datos, la manipulación de archivos desempeña un papel fundamental. Los archivos son una forma común de almacenar y recuperar información de manera persistente en una computadora. Ya sea que necesites leer datos desde un archivo existente, crear nuevos archivos para almacenar resultados o manipular datos estructurados en formatos específicos, la habilidad para trabajar con archivos es esencial en muchas aplicaciones.

Ya sea que estés desarrollando aplicaciones que necesitan acceder a archivos locales o procesando datos almacenados en múltiples formatos, comprender los conceptos y técnicas de manipulación de archivos te permitirá ser más eficiente y efectivo en tus tareas de programación y análisis de datos.

6.1. Lectura y Escritura de Archivos

La manipulación de archivos se refiere a la capacidad de un programa para leer y escribir archivos en el sistema de archivos de una computadora. Los archivos son una forma común de almacenar datos de manera persistente, lo que significa que los datos se mantienen incluso después de que se apague la computadora.

Ejemplo de lectura de un archivo en Python:

```python
# Abrir un archivo en modo lectura
with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()
    print(contenido)
```

Ejemplo de escritura en un archivo en Python:

```python
# Abrir un archivo en modo escritura
with open("nuevo_archivo.txt", "w") as archivo:
    archivo.write("Hola, mundo!")
```

6.2. Rutas de Archivos

Las rutas de archivos son la ubicación o dirección de un archivo en el sistema de archivos de una computadora. Las rutas pueden ser absolutas (desde la raíz del sistema) o relativas (desde el directorio actual).

Ejemplo de ruta absoluta en Windows: `C:\Users\Usuario\documento.txt`

Ejemplo de ruta relativa en Linux: `./carpeta/archivo.txt`

6.3. Archivos CSV

Los archivos CSV (Valores Separados por Comas) son utilizados para almacenar datos tabulares en formato de texto. Cada línea del archivo representa una fila de datos, y los valores se separan por comas u otro delimitador.

Ejemplo de lectura de un archivo CSV en Python:

```
import csv

with open("datos.csv", "r") as archivo:
    lector_csv = csv.reader(archivo)
    for fila in lector_csv:
        print(fila)
```

6.4. Archivos JSON

Los archivos JSON (Notación de Objetos de JavaScript) se utilizan para almacenar datos estructurados en formato de texto. Los datos se organizan en pares clave-valor, y pueden contener listas y objetos anidados.

Ejemplo de lectura de un archivo JSON en Python:

```
import json

with open("datos.json", "r") as archivo:
    datos = json.load(archivo)
    print(datos)
```

6.5. Archivos XML

Los archivos XML (Lenguaje de Marcado Extensible) se utilizan para almacenar datos estructurados en formato de texto. Los datos se organizan en etiquetas que pueden tener atributos y contenido.

Ejemplo de lectura de un archivo XML en Python:

```
import xml.etree.ElementTree as ET

tree = ET.parse("datos.xml")
root = tree.getroot()

for elemento in root:
    print(elemento.tag, elemento.text)
```

6.6. Caso Práctico

```
#!/usr/bin/env python3

import csv
import json
import xml.etree.ElementTree as ET

# Generar archivos de prueba
contenido_texto = "Hola, mundo"
datos_csv = [["Nombre", "Edad"], ["Juan", "25"], ["Ana", "30"]]
datos_json = {"Nombre": "Carlos", "Edad": 28}

nuevo_elemento_xml = ET.Element("Persona")
nombre_xml = ET.SubElement(nuevo_elemento_xml, "Nombre")
nombre_xml.text = "Laura"
edad_xml = ET.SubElement(nuevo_elemento_xml, "Edad")
edad_xml.text = "22"

tree_xml = ET.ElementTree(nuevo_elemento_xml)

# Escribir en archivos de prueba
with open("archivo.txt", "w") as archivo_texto:
    archivo_texto.write(contenido_texto)
```

```
23
24 with open("datos.csv", "w", newline="") as archivo_csv:
25     escritor_csv = csv.writer(archivo_csv)
26     for fila in datos_csv:
27         escritor_csv.writerow(fila)
28
29 with open("datos.json", "w") as archivo_json:
30     json.dump(datos_json, archivo_json, indent=4)
31
32 tree_xml.write("datos.xml")
33
34 # Leer y mostrar los archivos
35 with open("archivo.txt", "r") as archivo_texto:
36     contenido_texto = archivo_texto.read()
37     print("Contenido del archivo de texto:")
38     print(contenido_texto)
39
40 with open("datos.csv", "r") as archivo_csv:
41     lector_csv = csv.reader(archivo_csv)
42     print("\nContenido del archivo CSV:")
43     for fila in lector_csv:
44         print(fila)
45
46 with open("datos.json", "r") as archivo_json:
47     datos_json = json.load(archivo_json)
48     print("\nContenido del archivo JSON:")
49     print(datos_json)
50
51 tree = ET.parse("datos.xml")
52 root = tree.getroot()
53 print("\nContenido del archivo XML:")
54 for elemento in root:
55     print(elemento.tag, elemento.text)
56
57 print("\nArchivos generados y mostrados con éxito.")
```

```
1  Contenido del archivo de texto:
2  Hola, mundo
3
4  Contenido del archivo CSV:
5  ['Nombre', 'Edad']
6  ['Juan', '25']
7  ['Ana', '30']
8
9  Contenido del archivo JSON:
10 {'Nombre': 'Carlos', 'Edad': 28}
11
12 Contenido del archivo XML:
13 Nombre Laura
14 Edad 22
15
16 Archivos generados y mostrados con éxito.
```

 El lector puede bajar este programa desde el siguiente vínculo: 

Capítulo 7

Base de Datos `SQLite`

Las bases de datos son una parte integral de la mayoría de las aplicaciones modernas, proporcionando un medio eficiente y organizado para almacenar, recuperar y gestionar datos. En esta sección, se explorará la evolución de las bases de datos, sus tipos y modelos, y la importancia de los Sistemas de Gestión de Bases de Datos (SGBD). Se discutirá cómo las bases de datos no solo almacenan información, sino que también permiten operaciones complejas de búsqueda y manipulación de datos, lo cual es crucial en el mundo de la tecnología de la información.

7.1. Facilidades

`SQLite` es una biblioteca de software que proporciona un sistema de gestión de bases de datos relacional. Es notable por su ligereza, eficiencia y capacidad de ser integrado directamente en aplicaciones. A continuación, se presentan algunas ventajas clave de SQLite:

(1) **Independencia de la plataforma**: `SQLite` no depende de un sistema operativo específico, lo que facilita su uso en una variedad de plataformas, incluidas móviles, de escritorio y servidores.

(2) **Autonomía**: Al ser una base de datos autocontenida, `SQLite` no requiere un servidor de base de datos separado. Esto simplifica la configuración y reduce la sobrecarga administrativa.

(3) **Facilidad de integración**: Su diseño simple permite una integración fácil y directa en aplicaciones. Esto es especialmente útil para aplicaciones que necesitan una solución de almacenamiento local sin la complejidad de un sistema de base de datos completo.

(4) **Ligereza**: `SQLite` tiene un tamaño de biblioteca relativamente pequeño, lo que lo hace ideal para dispositivos con recursos limitados o para aplicaciones que necesitan ser ágiles y eficientes.

(5) **Flexibilidad en el manejo de tipos de datos**: Aunque es una base de datos relacional, `SQLite` ofrece cierta flexibilidad en el manejo de tipos de datos, lo que puede ser útil en ciertos casos de uso.

(6) **Facilidad de mantenimiento y actualización**: Al estar contenida en un único archivo, la gestión y actualización de una base de datos `SQLite` es sencilla, lo que facilita las tareas de mantenimiento y backup.

(7) **Transacciones ACID**: A pesar de su simplicidad, `SQLite` soporta transacciones ACID (Atomicidad, Consistencia, Aislamiento, Durabilidad), asegurando la fiabilidad y la integridad de los datos.

(8) **Alto rendimiento**: Para muchas operaciones comunes, `SQLite` ofrece un rendimiento sobresaliente, especialmente en entornos donde la carga de concurrencia es moderada o baja.

7.2. Componentes y Uso

Conexión a SQLite.

```
import sqlite3

# Conectar a la base de datos SQLite
conexion = sqlite3.connect('mi_base_de_datos.db')

# Crear un cursor
cursor = conexion.cursor()

# Cerrar la conexión
conexion.close()
```

SQLite es un sistema de gestión de bases de datos relacional ligero, que se integra con Python de manera eficiente. Esta sección aborda cómo utilizar SQLite en Python.

Operaciones Básicas con SQLite

```
import sqlite3

# Conectar a la base de datos
conexion = sqlite3.connect('mi_base_de_datos.db')

# Crear una tabla
conexion.execute('''CREATE TABLE IF NOT EXISTS estudiantes
                    (id INTEGER PRIMARY KEY, nombre TEXT, edad INTEGER)''')

# Insertar datos
conexion.execute("INSERT INTO estudiantes (nombre, edad) VALUES ('Ana', 21)")

# Consultar datos
cursor = conexion.cursor()
cursor.execute("SELECT * FROM estudiantes")
for fila in cursor.fetchall():
    print(fila)

# Cerrar la conexión
conexion.close()
```

MySQL es uno de los sistemas de gestión de bases de datos más populares. En esta parte, se explora cómo interactuar con una base de datos MySQL utilizando Python.

Conexión a MySQL y Operaciones.

```
import mysql.connector

# Conectar a la base de datos MySQL
conexion = mysql.connector.connect(user='tu_usuario',
                                   password='tu_contraseña',
                                   host='127.0.0.1',
                                   database='mi_base_de_datos')

# Crear un cursor
cursor = conexion.cursor()

# Crear una tabla
cursor.execute("CREATE TABLE IF NOT EXISTS estudiantes (id INT AUTO_INCREMENT PRIMARY KEY,
    nombre VARCHAR(255), edad INT)")

# Insertar datos
cursor.execute("INSERT INTO estudiantes (nombre, edad) VALUES ('Carlos', 23)")

# Hacer los cambios permanentes
conexion.commit()

# Consultar datos
```

```
cursor.execute("SELECT * FROM estudiantes")
for fila in cursor.fetchall():
    print(fila)

# Cerrar la conexión
conexion.close()
```

7.3. Caso Práctico

```
import sqlite3

# Conectar a la base de datos SQLite
conn = sqlite3.connect('agenda_telefonica.db')
cursor = conn.cursor()

# Crear tabla si no existe
cursor.execute('''
CREATE TABLE IF NOT EXISTS contactos (
    id INTEGER PRIMARY KEY,
    nombre TEXT NOT NULL,
    telefono TEXT NOT NULL
)
''')

# Funciones para manipular la agenda
def agregar_contacto(nombre, telefono):
    cursor.execute('INSERT INTO contactos (nombre, telefono) VALUES (?, ?)', (nombre,
    telefono))
    conn.commit()

def buscar_contacto(nombre):
    cursor.execute('SELECT * FROM contactos WHERE nombre LIKE ?', ('%'+nombre+'%',))
    return cursor.fetchall()

def actualizar_contacto(id, nuevo_nombre, nuevo_telefono):
    cursor.execute('UPDATE contactos SET nombre = ?, telefono = ? WHERE id = ?', (
    nuevo_nombre, nuevo_telefono, id))
    conn.commit()

def eliminar_contacto(id):
    cursor.execute('DELETE FROM contactos WHERE id = ?', (id,))
    conn.commit()

# Interfaz de usuario en la terminal
def menu():
    while True:
        print("\nAgenda Telefónica")
        print("1. Agregar Contacto")
        print("2. Buscar Contacto")
        print("3. Actualizar Contacto")
        print("4. Eliminar Contacto")
        print("5. Salir")
        opcion = input("Elige una opción: ")

        if opcion == '1':
            nombre = input("Nombre del Contacto: ")
            telefono = input("Teléfono del Contacto: ")
            agregar_contacto(nombre, telefono)

        elif opcion == '2':
            nombre = input("Nombre del Contacto a buscar: ")
            resultados = buscar_contacto(nombre)
            for contacto in resultados:
                print(contacto)

```

```
55          elif opcion == '3':
56              id = int(input("ID del Contacto a actualizar: "))
57              nuevo_nombre = input("Nuevo Nombre: ")
58              nuevo_telefono = input("Nuevo Teléfono: ")
59              actualizar_contacto(id, nuevo_nombre, nuevo_telefono)
60
61          elif opcion == '4':
62              id = int(input("ID del Contacto a eliminar: "))
63              eliminar_contacto(id)
64
65          elif opcion == '5':
66              break
67
68          else:
69              print("Opción inválida, intenta de nuevo.")
70
71  # Ejecutar el programa
72  menu()
73
74  # Cerrar la conexión a la base de datos
75  conn.close()
```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

Práctica 11.

Es conveniente que el lector ejecute este programa y haga modificaciones a las rutinas.

Capítulo 8

Importancia de las Pruebas Unitarias

Las pruebas unitarias son esenciales para cualquier proyecto de desarrollo de software. Su objetivo es validar que cada unidad de software funcione como se espera.

Beneficios Clave.

(1) Detección temprana de errores.

(2) Facilita los cambios y la mantenibilidad.

(3) Mejora del diseño del código.

(4) Actúa como documentación del código.

8.1. Commando `assert`

Ejemplo de Prueba Unitaria.

```
def suma(a, b):
    return a + b

def test_suma():
    try:
        assert suma(2, 3) == 5
        # Código a ejecutar si la verificación es exitosa
        print("La prueba ha sido exitosa.")
    except AssertionError:
        # Código a ejecutar si la verificación falla
        print("La prueba ha fallado.")

test_suma()
```

La instrucción `assert` en Python se utiliza para realizar comprobaciones durante la ejecución de un programa. La línea de código `assert suma(2, 3) == 5` realiza lo siguiente:

(1) **Ejecuta la Función** `suma`: La función `suma` se invoca con los argumentos `2` y `3`. Se espera que esta función realice una operación de suma con estos dos números.

(2) **Verificación con** `assert`: La palabra clave `assert` verifica si el resultado de la llamada a la función `suma(2, 3)` es igual a `5`.

(3) **Comportamiento en Caso de Éxito**: Si la condición `suma(2, 3) == 5` es verdadera, lo que significa que la función `suma` devuelve efectivamente `5` al sumar `2` y `3`, el programa continúa su ejecución sin interrupciones.

(4) **Comportamiento en Caso de Fallo**: Si la condición `suma(2, 3) == 5` es falsa, es decir, si la función `suma` no devuelve `5` al sumar `2` y `3`, Python lanza una excepción `AssertionError`. Este error puede ser capturado y manejado, o permitir que termine la ejecución del programa, dependiendo de cómo esté estructurado el código.

La instrucción `assert` se utiliza comúnmente para la depuración, ya que ayuda a verificar que ciertas condiciones se cumplan en el código. Es una herramienta útil para confirmar que el programa se comporta como se espera en determinadas situaciones, y ayuda a identificar problemas en fases tempranas del desarrollo.

8.2. Biblioteca `unittest`

`unittest` es una biblioteca de pruebas unitarias para el lenguaje de programación Python.

Características Principales.

(1) Estructura de pruebas en clases y métodos.

(2) Variedad de métodos de aserción.

(3) Descubrimiento automático de pruebas.

(4) Aislamiento de pruebas.

Este código Python utiliza el módulo `unittest` para implementar pruebas unitarias:

```
1 import unittest
2
3 class TestSuma(unittest.TestCase):
4     def test_suma(self):
5         self.assertEqual(suma(2, 3), 5)
6
7 if __name__ == '__main__':
8     unittest.main()
```

(1) **Importación de Unittest**: El código comienza importando el módulo `unittest`, que es una biblioteca de pruebas unitarias integrada en Python.

(2) **Definición de la Clase de Prueba**: Se define una clase llamada `TestSuma`, que hereda de `unittest.TestCase`. Esta clase se utiliza para agrupar las pruebas relacionadas con la función de suma.

(3) **Método de Prueba**: Dentro de la clase, se define un método llamado `test_suma`. Este método utiliza `self.assertEqual` para afirmar que el resultado de llamar a la función `suma` con los argumentos `2` y `3` es igual a `5`.

(4) **Ejecución Condicional**: Al final del código, hay una verificación condicional `if __name__ == '__main__':`. Esto asegura que `unittest.main()` se ejecute solo si el script se ejecuta como programa principal.

(5) **Llamada a Unittest**: `unittest.main()` se llama para iniciar la ejecución de las pruebas. Esto ejecutará automáticamente todos los métodos en la clase `TestSuma` que empiecen con la palabra `test`.

8.3. Biblioteca `pytest`

Pytest es una biblioteca de pruebas para el lenguaje de programación Python que se destaca por su simplicidad, flexibilidad y capacidad de escalabilidad. Con un enfoque en escribir pruebas mínimas mientras se obtiene la máxima información cuando fallan, *pytest* facilita tanto para desarrolladores como para QA la escritura de pruebas unitarias y de integración. Su sintaxis sencilla y el uso de aserciones de Python puro

hacen que las pruebas sean legibles y mantenibles. Además, *pytest* es conocido por su capacidad de ejecutar pruebas en paralelo, lo que mejora significativamente la velocidad de los ciclos de pruebas. La biblioteca también admite fixtures que proporcionan una forma poderosa de configurar datos, estado y entorno para las pruebas, haciendo que *pytest* sea una herramienta preferida para pruebas automatizadas en proyectos de Python de todos los tamaños.
Procedimiento para uso de la biblioteca `pytest`.

Primero, se define una función simple llamada `suma` en Python:

```
# archivo: my_math_module.py

def suma(a, b):
    return a + b
```

Script de Prueba Utilizando Pytest.

A continuación, se escribe un script de prueba para la función `suma` usando `pytest`:

```
# archivo: test_my_math_module.py

import my_math_module

def test_suma():
    assert my_math_module.suma(2, 3) == 5
    assert my_math_module.suma(0, 0) == 0
    assert my_math_module.suma(-1, 1) == 0
```

Ejecución de la Prueba y Resultados.

Para ejecutar la prueba, se utiliza el siguiente comando en la línea de comandos:

```
pytest test_my_math_module.py
```

Los resultados de la ejecución de la prueba son los siguientes:

```
============================================================ test session starts
    ============================================================
platform linux -- Python 3.8.10, pytest-7.4.3, pluggy-1.3.0
collected 1 item

test_my_math_module.py .
                                          [100%]

============================================================ 1 passed in 0.01s
    ============================================================
```

Este resultado indica que la prueba se ejecutó con éxito, y que la función `suma` pasó la prueba definida en el script `test_my_mathi_module.py`.

8.4. Estrategias de Pruebas

Las estrategias de pruebas incluyen diferentes enfoques y técnicas:

8.4.1. Test-Driven Development

El **Desarrollo Guiado por Pruebas (TDD, por sus siglas en inglés Test-Driven Development)** es una metodología de desarrollo de software que involucra iteraciones cortas de desarrollo en las que se escriben primero las pruebas, antes del código de producción. En TDD, un desarrollador comienza escribiendo un caso de prueba que define una mejora deseada o una nueva funcionalidad. Luego, se escribe el código mínimo necesario para pasar la prueba, seguido por un proceso de refactorización para mejorar el código mientras se mantiene su funcionalidad. TDD promueve un diseño de software más limpio y una mayor cobertura de pruebas.

En el enfoque de Test-Driven Development (TDD), se comienza escribiendo pruebas antes de desarrollar el código de la función. Este método enfatiza la importancia de las pruebas en el proceso de desarrollo de software. El siguiente es un ejemplo sencillo que demuestra TDD para una función de suma en Python.

```
# archivo: my_math.py
def suma(a, b):
    pass  # Implementación inicial vacía

# archivo: test_my_math.py
import unittest
import my_math

class TestSuma(unittest.TestCase):
    def test_suma(self):
        self.assertEqual(my_math.suma(2, 3), 5)

if __name__ == '__main__':
    unittest.main()
```

Inicialmente, la función ‘suma‘ no tiene una implementación funcional (se utiliza ‘pass‘). Se escribe primero un caso de prueba que falla, luego se desarrolla la función ‘suma‘ hasta que la prueba es exitosa.

8.4.2. Behavior-Driven Development

Desarrollo Guiado por Comportamiento (BDD, por sus siglas en inglés Behavior-Driven Development) es una técnica de desarrollo de software que combina los principios de TDD con ideas del diseño orientado a objetos y análisis de dominio. BDD se centra en obtener un claro entendimiento de los comportamientos deseados del software a través de la colaboración entre desarrolladores, QA y partes interesadas no técnicas. Utiliza un lenguaje específico de dominio (DSL) para describir el comportamiento y los resultados esperados, facilitando la comunicación entre todos los involucrados en el proyecto.

Behavior-Driven Development (BDD) es un enfoque de desarrollo de software que combina las técnicas de TDD con un lenguaje específico de dominio (DSL). BDD facilita la colaboración entre desarrolladores, QA y stakeholders no técnicos. El siguiente es un ejemplo en Python utilizando BDD:

```
# archivo: suma.feature
Feature: Suma de dos números
  Scenario: Suma de dos enteros
    Given dos enteros 2 y 3
    When los sumo
    Then el resultado debería ser 5

# archivo: steps/suma_steps.py
from behave import given, when, then
import my_math

@given('dos enteros {a:d} y {b:d}')
def step_impl(context, a, b):
    context.a = a
    context.b = b

@when('los sumo')
def step_impl(context):
    context.result = my_math.suma(context.a, context.b)

@then('el resultado debería ser {expected:d}')
def step_impl(context, expected):
    assert context.result == expected
```

En este ejemplo, se define un comportamiento deseado en un lenguaje natural (en el archivo ‘.feature‘) y luego se implementan los pasos en Python para cumplir con ese comportamiento.

8.4.3. Pruebas de Integración

Las **Pruebas de Integración** son un nivel de pruebas de software en el que se combinan unidades de código individual y se prueban como grupo. El objetivo principal de las pruebas de integración es detectar fallos en la interacción entre unidades integradas. Estas pruebas son fundamentales para garantizar que los diferentes módulos o servicios del software funcionen juntos correctamente. A menudo, las pruebas de integración siguen después de las pruebas unitarias y antes de las pruebas de sistema.

Las Pruebas de Integración se centran en combinar unidades de código y probarlas como grupo para asegurarse de que interactúan correctamente. Aquí hay un ejemplo de pruebas de integración en Python:

```
# archivo: my_services.py
def servicio_a(data):
    return data.upper()

def servicio_b(data):
    return data.lower()

# archivo: test_integration.py
import unittest
import my_services

class TestIntegration(unittest.TestCase):
    def test_integration_a_b(self):
        result = my_services.servicio_b(my_services.servicio_a("Texto"))
        self.assertEqual(result, "texto")

if __name__ == '__main__':
    unittest.main()
```

Este ejemplo demuestra la integración de dos servicios, ‘servicio_a‘ y ‘servicio_b‘, donde se prueba que funcionan correctamente cuando se usan en conjunto.

8.4.4. Pruebas de Regresión

Las **Pruebas de Regresión** son un tipo de pruebas que se realizan para verificar que un cambio reciente en el código no haya afectado adversamente las características existentes. Estas pruebas son cruciales para mantener la continuidad y la estabilidad del software, especialmente en entornos de desarrollo continuo. Las pruebas de regresión aseguran que las nuevas modificaciones o correcciones de errores no introduzcan nuevos fallos en partes previamente probadas y funcionales del software.

Las Pruebas de Regresión aseguran que los cambios recientes en el código no afecten negativamente a las funcionalidades existentes. A continuación, se presenta un ejemplo de pruebas de regresión en Python:

```
# archivo: my_module.py
def funcion_existente():
    return "algo"

# archivo: test_regression.py
import unittest
import my_module

class TestRegression(unittest.TestCase):
    def test_funcion_existente(self):
        self.assertEqual(my_module.funcion_existente(), "algo")

if __name__ == '__main__':
    unittest.main()
```

Este ejemplo incluye una prueba para una función existente para verificar que sigue funcionando como se espera después de realizar cambios en el código.

8.5. Caso Práctico

A continuación exponemos cuatro programas los cuales consisten establecen:

Definición de la Aplicación y las Funciones (TDD)

Definición de la Aplicación y las Funciones (TDD): En este primer programa, se aplica el enfoque de Desarrollo Guiado por Pruebas (TDD) para la creación de una serie de funciones matemáticas básicas, como suma, resta, multiplicación y división. Siguiendo los principios de TDD, se escribe inicialmente el esqueleto de las funciones sin implementación detallada. Este enfoque se centra en desarrollar soluciones que satisfagan las necesidades específicas dictadas por los casos de prueba, fomentando un diseño de software más eficiente y con un mayor enfoque en los requerimientos.

```python
# archivo: math_operations.py
def suma(a, b):
    return a + b

def resta(a, b):
    return a - b

def multiplicacion(a, b):
    return a * b

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

Pruebas Unitarias (TDD)

Pruebas Unitarias (TDD): El este segundo programa consiste en un conjunto de pruebas unitarias para las funciones matemáticas definidas en el primer programa. Cada función matemática tiene una prueba correspondiente que valida su correcto funcionamiento. Por ejemplo, para la función de suma, se comprueba si el resultado de sumar dos números específicos es el esperado. Estas pruebas unitarias son esenciales en TDD y ayudan a garantizar que cada componente del software funcione correctamente de manera aislada.

```python
# archivo: test_math_operations.py
import unittest
import math_operations

class TestMathOperations(unittest.TestCase):
    def test_suma(self):
        self.assertEqual(math_operations.suma(2, 3), 5)

    def test_resta(self):
        self.assertEqual(math_operations.resta(10, 5), 5)

    def test_multiplicacion(self):
        self.assertEqual(math_operations.multiplicacion(3, 3), 9)

    def test_division(self):
        self.assertEqual(math_operations.division(10, 2), 5)
        with self.assertRaises(ValueError):
            math_operations.division(10, 0)

if __name__ == '__main__':
    unittest.main()
```

Definición del Comportamiento (BDD)

Definición del Comportamiento (BDD): En este tercer programa, se utiliza el enfoque de Desarrollo Guiado por Comportamiento (BDD) para describir los escenarios de uso de la calculadora mediante un

lenguaje específico de dominio (DSL). Esto incluye la descripción de comportamientos esperados como "sumar dos números."[en] un formato que es comprensible tanto para los desarrolladores como para los interesados no técnicos. Luego, se implementan estos comportamientos en pasos concretos utilizando Python, vinculando las descripciones de alto nivel con el código de bajo nivel.

```
# archivo: suma.feature
Feature: Calculadora básica
  Scenario: Sumar dos números
    Given tengo los números 2 y 3
    When los sumo
    Then el resultado debería ser 5

# Otros escenarios para resta, multiplicación y división

# archivo: steps/calculator_steps.py
from behave import given, when, then
import math_operations

@given('tengo los números {a:d} y {b:d}')
def step_impl(context, a, b):
    context.a = a
    context.b = b

@when('los sumo')
def step_impl(context):
    context.result = math_operations.suma(context.a, context.b)

@then('el resultado debería ser {expected:d}')
def step_impl(context, expected):
    assert context.result == expected

# Pasos adicionales para resta, multiplicación y división
```

Pruebas de Integración y Regresión

Pruebas de Integración y Regresión: Finalmente, en este cuarto programa se enfoca en las pruebas de integración y regresión. Las pruebas de integración se diseñan para asegurar que las diferentes funciones matemáticas trabajen bien juntas, mientras que las pruebas de regresión verifican que los cambios recientes en el código no hayan roto ninguna funcionalidad existente. Estas pruebas son cruciales para mantener la integridad del sistema a medida que evoluciona y se expande.

```
# archivo: test_integration_and_regression.py
import unittest
import math_operations

class TestIntegrationAndRegression(unittest.TestCase):
    # Aquí se pueden agregar pruebas que combinen múltiples operaciones
    # y verifiquen la correcta integración y no-regresión de las funcionalidades

if __name__ == '__main__':
    unittest.main()
```

Capítulo 9

Automatización de Tareas

9.1. Procesamiento de Archivos en Lote

En esta sección, se abordará el proceso de automatización del procesamiento de archivos en lote. Se explorarán las técnicas y herramientas utilizadas para realizar tareas repetitivas en un conjunto de archivos de manera eficiente.

A continuación, se muestra un ejemplo en Python de procesamiento de archivos en lote que renombra archivos en un directorio:

```
import os

# Directorio con los archivos a procesar
directorio = '/ruta/al/directorio'

# Iterar a través de los archivos en el directorio
for nombre_archivo in os.listdir(directorio):
    if nombre_archivo.endswith('.txt'):
        # Realizar alguna acción en los archivos .txt, como cambiarles el nombre
        nuevo_nombre = nombre_archivo.replace('.txt', '_procesado.txt')
        os.rename(os.path.join(directorio, nombre_archivo), os.path.join(directorio,
    nuevo_nombre))
```

9.2. Automatización de Envío de Correos Electrónicos

En esta subsección, se discutirá cómo automatizar el envío de correos electrónicos. Se analizarán las herramientas y métodos disponibles para enviar mensajes de correo electrónico de forma programada y personalizada.

A continuación, se muestra un ejemplo en Python de automatización de envío de correos electrónicos utilizando la biblioteca ‘smtplib‘:

```
import smtplib
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart

# Configurar el servidor SMTP
smtp_server = 'smtp.example.com'
smtp_port = 587
smtp_user = 'tu_usuario'
smtp_pass = 'tu_contraseña'

# Crear el mensaje
mensaje = MIMEMultipart()
mensaje['From'] = 'remite@example.com'
mensaje['To'] = 'destinatario@example.com'
mensaje['Subject'] = 'Asunto del correo'
```

```
16
17 cuerpo_correo = 'Este es el cuerpo del correo.'
18 mensaje.attach(MIMEText(cuerpo_correo, 'plain'))
19
20 # Iniciar la conexión SMTP y enviar el correo
21 with smtplib.SMTP(smtp_server, smtp_port) as servidor:
22     servidor.starttls()
23     servidor.login(smtp_user, smtp_pass)
24     servidor.sendmail(mensaje['From'], mensaje['To'], mensaje.as_string())
25
26 print('Correo enviado con éxito.')
```

9.3. Tareas de Administración del Sistema

Esta subsección se enfocará en la automatización de tareas de administración del sistema. Se explorarán las técnicas para gestionar y mantener un sistema informático de manera eficiente y sin intervención manual constante.

A continuación, se muestra un ejemplo en Python de una tarea de administración del sistema que realiza una copia de seguridad de una base de datos:

```
1  import subprocess
2  import datetime
3
4  # Nombre de la base de datos
5  nombre_db = 'mi_base_de_datos'
6
7  # Nombre del archivo de respaldo con marca de tiempo
8  fecha_actual = datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
9  archivo_respaldo = f'{nombre_db}_respaldo_{fecha_actual}.sql'
10
11 # Comando para realizar la copia de seguridad utilizando mysqldump (MySQL)
12 comando = f'mysqldump -u usuario -p contraseña {nombre_db} > {archivo_respaldo}'
13
14 # Ejecutar el comando en la línea de comandos
15 subprocess.run(comando, shell=True)
16
17 print(f'Copia de seguridad de {nombre_db} creada en {archivo_respaldo}')
```

9.4. Caso Práctico

Este documento describe las tareas realizadas por un programa de automatización desarrollado en Python. Las tareas incluyen procesamiento de archivos, envío de correos electrónicos y administración del sistema.

Tareas de Automatización

Procesar archivos en un directorio

El programa cambiará la extensión de los archivos con extensión `.txt` a `.processed` en un directorio especificado.

Enviar un correo electrónico

El programa notificará vía correo electrónico cuando el procesamiento de archivos esté completo.

Ejecutar una tarea de administración del sistema

El programa listará los procesos en ejecución en el sistema y guardará esta lista en un archivo.

Configuración Requerida

Para este programa, será necesario ajustar las configuraciones del servidor de correo y las rutas de los archivos según el entorno en el que se ejecute.

```
import os
import smtplib
import subprocess
from email.mime.text import MIMEText

def procesar_archivos(directorio):
    for filename in os.listdir(directorio):
        if filename.endswith('.txt'):
            nuevo_nombre = filename.replace('.txt', '.processed')
            os.rename(os.path.join(directorio, filename), os.path.join(directorio,
    nuevo_nombre))
            print(f"Archivo procesado: {filename} -> {nuevo_nombre}")

def enviar_correo(sender_email, receiver_email, password, smtp_server, port):
    mensaje = MIMEText('El procesamiento de archivos ha sido completado.')
    mensaje['Subject'] = 'Notificación de Procesamiento de Archivos'
    mensaje['From'] = sender_email
    mensaje['To'] = receiver_email

    try:
        server = smtplib.SMTP(smtp_server, port)
        server.starttls()
        server.login(sender_email, password)
        server.sendmail(sender_email, receiver_email, mensaje.as_string())
        server.quit()
        print('Correo enviado exitosamente')
    except Exception as e:
        print('Error al enviar correo: ', e)

def listar_procesos():
    proceso = subprocess.Popen(['ps', '-aux'], stdout=subprocess.PIPE, stderr=subprocess.
    PIPE)
    stdout, stderr = proceso.communicate()

    if proceso.returncode == 0:
        with open('lista_procesos.txt', 'w') as file:
            file.write(stdout.decode())
        print('Lista de procesos guardada en "lista_procesos.txt"')
    else:
        print('Error al listar procesos: ', stderr.decode())

# Configura estos valores
directorio = 'path/to/directory'
sender_email = 'tu_email@example.com'
receiver_email = 'destinatario@example.com'
password = 'tu_contraseña'
smtp_server = 'smtp.example.com'
port = 587  # Cambiar si es necesario

# Ejecución del programa
procesar_archivos(directorio)
enviar_correo(sender_email, receiver_email, password, smtp_server, port)
listar_procesos()
```

Requerimientos del Sistema

Sistema Operativo

El programa está diseñado para ser ejecutado en **Fedora 39**. Se asume que el usuario tiene este sistema operativo instalado y configurado adecuadamente.

Instalación de Python

Fedora 39 generalmente viene con Python preinstalado. Para verificar la instalación y la versión de Python, ejecute el siguiente comando en la terminal:

```
python3 --version
```

Si Python no está instalado, puede instalarlo usando el siguiente comando:

```
sudo dnf install python3
```

Configuración del Entorno de Correo Electrónico

Para la función de envío de correos electrónicos, se requiere una configuración específica del servidor SMTP. Se deben proporcionar los detalles del servidor SMTP, el puerto, el correo electrónico del remitente y la contraseña.

Instalación de Bibliotecas Requeridas

Instale las bibliotecas necesarias para el envío de correos electrónicos y la ejecución de comandos del sistema. Ejecute el siguiente comando:

```
pip3 install smtplib
```

Configuración de Archivos y Permisos

Asegúrese de tener los permisos necesarios para leer y escribir en los directorios utilizados por el programa y para ejecutar comandos del sistema.

Permisos de Directorio

Configure los permisos del directorio donde el programa procesará los archivos. Por ejemplo:

```
chmod 755 /path/to/directory
```

Permisos de Ejecución

Para ejecutar comandos del sistema, el usuario debe tener los permisos adecuados. Esto puede requerir ejecutar el script de Python con privilegios de superusuario.

Ejecución del Programa

Una vez configurado el entorno, puede ejecutar el programa usando el comando:

```
python3 script_automatizacion.py
```

Capítulo 10

Seguridad en Python

En la era digital actual, la seguridad de las aplicaciones es más crucial que nunca. Python, siendo uno de los lenguajes de programación más populares y versátiles, se utiliza ampliamente en diversas aplicaciones, desde desarrollo web hasta ciencia de datos y automatización. Sin embargo, su popularidad también lo convierte en un objetivo común para ataques maliciosos. Este capítulo se dedica a explorar las prácticas, técnicas y estrategias esenciales para asegurar aplicaciones escritas en Python. Comenzaremos con las mejores prácticas generales de seguridad, seguido de una discusión sobre cómo evitar vulnerabilidades comunes. Finalmente, nos enfocaremos en métodos específicos para fortalecer la seguridad de las aplicaciones Python. A través de este capítulo, se proporcionarán ejemplos de código para ilustrar cómo implementar estas prácticas de seguridad en aplicaciones reales.

10.1. Mejores Prácticas de Seguridad

La validación de entradas es una de las mejores prácticas más importantes. El siguiente fragmento de código muestra cómo validar una entrada de usuario antes de procesarla:

```
def validar_entrada(entrada):
    if not entrada.isnumeric():
        raise ValueError("La entrada debe ser numérica")
    # Procesamiento adicional de la entrada
```

10.2. Evitar Vulnerabilidades Comunes

Una vulnerabilidad común en Python es la inyección SQL. El siguiente código utiliza SQLAlchemy, un ORM para Python, para evitar este tipo de vulnerabilidad:

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

engine = create_engine('sqlite:///database.db')
Session = sessionmaker(bind=engine)

session = Session()

# Uso seguro de parámetros
query = session.query(MyModel).filter(MyModel.column == param)
```

10.3. Protección de Aplicaciones

Para proteger una aplicación Python, es crucial implementar una autenticación segura. El siguiente ejemplo utiliza Flask y su extensión Flask-Login para gestionar la autenticación de usuarios:

```python
from flask import Flask, request, redirect, render_template
from flask_login import LoginManager, UserMixin, login_user

app = Flask(__name__)
login_manager = LoginManager()
login_manager.init_app(app)

# Definición de la clase de usuario
class User(UserMixin):
    # Implementación de la clase usuario

@login_manager.user_loader
def load_user(user_id):
    # Cargar usuario desde la base de datos
    return User.get(user_id)

@app.route('/login', methods=['GET', 'POST'])
def login():
    # Lógica de autenticación del usuario
    return render_template('login.html')
```

10.4. Caso Práctico

El programa incluye varias funcionalidades clave en el contexto de la seguridad de aplicaciones:

Hashing de Contraseñas

El programa utiliza la biblioteca `bcrypt` para:

(1) Generar un hash seguro de las contraseñas.

(2) Verificar contraseñas contra su hash almacenado.

Esto representa una práctica recomendada para el almacenamiento seguro de contraseñas.

Uso Seguro de Bases de Datos

Mediante el uso de `sqlite3`, el programa:

(1) Interactúa con una base de datos SQLite.

(2) Implementa consultas parametrizadas para prevenir inyecciones SQL.

Autenticación Basada en Token

El programa:

(1) Genera un token de seguridad usando `secrets`.

(2) Controla el acceso a funciones críticas mediante la validación de este token.

Inicialización y Manejo de la Base de Datos

El programa gestiona la base de datos realizando:

(1) Creación de la base de datos y las tablas necesarias si no existen.

(2) Inserción y recuperación segura de usuarios en la base de datos.

```python
import bcrypt
import sqlite3
import secrets

# Función para crear un hash de una contraseña
def hash_password(password):
    return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt())

# Función para verificar una contraseña contra un hash
def check_password(password, hashed):
    return bcrypt.checkpw(password.encode('utf-8'), hashed)

# Función para conectar a la base de datos y crear una tabla si no existe
def inicializar_base_de_datos():
    conn = sqlite3.connect('mi_base_de_datos.db')
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS usuarios (
            id INTEGER PRIMARY KEY,
            nombre TEXT,
            password_hash TEXT
        )
    ''')
    conn.commit()
    conn.close()

# Función segura para consultar la base de datos
def consulta_segura(usuario):
    conn = sqlite3.connect('mi_base_de_datos.db')
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM usuarios WHERE nombre = ?", (usuario,))
    resultado = cursor.fetchall()
    conn.close()
    return resultado

# Función que genera un token de seguridad
def generar_token():
    return secrets.token_hex(16)

# Función protegida por un token
def funcion_protegida(token_usuario):
    if token_usuario == TOKEN_SECRETO:
        print("Acceso concedido")
        # Lógica de la función protegida
    else:
        print("Acceso denegado")

# Inicializar base de datos
inicializar_base_de_datos()

# Crear un usuario de ejemplo
usuario = 'usuario_prueba'
contraseña = 'contraseña_segura'
hashed_password = hash_password(contraseña)

# Guardar usuario en la base de datos
conn = sqlite3.connect('mi_base_de_datos.db')
cursor = conn.cursor()
cursor.execute("INSERT INTO usuarios (nombre, password_hash) VALUES (?, ?)", (usuario,
    hashed_password))
conn.commit()
conn.close()

# Verificar contraseña
usuario_encontrado = consulta_segura(usuario)
if usuario_encontrado:
    verificacion = check_password('contraseña_segura', usuario_encontrado[0][2])
    print("Verificación de la contraseña:", verificacion)
```

```
68
69 # Generar y almacenar un token de seguridad
70 TOKEN_SECRETO = generar_token()
71 print("Token de seguridad:", TOKEN_SECRETO)
72
73 # Intentar acceder a la función protegida
74 funcion_protegida(TOKEN_SECRETO)  # Acceso concedido
75 funcion_protegida("token_incorrecto")  # Acceso denegado
```

Capítulo 11

Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) es un paradigma de programación que se basa en el concepto de objetos. Los objetos son instancias de clases, y las clases son plantillas que definen la estructura y el comportamiento de los objetos. La POO se basa en cuatro conceptos fundamentales:

11.1. Definiciones Básicas

11.1.1. Objetos

Un objeto es una instancia concreta de una clase y representa una entidad del mundo real. Los objetos tienen atributos (variables) que almacenan datos y métodos (funciones) que realizan acciones. Por ejemplo, en un sistema de gestión de bibliotecas, un objeto Libro podría tener atributos como título, autor y año de publicación, y métodos como prestar() y devolver().

11.1.2. Clases

Una clase es una plantilla que define la estructura y el comportamiento de los objetos. En una clase se especifican los atributos y los métodos que los objetos de esa clase tendrán. Por ejemplo, la clase Libro podría definir que todos los libros tienen un título y un autor como atributos, y métodos para prestar y devolver.

11.1.3. Herencia

La herencia es un mecanismo que permite que una clase herede atributos y métodos de otra clase. La clase que hereda se llama subclase o clase derivada, y la clase de la que hereda se llama superclase o clase base. La herencia permite la reutilización de código y la creación de jerarquías de clases. Por ejemplo, una subclase de Libro podría ser LibroDigital, que hereda los atributos de Libro pero tiene métodos adicionales para la descarga de libros digitales.

11.1.4. Polimorfismo

El polimorfismo permite que objetos de diferentes clases sean tratados de la misma manera. Esto significa que un objeto puede responder a un método de manera diferente según su clase. El polimorfismo permite la flexibilidad en el diseño y la extensión de programas. Por ejemplo, tanto un Libro como un LibroDigital pueden tener un método mostrarResumen(), pero cada uno lo implementará de manera diferente.

11.1.5. Clases y Objetos

En este capítulo, profundizaremos en los conceptos de clases y objetos en la Programación Orientada a Objetos.

Para definir una clase en Python, utilizamos la palabra clave `class`, seguida del nombre de la clase y dos puntos. Por ejemplo:

```
class Libro:
    # Atributos
    titulo = ""
    autor = ""
    ano_publicacion = 0

    # Metodos
    def prestar(self):
        print("El libro ha sido prestado")

    def devolver(self):
        print("El libro ha sido devuelto")
```

En este ejemplo, hemos definido una clase llamada `Libro`, que tiene atributos como `titulo`, `autor` y `ano_publicacion`, así como métodos como `prestar()` y `devolver()`.

11.1.6. Creación de Objetos

Para crear un objeto a partir de una clase, utilizamos la sintaxis de instanciación. Por ejemplo:

```
# Crear un objeto de la clase Libro
mi_libro = Libro()

# Asignar valores a los atributos
mi_libro.titulo = "La Odisea"
mi_libro.autor = "Homero"
mi_libro.ano_publicacion = 800 a.C.
```

Hemos creado un objeto `mi_libro` a partir de la clase `Libro` y hemos asignado valores a sus atributos.

11.1.7. Acceso a Atributos y Métodos

Para acceder a los atributos y métodos de un objeto, utilizamos la notación de punto. Por ejemplo:

```
# Acceder a los atributos
print("Título:", mi_libro.titulo)
print("Autor:", mi_libro.autor)
print("Año de Publicación:", mi_libro.ano_publicacion)

# Llamar a métodos
mi_libro.prestar()
mi_libro.devolver()
```

En este código, hemos accedido a los atributos y llamado a los métodos del objeto `mi_libro`.

11.2. Herencia y Polimorfismo

En este capítulo, exploraremos los conceptos de herencia y polimorfismo en la Programación Orientada a Objetos.

11.2.1. Herencia

La herencia permite que una clase herede atributos y métodos de otra clase. Para definir una subclase, utilizamos la palabra clave `class`, seguida del nombre de la subclase y el nombre de la superclase entre paréntesis. Por ejemplo:

```
class LibroDigital(Libro):
    # Atributos adicionales
    formato = ""
```

```
4
5       # Metodo adicional
6       def descargar(self):
7           print("El libro digital se ha descargado")
```

Hemos creado una subclase `LibroDigital` que hereda de la superclase `Libro` y agrega atributos y métodos adicionales.

11.2.2. Polimorfismo

El polimorfismo nos permite tratar objetos de diferentes clases de manera uniforme. Por ejemplo, podemos crear una lista de objetos y llamar a un método común en todos ellos:

```
1 libro1 = Libro()
2 libro2 = LibroDigital()
3
4 libros = [libro1, libro2]
5
6 for libro in libros:
7     libro.prestar()
```

En este código, tanto `libro1` como `libro2` responden al método `prestar()` a pesar de ser de diferentes clases.

11.2.3. Encapsulación y Abstracción

En este capítulo, exploraremos los conceptos de encapsulación y abstracción en la Programación Orientada a Objetos.

11.2.4. Encapsulación

La encapsulación es el principio de ocultar los detalles internos de un objeto y proporcionar una interfaz pública para interactuar con él. En Python, utilizamos métodos para acceder y modificar los atributos de un objeto. Por ejemplo:

```
1  class Libro:
2      def __init__(self, titulo, autor):
3          self.titulo = titulo
4          self.autor = autor
5          self.__prestado = False  # Atributo privado
6
7      def prestar(self):
8          if not self.__prestado:
9              print("El libro ha sido prestado")
10             self.__prestado = True
11         else:
12             print("El libro ya está prestado")
13
14     def devolver(self):
15         if self.__prestado:
16             print("El libro ha sido devuelto")
17             self.__prestado = False
18         else:
19             print("El libro no está prestado")
20
21 # Crear un objeto de la clase Libro
22 mi_libro = Libro("La Odisea", "Homero")
23
24 # Acceder a atributos y métodos
25 print("Título:", mi_libro.titulo)
26 mi_libro.prestar()
27 mi_libro.devolver()
```

En este ejemplo, hemos encapsulado el atributo `__prestado` y proporcionado métodos para acceder y modificar su estado.

11.2.5. Abstracción

La abstracción es el proceso de simplificar la complejidad de un objeto al enfocarse en los aspectos relevantes y ocultar los detalles innecesarios. Las clases y los objetos son ejemplos de abstracción, ya que representan entidades del mundo real de manera simplificada.

La Programación Orientada a Objetos (POO) es un paradigma poderoso que se basa en el concepto de objetos, clases, herencia, polimorfismo, encapsulación y abstracción. Estos conceptos permiten la creación de programas más estructurados, modularizados y reutilizables. Al comprender y aplicar estos principios, los programadores pueden desarrollar software más eficiente y mantenible.

En los capítulos anteriores, hemos explorado los conceptos básicos de POO, la creación de clases y objetos, la herencia y el polimorfismo, así como la encapsulación y la abstracción. Estos son fundamentos importantes que te ayudarán a construir una base sólida en el mundo de la programación orientada a objetos.

La POO es una herramienta poderosa en el arsenal de un programador, y su comprensión puede abrir la puerta a la creación de aplicaciones complejas y robustas. Continúa practicando y explorando este paradigma para convertirte en un desarrollador de software más hábil y versátil.

11.3. Caso Práctico

En el ámbito de la programación orientada a objetos (POO), los sistemas que gestionan entidades complejas y sus interacciones son ejemplos clásicos que demuestran la eficacia de este paradigma. Este documento presenta un sistema de gestión de concesionario de automóviles implementado en Python, que sirve como un ejemplo práctico de los conceptos fundamentales de la POO, como clases, objetos, herencia y encapsulación.

El programa está estructurado en torno a dos clases principales: `Vehiculo` y `Concesionario`. La clase `Vehiculo` define las propiedades y métodos básicos de un vehículo, como su marca, modelo y año de fabricación. Por otro lado, la clase `Concesionario` actúa como un contenedor para múltiples instancias de `Vehiculo`, permitiendo la gestión del inventario de vehículos de un concesionario. Este enfoque no solo facilita la representación de entidades del mundo real dentro del programa sino que también permite una gestión eficiente y organizada de los datos.

El sistema ofrece funcionalidades básicas como agregar vehículos al inventario del concesionario y mostrar detalles de los vehículos disponibles. A través de este sistema, se ilustra cómo la POO puede ser utilizada para construir programas estructurados, modulares y fácilmente escalables, lo cual es esencial en el desarrollo de software moderno.

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

    def mostrar_detalle(self):
        print(f"Marca: {self.marca}, Modelo: {self.modelo}, Año: {self.año}")

class Concesionario:
    def __init__(self):
        self.vehiculos = []

    def agregar_vehiculo(self, vehiculo):
        self.vehiculos.append(vehiculo)

    def mostrar_inventario(self):
        if not self.vehiculos:
            print("No hay vehículos en el inventario.")
```

```
20          for vehiculo in self.vehiculos:
21              vehiculo.mostrar_detalle()
22
23  # Crear vehículos
24  vehiculo1 = Vehiculo("Toyota", "Corolla", 2020)
25  vehiculo2 = Vehiculo("Ford", "Mustang", 2021)
26
27  # Crear un concesionario y agregar vehículos
28  mi_concesionario = Concesionario()
29  mi_concesionario.agregar_vehiculo(vehiculo1)
30  mi_concesionario.agregar_vehiculo(vehiculo2)
31
32  # Mostrar los vehículos en el concesionario
33  mi_concesionario.mostrar_inventario()
```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖵

Práctica 12.

Es conveniente que el lector ejecute el programa referido.

Capítulo 12

Machine Learning con Python

El Machine Learning (ML) con Python es un campo vibrante que combina análisis estadístico, minería de datos, reconocimiento de patrones y programación.

12.1. Definición de Machine Learning

El Machine Learning es un subcampo de la inteligencia artificial que se enfoca en desarrollar algoritmos para permitir que las computadoras aprendan de los datos. Los tipos principales incluyen:

(1) **Aprendizaje Supervisado**: Aprender una función que mapea entradas a salidas basado en ejemplos etiquetados.

(2) **Aprendizaje No Supervisado**: Modelar la estructura de los datos para aprender sobre ellos sin etiquetas.

12.2. Ventajas de Python en Machine Learning

Python es un lenguaje recomendable para recrear Machine Learning debido a:

(1) **Sintaxis Clara y Legible**: Facilita la escritura y comprensión de códigos complejos.

(2) **Bibliotecas Poderosas**: Como `Scikit-learn`, `Pandas`, `NumPy` y `Matplotlib`.

(3) **Comunidad y Soporte**: Una comunidad amplia y activa.

(4) **Flexibilidad y Eficiencia**: Integra diferentes tipos de datos y tecnologías.

(5) **Popularidad en Investigación y Industria**: Ampliamente utilizado en academias e industrias.

12.3. Machine Learning y `Scikit-learn`

El programa utiliza Scikit-learn para clasificar datos utilizando el conjunto de datos Iris y un clasificador de Árbol de Decisión. A continuación se presenta una explicación detallada de cada parte del código.

Ejemplo 12.

```
#!/usr/bin/env python3
# Importar las bibliotecas necesarias
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.metrics import accuracy_score

# Cargar el conjunto de datos Iris
iris = load_iris()
X = iris.data
y = iris.target

# Dividir el conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Crear el modelo de Árbol de Decisión
clf = DecisionTreeClassifier()

# Entrenar el modelo con el conjunto de datos de entrenamiento
clf.fit(X_train, y_train)

# Realizar predicciones en el conjunto de datos de prueba
y_pred = clf.predict(X_test)

# Calcular la precisión del modelo
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

```

```
./machine_learning.py
Accuracy: 1.0

```

El lector puede bajar este programa desde el siguiente vínculo:

12.4. Desglose del Código

Importación de Bibliotecas.

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
```

(1) `sklearn.datasets`: Para cargar conjuntos de datos.

(2) `sklearn.model_selection`: Herramientas para dividir los datos.

(3) `sklearn.tree`: Algoritmos basados en árboles.

(4) `sklearn.metrics`: Métricas de evaluación.

Carga del Conjunto de Datos Iris.

```
iris = load_iris()
X = iris.data
y = iris.target
```

Carga las características y etiquetas del conjunto de datos Iris.

Muestra del Conjunto de Datos Iris.

```
Sepal Length, Sepal Width, Petal Length, Petal Width, Species
5.1, 3.5, 1.4, 0.2, setosa
4.9, 3.0, 1.4, 0.2, setosa
...
```

División de Datos en Entrenamiento y Prueba.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

Divide el conjunto de datos en un conjunto de entrenamiento (70 %) y un conjunto de prueba (30 %).

Creación y Entrenamiento del Modelo.

```
clf = DecisionTreeClassifier()
clf.fit(X_train, y_train)
```

Crea y entrena un modelo de Árbol de Decisión.

Realizar Predicciones.

```
y_pred = clf.predict(X_test)
```

Utiliza el modelo para hacer predicciones sobre el conjunto de prueba.

Evaluación del Modelo.

```
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")
```

Calcula y muestra la precisión del modelo.

Ejemplo de salida del programa después de realizar la clasificación:

```
Accuracy: 0.955
```

```
#!/usr/bin/env python3

import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Cargar el conjunto de datos Iris
iris = load_iris()
X = iris.data
y = iris.target

# Dividir el conjunto de datos en un conjunto de entrenamiento y un conjunto de prueba
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Crear el modelo de Árbol de Decisión
clf = DecisionTreeClassifier()

# Entrenar el modelo con el conjunto de datos de entrenamiento
clf.fit(X_train, y_train)

# Realizar predicciones en el conjunto de datos de prueba
y_pred = clf.predict(X_test)

# Calcular la precisión del modelo
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy}")

# Graficar los datos de entrenamiento y prueba
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap='winter', edgecolor='k', label="
    Entrenamiento")
plt.xlabel('Sepal Length')
plt.ylabel('Sepal Width')
plt.title('Datos de Entrenamiento')
plt.legend()
```

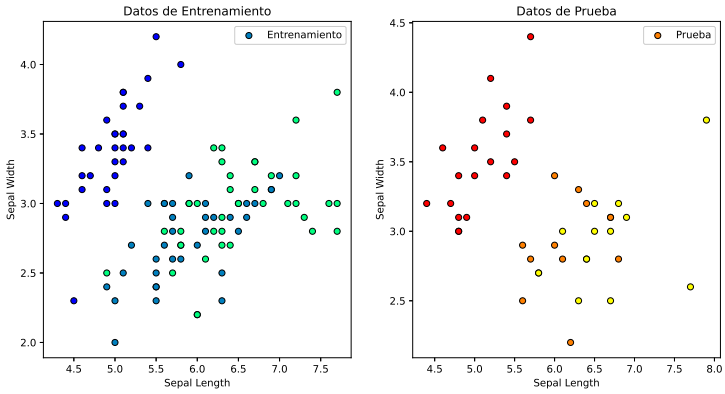


Figura 12.1: Gráficos del agrupamiento de los datos de entrenamiento y validación a partir de `Matplotlib`.

```
38
39 plt.subplot(1, 2, 2)
40 plt.scatter(X_test[:, 0], X_test[:, 1], c=y_test, cmap='autumn', edgecolor='k', label="
       Prueba")
41 plt.xlabel('Sepal Length')
42 plt.ylabel('Sepal Width')
43 plt.title('Datos de Prueba')
44 plt.legend()
45
46 # Guardar la gráfica en un archivo .eps
47 plt.savefig("agrupamiento.eps", format='eps')
```

El lector puede bajar este programa desde el siguiente vínculo:

Este programa muestra cómo utilizar un modelo de Árbol de Decisión para clasificar las especies de las flores de iris, utilizando medidas físicas como características. Se evalúa la precisión del modelo para determinar su eficacia en la predicción de datos no vistos anteriormente.

Análisis del Resultado.

(1) **Accuracy: 1.0**: Esta salida indica que el modelo de aprendizaje automático ha logrado una precisión perfecta en el conjunto de datos de prueba. Significa que en esta instancia de prueba, el modelo ha predicho correctamente el 100 % de las veces.

(2) **Consideraciones Importantes**:

   (a) Una precisión del 100 % es rara y puede sugerir sobreajuste, especialmente si el conjunto de datos es complejo.

   (b) El sobreajuste ocurre cuando un modelo aprende tanto los detalles como el ruido del conjunto de entrenamiento, lo cual puede perjudicar su rendimiento en datos nuevos.

   (c) Es crucial verificar la robustez del modelo y asegurarse de que los datos de prueba sean representativos y variados.

El resultado `Accuracy: 1.0` es indicativo de un alto rendimiento del modelo en el conjunto de datos de prueba, pero es recomendable realizar un análisis más detallado para confirmar que el modelo no está sobreajustado y es generalizable a nuevos datos.

12.5. Usos del Machine Learning con Python

El Machine Learning con Python se aplica en:

(1) **Reconocimiento de Imágenes y Visión Computacional**: Como reconocimiento facial y detección de objetos.

(2) **Procesamiento del Lenguaje Natural (NLP)**: En traducción automática y análisis de sentimientos.

(3) **Predicción y Análisis de Datos**: Utilizado en finanzas y medicina.

(4) **Sistemas de Recomendación**: En comercio electrónico y streaming.

(5) **Automatización y Robótica**: Vehículos autónomos y robótica.

(6) **Detección de Fraude y Análisis de Riesgos**: En el sector bancario y financiero.

12.6. Visualización con `Scikit-learn`

`Scikit-learn` es una biblioteca de Python que ofrece herramientas eficientes para el análisis de datos y la minería de datos. Es ampliamente reconocida en la comunidad científica y tecnológica por su facilidad de uso y versatilidad.

12.7. Funciones Principales

(1) **Aprendizaje Automático**: `Scikit-learn` soporta varios aspectos del aprendizaje automático, como clasificación, regresión y agrupamiento.

(2) **Preprocesamiento de Datos**: Ofrece herramientas para la normalización, estandarización y codificación de características.

(3) **Reducción de Dimensionalidad**: Incluye métodos para reducir la dimensionalidad de los datos, mejorando así la eficiencia de los algoritmos.

(4) **Selección de Modelos**: Proporciona funcionalidades para la validación cruzada y la búsqueda de hiperparámetros.

(5) **Algoritmos de Aprendizaje Automático**: Contiene una amplia gama de algoritmos, desde simples hasta avanzados.

(6) **Integración con Otras Bibliotecas**: Se integra eficientemente con bibliotecas como `NumPy`, `Pandas` y `Matplotlib`.

(7) **Comunidad y Documentación**: Cuenta con una extensa comunidad de usuarios y una documentación bien mantenida.

12.8. Modelos de Clasificación y Regresión

La clasificación y la regresión son dos de los problemas más comunes que se abordan en el campo del aprendizaje automático. Aunque ambos se utilizan para hacer predicciones, difieren significativamente en la naturaleza de sus predicciones y sus aplicaciones. Este documento tiene como objetivo brindar una comprensión clara de estos dos tipos de modelos, sus diferencias y cómo se aplican en el análisis de datos.

Ejemplo 12.

```python
#!/usr/bin/env python3

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

# Cargar el conjunto de datos de Viviendas de California
california_housing = fetch_california_housing()
X = california_housing.data
y = california_housing.target

# Convertir a DataFrame de Pandas
california_df = pd.DataFrame(X, columns=california_housing.feature_names)
california_df['MedHouseVal'] = y  # MedHouseVal es el valor medio de las viviendas

# Elegir una característica para la regresión, por ejemplo, MedInc (ingreso medio)
X_medinc = california_df[['MedInc']].values

# Dividir los datos en conjuntos de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_medinc, y, test_size=0.3, random_state
    =42)

# Crear y entrenar el modelo de Regresión Lineal
regressor = LinearRegression()
regressor.fit(X_train, y_train)

# Realizar predicciones sobre el conjunto de prueba
y_pred = regressor.predict(X_test)

# Calcular métricas de rendimiento
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"R^2 Score: {r2}")

# Visualizar los resultados
plt.figure(figsize=(14, 6))

# Histograma de los precios de las viviendas
plt.subplot(1, 2, 1)
plt.hist(california_df['MedHouseVal'], bins=30, color='green', edgecolor='black')
plt.xlabel('Valor Medio de la Vivienda (en $100,000s)')
plt.ylabel('Frecuencia')
plt.title('Distribución de los Precios de las Viviendas')

# Gráfico de dispersión de la Regresión Lineal
plt.subplot(1, 2, 2)
plt.scatter(X_test, y_test, color='blue', label='Datos Reales')
plt.plot(X_test, y_pred, color='red', linewidth=2, label='Predicción de Regresión')
plt.xlabel('Ingreso Medio')
plt.ylabel('Valor Medio de la Vivienda (en $100,000s)')
plt.title('Regresión Lineal - California Housing Dataset')
plt.legend()

# Guardar la gráfica en un archivo .eps
plt.savefig("regresion1.eps", format='eps')
```

⚠ El lector puede bajar este programa desde el siguiente vínculo: 🖳

Descripción del Programa de Regresión Lineal en Python. El objetivo de este programa es predecir el valor medio de las viviendas en California utilizando un modelo de regresión lineal basado en el ingreso medio del área. Se busca comprender la influencia del ingreso medio en el valor de la vivienda y visualizar esta relación mediante gráficos.

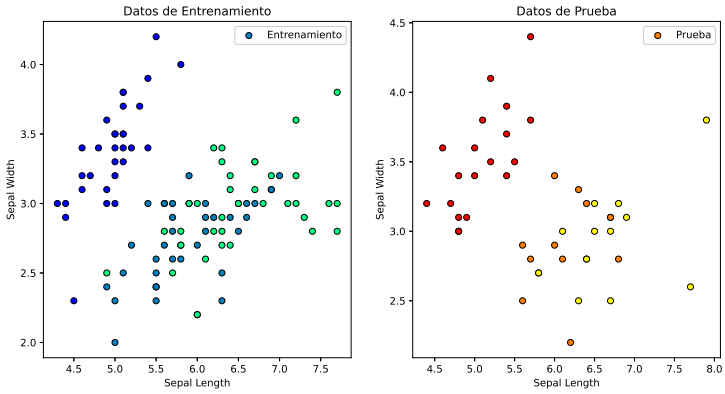


Figura 12.2: Gráficos sobre el análisis de regresión a partir de `Matplotlib`.

Importación de Bibliotecas.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score
```

Cargar y Preparar los Datos.

```
california_housing = fetch_california_housing()
X = california_housing.data
y = california_housing.target

california_df = pd.DataFrame(X, columns=california_housing.feature_names)
california_df['MedHouseVal'] = y
X_medinc = california_df[['MedInc']].values
```

División en Conjuntos de Entrenamiento y Prueba.

```
X_train, X_test, y_train, y_test = train_test_split(X_medinc, y, test_size=0.3, random_state
    =42)
```

Muestra del Conjunto de Entrenamiento.

```
# Muestra de las primeras filas de los datos de entrenamiento
X_train[:5], y_train[:5]

# Ingreso Medio (MedInc), Valor Medio de la Vivienda (MedHouseVal)
[[ 8.3252 ], 4.526 ]
[[ 8.3014 ], 3.585 ]
[[ 7.2574 ], 3.521 ]
[[ 5.6431 ], 3.413 ]
[[ 3.8462 ], 3.422 ]
```

Muestra del Conjunto de Prueba.

```
# Muestra de las primeras filas de los datos de prueba
X_test[:5], y_test[:5]

# Ingreso Medio (MedInc), Valor Medio de la Vivienda (MedHouseVal)
[[ 2.0804 ], 0.477 ]
[[ 3.6912 ], 0.458 ]
```

```
[[ 3.2031 ], 5.000 ]
[[ 3.2705 ], 2.186 ]
[[ 3.0750 ], 2.780 ]
```

Creación y Entrenamiento del Modelo.

```
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

Evaluación del Modelo.

```
y_pred = regressor.predict(X_test)
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)
print(f"Mean Squared Error: {mse}")
print(f"R^2 Score: {r2}")
```

Visualización de Resultados.

```
plt.figure(figsize=(14, 6))
...
plt.savefig("regresion.eps", format='eps')
plt.show()
```

El histograma muestra la distribución del valor medio de las viviendas, proporcionando una idea de la variedad y frecuencia de los diferentes valores de vivienda en el conjunto de datos.

El gráfico de dispersión con la línea de regresión ilustra la relación entre el ingreso medio y el valor medio de la vivienda. La línea de regresión predice cómo el valor de la vivienda aumenta con el ingreso medio, proporcionando una herramienta visual para comprender esta tendencia.

Discusión de Resultados de Regresión Lineal. El modelo de regresión lineal ha producido dos métricas clave: el Error Cuadrático Medio (MSE) y el coeficiente de determinación (R^2 Score). A continuación se discuten estos resultados y su significado.

Mean Squared Error (MSE).
El MSE obtenido es aproximadamente 0.6918. Este valor representa el promedio de los cuadrados de las diferencias entre los valores observados y los predichos por el modelo. Un MSE de 0.6918 indica que:

(1) El cuadrado de la diferencia entre el valor real y el valor predicho por el modelo es en promedio 0.6918.

(2) La interpretación del MSE depende de la escala de la variable objetivo. Si los valores objetivo tienen una amplia variación, un MSE de 0.6918 podría ser considerado bajo. Sin embargo, si los valores objetivo son menores, el mismo MSE podría ser alto.

(3) El MSE solo puede no ser suficiente para evaluar la precisión absoluta o comparar modelos sin considerar la escala de los datos.

R^2 Score.
El coeficiente de determinación, R^2, obtenido es aproximadamente 0.4729. Este valor indica que:

(1) Alrededor del 47.3 % de la variabilidad en el valor medio de la vivienda está explicada por el modelo.

(2) Un R^2 de 0.4729 sugiere una capacidad de predicción moderada, lo cual implica que hay una cantidad significativa de variabilidad en el valor de la vivienda que el modelo no puede explicar.

(3) Un R^2 más bajo puede indicar la ausencia de variables importantes en el modelo, relaciones no lineales, o que el modelo no se ajusta bien a los datos.

Los resultados muestran un rendimiento moderado del modelo. El MSE indica un error promedio cuadrado de 0.6918 en las predicciones, y el R^2 muestra que el modelo explica aproximadamente el 47.3 % de la variabilidad en el valor de la vivienda. Estos resultados pueden motivar la exploración de modelos más complejos o la inclusión de más características para mejorar el rendimiento.

12.9. Caso Práctico

Python es una herramienta poderosa y versátil para el aprendizaje automático (machine learning). En este artículo, exploraremos algunas de las razones por las cuales Python se destaca en este campo, como su amplia variedad de bibliotecas especializadas, su comunidad activa de desarrolladores, su facilidad de uso y su capacidad para visualizar datos. Además, presentaremos un caso de estudio que utiliza Python para resolver problemas de regresión y clasificación, demostrando su eficacia en la implementación de modelos de aprendizaje automático.

Utilizaremos Python y sus bibliotecas de aprendizaje automático para abordar un caso práctico que involucra regresión y clasificación. Mostraremos cómo Python simplifica la adquisición de datos, el preprocesamiento, el entrenamiento de modelos y la visualización de resultados, destacando así su utilidad en aplicaciones del mundo real en el campo del aprendizaje automático.

```
#!/usr/bin/env python3

# Importar las bibliotecas necesarias
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import mean_squared_error, accuracy_score

# Generar datos de ejemplo para regresión
np.random.seed(0)
X_regresion = np.random.rand(100, 1)  # Característica de entrada
y_regresion = 2 * X_regresion + 1 + 0.1 * np.random.randn(100, 1)  # Variable objetivo

# Dividir los datos en conjuntos de entrenamiento y prueba para regresión
X_regresion_train, X_regresion_test, y_regresion_train, y_regresion_test = train_test_split(
    X_regresion, y_regresion, test_size=0.2, random_state=0)

# Entrenar un modelo de regresión lineal
regresion_model = LinearRegression()
regresion_model.fit(X_regresion_train, y_regresion_train)

# Predecir valores con el modelo de regresión
y_regresion_pred = regresion_model.predict(X_regresion_test)

# Calcular el error cuadrático medio (MSE) para la regresión
mse = mean_squared_error(y_regresion_test, y_regresion_pred)
print(f"Error cuadrático medio para regresión: {mse:.2f}")

# Graficar los datos y la línea de regresión
plt.figure(figsize=(8, 6))
plt.scatter(X_regresion_test, y_regresion_test, label='Datos reales')
plt.plot(X_regresion_test, y_regresion_pred, color='red', linewidth=2, label='Regresión
    lineal')
plt.xlabel('Característica de entrada')
plt.ylabel('Variable objetivo')
plt.legend()
plt.title('Regresión lineal')
plt.savefig('regresion_plot.eps', format='eps', dpi=300)  # Guardar la figura en formato EPS
plt.show()

# Generar datos de ejemplo para clasificación
X_clasificacion = np.random.rand(100, 2)  # Características de entrada
y_clasificacion = (X_clasificacion[:, 0] + X_clasificacion[:, 1] > 1).astype(int)  #
    Variable objetivo (0 o 1)

# Dividir los datos en conjuntos de entrenamiento y prueba para clasificación
X_clasificacion_train, X_clasificacion_test, y_clasificacion_train, y_clasificacion_test =
    train_test_split(X_clasificacion, y_clasificacion, test_size=0.2, random_state=0)

```

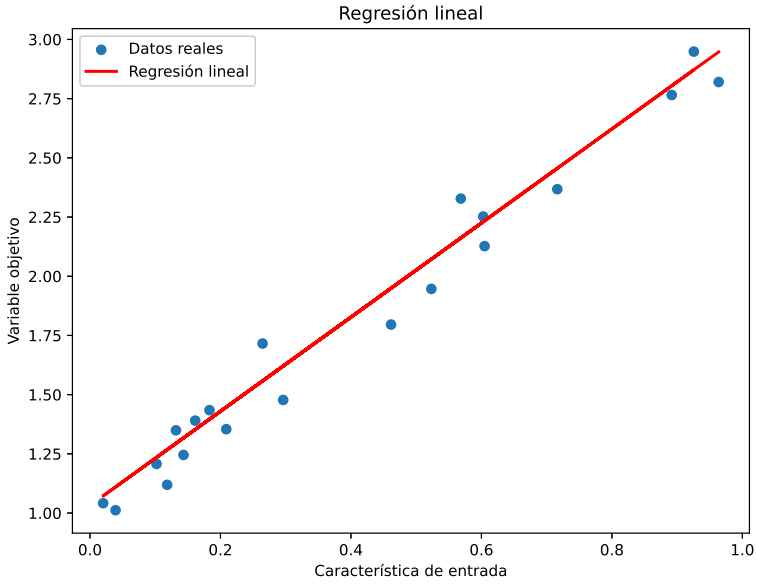


Figura 12.3: Gráfico de regresión a partir de `Matplotlib`.

```
# Entrenar un modelo de clasificación (Random Forest)
clasificacion_model = RandomForestClassifier()
clasificacion_model.fit(X_clasificacion_train, y_clasificacion_train)

# Predecir clases con el modelo de clasificación
y_clasificacion_pred = clasificacion_model.predict(X_clasificacion_test)

# Calcular la precisión para la clasificación
accuracy = accuracy_score(y_clasificacion_test, y_clasificacion_pred)
print(f"Precisión para clasificación: {accuracy:.2f}")

# Graficar la clasificación
plt.figure(figsize=(8, 6))
plt.scatter(X_clasificacion_test[:, 0], X_clasificacion_test[:, 1], c=y_clasificacion_pred,
    cmap=plt.cm.Paired)
plt.xlabel('Característica 1')
plt.ylabel('Característica 2')
plt.title('Clasificación con Random Forest')
plt.savefig('clasificacion_plot.eps', format='eps', dpi=300)  # Guardar la figura en formato
     EPS
```

```
./caso_regresion.py
Error cuadrático medio para regresión: 0.01
The PostScript backend does not support transparency; partially transparent artists will be
    rendered opaque.
Precisión para clasificación: 1.00
```

Los resultados del programa `caso_regresion.py` proporcionan métricas de rendimiento para dos problemas de aprendizaje automático: regresión y clasificación.

Para el problema de regresión, el programa informa un Error Cuadrático Medio (MSE) de 0.01. El MSE es una métrica que cuantifica cuán cerca están las predicciones del modelo de regresión de los valores reales. En este caso, un MSE de 0.01 indica que el modelo de regresión lineal tiene un buen rendimiento, ya que el error cuadrático medio es bajo, lo que sugiere que las predicciones se ajustan bien a los datos reales. Esto es un indicador positivo de la calidad del modelo de regresión.

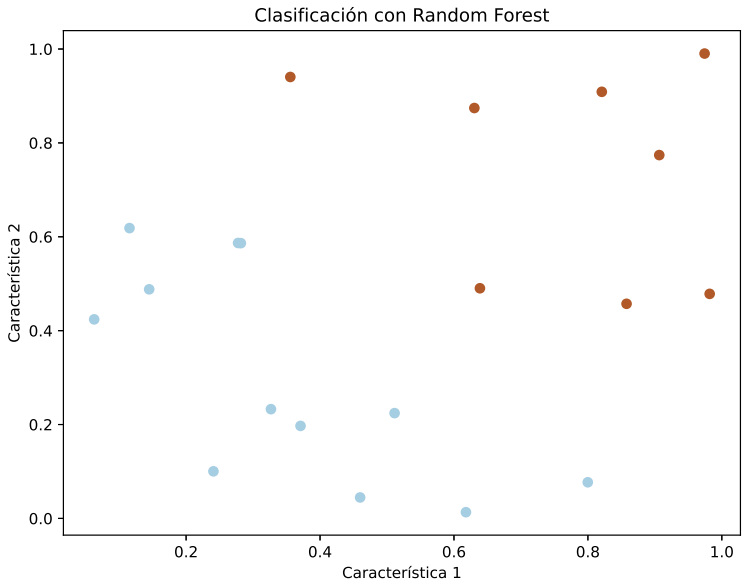


Figura 12.4: Gráfico de clasificación a partir de `Matplotlib`.

Para el problema de clasificación, el programa muestra una precisión del 100 %. La precisión es una métrica que mide la proporción de ejemplos clasificados correctamente por el modelo. Una precisión del 100 % indica que el modelo de clasificación (Random Forest) ha clasificado correctamente todos los ejemplos en el conjunto de prueba. Esto sugiere un alto nivel de eficacia en el modelo de clasificación, aunque también puede indicar que el conjunto de datos de clasificación utilizado es relativamente simple y fácil de separar en clases.

Los resultados son positivos en ambos casos. El modelo de regresión lineal tiene un bajo error cuadrático medio, lo que indica una buena capacidad de predicción en el problema de regresión. Además, el modelo de clasificación tiene una precisión del 100 %, lo que indica una clasificación perfecta en el problema de clasificación. Sin embargo, es importante considerar el contexto y la complejidad de los conjuntos de datos y modelos utilizados al interpretar estos resultados.

El lector puede bajar este programa desde el siguiente vínculo:

Práctica 13.

Es conveniente que el lector ejecute el programa referido.

Bibliografía

[1] John D. Hunter. Matplotlib: A 2d graphics environment, 2007.

[2] OpenAI. `ChatGPT`. `https://chat.openai.com/auth/login`, 2023.