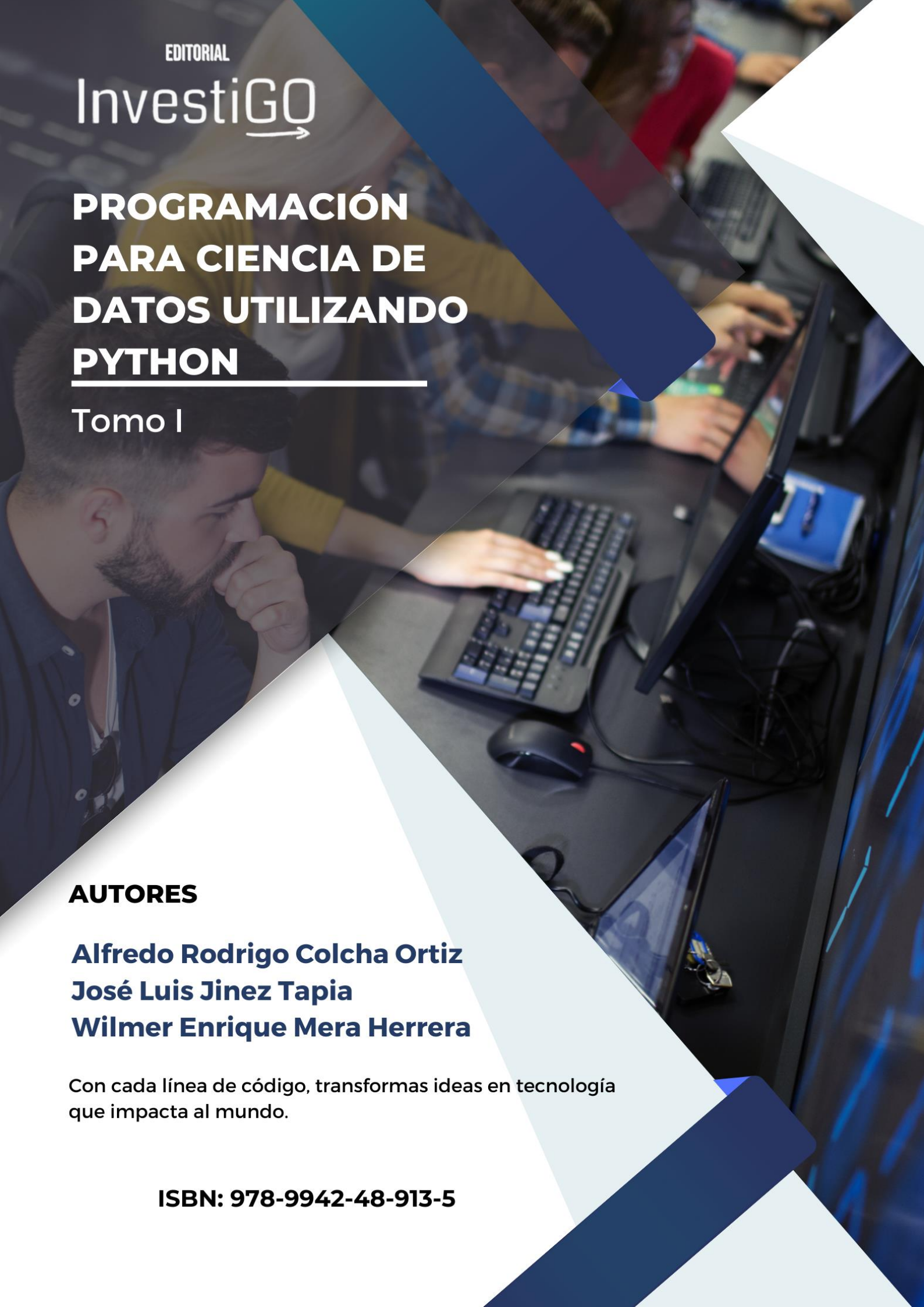
EDITORIAL
InvestiGO

PROGRAMACIÓN PARA CIENCIA DE DATOS UTILIZANDO PYTHON

Tomo I



AUTORES

Alfredo Rodrigo Colcha Ortiz
José Luis Jinez Tapia
Wilmer Enrique Mera Herrera

Con cada línea de código, transformas ideas en tecnología que impacta al mundo.

ISBN: 978-9942-48-913-5

PROGRAMACIÓN PARA CIENCIA DE DATOS UTILIZANDO PYTHON

TOMO I

AUTORES:

Alfredo Rodrigo Colcha Ortiz

José Luis Jinez Tapia

Wilmer Enrique Mera Herrera

ISBN: 978-9942-48-913-5

EDITORIAL InvestiGO

Este libro ha sido debidamente examinado y valorado en la modalidad doble par ciego con fin de garantizar la calidad científica del mismo.

©Publicaciones Editorial InvestiGo

Riobamba – Ecuador

investigoeditorial@gmail.com

https://grupoblr.com/2025/01/04/libros-investigo/

REPOSITORIO

Colcha, A., Jinez, J., Mera, W. (2025) Programación para ciencia de datos utilizando Python. Editorial InvestiGo.

ISBN: 978-9942-48-913-5

FILIACIÓN DE AUTORES

Alfredo Rodrigo Colcha Ortiz

Docente Investigador

Facultad de Ingeniería, Universidad Nacional de Chimborazo (UNACH)

Correo electrónico: alfredo.colcha@unach.edu.ec

ORCID: https://orcid.org/0009-0005-2280-5189

José Luis Jinez Tapia

Docente Investigador

Facultad de Ingeniería, Universidad Nacional de Chimborazo (UNACH)

Correo Electrónico: jjinez@unach.edu.ec

ORCID: https://orcid.org/0000-0002-4113-0579

Wilmer Enrique Mera Herrera

Coordinador de Maestrías de Posgrado

Instituto de Posgrado, Universidad Nacional de Chimborazo (UNACH)

Correo Electrónico: wilmer.mera@unach.edu.ec

ORCID: https://orcid.org/0009-0006-4484-950X

EDITORIAL InvestiGO

ÍNDICE

Código Python

DEDICATORIA

A todas aquellas personas que, con curiosidad y determinación, se aventuran a descubrir el fascinante mundo de la programación y el análisis de datos.

A los estudiantes, profesionales y autodidactas que, sin importar sus condiciones, se comprometen a aprender y mejorar constantemente, este libro es un homenaje a su espíritu de superación, su pasión por el conocimiento y su deseo de transformar la información en herramientas poderosas para resolver los retos del mundo actual.

A nuestras familias, cuya paciencia, apoyo incondicional y palabras de aliento nos inspiraron en cada paso de este proyecto. Gracias por creer en nosotros y en la importancia de crear un recurso accesible y práctico para todos aquellos que desean iniciarse en este campo.

RESUMEN

Programación para Ciencia de Datos utilizando Python. Tomo I tiene como objetivo proporcionar una base sólida en programación y análisis de datos utilizando Python. Facilita el aprendizaje de conceptos fundamentales para quienes se inician en la ciencia de datos. Este libro está dirigido a estudiantes, profesionales y entusiastas que buscan desarrollar habilidades técnicas aplicables en la resolución de problemas reales.

El primer capítulo se centra en la Programación Orientada a Objetos (POO); a través de un enfoque práctico, se explica cómo utilizar clases y objetos para modelar situaciones del mundo real. El lector aprenderá a aplicar los pilares de la POO, como la herencia, el polimorfismo y la encapsulación, en el desarrollo de programas eficientes y mantenibles. Este capítulo resalta la importancia de la POO para estructurar código de manera lógica y escalable, lo cual facilita la creación de aplicaciones complejas.

El segundo capítulo aborda el Manejo Eficiente de Arreglos Multidimensionales con la librería NumPy, aquí se profundiza en la creación y manipulación de arreglos de diversas dimensiones, fundamentales para operaciones matemáticas y estadísticas avanzadas. El capítulo destaca cómo NumPy optimiza el procesamiento de datos en grandes volúmenes y permite la ejecución de cálculos complejos de manera eficiente.

Con ejemplos claros y ejercicios prácticos, se guía al lector en cada paso del proceso, esto asegura una comprensión profunda de los conceptos. Al final, el lector estará preparado para enfrentar desafíos más avanzados en el análisis y visualización de datos, sentando una base firme para continuar en su aprendizaje de programación.

Palabras Clave: Algoritmos, análisis de datos, aprendizaje automático, ciencia de datos, programación informática.

PRESENTACIÓN

En la era digital actual, el manejo y la interpretación de grandes volúmenes de datos se ha convertido en una habilidad esencial en una amplia variedad de campos, desde la economía y la medicina hasta el marketing y la ingeniería. En este contexto, Python ha emergido como uno de los lenguajes de programación más populares y poderosos para el análisis de datos. Su versatilidad, simplicidad y la gran cantidad de bibliotecas especializadas disponibles lo convierten en una herramienta indispensable para cualquier profesional que desee aprovechar al máximo el potencial de los datos.

El lenguaje de programación Python ha ganado un protagonismo indiscutible en el análisis de datos debido a su facilidad de uso y su capacidad para integrar diferentes tecnologías. A diferencia de otros lenguajes más tradicionales, Python ofrece una curva de aprendizaje suave, lo que permite a los principiantes familiarizarse rápidamente con conceptos básicos y avanzar hacia técnicas más complejas en poco tiempo. Esta característica hace que Python sea ideal tanto para quienes se inician en la programación como para expertos que buscan una herramienta eficiente y flexible.

La popularidad de Python en la comunidad de ciencia de datos se debe en gran medida a su rica colección de bibliotecas y paquetes, como NumPy, Pandas, Matplotlib y Scikit-learn. Estas bibliotecas proporcionan funcionalidades avanzadas para la manipulación, visualización y modelado de datos, facilita el desarrollo de proyectos complejos de manera rápida y efectiva. Además, la comunidad global de desarrolladores contribuye constantemente con nuevos recursos, lo que asegura que Python se mantenga a la vanguardia de los avances tecnológicos.

El uso de Python no se limita al análisis de datos tradicional. También es fundamental en áreas emergentes como el aprendizaje automático, la inteligencia artificial y el análisis de grandes volúmenes de datos (Big Data). La capacidad de Python para manejar grandes conjuntos de datos y su integración con plataformas de

procesamiento como Apache Spark, lo hacen especialmente útil para explorar, analizar y visualizar datos en tiempo real.

Por tanto, aprender a programar en Python no solo proporciona habilidades técnicas valiosas, sino que también abre la puerta a un mundo de oportunidades en la investigación y el desarrollo tecnológico.

El principal objetivo de este libro es proporcionar a los lectores una base sólida en la programación con Python aplicada al análisis de datos, sin necesidad de experiencia previa en el campo. A lo largo de sus páginas, se busca que los lectores adquieran una comprensión profunda de los conceptos fundamentales de la programación y desarrollen habilidades prácticas que les permitan enfrentar problemas reales en el ámbito de la ciencia de datos.

El conocimiento que se quiere transmitir no se limita a la sintaxis del lenguaje o al uso de librerías específicas. En cambio, se busca que los lectores comprendan el "por qué" y el "cómo" de las herramientas que se utilizan. ¿Por qué es importante la programación orientada a objetos en el diseño de software modular? ¿Cómo la manipulación eficiente de arreglos multidimensionales puede mejorar el rendimiento de un análisis de datos?

Estas son algunas de las preguntas que se abordarán a lo largo del libro, con el fin de formar a profesionales capaces de tomar decisiones informadas y aplicar estas técnicas en contextos variados.

El enfoque va más allá de la simple instrucción técnica; se aspira a que los lectores desarrollen un pensamiento analítico y crítico que les permita identificar problemas, diseñar soluciones y evaluar los resultados de manera efectiva. La programación, en este sentido, se convierte en una herramienta para modelar, simular y entender fenómenos complejos, que van desde patrones de consumo en una tienda en línea hasta la propagación de enfermedades en una población.

Uno de los aspectos más destacados de esta guía es su carácter eminentemente práctico. Cada concepto teórico presentado va acompañado de ejemplos concretos y ejercicios diseñados para reforzar el aprendizaje. Esta metodología garantiza que el lector no solo entienda los principios subyacentes, sino que también los ponga en práctica inmediatamente, enfrentándose a problemas que simulan escenarios del mundo real.

Los ejercicios están pensados para llevar al lector de la mano, se inicia con tareas simples y se aumenta gradualmente en complejidad. Por ejemplo, al introducir la programación orientada a objetos, los lectores no solo aprenderán a definir clases y objetos, sino que también implementarán sus propias estructuras de datos y funciones, replicando el comportamiento de sistemas complejos. En el capítulo sobre NumPy, no solo se tratarán las operaciones básicas con arreglos, sino que se propondrán retos que involucren el análisis y transformación de datos de manera eficiente.

Además, al final de cada capítulo se incluyen proyectos integradores que desafían al lector a combinar los conocimientos adquiridos y aplicarlos en casos de estudio más amplios. Estos proyectos están diseñados para simular desafíos que podrían encontrarse en un entorno profesional, como el análisis de datos financieros, la visualización de tendencias en datos de ventas, o la simulación de escenarios epidemiológicos.

De esta manera, el lector no solo adquiere conocimientos teóricos, sino que también desarrolla la capacidad de aplicarlos de manera práctica y efectiva.

Al finalizar este primer tomo, los lectores habrán logrado dominar los conceptos esenciales de la programación en Python y el análisis de datos, serán capaces de desarrollar programas modulares y eficientes con la programación orientada a objetos, lo que les permitirá diseñar sistemas complejos de manera organizada y escalable. También habrán adquirido la habilidad para manipular grandes volúmenes de datos

con la biblioteca NumPy, se optimiza el rendimiento en operaciones matemáticas y estadísticas.

A lo largo del libro, los lectores aplicarán estos conocimientos en la resolución de problemas prácticos, lo que fortalecerá su capacidad para enfrentar desafíos reales y mejora su pensamiento crítico y analítico, con esta base sólida, estarán preparados para abordar temas más avanzados, como el aprendizaje automático y la inteligencia artificial, y enfrentar con éxito nuevos retos en el campo de la ciencia de datos.

Este libro tiene como objetivo principal que los lectores, al finalizar su estudio, logren varios resultados de aprendizaje clave. En primer lugar, comprenderán los principios de la Programación Orientada a Objetos (POO), lo que les permitirá aplicar conceptos fundamentales como clases, objetos, herencia, polimorfismo y encapsulamiento en la solución de problemas prácticos.

Esta comprensión es esencial para desarrollar programas estructurados y modulares que puedan evolucionar en complejidad y funcionalidad.

Además, los lectores aprenderán a manejar eficientemente arreglos multidimensionales a través de la librería NumPy. Conocerán y utilizarán las capacidades de esta herramienta para crear y manipular arreglos de diferentes dimensiones, aplicando técnicas de optimización que mejoren el rendimiento del procesamiento de datos. Este conocimiento es fundamental para cualquier trabajo relacionado con análisis de datos a gran escala.

Por otro lado, el libro se enfoca en el desarrollo de habilidades prácticas en ciencia de datos, ya que los lectores aprenderán a utilizar Python y sus bibliotecas más importantes para realizar análisis de datos completos, que incluyan la limpieza, transformación y visualización de datos en distintos formatos. Esta parte práctica permitirá que los estudiantes adquieran una experiencia sólida y directa en el manejo de datos reales.

Asimismo, el contenido del libro está diseñado para que los lectores sean capaces de integrar herramientas y técnicas en proyectos reales. Se busca que puedan desarrollar proyectos completos que utilicen los conocimientos adquiridos en cada capítulo, aplicándolos en diversos contextos como el análisis financiero, la investigación científica o el marketing digital, fortaleciendo así su capacidad para resolver problemas concretos con un enfoque analítico.

Finalmente, los lectores aprenderán a evaluar y mejorar soluciones de programación, mediante la revisión y optimización del código que desarrollen. Aplicarán buenas prácticas de programación y harán uso eficiente de los recursos computacionales, lo que les permitirá escribir código más limpio, eficiente y escalable, preparándolos para enfrentar con éxito futuros desafíos en su carrera profesional.

Estos resultados de aprendizaje garantizan que los lectores no solo adquieran conocimientos teóricos, sino también habilidades prácticas aplicables a diversos campos del análisis de datos y la programación.

El libro se divide en dos capítulos principales, cada uno diseñado para abordar aspectos fundamentales de la programación y el análisis de datos.

Capítulo 1: Programación Orientada a Objetos (POO), este capítulo introduce al lector en el paradigma de la programación orientada a objetos, un enfoque que permite organizar el código de manera modular y eficiente. Se abordan conceptos como clases, objetos, métodos y atributos, así como los principios de herencia, polimorfismo, encapsulamiento y abstracción, el lector aprenderá a construir programas que simulen situaciones del mundo real, aplicará estos principios para diseñar sistemas más complejos y escalables.

Este capítulo resalta cómo la POO es fundamental para estructurar el código de manera que sea reutilizable y fácil de mantener. Proporciona una base sólida para el desarrollo de proyectos de análisis de datos y otras aplicaciones avanzadas.

Capítulo 2: Manejo Eficiente de Arreglos Multidimensionales, explora la biblioteca NumPy, una herramienta esencial para el análisis de datos en Python, se presentan desde los conceptos básicos de arreglos unidimensionales hasta estructuras de datos más complejas como matrices y tensores, los lectores aprenderán a realizar operaciones matemáticas y estadísticas de manera eficiente, aprovechando la capacidad de NumPy para manejar grandes volúmenes de datos.

Este capítulo subraya la importancia de NumPy en el ecosistema de la ciencia de datos, destaca cómo su uso permite optimizar operaciones críticas y facilita el manejo de datos en múltiples dimensiones.

Capítulo 3: El capítulo presenta una serie de ejercicios prácticos diseñados para desarrollar habilidades en programación aplicada a la ciencia de datos y resolución de problemas específicos, promoviendo una comprensión profunda y práctica de conceptos avanzados de programación.

El manuscrito es una guía ideal para quienes buscan fortalecer sus conocimientos en programación aplicada y ciencia de datos. Este libro no solo enseñará a programar, sino a resolver problemas con un enfoque estructurado y eficiente.

INTRODUCCIÓN

La transformación digital que experimenta el mundo actual ha impactado a diversos niveles de la sociedad, desde el ámbito global hasta el contexto más local, marcando una profunda revolución en la forma en que se genera, analiza y utiliza la información. Este cambio ha sido impulsado en gran parte por el avance tecnológico y la capacidad de procesar y analizar grandes volúmenes de datos de manera eficiente.

En este sentido, el lenguaje de programación Python ha emergido como una herramienta clave en la ciencia de datos, gracias a su versatilidad, accesibilidad y la amplia gama de bibliotecas que ofrece para la manipulación y visualización de datos. Esta introducción busca contextualizar la relevancia de este libro desde una perspectiva macro, meso y micro, enfatiza su importancia para América Latina, Ecuador y las instituciones de educación superior del país.

A nivel global, la demanda de habilidades en análisis de datos y programación ha crecido exponencialmente en las últimas décadas. Empresas de todos los sectores, desde la tecnología hasta la salud, buscan profesionales capaces de extraer conocimiento valioso de grandes volúmenes de información. En este contexto, Python ha ganado terreno como uno de los lenguajes de programación más utilizados en ciencia de datos.

Autores como Lott y Phillips (2021) destacan la capacidad de Python para construir aplicaciones robustas y mantenibles, lo que lo convierte en una opción preferida para desarrolladores e investigadores a nivel mundial. La relevancia de esta tecnología no se limita solo a su popularidad, sino que también radica en su capacidad para democratizar el acceso al análisis de datos, lo que permite a personas de diversos contextos y niveles de experiencia participar en esta revolución digital.

En América Latina, y particularmente en Ecuador, la adopción de tecnologías avanzadas para el análisis de datos ha comenzado a ganar impulso en los últimos años. Sin embargo, aún existen brechas significativas en comparación con otras regiones del

mundo. Los esfuerzos por integrar estas tecnologías en sectores clave como la educación, la salud y la industria son fundamentales para el desarrollo sostenible de la región.

Es aquí donde las instituciones de educación superior juegan un papel crucial en la formación de profesionales capacitados en el manejo de herramientas tecnológicas modernas.

Autores como Rodríguez Guerrero, Vanegas y Castang Montiel (2020) destacan la importancia de llevar el conocimiento sobre Python a un público hispanohablante, lo que facilita el acceso a recursos educativos adaptados a las necesidades de la región. Este libro responde a esa necesidad, proporciona una guía práctica y accesible para que estudiantes y profesionales de Ecuador y América Latina puedan adquirir las habilidades necesarias para enfrentar los desafíos del siglo XXI.

Dentro del contexto ecuatoriano, la necesidad de formación en tecnologías de análisis de datos es aún más apremiante. Las instituciones de educación superior en el país tienen la responsabilidad de formar profesionales que no solo comprendan los fundamentos teóricos de la ciencia de datos, sino que también sean capaces de aplicar estos conocimientos en la resolución de problemas concretos que afectan a la sociedad ecuatoriana.

La Universidad Nacional de Chimborazo, entre otras, ha sido pionera en la incorporación de programas educativos orientados a la ciencia de datos, reconoce la importancia de preparar a sus estudiantes para un entorno laboral cada vez más digitalizado.

Este libro busca cerrar la brecha existente entre el conocimiento teórico y la práctica aplicada. Utiliza ejemplos y ejercicios adaptados al contexto local, de modo que los lectores podrán entender cómo aplicar las técnicas aprendidas en casos concretos que podrían encontrarse en su vida profesional.

El objetivo principal de este libro es proporcionar a los lectores una base sólida en la programación y análisis de datos utilizando Python. Facilita su aprendizaje a través de un enfoque práctico y contextualizado. Se espera que, al finalizar el estudio, los lectores puedan no solo comprender los conceptos fundamentales de la programación orientada a objetos y el manejo de arreglos multidimensionales, sino también aplicarlos en proyectos reales que aborden problemas relevantes para su entorno profesional y social.

La construcción de esta obra se basa en la rica contribución de varios autores que han abordado el desarrollo y aplicación de Python en la ciencia de datos. Lott y Phillips (2021), con su enfoque en la programación orientada a objetos, han sido fundamentales para la estructura del primer capítulo, proporciona un marco teórico y práctico que guía al lector desde los conceptos básicos hasta la implementación de sistemas complejos. Jorge Santiago Nolasco Valenzuela (2018), con su obra "Python Aplicaciones prácticas", ha sido un referente para el diseño de ejercicios y ejemplos que permitan al lector aplicar los conocimientos adquiridos de manera efectiva.

Por otro lado, la obra de Óscar Ramírez Jiménez (2021), "Python a fondo", ha sido fundamental para entender las capacidades avanzadas de Python en la manipulación de datos, aporta una perspectiva que combina teoría y práctica de manera equilibrada. Finalmente, la guía proporcionada por el equipo de NumPy (s.f.) ha sido indispensable para la construcción del segundo capítulo, ya que se asegura que el contenido sea no solo técnicamente preciso, sino también accesible para principiantes.

CAPÍTULO I: PROGRAMACIÓN ORIENTADA A OBJETOS (POO)

La Programación Orientada a Objetos (POO) ha revolucionado el desarrollo de software al introducir un enfoque modular y centrado en la creación de objetos que representan entidades del mundo real. Este paradigma, fundamentado en conceptos como encapsulamiento, herencia, polimorfismo y abstracción, ofrece una estructura eficiente y reutilizable para diseñar sistemas complejos y escalables. En este capítulo, exploraremos los principios fundamentales de la POO, su evolución histórica y su importancia en la industria del software. Además, se analizarán las ventajas de adoptar este enfoque en comparación con paradigmas tradicionales, proporcionando una base sólida para comprender y aplicar este modelo en distintos lenguajes de programación.

1.1 Fundamentos de la Programación Orientada a Objetos

La Programación Orientada a Objetos (POO) constituye un paradigma esencial en el desarrollo de software, cuyo propósito principal es modelar sistemas complejos mediante la representación de entidades reales o conceptuales como objetos. Esta metodología organiza el código en torno a elementos que combinan datos (atributos) y comportamientos (métodos), lo que facilita la creación de programas modulares y reutilizables.

En su núcleo, la POO se basa en cuatro pilares fundamentales: encapsulamiento, abstracción, herencia y polimorfismo.

1.1.1 Introducción a la Programación Orientada a Objetos.

La Programación Orientada a Objetos (**POO**) es un paradigma de programación que se centra en el uso de **objetos** para diseñar software. Los **objetos** representan entidades del mundo real con características y comportamientos definidos, organizados en **clases**. Esta metodología ayuda a hacer el código más organizado, reutilizable y fácil de mantener, permitiendo que los desarrolladores simulen situaciones y entidades del mundo real de manera más natural y eficiente.

1.1.2 Python para Programación Orientada a Objetos

Python ha sido seleccionado como el lenguaje de programación para desarrollar la asignatura de Programación 2 debido a su sintaxis sencilla y clara, lo que simplifica tanto la enseñanza como el aprendizaje de la Programación Orientada a Objetos (**POO**). La combinación de su tipado dinámico, un eficiente gestor de memoria y una amplia biblioteca de recursos potencia la capacidad del lenguaje para analizar datos y desarrollar complejos algoritmos de inteligencia artificial. Aprender **POO** en Python habilita a los estudiantes con habilidades prácticas en Ciencia de Datos e Inteligencia Artificial, permitiéndoles modelar datos y funcionalidades de manera eficaz.

- **Clases y objetos**

Son los bloques fundamentales de la Programación Orientada a Objetos, donde una clase define la estructura y comportamiento de los objetos, mientras que los objetos son instancias concretas de una clase.

- **Clase**

Una **clase** es un plano, modelo o guía que define las características y comportamientos (atributos y métodos) de los **objetos** que se crearán a partir de ella. Es una forma de **encapsular** datos junto con métodos para operar esos datos, lo que facilita la organización, modularidad y reutilización del código (*Classes (OOP) | Brilliant Math & Science Wiki*, s. f.).

Ejemplo: Creación de una Clase

Código Python 1
Creación de una Clase

Código Python

```
# Crea una clase vacía Automóvil
class Automovil:
    pass
```

Resultado: Código Python 1, formulación de los autores para la creación de una Clase.

Una convención en Python al definir los nombres de las **clases** es que el nombre de la **clase** debe empezar con una letra mayúscula.

- **Objeto**

Se define un **objeto** como una instancia de una **clase**, un **objeto** encapsula tanto datos como comportamientos, que se representan a través de atributos y métodos específicos del **objeto**, cada **objeto** es una implementación concreta de una clase, que actúa como una entidad específica dentro del software.

Ejemplo: En una **clase** denominada Automóvil, un **objeto** específico podría ser un Toyota Corolla con atributos como color, modelo y kilometraje, y métodos que permiten acciones como arrancar, detener y acelerar.

Ejemplo: Creación de un Objeto

Código Python 2
Creación de un Objeto

Código Python

```
# Objeto de la clase Automóvil
mi_auto=Automovil()
```

Resultado: Código Python 2, formulación de los autores para la creación de un Objeto.

- **Atributo**

Un atributo se refiere a las características o propiedades que definen un objeto, describen su estado o cualidades. Por ejemplo: En una **clase** Automóvil, los atributos podrían incluir el color, la marca, el modelo y el kilometraje. En cambio, el **objeto** creado a partir de la **clase** Automóvil tendría su propio conjunto de valores para estos atributos como:

color= “rojo”

marca =“Toyota”

modelo =” Fortuner”

kilometraje = 4500

Ejemplo: Declaración de Atributos

Código Python 3
Declaración de Atributos

Código Python

```
class Automovil:
   #atributos
    color='Rojo'
    marca ='Toyota'
    modelo='Fortuner'
    kilometraje=4500

# Objeto Automovil
mi_auto=Automovil()

#Se imprime un atributo del Objeto
print(mi_auto.color)
```

Resultado: Código Python 3, formulación de los autores para la declaración de Atributos.

- **Método**

Un método es una función definida dentro de una **clase** que describe cómo los objetos de esa **clase** se comportarán o interactuarán. Los métodos son esenciales para manipular los atributos de los **objetos** y para definir acciones específicas que estos **objetos** pueden realizar.

Ejemplo: En una **clase Automóvil**, un método podría ser **arrancar**, que cambia el estado del automóvil de **apagado** a **encendido**. Este concepto permite que los **objetos** actúen y reaccionen de manera específica, lo que facilita así la simulación de comportamientos complejos y la interacción entre diferentes objetos.

Código Python 4

Creación de la clase Automóvil

Código Python

```
class Automovil:
   #atributos
    color='Rojo'
    marca ='Toyota'
    modelo='Fortuner'
    kilometraje=4500
    estado='apagado'
    # Método
    def arrancar(self):
        self.estado='encendido'

# Objeto
mi_auto=Automovil()

#Imprime un atributo del objeto
print(mi_auto.estado)
# Imprime apagado

#Llama al método arrancar
mi_auto.arrancar()

print(mi_auto.estado)
#Imprime encendido
```

Resultado: Código Python 4, formulación de los autores para la creación de la clase Automóvil.

- **Constructor**

Un constructor es un método especial dentro de una **clase** que se ejecuta automáticamente cuando se crea un nuevo **objeto** de esa **clase**. Este método tiene como principal función inicializar el **objeto**; establece valores iniciales para sus atributos o ejecuta cualquier configuración inicial necesaria.

Ejemplo: En una **clase Automóvil**, el constructor podría establecer el **modelo**, el **color** y el **kilometraje** inicial del automóvil basándose en los valores proporcionados en el momento de la creación del **objeto**. El uso de constructores asegura que el **objeto** empiece su vida en un estado válido y consistente.

Código Python 5
Constructor de una clase

Código Python
```
class Automovil:
    def __init__(self, modelo, color, kilometraje):
```

Resultado: Código Python 5, formulación de los autores para el constructor de una clase

La función **__init__ (self, <parámetros>)** es el constructor de una **clase** en Python.

Código Python 6
Constructor en la clase Automóvil

Código Python
```
#Clase con Constructor
class Automovil:
    def __init__(self, modelo, color, kilometraje):
        self.modelo=modelo
        self.color=color
        self.kilometraje=kilometraje

# Crea un Objeto enviándole parámetros
auto=Automovil('RAV4','blanco',5600)
#Imprimo el color
print(auto.color)
#Imprime blanco
```

Resultado: Código Python 6, formulación de los autores para el constructor en la clase Automóvil.

1.1.3 Pilares de la Programación Orientada a Objetos

La Programación Orientada a Objetos (**POO**) se basa en cuatro pilares fundamentales que nos ayudarán a construir código modular, mantenible y extensible. Estos pilares son la **abstracción**, la **encapsulación**, la **herencia** y el **polimorfismo**. Cada uno de estos conceptos contribuirá a conseguir una estructura de programación más eficiente y la capacidad de manejar complejidades en aplicaciones de análisis de problemas de Ciencia de Datos e Inteligencia Artificial (Lott & Phillips, 2021).

- **Abstracción**

Este pilar permite manejar la complejidad al esconder los detalles menos relevantes y mostrar solo los necesarios, lo que facilita la gestión y comprensión del código.

- **Encapsulación**

Se refiere al embalaje de los datos (atributos) y el código (métodos) juntos de forma segura dentro de una **clase**. Protege la información y limita la forma en que se accede o se modifica desde fuera de la **clase**.

- **Herencia**

Permite a una **clase heredar** características y métodos de otra **clase**. Esto facilita la reutilización del código y la creación de nuevas **clases** a partir de **clases** existentes sin necesidad de reescribir el código.

- **Polimorfismo**

Este principio permite a las **clases** derivadas de una misma **clase** base tener métodos que pueden actuar de manera diferente, aunque compartan el mismo nombre. Esto es útil para manejar diferentes tipos de **objetos** de manera uniforme.

- **Herencia**

La herencia es un mecanismo que permite que una **clase**, conocida como **subclase**, derive o **herede** propiedades y métodos de otra **clase**, conocida como **superclase**. Este principio es fundamental para reutilizar código ya existente, además nos permitirá la extensión y modificación de comportamientos de **clases** sin necesidad de modificar la **clase** original (Óscar Ramírez Jiménez, 2021).

Ejemplo: Si se tiene una **clase** llamada **Vehículo** que incluye métodos como **arrancar** y **detener**, una **subclase** llamada **Automóvil** puede heredar estos métodos y agregar otros específicos, como **abrirPuerta**

Código Python 7
Implementación de Herencia en una Clase

Código Python

```
# Definimos la clase Vehiculo que es la clase base para todos los vehículos
class Vehiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

    # Método para arrancar el vehículo
    def arrancar(self):
        print(f"El {self.marca} {self.modelo} ha arrancado.")

    # Método para detener el vehículo
    def detener(self):
        print(f"El {self.marca} {self.modelo} se ha detenido.")

# Definimos la subclase Automovil que hereda de Vehiculo
class Automovil(Vehículo):
    def __init__(self, marca, modelo, numero_puertas):
        super().__init__(marca, modelo)  # Llamada al constructor de la clase
base
        self.numero_puertas = numero_puertas

    # Método específico de Automovil para abrir las puertas
    def abrirPuerta(self):
            print(f"Se  han  abierto  las  {self.numero_puertas}  puertas  del
automóvil.")

# Creación de un objeto de la clase Automovil
mi_auto = Automovil("Toyota", "Corolla", 4)

# Uso de los métodos heredados y del método específico
mi_auto.arrancar()
mi_auto.abrirPuerta()
mi_auto.detener()
```

Resultado: Código Python 7, formulación de los autores para la implementación de Herencia en una Clase.

Explicación del código:

- **Clase Vehiculo:** Es la **clase** base y tiene dos métodos: **arrancar ()** y **detener ()**. Ambos imprimen mensajes que indican la acción que realiza el vehículo.

- **Constructor de Vehiculo:** Inicializa la instancia con atributos como la marca y el modelo del vehículo.
- **Subclase Automovil:** Hereda de **vehículo** y agrega un atributo adicional (**numero_puertas**) y un método adicional (**abrirPuerta()**), que también imprime un mensaje.
- **Constructor de Automovil:** Usa **super()** para llamar al constructor de la **clase** base y así asegurar que los atributos heredados sean inicializados correctamente, además de inicializar el nuevo atributo **numero_puertas**.
- **Creación y uso de un objeto Automovil:** Se crea una instancia de **Automovil**, y se utilizan tanto los métodos heredados como el método específico de la **subclase**.

El ejemplo muestra cómo la **herencia** permite reutilizar y extender el comportamiento en la programación orientada a objetos.

1.1.4 Comando super()

El comando super() es una función incorporada en Python que permite acceder a métodos de la **clase** base (o **superclase**) desde la **subclase** que los **hereda**. Esto es especialmente útil en la **herencia**, donde una **subclase** puede querer extender o modificar el comportamiento de la **clase** base.

En el contexto de los constructores (métodos **__init__**), **super()** se usa para asegurarse de que la inicialización que ocurre en la **clase** base también se aplique a las instancias de la **subclase**. Esto es importante porque la **subclase** puede depender de la lógica de inicialización definida en la **clase** base.

Ejemplo: Uso del comando **super()**

Código Python 8
Uso del comando super()

Código Python

```
# Definimos la subclase Automovil que hereda de Vehiculo
class Automovil(Vehiculo):
    def __init__(self, marca, modelo, numero_puertas):
        super().__init__(marca, modelo)  # Llamada al constructor de la clase
base
        self.numero_puertas = numero_puertas
```

Resultado: Código Python 8, formulación de los autores para el uso del comando super().

Beneficios de usar super():

- Mantenimiento del Código: Facilita la actualización y mantenimiento del código, ya que los cambios en la inicialización de la **clase** base no requieren cambios en cada **subclase**.
- Cooperación entre **Clases**: Permite a múltiples **clases** base ser inicializadas correctamente, útil en la herencia múltiple.
- Evita Problemas Comunes: Evita problemas comunes como llamar explícitamente al constructor de la **clase** base por su nombre, lo cual puede llevar a errores si el nombre de la **clase** base cambia.

Ejemplo: Crear una **clase** llamada **Persona** que incluye atributos como **nombre**, **edad**, **nacionalidad** y métodos como **hablar** y **caminar**. Una subclase llamada **Empleado** puede heredar estos atributos y métodos y agregar otros atributos específicos, como **trabajo** y **salario** y métodos como **trabajar.**

Código Python 9

Heredar atributos y métodos, agregar otros atributos específicos

Código Python

```
# Definimos la clase Persona, que es la clase base para personas
class Persona:
    def __init__(self, nombre, edad, nacionalidad):
        self.nombre = nombre
        self.edad = edad
        self.nacionalidad = nacionalidad

    # Método para que la persona hable
    def hablar(self, mensaje):
        print(f"{self.nombre} dice: {mensaje}")

    # Método para que la persona camine
    def caminar(self):
        print(f"{self.nombre} está caminando.")

# Definimos la subclase Empleado que hereda de Persona
class Empleado(Persona):
    def __init__(self, nombre, edad, nacionalidad, trabajo, salario):
       super().__init__(nombre, edad, nacionalidad)  # Llamada al constructor
de la clase base
        self.trabajo = trabajo
        self.salario = salario

    # Método específico de Empleado para trabajar
    def trabajar(self):
         print(f"{self.nombre} está trabajando como {self.trabajo} y gana
{self.salario} al mes.")

# Creación de un objeto de la clase Empleado
mi_empleado = Empleado("Juan", 30, "Española", "Ingeniero", 3000)

# Uso de los métodos heredados y del método específico
mi_empleado.hablar("¡Hola, estoy en el trabajo!")
mi_empleado.caminar()
mi_empleado.trabajar()
```

Resultado: Código Python 9, formulación de los autores para heredar atributos y métodos, agregar otros atributos específicos

Explicación del código:

- **Clase Persona:** Es la **clase** base que tiene atributos como **nombre**, **edad** y **nacionalidad**, además de métodos básicos como **hablar(mensaje)** y **caminar()**, que permiten realizar acciones generales de una persona.
- **Constructor de Persona:** Inicializa los atributos de la instancia con los valores proporcionados.
- **Subclase Empleado:** Hereda de **Persona** y añade atributos específicos para un empleado: **trabajo** y **salario**. También incluye un método adicional, **trabajar()**, que muestra un mensaje específico sobre el trabajo del empleado.
- **Constructor de Empleado:** Utiliza **super()** para llamar al constructor de la **clase** base **Persona** para inicializar los atributos **heredados**. Luego, inicializa los atributos específicos de **Empleado**.
- **Creación y uso de un objeto Empleado:** Se crea una instancia de **Empleado** y se utilizan tanto los métodos **heredados** de la **clase** base como los métodos específicos de la **subclase**.

1.1.5 Herencia múltiple

La **herencia** múltiple permitirá que una **clase herede** las características de varias **clases**, es decir tanto atributos como métodos. Hace que la **subclase** pueda construirse sobre las funcionalidades de las **clases** base, adaptándolas y extendiéndolas según sea necesario.

Ejemplo: Se crear la **clase Vehículo** con los métodos **arrancar** y **detener**, la clase **Acuático** con el método **navegar**, y luego la **subclase Anfibio** podría heredar de ambas para poseer todas estas capacidades.

Código Python 10

Herencia múltiple, una clase herede las características de varias clases

Código Python

```
# Clase Vehiculo con métodos básicos para cualquier tipo de vehículo
class Vehiculo:
    def arrancar(self):
        print("El vehículo ha arrancado.")

    def detener(self):
        print("El vehículo se ha detenido.")

# Clase Acuatico específica para vehículos que navegan en el agua
class Acuatico:
    def navegar(self):
        print("El vehículo está navegando.")

# Clase Anfibio que hereda de Vehiculo y Acuatico
class Anfibio(Vehiculo, Acuatico):
    def __init__(self, nombre):
       self.nombre = nombre  # Nombre del vehículo anfibio para identificación

    # Método adicional para demostrar todas las capacidades
    def demostrar_capacidades(self):
        print(f"Demostración de {self.nombre}:")
        self.arrancar()
        self.navegar()
        self.detener()

# Creación de un objeto de la clase Anfibio
mi_anfibio = Anfibio("AquaRanger")

# Uso del método que demuestra todas las capacidades del anfibio
mi_anfibio.demostrar_capacidades()
```

Resultado: Código Python 10, formulación de los autores para la herencia múltiple, una clase herede las características de varias clases.

Explicación del código

- **Clase Vehiculo:** Es una **clase** base que proporciona métodos básicos **arrancar()** y **detener()**, comunes a todos los vehículos.

- **Clase Acuatico:** Otra **clase** base diseñada para vehículos que operan en el agua, con un método **navegar()** específico para la navegación.
- **Clase Anfibio:** Esta es una **subclase** que hereda de **Vehiculo** y **Acuatico**. Hereda los métodos de ambas **clases**, lo que le permite tanto arrancar y detener como navegar.
- **Constructor __init__(self, nombre):** Inicializa la instancia de **Anfibio** con un nombre específico y proporciona un contexto útil para identificar el vehículo durante las demostraciones.
- **Método demostrar_capacidades():** Utiliza los métodos heredados para demostrar que puede operar tanto en tierra como en agua.

1.2 Estructuras de programación que utilizan los pilares de la POO

En las estructuras de programación que utilizan los pilares de la **POO**, se explorarán conceptos fundamentales para el desarrollo de software estructurado y eficiente. Este módulo cubrirá el polimorfismo, destacando cómo los diferentes **objetos** pueden compartir una interfaz común para ejecutar acciones específicas. También se profundizará en el **encapsulamiento** y la **abstracción**, técnicas esenciales para ocultar la complejidad y exponer solo lo necesario. Esto facilitará el mantenimiento y la escalabilidad del código, además, se estudiarán las **clases abstractas**.

1.2.1 Polimorfismo

Es un principio que permite a **objetos** de diferentes **clases** ser tratados como instancias de una **clase** común. Esto se logra mediante la implementación de métodos con el mismo nombre, pero comportamientos distintos en cada **clase**. El **polimorfismo** también facilita la flexibilidad en el código, permitiendo que una función o método interactúe con **objetos** de múltiples tipos y seleccionando automáticamente la implementación adecuada del método según el tipo de objeto que lo invoca (Lott & Phillips, 2021).

Ejemplo: Se construye una **clase** base llamada **Animal** con un método **hacerSonido()**. **Clases** derivadas como **Perro** y **Gato** pueden implementar este método de maneras específicas a su especie, de modo que un **Perro** ladre y un **Gato** maúlle al momento de invocar **hacerSonido()**. El polimorfismo permite escribir código que opera en una colección de **Animales**, se llama al método **hacerSonido()** sin necesidad de saber si el animal es un perro o un gato, ya que se proporciona una interfaz común para interactuar con todos los objetos derivados de **Animal**.

Código Python 11

Polimorfismo, método que puede implementarse de varias maneras.

Código Python

```python
# Clase base Animal con un método hacerSonido
class Animal:
    def hacerSonido(self):
        raise NotImplementedError("Este método debe ser implementado por subclases.")
# Clase derivada Perro que implementa su versión específica de hacerSonido
class Perro(Animal):
    def hacerSonido(self):
        print("Guau guau!")
# Clase derivada Gato que implementa su versión específica de hacerSonido
class Gato(Animal):
    def hacerSonido(self):
        print("Miau miau!")
# Función que acepta un objeto Animal y llama a su método hacerSonido
def emitirSonido(animal):
    animal.hacerSonido()

# Creación de objetos de las clases Perro y Gato
mi_perro = Perro()
mi_gato = Gato()

# Lista de animales
animales = [mi_perro, mi_gato]

# Uso del polimorfismo: se llama a hacerSonido sin saber exactamente si el animal es un perro o un gato
for animal in animales:
    emitirSonido(animal)
```

Resultado: Código Python 11, formulación de los autores para el uso del Polimorfismo, clase con un método que puede implementarse de varias maneras.

Explicación del código:

- **Clase Animal:** Esta es la **clase** base que define un método **hacerSonido()**. El método en la **clase** base lanza una excepción **NotImplementedError** para asegurar que cualquier **subclase** debe implementar este método. Esto garantiza que todas las **subclases** sigan una interfaz común.

- **Clases Perro y Gato:** Son **subclases** de **Animal**. Cada una de estas **clases** implementa el método **hacerSonido()** de manera que refleje el sonido típico de cada especie.

- **Función emitirSonido(animal):** Esta función toma un objeto **Animal** como parámetro y llama a su método **hacerSonido()**. La función no necesita saber el tipo específico del objeto **Animal** para llamar a este método, lo que demuestra el uso del polimorfismo.

- **Creación y manejo de objetos:** Se crean instancias de **Perro** y **Gato**, y se agregan a una lista **animales**. Luego, se itera sobre esta lista y se llama a **emitirSonido()** para cada animal. A pesar de que **emitirSonido()** solo sabe que maneja objetos **Animal**, el comportamiento correcto (ladrido o maullido) se activa según el tipo de objeto real.

Ejemplo: Se construye una **clase** base llamada **Figura** con un método **dibujar(),** y dos **clases** derivadas, **Círculo** y **Cuadrado**, que implementan el método **dibujar()** de maneras específicas a su forma.

Código Python 11
Polimorfismo, método dibujar()

Código Python

```
# Clase base Figura con un método dibujar
class Figura:
    def dibujar(self):
         raise NotImplementedError("Este método debe ser implementado por
subclases.")

# Clase derivada Circulo que implementa su versión específica de dibujar
class Circulo(Figura):
    def dibujar(self):
        print("Dibujando un círculo.")

# Clase derivada Cuadrado que implementa su versión específica de dibujar
class Cuadrado(Figura):
    def dibujar(self):
        print("Dibujando un cuadrado.")

# Función que acepta un objeto Figura y llama a su método dibujar
def dibujarFigura(figura):
    figura.dibujar()

# Creación de objetos de las clases Circulo y Cuadrado
mi_circulo = Circulo()
mi_cuadrado = Cuadrado()

# Lista de figuras
figuras = [mi_circulo, mi_cuadrado]

# Uso del polimorfismo: se llama a dibujar sin saber exactamente si la figura
es un círculo o un cuadrado
for figura in figuras:
    dibujarFigura(figura)
```

Resultado: Código Python 12, formulación de los autores para el uso del Polimorfismo, método dibujar().

Explicación del código:

- **Clase Figura:** Es la **clase** base que define un método abstracto **dibujar()**. Al lanzar **NotImplementedError**, se asegura que cualquier **subclase** implementará este método y proporcionará una interfaz común para todas las figuras.
- **Clases Círculo y Cuadrado:** Son **subclases** de **Figura** y cada una implementa el método **dibujar()** de una manera que es coherente con la forma que representan.
- **Función dibujarFigura(figura):** Esta función acepta un **objeto Figura** y llama a su método **dibujar()**. La función no necesita conocer el tipo específico de la figura para invocar este método, lo que demuestra el principio del polimorfismo.
- **Manejo de objetos:** Se crean instancias de **Círculo** y **Cuadrado**, que se almacenan en una lista **figuras**. Durante la iteración sobre esta lista, se llama a **dibujarFigura()** para cada **objeto**, se activa el comportamiento correcto y depende del tipo de figura.

1.2.2 Encapsulamiento

El **encapsulamiento** es un principio fundamental de la **POO** que implica el empaquetamiento de los datos (atributos) y el código (métodos) que opera sobre los datos dentro de una unidad o clase. Este principio asegura que los datos de un **objeto** están ocultos al mundo exterior, accesibles solo a través de métodos definidos en la **clase**, lo que ayuda a proteger la integridad de los datos y a evitar usos indebidos. El encapsulamiento también permite un mejor control sobre cómo se accede y se modifica el estado interno de los objetos, promueve así un diseño más seguro y robusto (Lott & Phillips, 2021).

Ejemplo: Una **clase CuentaBancaria** puede tener atributos como **saldo** y métodos como **depositar()** o **retirar()**, que manipulan estos atributos de manera controlada. Previne el accesos o modificaciones directas al **saldo** por medios no autorizados.

Código Python 12
Encapsulamiento, clase para manipular atributos de manera controlada.

Código Python

```
# Clase CuentaBancaria que encapsula el atributo saldo
class CuentaBancaria:
    def __init__(self, saldo_inicial=0):
         self.__saldo = saldo_inicial  # El saldo es un atributo privado,
encapsulado

    def depositar(self, cantidad):
        if cantidad > 0:
            self.__saldo += cantidad
            print(f"Depósito exitoso. Saldo actual: {self.__saldo}")
        else:
            print("El monto a depositar debe ser positivo.")

    def retirar(self, cantidad):
        if cantidad > 0 and cantidad <= self.__saldo:
            self.__saldo -= cantidad
            print(f"Retiro exitoso. Saldo actual: {self.__saldo}")
        else:
            print("Fondos insuficientes o monto inválido.")

    def mostrar_saldo(self):
        print(f"El saldo actual es: {self.__saldo}")

# Creación y uso de una cuenta bancaria
mi_cuenta = CuentaBancaria(100)
mi_cuenta.depositar(50)
mi_cuenta.retirar(20)
mi_cuenta.mostrar_saldo()
```

Resultado: Código Python 13, formulación de los autores para el uso del Encapsulamiento, clase para manipular atributos de manera controlada.

Explicación del código:

- **Atributo __saldo:** El uso de dos guiones bajos antes del nombre **saldo** hace que este atributo sea privado. Esto significa que el atributo no puede ser modificado directamente desde fuera de la **clase**, lo cual es una práctica de **encapsulamiento**.

- **Método depositar():** Permite agregar una cantidad al saldo. Verifica que la cantidad sea positiva antes de realizar el depósito.
- **Método retirar():** Permite retirar una cantidad del saldo. Verifica que haya suficientes fondos y que la cantidad sea positiva antes de realizar el retiro.
- **Método mostrar_saldo():** Proporciona un medio controlado para acceder al saldo, mostrándolo en consola. Esto es útil para evitar el acceso directo al atributo privado **__saldo**.

Este ejemplo muestra cómo el **encapsulamiento** puede ser usado para proteger los datos y asegurar que solo sean modificados de manera controlada a través de métodos específicos de la **clase**. Es una buena práctica en la **POO** porque mejora la seguridad de los datos y reduce los errores de programación.

Ejemplo: Se crea una **clase Coche** que encapsula tanto los detalles del estado del motor como los métodos para manipular esos detalles. Se evita el acceso directo y asegura que el uso sea a través de métodos específicos y controlados.

Código Python 13

Encapsulamiento puede ser usado para proteger los datos

Código Python

```
# Clase Coche que encapsula tanto los atributos como los métodos relacionados con el motor
class Coche:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo
        self.__motor_encendido = False  # Atributo privado que almacena el estado del motor
    def __encender_motor(self):
        if not self.__motor_encendido:
            self.__motor_encendido = True
            print(f"Motor encendido. {self.marca} {self.modelo} listo para conducir.")
        else:
            print("El motor ya está encendido.")

    def __apagar_motor(self):
        if self.__motor_encendido:
            self.__motor_encendido = False
            print("Motor apagado.")
        else:
            print("El motor ya está apagado.")

    def conducir(self):
        if not self.__motor_encendido:
            print("El motor está apagado. Enciende el motor primero.")
        else:
            print(f"Conduciendo {self.marca} {self.modelo}.")

    def encender(self):
        self.__encender_motor()

    def apagar(self):
        self.__apagar_motor()
# Creación y uso de un objeto Coche
mi_coche = Coche("Toyota", "Corolla")
mi_coche.encender()
mi_coche.conducir()
mi_coche.apagar()
```

Resultado: Código Python 14, formulación de los autores para el uso del Encapsulamiento.

Explicación del código:

- **Atributos y Métodos Privados:** El atributo **__motor_encendido** y los métodos **__encender_motor()** y **__apagar_motor()** son privados (denotados por dos guiones bajos __), lo que significa que no pueden ser accedidos o modificados directamente desde fuera de la **clase**.
- **Métodos Públicos: encender()**, **apagar()** y **conducir()** son métodos públicos que los usuarios de la **clase** pueden llamar. Estos métodos actúan como interfaces públicas para las funcionalidades privadas del coche, permiten operaciones como encender y apagar el motor, además, conducir, pero siempre a través de los métodos controlados.
- **Encapsulamiento de Funcionalidades:** Los métodos privados controlan el estado del motor y garantizan que el coche solo se pueda conducir si el motor está encendido, protege el estado del motor contra manipulaciones inadecuadas.

1.2.3 Abstracción

La **abstracción** es un principio que se enfoca en ocultar la complejidad detallada de las implementaciones y exponer solo las características esenciales de un **objeto** o **clase**. Esto permite a los desarrolladores trabajar con conceptos a un nivel más general y menos detallado. La abstracción también ayuda a reducir la complejidad y permite manejar sistemas más grandes y complejos de manera más eficiente. Se facilita la reutilización de código y mejora la mantenibilidad del software (Lott & Phillips, 2021).

Ejemplo: Se diseña una aplicación de software para una biblioteca que gestiona préstamos de libros. La **abstracción** se podría utilizar para crear una **clase Libro** que contenga métodos como **prestar()** y **devolver()**, estos métodos simplifican las

operaciones que se pueden realizar con un **objeto** libro, como cambiar su estado de "disponible" a "prestado" y viceversa. El usuario o el programador no necesita conocer los detalles de cómo se actualiza el inventario, cómo se registra un préstamo en la base de datos o cómo se notifica a los usuarios sobre la disponibilidad del libro. La **clase Libro** abstrae todos estos detalles y permite que el usuario interactúe con los libros de la biblioteca de manera sencilla y directa sin preocuparse por la complejidad subyacente del sistema.

Código Python 14
Abstracción, clase que contenga métodos que simplifican las operaciones

Código Python

```python
# Clase Libro que abstrae los detalles de las operaciones de préstamo y devolución
class Libro:
    def __init__(self, titulo, autor):
        self.titulo = titulo
        self.autor = autor
        self.esta_prestado = False

    def prestar(self):
        if not self.esta_prestado:
            self.esta_prestado = True
            self.__actualizar_inventario()
            print(f"El libro '{self.titulo}' ha sido prestado.")
        else:
            print("El libro ya está prestado.")

    def devolver(self):
        if self.esta_prestado:
            self.esta_prestado = False
            self.__actualizar_inventario()
            print(f"El libro '{self.titulo}' ha sido devuelto.")
        else:
            print("Este libro no estaba prestado.")
    def __actualizar_inventario(self):
# Detalles de cómo se actualiza el inventario o se registra en la base de datos
print("Actualizando el estado del inventario en la base de datos.")
# Uso de la clase Libro
libro1 = Libro("1984", "George Orwell")
libro1.prestar()
libro1.devolver()
```

Resultado: Código Python 15, formulación de los autores para el uso de la Abstracción.

Explicación del código:

- **Clase Libro:** Esta **clase** representa un libro y **abstrae** las operaciones de préstamo y devolución de libros.
- **Atributos: título**, **autor**, y **esta_prestado** son atributos que definen las características básicas de un libro y su estado de préstamo.
- **Métodos prestar() y devolver():** Estos métodos permiten cambiar el estado de disponibilidad del libro, imprimen mensajes relevantes y llama a un método privado que simula la actualización del inventario.
- **Método Privado __actualizar_inventario():** Este método es privado y representa la abstracción de los detalles de la actualización de inventario. Esto podría involucrar interactuar con una base de datos o con un sistema de gestión de biblioteca, pero esos detalles están ocultos al usuario.

Este ejemplo demuestra cómo la **abstracción** permite simplificar la interacción con **objetos** complejos. Los usuarios de la **clase Libro** pueden prestar y devolver libros usando métodos simples sin necesidad de entender o manejar la complejidad del sistema de gestión de la biblioteca que opera detrás de escena. Esta simplificación hace que el código sea más legible, mantenible y fácil de usar, especialmente para programadores que no necesitan preocuparse por los detalles de implementación internos de las operaciones de inventario.

Ejemplo: Se crea una **clase Impresora** que encapsule las operaciones de **impresión** y **mantenimiento** de una manera simplificada

Código Python 15

Crea una clase Impresora que encapsule varias operaciones

Código Python

```
# Clase Impresora que abstrae los detalles de imprimir documentos y realizar
mantenimiento
class Impresora:
    def __init__(self, modelo):
        self.modelo = modelo
        self.tinta = 100  # Nivel inicial de tinta en porcentaje

    def imprimir(self, paginas):
        if self.tinta >= paginas * 0.5:  # Cada página impresa consume 0.5%
de la tinta
            self.tinta -= paginas * 0.5
                  print(f"Imprimiendo  {paginas}  páginas.  Tinta  restante:
{self.tinta}%")
        else:
            print("No hay suficiente tinta, por favor recargue.")
            self.recargar_tinta()

    def recargar_tinta(self):
        self.tinta = 100
        print("La tinta ha sido recargada al 100%.")

    def realizar_mantenimiento(self):
        self.__limpiar_cabezales()
       print("Mantenimiento completado. La impresora está lista para usar.")

    def __limpiar_cabezales(self):
       # Este método privado simula la limpieza de los cabezales de impresión
        print("Limpiando los cabezales de impresión.")

# Uso de la clase Impresora
mi_impresora = Impresora("Epson L220")
mi_impresora.imprimir(10)
mi_impresora.imprimir(150)
mi_impresora.realizar_mantenimiento()
```

Resultado: Código Python 16, formulación de los autores para crea una clase Impresora que encapsule varias operaciones.

Explicación del Código:

Clase Impresora: Representa una impresora que puede imprimir documentos y realizar su propio mantenimiento.

Atributos:

- **modelo**: Identifica el modelo de la impresora.
- **Tinta**: Representa el nivel de tinta actual en la impresora, inicia al 100%.

Métodos Públicos:

- **imprimir(paginas)**: Permite imprimir un número especificado de páginas, verifica si hay suficiente tinta y reduce el nivel de tinta de acuerdo con el uso. Si la tinta no es suficiente, llama al método de recargar tinta.
- **recargar_tinta()**: Recarga la tinta de la impresora al 100% y notifica al usuario.
- **realizar_mantenimiento()**: Realiza el mantenimiento de la impresora, abstrae la limpieza de cabezales mediante un método privado.

Método Privado:

__limpiar_cabezales(): Simula la limpieza de los cabezales de impresión, un detalle técnico que no necesita ser expuesto al usuario de la clase.

Beneficios de la abstracción en este ejemplo

- **Simplicidad:** Los usuarios de la **clase Impresora** pueden realizar tareas complejas como imprimir y mantener la impresora usando métodos simples sin preocuparse por los detalles internos.
- **Encapsulamiento:** Los detalles técnicos como la limpieza de cabezales y el cálculo del uso de tinta están ocultos, protege estos procesos de intervenciones externas y uso incorrecto.
- **Mantenibilidad:** Cambios internos en cómo se realiza el mantenimiento o cómo se gestiona la tinta no afectan al código que utiliza esta **clase**, facilita las actualizaciones y el mantenimiento del código.

1.2.4 Clases abstractas

Las **clases abstractas** son estructuras que no pueden ser instanciadas directamente; es decir, no se pueden crear **objetos** de una clase **abstracta** de manera directa. Estas clases están diseñadas para ser **clases** base de otras **clases**, el propósito de una **clase abstracta** es proporcionar una definición base para otras **clases** que extiendan e implementen sus propios métodos. Las **clases abstractas** suelen contener uno o más métodos **abstractos**, que son métodos declarados, pero no implementados en la **clase abstracta**.

Por lo tanto, cualquier clase que herede de una clase abstracta debe implementar todos los métodos **abstractos** de su **clase** base. Esto proporciona las acciones específicas que esos métodos realizarán (Lott & Phillips, 2021).

Ejemplo: Una **clase abstracta Vehículo** podría tener un método **abstracto moverse()**, que luego sería implementado de manera específica en las **clases** derivadas como **Automóvil**, **Bicicleta** y **Motocicleta** que hereden de **Vehículo**.

Código Python 16
Clase Abstracta

Código Python

```
from abc import ABC, abstractmethod

# Clase abstracta Vehiculo
class Vehiculo(ABC):
    def __init__(self, nombre):
        self.nombre = nombre

    @abstractmethod
    def moverse(self):
         pass  # Los detalles de implementación serán definidos en las
subclases

# Clase derivada Automovil
class Automovil(Vehiculo):
    def moverse(self):
        print(f"El automóvil {self.nombre} se está moviendo rápidamente por
la carretera.")

# Clase derivada Bicicleta
class Bicicleta(Vehiculo):
    def moverse(self):
        print(f"La bicicleta {self.nombre} se está moviendo tranquilamente
por el carril bici.")

# Clase derivada Motocicleta
class Motocicleta(Vehiculo):
    def moverse(self):
        print(f"La motocicleta {self.nombre} se está moviendo ágilmente por
la ciudad.")

# Creación y uso de objetos de las subclases
mi_auto = Automovil("Ford Focus")
mi_bici = Bicicleta("Trek 820")
mi_moto = Motocicleta("Harley Davidson")

mi_auto.moverse()
mi_bici.moverse()
mi_moto.moverse()
```

Resultado: Código Python 17, formulación de los autores para una clase Abstracta

Explicación del código.

Clase Abstracta Vehiculo:

@abstractmethod: Este decorador indica que el método **moverse()** debe ser implementado por todas las **subclases** que hereden de **Vehiculo**. No se proporciona una implementación en la **clase Vehiculo**, solo se define la firma del método.

Constructor __init__: Inicializa el objeto **Vehiculo** con un nombre, que se usa en las implementaciones específicas de las subclases para personalizar los mensajes.

Clases Derivadas (Automovil, Bicicleta, Motocicleta):

Cada una de estas **clases** implementa el método **moverse()** de manera que refleje el modo específico de movimiento del vehículo. Por ejemplo, los automóviles se mueven rápidamente por carreteras, mientras que las bicicletas lo hacen más tranquilamente por carriles bici.

Uso de Clases Derivadas:

Se crean instancias de cada **clase** derivada y se llama al método **moverse()** para cada una. Esto muestra cómo el mismo método puede tener comportamientos diferentes y depende del tipo de objeto que lo invoque, ilustra el **polimorfismo** en acción.

1.2.5 Instrucción from abc import ABC, abstractmethod

En Python, esta instrucción es esencial para utilizar las características de las **clases abstractas** en la programación orientada a objetos. A continuación, el detalle para qué sirve cada parte de esta instrucción:

- **abc:** Es el módulo de "Abstract Base Classes" (Clases Base Abstractas) proporcionado por Python en su biblioteca estándar. Este módulo ofrece la infraestructura para definir **clases abstractas** en Python.
- **ABC:** Es una **clase** de ayuda que se utiliza como la **clase** base para definir una **clase abstracta**. Heredar de **ABC** asegura que la **clase** que se define es reconocida por

Python como una clase abstracta, lo que implica ciertas restricciones y comportamientos específicos, como la imposibilidad de instanciarla directamente.

abstractmethod: Es un decorador que se aplica a los métodos en una **clase abstracta**. Este decorador marca los métodos como **abstractos**, lo que significa que estos métodos **deben ser implementados** por cualquier **subclase** no **abstracta** que **herede** de esta clase **abstracta**. Un método **abstracto** puede tener una declaración, pero no una implementación efectiva en la **clase abstracta** misma.

Ejemplo: El uso de **ABC** y **abstractmethod** permite diseñar estructuras como la siguiente, donde no puedes crear una instancia de **Vehiculo** directamente, pero puedes utilizar sus **subclases** que implementan el método **moverse().**

Código Python 17
El uso de ABC y abstractmethod

Código Python

```python
from abc import ABC, abstractmethod

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

# Esto fallará: v = Vehiculo()
# Esto funcionará si se implementa moverse correctamente en la subclase.
class Bicicleta(Vehiculo):
    def moverse(self):
        print("La bicicleta se mueve.")

b = Bicicleta()
b.moverse()
```

Resultado: Código Python 18, formulación de los autores para el uso de ABC y abstractmethod.

Propósitos y beneficios:

- **Forzar la implementación de métodos:** El uso de **abstractmethod** obliga a las **subclases** a implementar los métodos abstractos definidos en la **clase** base **abstracta**. Esto es crucial en proyectos grandes y complejos donde se espera que ciertas interfaces tengan implementaciones garantizadas en todas las **subclases**.

- **Evitar la instanciación directa:** Al heredar de **ABC**, se evita la instanciación directa de la **clase** base. Esto asegura que solo las **subclases** con implementaciones completas puedan ser creadas. Además, ayuda a mantener la integridad y la coherencia del diseño orientado a objetos, ya que las **clases abstractas** a menudo representan conceptos de alto nivel o plantillas incompletas que no deben tener instancias por sí mismas.

- **Promover el polimorfismo:** Las **clases abstractas** son una excelente manera de asegurar que diferentes **clases** implementen interfaces comunes de manera uniforme, lo que facilita el diseño polimórfico y mejora la flexibilidad del código.

1.3 Ejercicios de aplicación

En esta sección se presentan actividades prácticas diseñadas para consolidar los conceptos fundamentales de clases, objetos y herencia en programación orientada a objetos.

1.3.1 Ejercicio 1: Aplicación de conceptos de clases, objetos y herencia

Se crea una clase **Estudiante** que tenga los atributos nombre, edad, semestre y se agrega un método llamado estudiar que imprima el mensaje: "El estudiante {nombre del estudiante} está estudiando". Para trabajar con instancias, se crea una instancia de esta **clase** para usar el método. Sin embargo, para esto habría que generar una interacción con el usuario, es decir, debe ser un requerimiento pedir el nombre, edad y grado.

Luego, se instanciará esta clase y mostrarán los datos de la **clase** cread. Después de registrar al estudiante, se pondrá una condición: si el usuario escribe la palabra estudiar, el estudiante puede estudiar indistintamente de mayúsculas o minúsculas. Si no, el usuario deberá ingresar más palabras.

Código Python 18
Solución ejercicio de aplicación 1: Conceptos de Clases, Objetos y Herencia

Código Python

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

    def estudiar(self):
        print(f"El estudiante {self.nombre} está estudiando.")

# Solicitar datos del estudiante al usuario
nombre = input("Ingrese el nombre del estudiante: ")
edad = int(input("Ingrese la edad del estudiante: "))
semestre = int(input("Ingrese el semestre actual del estudiante: "))

# Crear una instancia de la clase Estudiante
estudiante = Estudiante(nombre, edad, semestre)

# Mostrar datos del estudiante
print(f"Registrado: {estudiante.nombre}, Edad: {estudiante.edad}, Semestre:
{estudiante.semestre}")

# Bucle para esperar comando de estudio
while True:
    comando = input("¿Qué debe hacer el estudiante? (escriba 'estudiar' para
estudiar): ")
    if comando.lower() == 'estudiar':
        estudiante.estudiar()
        break  # Opcional: salir del bucle después de estudiar
    else:
        print("Comando no reconocido, intente de nuevo.")
```

Resultado: Código Python 19, formulación de los autores de la solución ejercicio de aplicación 1: Conceptos de clases, objetos y herencia.

Explicación del Código:

Clase Estudiante:

- La **clase Estudiante** tiene un constructor (**__init__**) que inicializa tres atributos: **nombre**, **edad**, y **semestre**. También tiene un método **estudiar()** que imprime un mensaje que indica que el estudiante está estudiando.

Interacción con el Usuario:

- El código solicita al usuario que ingrese el nombre, la edad y el semestre del estudiante. Estos datos se utilizan para crear una instancia de la clase **Estudiante**.
- Una vez creada la instancia, se imprime una confirmación con los datos del estudiante.

Bucle de Comando:

- Se entra en un bucle infinito que solicita al usuario escribir un comando.
- Si el usuario escribe "estudiar" (sin considerar mayúsculas o minúsculas), el método **Estudiar()** del objeto **estudiante** es llamado y el estudiante "estudia". Después de esto, el bucle se rompe y el programa podría finalizar o continuar con otra lógica.
- Si el usuario escribe cualquier otra cosa, el programa le indica que el comando no es reconocido y pide que intente de nuevo.

El ejercicio de aplicación 1 demuestra cómo se pueden combinar conceptos básicos de la **POO** con interacciones dinámicas del usuario para crear aplicaciones interactivas y funcionales.

1.3.2 Ejercicio 2: Aplicación de conceptos de clases, objetos, herencia, polimorfismo, encapsulamiento, abstracción y clases abstractas

Se crea clases para manejar atracciones, empleados y visitantes del parque. Las atracciones serán clasificadas en diferentes tipos, como Montaña Rusa, Carrusel y Casa Embrujada, cada una con comportamientos específicos. Los empleados tendrán roles

como Operadores de Atracción y Supervisores, y los visitantes podrán disfrutar de las atracciones y comprar comida.

Código Python 19
Solución ejercicio de aplicación 2: Polimorfismo, Encapsulamiento, Abstracción y Clases Abstractas

Código Python Parte 2

```
# Clase para los visitantes
class Visitante(ElementoParque):
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def interactuar(self):
          print(f"{self.nombre}, de {self.edad} años, está explorando el
parque.")

# Instanciación y demostración de polimorfismo
elementos = [
    MontañaRusa("Cyclone", 20),
    Carrusel("Merry-Go-Round", 30),
    CasaEmbrujada("Ghost House", 15),
    Empleado("Juan", "Operador de Atracción"),
    Visitante("Ana", 12)
]

for elemento in elementos:
    elemento.interactuar()
```

Resultado: Código Python 20, formulación de los autores para la solución ejercicio de aplicación 2: Polimorfismo, Encapsulamiento, Abstracción y Clases Abstractas

Explicación del código:

- **Clase Abstracta ElementoParque**: Define una interfaz común para todos los elementos del parque. Esta **abstracción** permite tratar a todas las entidades del parque de manera uniforme.

- **Clase Atraccion y Sus Subclases**: **Atraccion** es una **clase** base para todas las atracciones del parque. **Clases** como **MontañaRusa**, **Carrusel** y **CasaEmbrujada heredan** de ella y personalizan el método **interactuar()** para reflejar sus características únicas.

- **Clase Empleado y Visitante**: Estas **clases** modelan otras entidades del parque y también implementan **interactuar()**, mostrando cómo diferentes tipos de objetos pueden compartir la misma interfaz.

- **Demostración de Polimorfismo**: Al iterar sobre una lista de **ElementoParque** y llamar a **interactuar()**, cada objeto responde de una manera que es apropiada a su tipo concreto. Esto muestra el **polimorfismo** en acción.

El ejercicio de aplicación 2 proporciona una forma clara de entender cómo se aplican en la práctica los conceptos de **herencia**, **polimorfismo**, **encapsulamiento** y **abstracción** en el diseño de software orientado a objetos.

1.3.3 Ejercicio 3: Aplicación de conceptos POO (Programación Orientada a Objetos) en un contexto práctico Gestión de Reserva de Espacios para Eventos

Este ejercicio consiste en desarrollar un sistema para gestionar la reserva de diferentes tipos de espacios para eventos, como salones de conferencias, auditorios y salas de reuniones.

Cada tipo de espacio tiene características únicas y diferentes opciones de reserva, y el sistema debe ser capaz de gestionar estos espacios de manera flexible y segura, permitiendo a los usuarios consultar la disponibilidad, reservar espacios, y ver los detalles específicos de cada tipo de reserva.

Código Python 20
Solución al ejercicio de aplicación 3: Caso práctico: sistema de gestión de reserva de espacios para eventos

Código Python Parte 2

```
# Clase derivada Auditorio
class Auditorio(Espacio):
    def reservar(self):
        if self._disponible:
            self._disponible = False
            print(f"Auditorio '{self._nombre}' reservado.")
        else:
            print(f"Auditorio '{self._nombre}' no está disponible.")

    def cancelar_reserva(self):
        self._disponible = True
        print(f"Reserva del auditorio '{self._nombre}' cancelada.")

    def mostrar_detalle(self):
        estado = "disponible" if self._disponible else "ocupado"
         print(f"Auditorio: {self._nombre}, Capacidad: {self._capacidad},
Estado: {estado}")

# Clase derivada SalonConferencias
class SalonConferencias(Espacio):
    def reservar(self):
        if self._disponible:
            self._disponible = False
            print(f"Salón de conferencias '{self._nombre}' reservado.")
        else:
                print(f"Salón de conferencias '{self._nombre}' no está
disponible.")

    def cancelar_reserva(self):
        self._disponible = True
           print(f"Reserva del salón de conferencias '{self._nombre}'
cancelada.")

    def mostrar_detalle(self):
        estado = "disponible" if self._disponible else "ocupado"
            print(f"Salón de Conferencias: {self._nombre}, Capacidad:
{self._capacidad}, Estado: {estado}")
```

Código Python Parte 3

```python
# Ejemplo de uso
sala = SalaReuniones("Sala 101", 10)
auditorio = Auditorio("Auditorio Principal", 200)
salon = SalonConferencias("Conferencias A", 50)

# Interacción
sala.mostrar_detalle()
sala.reservar()
sala.mostrar_detalle()
sala.cancelar_reserva()
sala.mostrar_detalle()
```

Resultado: Código Python 21, formulación de los autores para la solución ejercicio de aplicación 3: Caso práctico sistema de gestión de reserva de espacios para eventos.

Explicación del Código

El código está diseñado para gestionar la reserva de distintos tipos de espacios de eventos, como **salas de reuniones**, **auditorios** y **salones de conferencias**. Utiliza conceptos de Programación Orientada a Objetos (POO) como **clases abstractas**, **herencia**, **polimorfismo**, **encapsulamiento** y **abstracción**.

Clase Abstracta Espacio:

La clase Espacio es una clase abstracta que define una interfaz común para todos los tipos de espacios de eventos.

Posee tres métodos abstractos:

- **reservar()**: Método para reservar el espacio.
- **cancelar_reserva()**: Método para cancelar una reserva y volver a poner el espacio como disponible.
- **mostrar_detalle()**: Método que muestra información detallada sobre el espacio.
- Tiene atributos encapsulados:
- **_nombre**: El nombre del espacio.
- **_capacidad**: La capacidad máxima del espacio.
- **_disponible**: Un atributo booleano que indica si el espacio está disponible para ser reservado.

La clase **Espacio** utiliza el módulo ABC (Abstract Base Classes) y el decorador @abstractmethod para definir métodos abstractos, lo que obliga a las subclases a implementar estos métodos.

Clases Derivadas (Subclases):

Las subclases **SalaReuniones**, **Auditorio**, y **SalonConferencias** heredan de **Espacio** y personalizan su funcionalidad, adaptándose a las características de cada tipo de espacio.

Clase SalaReuniones:

- Implementa los métodos abstractos reservar(), cancelar_reserva(), y mostrar_detalle() para manejar las reservas de salas de reuniones.
- **reservar()**: Cambia el estado de disponibilidad y muestra un mensaje de confirmación. Si el espacio ya está reservado, muestra un mensaje indicando que no está disponible.
- **cancelar_reserva()**: Devuelve el espacio a su estado disponible y confirma la cancelación.
- **mostrar_detalle()**: Muestra la capacidad y el estado de disponibilidad del espacio.

Clase Auditorio:

Similar a SalaReuniones, pero puede incluir atributos y comportamientos específicos de un auditorio en proyectos más complejos.

Clase SalonConferencias:

También sigue la misma estructura, pero podría incluir atributos específicos como configuraciones de mesas y sillas.

Cada subclase es responsable de definir su propia versión de los métodos reservar(), cancelar_reserva() y mostrar_detalle(). Gracias al polimorfismo, los métodos se llaman de la misma manera, pero su comportamiento es diferente dependiendo de la subclase que los implemente.

Encapsulamiento:

- Los atributos nombre, capacidad y disponible están encapsulados usando el prefijo de un guion bajo (_). Esto indica que estos atributos son internos a la clase y no deberían ser accesibles directamente desde fuera de la clase.
- El encapsulamiento asegura que los datos no sean modificados de forma incorrecta y que el acceso a ellos esté controlado mediante métodos.

Ejemplo de Uso:

- Se crean instancias de SalaReuniones, Auditorio, y SalonConferencias.
- Se llama a los métodos mostrar_detalle(), reservar() y cancelar_reserva() para demostrar cómo se reserva y cancela un espacio también cómo se muestra su disponibilidad.

El código proporciona una estructura modular y reutilizable para gestionar diferentes tipos de espacios, haciendo uso de:

- **Abstracción:** La clase Espacio define la estructura y métodos comunes.
- **Herencia:** Las subclases heredan de Espacio.
- **Polimorfismo:** Los métodos reservar(), cancelar_reserva(), y mostrar_detalle() se implementan de forma personalizada en cada subclase.
- **Encapsulamiento:** Atributos privados protegen los datos del objeto.

Este ejercicio da a los estudiantes una comprensión profunda de cómo aplicar los principios de la POO para resolver problemas reales en un contexto de gestión de espacios para eventos.

1.3.4 Ejercicio 4: Aplicación de conceptos POO (Programación Orientada a Objetos) en un contexto práctico: Gestión de citas médicas

El ejercicio consiste en desarrollar un sistema que permita gestionar citas para diferentes tipos de consultas médicas, como consultas generales, consultas de especialidades y consultas de urgencias.

Cada tipo de consulta tiene características específicas, como la duración, los requisitos previos y las limitaciones de tiempo. El sistema permitirá a los usuarios reservar y cancelar citas, además de mostrar detalles específicos de cada tipo de consulta.

Código Python 21
Solución al ejercicio de aplicación 4, caso práctico sistema de gestión de citas médicas

Código Python Parte 1

```
from abc import ABC, abstractmethod

# Clase base abstracta ConsultaMedica
class ConsultaMedica(ABC):
    def __init__(self, paciente, hora):
        self._paciente = paciente
        self._hora = hora
        self._disponible = True
    @abstractmethod
    def reservar_cita(self):
        pass
    @abstractmethod
    def cancelar_cita(self):
        pass

    @abstractmethod
    def mostrar_detalle(self):
        pass

# Clase derivada ConsultaGeneral
class ConsultaGeneral(ConsultaMedica):
    def reservar_cita(self):
        if self._disponible:
            self._disponible = False
           print(f"Cita de consulta general para {self._paciente} reservada
a las {self._hora}.")
        else:
            print("Cita de consulta general no está disponible.")

    def cancelar_cita(self):
        self._disponible = True
         print(f"Cita de consulta general para {self._paciente} ha sido
cancelada.")

    def mostrar_detalle(self):
        estado = "disponible" if self._disponible else "reservada"
          print(f"Consulta General - Paciente: {self._paciente}, Hora:
{self._hora}, Estado: {estado}")
```

Código Python Parte 2

```
# Clase derivada ConsultaEspecialidad
class ConsultaEspecialidad(ConsultaMedica):
    def reservar_cita(self):
        if self._disponible:
            self._disponible = False
            print(f"Cita de especialidad para {self._paciente} reservada a
las {self._hora}.")
        else:
            print("Cita de especialidad no está disponible.")

    def cancelar_cita(self):
        self._disponible = True
           print(f"Cita  de  especialidad  para  {self._paciente}  ha  sido
cancelada.")

    def mostrar_detalle(self):
        estado = "disponible" if self._disponible else "reservada"
         print(f"Consulta Especialidad - Paciente: {self._paciente}, Hora:
{self._hora}, Estado: {estado}")

# Clase derivada ConsultaUrgencias
class ConsultaUrgencias(ConsultaMedica):
    def reservar_cita(self):
        if self._disponible:
            self._disponible = False
           print(f"Cita de urgencias para {self._paciente} reservada a las
{self._hora}.")
        else:
            print("Cita de urgencias no está disponible.")

    def cancelar_cita(self):
        self._disponible = True
       print(f"Cita de urgencias para {self._paciente} ha sido cancelada.")

    def mostrar_detalle(self):
        estado = "disponible" if self._disponible else "reservada"
          print(f"Consulta  Urgencias  -  Paciente:  {self._paciente},  Hora:
{self._hora}, Estado: {estado}")
```

Código Python Parte 3

```
# Ejemplo de uso
cita_general = ConsultaGeneral("Juan Perez", "10:00 AM")
cita_especialidad = ConsultaEspecialidad("Maria Lopez", "11:00 AM")
cita_urgencias = ConsultaUrgencias("Carlos Diaz", "9:00 AM")

# Interacción
cita_general.mostrar_detalle()
cita_general.reservar_cita()
cita_general.mostrar_detalle()
cita_general.cancelar_cita()
cita_general.mostrar_detalle()
```

Resultado: Código Python 22, formulación de los autores para la solución ejercicio de aplicación 4: caso práctico sistema de gestión de citas médicas.

Explicación del Código

- **Clases Abstractas y Subclases:** La clase abstracta ConsultaMedica define la estructura de las consultas y obliga a sus subclases a implementar métodos para reservar, cancelar y mostrar detalles de las citas
- **Encapsulamiento:** Los datos del paciente, la hora y el estado de disponibilidad están encapsulados en atributos privados, accesibles y modificables solo a través de métodos definidos.
- **Polimorfismo y Abstracción:** Los métodos reservar_cita(), cancelar_cita() y mostrar_detalle() permiten interactuar con diferentes tipos de consultas de manera uniforme, haciendo que el sistema sea fácil de usar.

1.3.5 Ejercicio 5: Aplicación de conceptos POO (Programación Orientada a Objetos) en un contexto práctico: Gestión de cuentas bancarias

El ejercicio desarrolla un sistema de gestión de cuentas bancarias que permita a los usuarios manejar diferentes tipos de cuentas, como cuentas de ahorro, cuentas corrientes y cuentas de inversión. Cada tipo de cuenta tiene características y operaciones específicas, como realizar depósitos, retiros y consultas de saldo. El

sistema debe ser capaz de manejar estos diferentes tipos de cuentas de manera flexible y segura, aplicando cargos o beneficios particulares según el tipo de cuenta.

Código Python Parte 1

```python
from abc import ABC, abstractmethod

# Clase base abstracta CuentaBancaria
class CuentaBancaria(ABC):
    def __init__(self, numero_cuenta, titular, saldo_inicial=0):
        self._numero_cuenta = numero_cuenta
        self._titular = titular
        self._saldo = saldo_inicial

    @abstractmethod
    def depositar(self, cantidad):
        pass

    @abstractmethod
    def retirar(self, cantidad):
        pass

    def consultar_saldo(self):
            print(f"El saldo de la cuenta {self._numero_cuenta} es: {self._saldo}")

# Clase derivada CuentaAhorro
class CuentaAhorro(CuentaBancaria):
    def depositar(self, cantidad):
        if cantidad > 0:
            self._saldo += cantidad
            print(f"Depósito realizado. Nuevo saldo: {self._saldo}")
        else:
            print("La cantidad debe ser positiva.")

    def retirar(self, cantidad):
        if cantidad > self._saldo:
            print("Fondos insuficientes para el retiro.")
        elif cantidad > 500:
                print("No se pueden retirar más de 500 en una sola transacción.")
        else:
            self._saldo -= cantidad
            print(f"Retiro realizado. Nuevo saldo: {self._saldo}")
```

Código Python 22
Solución al ejercicio de aplicación 5: Caso práctico: sistema de gestión de cuentas bancarias

Código Python Parte 2

```
# Clase derivada CuentaCorriente
class CuentaCorriente(CuentaBancaria):
      def  __init__(self,  numero_cuenta,  titular,  saldo_inicial=0,
limite_sobregiro=200):
        super().__init__(numero_cuenta, titular, saldo_inicial)
        self._limite_sobregiro = limite_sobregiro

    def depositar(self, cantidad):
        if cantidad > 0:
            self._saldo += cantidad
            print(f"Depósito realizado. Nuevo saldo: {self._saldo}")
        else:
            print("La cantidad debe ser positiva.")

    def retirar(self, cantidad):
        if cantidad > self._saldo + self._limite_sobregiro:
            print("Excede el límite de sobregiro permitido.")
        else:
            self._saldo -= cantidad
            print(f"Retiro realizado. Nuevo saldo: {self._saldo}")

# Clase derivada CuentaInversion
class CuentaInversion(CuentaBancaria):
    def depositar(self, cantidad):
        if cantidad >= 1000:
            self._saldo += cantidad
            print(f"Depósito realizado. Nuevo saldo: {self._saldo}")
        else:
            print("Depósito mínimo de inversión es 1000.")

    def retirar(self, cantidad):
          print("No  se  permiten  retiros  hasta  el  final  del  plazo  de
inversión.")
```

Código Python Parte 3

```python
# Ejemplo de uso
cuenta_ahorro = CuentaAhorro("123", "Ana Perez", 1000)
cuenta_corriente = CuentaCorriente("456", "Carlos López", 500)
cuenta_inversion = CuentaInversion("789", "María García", 3000)

# Interacción con las cuentas
cuenta_ahorro.depositar(200)
cuenta_ahorro.retirar(700)
cuenta_ahorro.consultar_saldo()

cuenta_corriente.retirar(600)
cuenta_corriente.consultar_saldo()

cuenta_inversion.depositar(500)
cuenta_inversion.retirar(100)
cuenta_inversion.consultar_saldo()
```

Resultado: Código Python 23, formulación de los autores para la solución ejercicio de aplicación 5: caso práctico sistema de gestión de cuentas bancarias

Explicación del Código

Clases Abstractas y Subclases:

- La clase CuentaBancaria es abstracta y define la estructura para las cuentas bancarias, obligando a sus subclases a implementar los métodos de depósito y retiro.
- Cada subclase (CuentaAhorro, CuentaCorriente, CuentaInversion) define reglas específicas para la operación de sus métodos.

Encapsulamiento:

- Los atributos _numero_cuenta, _titular y _saldo están encapsulados, protegiendo la información sensible y controlando el acceso a los datos.

Polimorfismo y Abstracción:

Los métodos depositar() y retirar() se implementan de manera diferente en cada subclase, permitiendo manejar todos los tipos de cuentas de manera flexible mediante el polimorfismo.

1.3.6 Ejercicio 6: Aplicación de conceptos POO en el análisis de datos

Diseño del Sistema para Medidas de Tendencia Central, se crea una **clase abstracta AnalizadorDeDatos** que define métodos para calcular estas estadísticas, y luego se implementa **clases** concretas para cada medida de tendencia central (Media, Mediana y Moda).

Código Python 23
Solución ejercicio de aplicación 4: POO en el análisis de datos

Código Python Parte 1

Código Python Parte 2

```
# Clase para calcular la Moda
class Moda(AnalizadorDeDatos):
    def calcular(self):
        try:
            resultado = mode(self.datos)
            self.mostrar_resultado(resultado)
        except Exception as e:
            print(f"Error al calcular la moda: {str(e)}")

# Datos de ejemplo
datos = [1, 2, 2, 3, 4, 4, 4, 5, 5, 6]

# Instanciación y uso de las clases
calculadora_media = Media(datos)
calculadora_media.calcular()

calculadora_mediana = Mediana(datos)
calculadora_mediana.calcular()

calculadora_moda = Moda(datos)
calculadora_moda.calcular()
```

```
        self.mostrar_resultado(resultado)
```

Resultado: Código Python 24, formulación de los autores para la solución ejercicio de aplicación 3: POO en el análisis de datos

Explicación del código:

- **Clase Abstracta AnalizadorDeDatos**:

Esta **clase** sirve como una plantilla para los diferentes tipos de análisis de datos.

Posee un método abstracto **calcular()** que las **subclases** deben implementar para calcular diferentes estadísticas. También tiene un método **mostrar_resultado()** para imprimir el resultado del cálculo.

- **Subclases Media, Mediana y Moda**:

Cada una de estas **clases** extiende **AnalizadorDeDatos** y sobrescribe el método **calcular()** para realizar sus cálculos específicos, utilizando funciones del módulo **statistics** de Python.

Moda incluye manejo de excepciones para tratar casos donde no se pueda determinar un único valor más frecuente.

El ejercicio de aplicación 3 demuestra cómo la Programación Orientada a Objetos puede ser utilizada para estructurar un programa de análisis de datos, haciendo que el código sea más modular, reusable y fácil de mantener. Además, enseña cómo las **clases** pueden ser diseñadas para **encapsular** comportamientos específicos relacionados con el análisis de datos, proporcionan una base robusta para futuras extensiones y aplicaciones.

1.3.7 Ejercicio para desarrollar: Fundamentos de la Programación Orientada a Objetos

Desarrollar el Sistema de Gestión de Empleados Multifuncionales

Objetivos

Clases y Objetos:

- Entender cómo definir clases y crear objetos en Python.
- Aprender a encapsular datos y funcionalidades relacionadas en objetos.

Herencia:

- Utilizar la herencia para crear una jerarquía de clases que modelen empleados de diferentes especialidades.
- Comprender cómo la herencia permite reutilizar y extender el código.

Herencia Múltiple:

- Implementar herencia múltiple para permitir que un empleado tenga múltiples roles.
- Resolver conflictos y complicaciones que surgen con la herencia múltiple, como el problema del diamante.

Instrucciones

Esta actividad consiste en desarrollar un sistema para gestionar información sobre empleados en una empresa, donde algunos empleados pueden tener múltiples roles (como técnicos, gerentes y vendedores) simultáneamente. Los estudiantes crearán un sistema que permita registrar empleados, asignarles múltiples roles y gestionar tareas específicas que dependan de los roles que desempeñan.

Desarrollar las clases de acuerdo con las siguientes especificaciones

Clase Base Empleado:

- Atributos comunes como nombre, ID, email.
- Métodos comunes como mostrar_info() para mostrar información básica del empleado.

Subclases de Rol Específico:

- Clase Técnico: Implementa métodos específicos relacionados con las tareas técnicas.
- Clase Gerente: Añade métodos para la gestión de equipos y toma de decisiones.
- Clase Vendedor: Incluye métodos para manejar ventas y interactuar con clientes.

Clase de Herencia Múltiple EmpleadoMultifuncional:

- Hereda de Técnico, Gerente, y Vendedor.
- Maneja la asignación de tareas que involucran capacidades de más de una de sus clases base.

Desafíos Propuestos:

Gestión de Tareas Complejas:

- Desarrollar un sistema que asigne tareas a empleados multifuncionales basándose en sus roles activos y habilidades.
- Implementar una función que verifique las habilidades antes de asignar una tarea para asegurar que el empleado es apto para realizarla.

Interfaz de Usuario: Crear una interfaz de usuario amigable que permita a los supervisores gestionar los roles y tareas de los empleados fácilmente.

Flexibilidad en la Asignación de Roles: Permitir que los roles de los empleados sean dinámicamente modificables, permitiendo agregar o eliminar roles según las necesidades operativas de la empresa.

El ejercicio permitirá que los lectores apliquen todos los conocimientos adquiridos sobre los fundamentos de la Programación Orientada a Objetos. Al tener una comprensión básica de la programación orientada a objetos están listos para enfrentarse a desafíos más complejos. Además, les permitirá aplicar teoría en

situaciones de la vida real y aprender a manejar la complejidad que se presenta en el diseño de software.

CAPÍTULO II: MANEJO EFICIENTE DE ARREGLOS MULTIDIMENSIONALES

El manejo eficiente de arreglos multidimensionales es un pilar fundamental en la programación moderna, especialmente en aplicaciones que requieren manipular grandes volúmenes de datos o realizar cálculos complejos. Los arreglos multidimensionales, también conocidos como matrices, permiten organizar y procesar datos en estructuras ordenadas que simulan tablas, gráficos o espacios multidimensionales, lo que los convierte en herramientas esenciales en áreas como la inteligencia artificial, el análisis de datos, la simulación científica y el diseño de videojuegos.

En este capítulo, exploraremos las técnicas y buenas prácticas para trabajar con arreglos multidimensionales, centrándonos en cómo optimizar el uso de memoria, mejorar la velocidad de acceso y garantizar la legibilidad del código. Además, se analizarán casos prácticos en diversos lenguajes de programación, mostrando cómo implementar operaciones comunes, como la inicialización, manipulación, búsqueda y transformación de datos en estas estructuras. Este enfoque no solo fortalecerá la comprensión de su funcionamiento, sino que también destacará su importancia en la resolución de problemas complejos de manera eficiente.

1.4 Introducción a arreglos multidimensionales

Los arreglos multidimensionales son estructuras de datos que permiten almacenar información en forma de tablas, matrices o estructuras de mayor dimensión. Estas herramientas son útiles para organizar datos de manera eficiente y facilitar el acceso a ellos mediante índices. En este tema se explorarán las definiciones, características y operaciones básicas asociadas con arreglos de dos y más dimensiones.

1.4.1 Conceptos básicos de arreglos multidimensionales.

Los **arreglos multidimensionales** son estructuras de datos que permiten almacenar información en múltiples dimensiones, y organiza los datos en una forma tabular o en una matriz. En el contexto de la programación y la ciencia de datos, un arreglo puede ser visto como una colección de elementos del mismo tipo, dispuestos en filas y columnas, donde cada dimensión añade un nivel de profundidad y complejidad al arreglo. Estos **arreglos** son especialmente útiles en aplicaciones que requieren operaciones matemáticas y estadísticas complejas, como el procesamiento de imágenes, análisis de series temporales y modelado de datos **multidimensionales** (*Python Data Science Handbook | Python Data Science Handbook*, s. f.)

1.4.2 Librería NumPy del lenguaje Python

NumPy es una biblioteca esencial en Python para el manejo eficiente de **arreglos multidimensionales**. Ofrece una estructura de datos que optimiza tanto el uso de memoria como el rendimiento en tiempo de procesamiento y proporciona un amplio rango de funciones matemáticas como estadísticas que facilitan desde operaciones básicas hasta complejas transformaciones numéricas. Además, su capacidad de **broadcasting** permite realizar operaciones aritméticas entre **arreglos** de diferentes tamaños de manera eficiente.

NumPy es fundamental en el ecosistema de la ciencia de datos, ya que muchas otras bibliotecas, como Pandas, Matplotlib y scikit-learn se basan en ella para la manipulación y análisis de datos. Su versatilidad en la transformación de datos, como cambiar la forma, dividir y **combinar** arreglos, es crucial para el preprocesamiento en tareas de análisis avanzado y aprendizaje automático. Todo esto hace de NumPy una herramienta indispensable para los científicos de datos (*Guía del usuario de NumPy — Manual de NumPy v1.22*, s. f.).

1.4.3 Instalar NumPy

Paso 1: Verificar Python

Antes de instalar NumPy, es importante asegurarse de que Python está correctamente instalado en tu sistema. Puedes verificar esto si se ejecuta el siguiente comando en tu terminal o consola de comandos:

```
Bash
>python --version
```

Paso 2: Verificar o Instalar pip:

Pin es el gestor de paquetes para Python que facilita la instalación de librerías. Para verificar si ya tienes **pip** instalado, puedes ejecutar:

```
Bash
>pip --version
```

Paso 3: Instalar NumPy:

Una vez que tienes **pip**, puedes instalar NumPy. Se ejecuta el siguiente comando:

```
Bash
>pip install numpy
```

Paso 4: Verificar la Instalación

Para verificar que NumPy se ha instalado correctamente, puedes intentar importarlo en el intérprete de Python:

```
Código Python
import numpy as np
print(np.__version__)
```

1.5 Creación de Arreglos Multidimensionales

La creación de arreglos multidimensionales implica definir su tamaño, tipo de datos y estructura mediante código. Dependiendo del lenguaje de programación, se utilizan

diferentes métodos para declarar y llenar estos arreglos con valores iniciales. En este apartado se explicarán las técnicas fundamentales para crear arreglos de dos o más dimensiones.

1.5.1 Arreglos de una dimensión (Vectores)

Estos son los tipos más simples de arreglos, similares a una lista en Python, pero con un rendimiento optimizado para operaciones numéricas. Se utilizan comúnmente para almacenar líneas de datos, como series temporales o señales unidimensionales.

Luego de importar la librería NumPy, para crear un vector se usará la función **np.array()**, se ingresa como parámetro una lista de elementos. Aquí está la sintaxis básica:

Código Python 24
Arreglos de una dimensión

Código Python

```
# Importamos la libreria Numpy con el alias np
import numpy as np

# Crear un vector simple
vector = np.array([1, 2, 3, 4, 5])
print("Vector:", vector)

# Vector de ceros
vector_ceros = np.zeros(5)  # Crea un vector de 5 elementos, todos siendo
ceros
print("Vector de ceros:", vector_ceros)

# Vector de unos
vector_unos = np.ones(5)  # Crea un vector de 5 elementos, todos siendo unos
print("Vector de unos:", vector_unos)

# Vector de un rango específico de números
vector_rango = np.arange(1, 6)  # Crea un vector con números del 1 al 5
print("Vector de rango:", vector_rango)

# Vector de espacios lineales
vector_linspace = np.linspace(0, 1, 6)  # Crea un vector de 5 números
espaciados uniformemente entre 0 y 1
print("Vector linspace:", vector_linspace)
```

Resultado: Código Python 25, formulación de los autores para arreglos de una dimensión

1.5.2 Arreglos de dos dimensiones (Matrices)

Estos **arreglos** tienen dos dimensiones, lo que los hace ideales para representar matrices matemáticas, tablas de datos, o cualquier tipo de información organizada en forma de cuadrícula. Son fundamentales en operaciones de álgebra lineal, transformaciones de matrices, y en la manipulación de imágenes.

Para crear una matriz, se utilizará la función **np.array()** y se proporciona una lista de listas, donde cada lista representa una fila de la matriz.

Código Python 25
Arreglos de dos dimensiones

Código Python

```
import numpy as np

# Crear una matriz simple
matriz = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Matriz:\n", matriz)
```

Resultado: Código Python 26, formulación de los autores para arreglos de dos dimensiones

1.5.3 Arreglos de tres dimensiones

Utilizados frecuentemente en aplicaciones de ciencias de datos para representar datos espaciales o series de tiempo donde cada matriz de dos dimensiones es una capa en un punto de tiempo específico. También son muy utilizados en procesamiento de imágenes, donde cada capa puede representar un canal de color.

Código Python 26
Arreglos de tres dimensiones

Código Python

```
import numpy as np

arreglo_3d = np.array([
    [[1, 2, 3], [4, 5, 6]],
    [[7, 8, 9], [10, 11, 12]],
    [[13, 14, 15], [16, 17, 18]]
])
print("Arreglo 3D:\n", arreglo_3d)
```

Resultado: Código Python 27, formulación de los autores para arreglos de tres dimensiones

1.5.4 Arreglos de mayor dimensionalidad

A medida que aumentan las dimensiones, los **arreglos** se utilizan para aplicaciones más complejas, como el manejo de datos de múltiples variables a la vez, simulaciones en física que requieren representar varios parámetros espaciales y temporales, o para operaciones en aprendizaje automático y redes neuronales, donde las estructuras de datos pueden ser extremadamente **multidimensionales**.

Código Python 27
Arreglos multidimensionales

Código Python

```
import numpy as np

# Crear un arreglo de cuatro dimensiones
arreglo_4d = np.zeros((3, 2, 4, 5))  # 3 grupos, cada uno con 2 matrices, cada matriz de 4x5
print("Arreglo 4D de ceros:\n", arreglo_4d)
```

Resultado: Código Python 28, formulación de los autores para arreglos multidimensionales

1.5.5 Características de los Arreglos

Axis: Corresponde a cada una de las dimensiones del arreglo. Axis especificará con cuál de las dimensiones trabajar a lo largo de las diferentes operaciones.

Length: Es el número de elementos en una dimensión específica en el arreglo.

Index: El índice se refiere a la posición de un elemento dentro de un arreglo. Los índices en NumPy inician en cero, lo que significa que el primer elemento de cualquier dimensión tiene un índice de 0. La **indexación** es una herramienta poderosa para acceder a elementos específicos, rangos de elementos, o subconjuntos de un arreglo, y se puede realizar de varias maneras.

Value: Se refiere al valor específico almacenado en cada elemento del arreglo. En NumPy, cada valor en un arreglo tiene un tipo de dato definido, y todos los elementos de un arreglo en particular comparten el mismo tipo de dato, lo que garantiza una operación eficiente y homogénea.

Rank: El rank de un arreglo se refiere a la cantidad de dimensiones que tiene el arreglo. También se le conoce como el "rango" del arreglo, este término es especialmente importante para entender cómo se estructuran los datos dentro del arreglo y cómo se pueden realizar operaciones sobre los mismos.

Size: Indica el número total de elementos que contiene el arreglo, independientemente de sus dimensiones. Este atributo es útil para obtener una visión rápida del volumen de datos con los que se trabaja.

Shape: Es un atributo que describe la estructura del arreglo en términos de dimensiones. Representa una tupla de enteros que indica el tamaño del arreglo en cada dimensión.

1.5.6 Creación de arreglos

Una dimensión (Vector): Se trabaja en un proyecto que requiere calcular el cuadrado de cada número en un rango de valores, el código es:

Código Python 28
Crear un vector con un rango de valores

Código Python

```python
import numpy as np

# Crear un vector con un rango de valores del 1 al 10
vector = np.arange(1, 11)
print("Vector original:", vector)

# Calcular el cuadrado de cada elemento
vector_cuadrados = vector ** 2
print("Cuadrados del vector:", vector_cuadrados)
```

Resultado: Código Python 29, formulación de los autores para crear un vector con un rango de valores

El ejemplo muestra cómo manipular vectores de manera eficiente con NumPy. Realiza operaciones matemáticas sobre cada elemento del vector de forma vectorizada, lo que es mucho más rápido que usar bucles en Python.

Dos dimensiones (Matrices): Se necesita calcular la transpuesta de una matriz y luego sumar una matriz de unos:

Código Python 29

Crea una Matriz

Código Python

```python
import numpy as np

# Crear una matriz original
matriz = np.array([[1, 2], [3, 4], [5, 6]])
print("Matriz original:\n", matriz)

# Calcular la transpuesta de la matriz
matriz_transpuesta = matriz.T
print("Matriz transpuesta:\n", matriz_transpuesta)

# Crear una matriz de unos del mismo tamaño que la transpuesta
matriz_unos = np.ones(matriz_transpuesta.shape)
print("Matriz de unos:\n", matriz_unos)

# Sumar la matriz transpuesta y la matriz de unos
matriz_suma = matriz_transpuesta + matriz_unos
print("Resultado de la suma:\n", matriz_suma)
```

Resultado: Código Python 30, formulación de los autores par crea una Matriz

El ejemplo ilustra cómo manipular matrices, NumPy realiza operaciones típicas como transposiciones y sumas de matrices. Utilizar NumPy para estas operaciones hace que el código no solo sea más compacto y fácil de leer, sino también mucho más rápido que implementarlo manualmente.

Tres Dimensiones: Se trabaja en un proyecto que involucra el análisis de datos de temperatura a lo largo del tiempo en diferentes ubicaciones (las filas representan ubicaciones y columnas representan tiempos del día).

Código Python 30

Crear un arreglo tridimensional

Código Python

```
import numpy as np
# Crear un arreglo tridimensional para almacenar datos de temperatura
# 3 días, 2 ubicaciones, 4 mediciones de tiempo por día
temperaturas = np.array([
    [[21, 22, 23, 20], [19, 20, 21, 18]],
    [[22, 21, 23, 24], [20, 19, 21, 22]],
    [[23, 24, 22, 21], [22, 21, 23, 24]]
])
print("Datos de temperatura:\n", temperaturas)

# Calcular la temperatura promedio por ubicación a lo largo de todos los días
y tiempos
promedio_ubicaciones = np.mean(temperaturas, axis=(0, 2))
print("Promedio de temperatura por ubicación:", promedio_ubicaciones)
```

Resultado: Código Python 31, formulación de los autores para crear un arreglo tridimensional

El ejemplo muestra cómo crear y manipular **arreglos** tridimensionales, lo que facilita el cálculo de estadísticas a través de múltiples dimensiones de datos.

Más de tres dimensiones: Se Crea un arreglo donde cada "capa" representa un día diferente, cada "fila" una ubicación diferente, cada "columna" un producto diferente, y otra dimensión para diferentes métricas de evaluación del producto.

Código Python 31

Crear arreglos de más de tres dimensiones

Código Python

```
import numpy as np
# Crear un arreglo de cinco dimensiones
# Dimensiones: 2 semanas, 3 ubicaciones, 4 productos, 2 métricas, 5 días por semana
datos = np.random.rand(2, 3, 4, 2, 5)  # Usar random.rand para generar datos aleatorios
# Acceder a datos específicos
# Obtener los datos de la primera semana, todas las ubicaciones, primer producto, ambas métricas, todos los días
producto_1_datos = datos[0, :, 0, :, :]
print("Datos del primer producto en la primera semana:\n", producto_1_datos)
# Calcular promedios a lo largo de una dimensión específica
# Promedio de ventas del primer producto a lo largo de todas las ubicaciones y días en la primera semana
# Asumiendo que la métrica de ventas es el índice 0
ventas_promedio = np.mean(datos[0, :, 0, 0, :], axis=(0, 1))
print("Promedio de ventas del primer producto en la primera semana:", ventas_promedio)
```

Resultado: Código Python 32, formulación de los autores para crear arreglos de más de tres dimensiones

El Ejemplo ilustra cómo trabajar con **arreglos** de mayor dimensionalidad para organizar y analizar datos complejos. NumPy proporciona herramientas potentes que permiten manipular estos **arreglos** de manera eficiente, incluyendo la realización de operaciones matemáticas y estadísticas a lo largo de múltiples ejes.

1.6 Indexación y segmentación

La **indexación** en **arreglos** es un mecanismo que permite acceder y manipular elementos específicos dentro de un arreglo multidimensional. Esta técnica es fundamental para operaciones de selección y asignación de valores a segmentos de un arreglo. NumPy ofrece varias formas de **indexación**, que incluyen **indexación** básica con enteros y segmentos, **indexación** booleana que utiliza condiciones para seleccionar elementos, e **indexación** avanzada que permite la selección de elementos no contiguos y en patrones complejos (*NumPy*, s. f.)

Formas de indexación

Existen varias formas de **indexación** que permiten acceder y manipular los datos en un arreglo de manera flexible y eficiente. Estas son las principales formas de **indexación**:

1.6.1 Indexación básica con enteros

Puedes acceder a un elemento específico del arreglo con un índice para cada dimensión. Por ejemplo, arr[2, 3] accede al elemento en la tercera fila y cuarta columna. Ejemplo: Se crea un arreglo unidimensional para acceder a varios elementos, usa la **indexación** básica.

Código Python 32
Indexación básica con enteros

Código Python

```
import numpy as np

# Crear un arreglo 1D
arreglo_1d = np.array([10, 20, 30, 40, 50])
print("Arreglo 1D:", arreglo_1d)

# Acceder al primer elemento
print("Primer elemento:", arreglo_1d[0])

# Acceder al tercer elemento
print("Tercer elemento:", arreglo_1d[2])

# Acceder al último elemento
print("Último elemento:", arreglo_1d[-1])
```

Resultado: Código Python 33, formulación de los autores para la indexación básica con enteros

Ejemplo: Para un arreglo bidimensional se puede acceder de la siguiente forma

Código Python 33
Acceder a un arreglo bidimensional

Código Python

```python
import numpy as np
# Crear un arreglo 2D
arreglo_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Arreglo 2D:\n", arreglo_2d)

# Acceder al elemento en la primera fila, segunda columna
print("Elemento en la primera fila, segunda columna:", arreglo_2d[0, 1])

# Acceder al elemento en la última fila, última columna
print("Elemento en la última fila, última columna:", arreglo_2d[-1, -1])
```

Resultado: Código Python 34, formulación de los autores para acceder a un arreglo bidimensional

1.6.2 Indexación con slices (segmentos)

Permite seleccionar un rango de elementos dentro del arreglo, es similar a la **indexación** de listas en Python, pero aplicada a múltiples dimensiones, por ejemplo, arr[1:5, :3].

Ejemplo: Se crea un arreglo unidimensional para acceder a varios segmentos (slices) del mismo.

Código Python 34
Indexación con slices (segmentos)

Código Python

```python
import numpy as np

# Crear un arreglo 1D
arreglo_1d = np.array([10, 20, 30, 40, 50, 60])
print("Arreglo 1D:", arreglo_1d)

# Acceder a los primeros tres elementos
print("Primeros tres elementos:", arreglo_1d[:3])

# Acceder a los últimos dos elementos
print("Últimos dos elementos:", arreglo_1d[-2:])

# Acceder a los elementos desde el segundo hasta el cuarto
print("Elementos del segundo al cuarto:", arreglo_1d[1:4])

# Acceder a los elementos con un paso de 2
print("Elementos con un paso de 2:", arreglo_1d[::2])
```

Resultado: Código Python 35, formulación de los autores para la indexación con slices (segmentos)

Ejemplo: Para un arreglo bidimensional.

Código Python 35
Indexación de un arreglo bidimensional

Código Python

```
import numpy as np
# Crear un arreglo 2D
arreglo_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print("Arreglo 2D:\n", arreglo_2d)

# Acceder a todas las filas y las primeras dos columnas
print("Todas las filas, primeras dos columnas:\n", arreglo_2d[:, :2])

# Acceder a las primeras dos filas y todas las columnas
print("Primeras dos filas, todas las columnas:\n", arreglo_2d[:2, :])

# Acceder a las últimas dos filas y las últimas dos columnas
print("Últimas dos filas, últimas dos columnas:\n", arreglo_2d[-2:, -2:])

# Acceder a las filas alternas (es decir, cada dos filas)
print("Filas alternas:\n", arreglo_2d[::2, :])
```

Resultado: Código Python 36, formulación de los autores para la indexación de un arreglo bidimensional

1.6.3 Indexación con booleanos

Utiliza un arreglo de valores booleanos para seleccionar elementos, si el arreglo booleano tiene la misma forma que el arreglo de datos, selecciona los elementos correspondientes a los valores True.

Ejemplo: Se crea un arreglo unidimensional para utilizar condiciones booleanas y, de esta forma, filtrar elementos.

Código Python 36
Indexación con booleanos

Código Python

```
import numpy as np
# Crear un arreglo 1D
arreglo_1d = np.array([10, 20, 30, 40, 50, 60])
print("Arreglo 1D:", arreglo_1d)

# Crear una máscara booleana para filtrar elementos mayores a 30
mascara = arreglo_1d > 30
print("Máscara booleana:", mascara)

# Usar la máscara booleana para filtrar el arreglo
elementos_filtrados = arreglo_1d[mascara]
print("Elementos mayores a 30:", elementos_filtrados)

# Filtrar y modificar los elementos que cumplan la condición
arreglo_1d[arreglo_1d > 30] = 99
print("Arreglo 1D después de modificar elementos mayores a 30:", arreglo_1d)
```

Código Python 37
Indexación con booleanos arreglos bidimensionales

Código Python

```
import numpy as np
# Crear un arreglo 2D
arreglo_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print("Arreglo 2D:\n", arreglo_2d)
# Crear una máscara booleana para filtrar elementos mayores a 5
mascara = arreglo_2d > 5
print("Máscara booleana:\n", mascara)

# Usar la máscara booleana para filtrar el arreglo
elementos_filtrados = arreglo_2d[mascara]
print("Elementos mayores a 5:", elementos_filtrados)

# Filtrar y modificar los elementos que cumplan la condición
arreglo_2d[arreglo_2d > 5] = 0
print("Arreglo 2D después de modificar elementos mayores a 5:\n", arreglo_2d)
```

Resultado: Código Python 38, formulación de los autores para la indexación con

Ejemplo: Combinación de Condiciones Booleanas, se puede **combinar** múltiples condiciones booleanas, utiliza operadores lógicos (& para AND, | para OR).

Código Python 38
Combinación de Condiciones Booleanas

Código Python

```
import numpy as np
# Crear un arreglo 1D
arreglo_1d = np.array([10, 20, 30, 40, 50, 60])
print("Arreglo 1D:", arreglo_1d)

# Crear una máscara booleana para filtrar elementos entre 20 y 50 (inclusive)
mascara = (arreglo_1d >= 20) & (arreglo_1d <= 50)
print("Máscara booleana:", mascara)

# Usar la máscara booleana para filtrar el arreglo
elementos_filtrados = arreglo_1d[mascara]
print("Elementos entre 20 y 50:", elementos_filtrados)
```

Resultado: Código Python 39, formulación de los autores para la combinación de Condiciones Booleanas.

1.6.4 Indexación avanzada o fancy indexing

Se utiliza si los índices son **arreglos** de enteros o booleanos, permite seleccionar un subconjunto de un arreglo a partir de una secuencia arbitraria de índices.

Ejemplo: Se crea un arreglo unidimensional para acceder a varios elementos y utiliza listas de índices.

Código Python 39
Acceder a varios elementos utilizando listas de índices

Código Python

```python
import numpy as np

# Crear un arreglo 1D
arreglo_1d = np.array([10, 20, 30, 40, 50, 60])
print("Arreglo 1D:", arreglo_1d)

# Utilizar una lista de índices para acceder a elementos específicos
indices = [0, 2, 4]
elementos = arreglo_1d[indices]
print("Elementos en los índices [0, 2, 4]:", elementos)

# Modificar elementos específicos utilizando una lista de índices
arreglo_1d[indices] = [100, 200, 300]
print("Arreglo 1D después de modificar elementos en los índices [0, 2, 4]:", arreglo_1d)
```

Resultado: Código Python 40, formulación de los autores para acceder a varios elementos utilizando listas de índices

Ejemplo: Para un arreglo bidimensional, se utiliza listas de índices para acceder a elementos específicos.

Código Python 40

Utilizar listas de índices para acceder a elementos específicos

Código Python

```python
import numpy as np
# Crear un arreglo 2D
arreglo_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print("Arreglo 2D:\n", arreglo_2d)

# Utilizar listas de índices para acceder a elementos específicos
filas = [0, 1, 2]
columnas = [1, 3, 0]
elementos = arreglo_2d[filas, columnas]
print("Elementos en los índices [(0,1), (1,3), (2,0)]:", elementos)

# Modificar elementos específicos utilizando listas de índices
arreglo_2d[filas, columnas] = [20, 40, 60]
print("Arreglo 2D después de modificar elementos en los índices [(0,1), (1,3), (2,0)]:\n", arreglo_2d)
```

Resultado: Código Python 41, formulación de los autores para utilizar listas de índices para acceder a elementos específicos.

1.6.5 Cambio de forma, combinar y separar Arreglos

Cambio de Forma (Reshape): Esta operación permite modificar la estructura de un arreglo sin cambiar los datos que contiene. Es fundamental para adaptar un arreglo a una nueva forma para realizar operaciones específicas y mantiene el mismo número total de elementos.

Ejemplo: Se crea un arreglo original para cambiarlo de forma

Código Python 41
Cambio de forma de arreglos

Código Python

```
import numpy as np

# Crear un arreglo 1D con 12 elementos
arreglo_1d = np.arange(12)
print("Arreglo 1D original:")
print(arreglo_1d)

# Cambiar la forma a 2D (3 filas y 4 columnas)
arreglo_2d = arreglo_1d.reshape(3, 4)
print("\nArreglo 2D (3x4):")
print(arreglo_2d)

# Cambiar la forma a 2D (4 filas y 3 columnas)
arreglo_2d_4x3 = arreglo_1d.reshape(4, 3)
print("\nArreglo 2D (4x3):")
print(arreglo_2d_4x3)

# Cambiar la forma a 3D (2 matrices de 3 filas y 2 columnas)
arreglo_3d = arreglo_1d.reshape(2, 3, 2)
print("\nArreglo 3D (2x3x2):")
print(arreglo_3d)
```

Resultado: Código Python 42, formulación de los autores para el cambio de forma de arreglos

- El número total de elementos debe mantenerse constante. Por ejemplo, un arreglo con 12 elementos puede ser cambiado a una forma (3, 4), (4, 3), (2, 3, 2), etc., pero no a una forma que no multiplique a 12 (como (3, 5)).
- **reshape** devuelve una nueva vista del arreglo original siempre que sea posible. Si no es posible, se devuelve una copia.

Combinar Arreglos (Concatenate): La combinación de **arreglos** permite unir dos o más **arreglos** a lo largo de un eje específico. Esta funcionalidad es crucial para agrupar datos de diferentes fuentes que comparten dimensiones compatibles.

Ejemplo: Se crea varios **arreglos** para **combinarlos**

Código Python 42
Combinar Arreglos (Concatenate)

Código Python

```
import numpy as np

# Crear dos arreglos 1D
arreglo_1d_1 = np.array([1, 2, 3])
arreglo_1d_2 = np.array([4, 5, 6])
print("Arreglo 1D 1:", arreglo_1d_1)
print("Arreglo 1D 2:", arreglo_1d_2)

# Combinar los arreglos 1D
arreglo_1d_combinado = np.concatenate((arreglo_1d_1, arreglo_1d_2))
print("Arreglo 1D combinado:", arreglo_1d_combinado)

# Crear dos arreglos 2D
arreglo_2d_1 = np.array([[1, 2], [3, 4]])
arreglo_2d_2 = np.array([[5, 6], [7, 8]])
print("Arreglo 2D 1:\n",arreglo_2d_1)
print("Arreglo 2D 2:\n", arreglo_2d_2)

# Combinar los arreglos 2D a lo largo del eje 0 Filas
arreglo_2d_combinado_eje0 = np.concatenate((arreglo_2d_1, arreglo_2d_2),
axis=0)
print("Arreglo 2D combinado a lo largo del eje 0:\n",
arreglo_2d_combinado_eje0)

# Combinar los arreglos 2D a lo largo del eje 1 Columnas
arreglo_2d_combinado_eje1 = np.concatenate((arreglo_2d_1, arreglo_2d_2),
axis=1)
print("Arreglo 2D combinado a lo largo del eje 1:\n",
arreglo_2d_combinado_eje1)
```

Resultado: Código Python 43, formulación de los autores para combinar Arreglos (Concatenate).

- Al usar **np.concatenate**, se debe asegurar de que las dimensiones de los **arreglos** a **combinar** sean compatibles según el eje especificado. Por ejemplo, al **combinar** a lo largo del eje 0, el número de columnas debe coincidir en ambos **arreglos**, y al **combinar** a lo largo del eje 1, el número de filas debe coincidir.
- **np.concatenate** puede **combinar** más de dos **arreglos** si se pasan como una tupla o lista de arreglos.

Separar Arreglos (Split): La separación de un arreglo en varios sub-**arreglos** permite dividir datos complejos en componentes más manejables para análisis o procesamiento individual. NumPy ofrece varias funciones para dividir arreglos, permite especificar el número de divisiones o los índices donde se realizarán los cortes.

Código Python 43
Separar Arreglos (Split)

Código Python

```
import numpy as np

# Crear un arreglo 1D con 12 elementos
arreglo_1d = np.arange(12)
print("Arreglo 1D original:")
print(arreglo_1d)

# Dividir el arreglo en 3 sub-arreglos iguales
sub_arreglos = np.array_split(arreglo_1d, 3)
print("\nSub-arreglos después de dividir en 3 partes:")
for i, sub_arreglo in enumerate(sub_arreglos):
    print(f"Sub-arreglo {i+1}: {sub_arreglo}")

# Crear un arreglo 2D
arreglo_2d = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14,
15, 16]])
print("\nArreglo 2D original:")
print(arreglo_2d)

# Dividir el arreglo 2D en 2 sub-arreglos a lo largo del eje 0
sub_arreglos_2d_eje0 = np.array_split(arreglo_2d, 2, axis=0)
print("\nSub-arreglos 2D después de dividir a lo largo del eje 0:")
for i, sub_arreglo in enumerate(sub_arreglos_2d_eje0):
    print(f"Sub-arreglo {i+1}:\n{sub_arreglo}")
```

Resultado: Código Python 44, formulación de los autores para separar Arreglos (Split).

La función **split** de NumPy es muy útil para dividir **arreglos** en partes más pequeñas para diversas tareas de análisis y procesamiento. El ejemplo muestra cómo usar esta función para dividir **arreglos** unidimensionales y bidimensionales a lo largo de diferentes ejes. Esto te permite gestionar y reorganizar datos de manera eficiente.

1.7 Operaciones con arreglos multidimensionales

Las operaciones con arreglos multidimensionales permiten manipular datos organizados en estructuras de varias dimensiones. Entre las operaciones más comunes se encuentran el acceso, modificación, recorrido e iteración de elementos. Este tema abordará estas técnicas y su implementación en diferentes lenguajes de programación.

1.7.1 Operaciones matemáticas y estadísticas

Suma y Resta: Puedes sumar o restar dos **arreglos** elemento a elemento, siempre que tengan la misma forma. También es posible sumar o restar un escalar a cada elemento del arreglo.

Multiplicación y División: Similar a la suma y resta, la multiplicación y división se realizan elemento a elemento. NumPy también soporta la multiplicación de matrices con la función **np.dot** o el operador **@** para la multiplicación matricial conforme a las reglas del álgebra lineal.

Funciones Trigonométricas y Exponenciales: NumPy incluye funciones como **np.sin**, **np.cos**, **np.tan** para cálculos trigonométricos, y **np.exp** para calcular la exponencial de todos los elementos en un arreglo.

Reducción: Funciones como **np.sum**, **np.prod**, **np.cumsum** y **np.cumprod** permiten realizar sumas y productos acumulativos o totales de los elementos en un arreglo.

Ejemplo: Operaciones Elemento por Elemento

Código Python 44
Operaciones Elemento por Elemento

Código Python

```python
import numpy as np

# Crear dos arreglos 1D
arreglo_1 = np.array([1, 2, 3, 4])
arreglo_2 = np.array([5, 6, 7, 8])

# Suma
suma = arreglo_1 + arreglo_2
print("Suma:", suma)

# Resta
resta = arreglo_1 - arreglo_2
print("Resta:", resta)

# Multiplicación
multiplicacion = arreglo_1 * arreglo_2
print("Multiplicación:", multiplicacion)

# División
division = arreglo_1 / arreglo_2
print("División:", division)
```

Código Python

```python
# Suma escalar
suma_escalar = arreglo_1 + 10
print("Suma con escalar:", suma_escalar)

# Multiplicación escalar
multiplicacion_escalar = arreglo_1 * 3
print("Multiplicación con escalar:", multiplicacion_escalar)
```

Resultado: Código Python 46, formulación de los autores para operaciones con Escalares.

Ejemplo: Funciones Universales (ufuncs)

Código Python 46
Funciones Universales (ufuncs)

Código Python

```
# Exponenciación
exponenciacion = np.power(arreglo_1, 2)
print("Exponenciación (cuadrado):", exponenciacion)

# Raíz cuadrada
raiz_cuadrada = np.sqrt(arreglo_2)
print("Raíz cuadrada:", raiz_cuadrada)

# Logaritmo natural
logaritmo = np.log(arreglo_2)
print("Logaritmo natural:", logaritmo)
```

Resultado: Código Python 47, formulación de los autores para funciones Universales (ufuncs).

Ejemplo: Funciones Trigonométricas

Código Python 47
Funciones Trigonométricas

Código Python

```
# Crear un arreglo de ángulos en radianes
angulos = np.array([0, np.pi/2, np.pi, 3*np.pi/2])

# Seno
seno = np.sin(angulos)
print("Seno:", seno)

# Coseno
coseno = np.cos(angulos)
print("Coseno:", coseno)

# Tangente
tangente = np.tan(angulos)
print("Tangente:", tangente)
```

Resultado: Código Python 48, formulación de los autores para funciones Trigonométricas.

Ejemplo: Operaciones de Reducción

Código Python 48
Operaciones de Reducción

Código Python

```
# Suma de todos los elementos
suma_total = np.sum(arreglo_1)
print("Suma total de los elementos:", suma_total)

# Producto de todos los elementos
producto_total = np.prod(arreglo_1)
print("Producto total de los elementos:", producto_total)

# Valor máximo
maximo = np.max(arreglo_1)
print("Valor máximo:", maximo)

# Valor mínimo
minimo = np.min(arreglo_1)
print("Valor mínimo:", minimo)
```

Resultado: Código Python 49, formulación de los autores para operaciones de Reducción

Operaciones Estadísticas: NumPy ofrece varias funciones para realizar cálculos estadísticos con **arreglos** como:

a) Media: Es el promedio de todos los elementos del arreglo.

b) Mediana: Es el valor medio de los elementos del arreglo ordenado.

c) Desviación Estándar: Mide la dispersión de los elementos alrededor de la media.

d) Varianza: Es la media de las desviaciones al cuadrado respecto a la media.

e) Percentiles: Indican la posición de un valor en el arreglo ordenado. El percentil 25 es el valor por debajo del cual se encuentra el 25% de los datos.

f) Valor Mínimo y Máximo: Los valores más bajos y altos del arreglo, respectivamente.

g) Rango: Es la diferencia entre el valor máximo y el valor mínimo.

Código Python 49
Operaciones Estadísticas

Código Python

```
import numpy as np

# Crear un arreglo 1D con 10 elementos
arreglo = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print("Arreglo:", arreglo)
# Calcular la media
media = np.mean(arreglo)
print("Media:", media)
# Calcular la mediana
mediana = np.median(arreglo)
print("Mediana:", mediana)
# Calcular la desviación estándar
desviacion_estandar = np.std(arreglo)
print("Desviación estándar:", desviacion_estandar)
# Calcular la varianza
varianza = np.var(arreglo)
print("Varianza:", varianza)
# Calcular el percentil 25
percentil_25 = np.percentile(arreglo, 25)
print("Percentil 25:", percentil_25)

# Calcular el percentil 50 (mediana)
percentil_50 = np.percentile(arreglo, 50)
print("Percentil 50:", percentil_50)

# Calcular el percentil 75
percentil_75 = np.percentile(arreglo, 75)
print("Percentil 75:", percentil_75)

# Calcular el valor mínimo
minimo = np.min(arreglo)
print("Valor mínimo:", minimo)

# Calcular el valor máximo
maximo = np.max(arreglo)
print("Valor máximo:", maximo)

# Calcular el rango
rango = np.ptp(arreglo)
print("Rango:", rango)
```

Resultado: Código Python 50, formulación de los autores para operaciones Estadísticas

1.7.2 Manipulación de Arreglos

La manipulación de **arreglos** en NumPy se refiere al conjunto de operaciones que permiten modificar la estructura, contenido o forma de un arreglo sin necesariamente cambiar los datos subyacentes. Estas operaciones son esenciales para la preparación y análisis de datos en ciencia de datos, programación científica y muchas otras aplicaciones que requieren cálculos numéricos eficientes. Algunas de las principales técnicas de manipulación incluyen:

Transposición (Transpose): Invertir los ejes de un arreglo, lo cual es común en cálculos matriciales.

Código Python 50
Transposición

Código Python

```python
import numpy as np

# Crear un arreglo 2D
arreglo_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Arreglo 2D original:")
print(arreglo_2d)

# Transponer el arreglo 2D
arreglo_transpuesto = np.transpose(arreglo_2d)
print("\nArreglo 2D transpuesto:")
print(arreglo_transpuesto)

# Transponer el arreglo 2D usando el atributo T
arreglo_transpuesto_T = arreglo_2d.T
print("\nArreglo 2D transpuesto usando T:")
print(arreglo_transpuesto_T)
```

Resultado: Código Python 51, formulación de los autores para Transposición

- Transposición de **Arreglos** de Dimensiones Mayores: La transposición también funciona con **arreglos** de más dimensiones, reordena los ejes según el orden especificado o el predeterminado.

- Inmutabilidad: La transposición devuelve una nueva vista del arreglo original, no modifica el arreglo original.

Copia y Vista (Copy and View): Crear copias independientes de los **arreglos** o vistas que son simplemente referencias a los datos originales sin duplicarlos. Entender la diferencia entre una copia y una vista en NumPy es importante para manejar eficientemente la memoria y evitar errores inesperados en el procesamiento de datos.

Código Python 51
Copia y Vista

Código Python

```
import numpy as np
# Crear un arreglo original
arreglo_original = np.array([1, 2, 3, 4, 5])
print("Arreglo original:")
print(arreglo_original)
# Crear una vista del arreglo
vista = arreglo_original.view()
print("\nVista del arreglo original:")
print(vista)
# Modificar la vista
vista[0] = 99
print("\nVista después de modificar el primer elemento:")
print(vista)
# Verificar el arreglo original
print("\nArreglo original después de modificar la vista:")
print(arreglo_original)
# Crear una copia del arreglo
copia = arreglo_original.copy()
print("\nCopia del arreglo original:")
print(copia)
# Modificar la copia
copia[1] = 88
print("\nCopia después de modificar el segundo elemento:")
print(copia)
# Verificar el arreglo original
print("\nArreglo original después de modificar la copia:")
print(arreglo_original)
```

Resultado: Código Python 52, formulación de los autores para la Copia y Vista

- Vista: Modificar una vista afecta el arreglo original y viceversa porque comparten los mismos datos subyacentes.

- Copia: Modificar una copia no afecta el arreglo original porque la copia tiene sus propios datos independientes.

Ordenamiento: Reordenar los elementos de un arreglo según algún criterio. El ordenamiento de **arreglos** es una operación común y NumPy proporciona la función sort para esta tarea.

Código Python 52
Ordenamiento

Código Python

```
import numpy as np

# Crear un arreglo 1D desordenado
arreglo_1d = np.array([3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5])
print("Arreglo 1D original:")
print(arreglo_1d)

# Ordenar el arreglo 1D
arreglo_1d_ordenado = np.sort(arreglo_1d)
print("\nArreglo 1D ordenado:")
print(arreglo_1d_ordenado)

# Crear un arreglo 2D desordenado
arreglo_2d = np.array([[6, 5, 4], [3, 2, 1], [9, 8, 7]])
print("\nArreglo 2D original:")
print(arreglo_2d)

# Ordenar el arreglo 2D a lo largo del eje 0
arreglo_2d_ordenado_eje0 = np.sort(arreglo_2d, axis=0)
print("\nArreglo 2D ordenado a lo largo del eje 0:")
print(arreglo_2d_ordenado_eje0)

# Ordenar el arreglo 2D a lo largo del eje 1
arreglo_2d_ordenado_eje1 = np.sort(arreglo_2d, axis=1)
print("\nArreglo 2D ordenado a lo largo del eje 1:")
print(arreglo_2d_ordenado_eje1)
```

Resultado: Código Python 53, formulación de los autores para el Ordenamiento.

- Inmutabilidad: La función **np.sort** devuelve un nuevo arreglo ordenado y no modifica el arreglo original.
- Ordenamiento en Ejes: Al ordenar **arreglos multidimensionales**, puedes especificar el eje a lo largo del cual se debe realizar el ordenamiento, utiliza el parámetro **axis**.
- Estabilidad: El método de ordenamiento de NumPy es estable, lo que significa que preserva el orden relativo de los elementos iguales.

1.7.3 Principios y reglas del broadcasting para operar con arreglos de diferentes tamaños

El **broadcasting** es una poderosa característica de NumPy que permite realizar operaciones aritméticas entre **arreglos** de diferentes tamaños sin necesidad de duplicar datos. Este proceso sigue un conjunto de reglas para hacer que los **arreglos** sean compatibles en tamaño durante las operaciones. Aquí están los principios y reglas fundamentales del **broadcasting**:

a) **Regla de Preparación**: Si los **arreglos** no tienen el mismo rango, se añaden dimensiones de tamaño 1 al principio del arreglo de menor rango hasta que ambos **arreglos** tengan el mismo número de dimensiones.

b) **Regla de Extensión**: Dos dimensiones son compatibles para el **broadcasting** si son iguales o si una de ellas es 1. Si una dimensión es 1, el arreglo con esa dimensión actúa como si tuviera el tamaño de la dimensión correspondiente del otro arreglo, repite los valores a lo largo de esa dimensión.

c) **Resultado del Broadcasting**: El tamaño de cada dimensión en el resultado es el máximo de las dimensiones de los **arreglos** originales en esa dimensión.

Por ejemplo, si se suma dos arreglos, uno con forma (5,1) y otro con forma (1,4), el **broadcasting** los extiende ambos a la forma (5,4) antes de realizar la suma. Cada

elemento de los **arreglos** originales se "difunde" o repite a través de la dimensión correspondiente para coincidir con la forma del otro arreglo.

El **broadcasting** es especialmente útil porque reduce la necesidad de crear manualmente **arreglos** de tamaños más grandes, lo que ahorra memoria y computación, permite simplificar y acelerar operaciones que de otro modo requerirían bucles explícitos en un lenguaje de programación de alto nivel.

El siguiente código explica la aplicación del **broadcasting:**

Código Python 53
Suma de un Arreglo de 1D y un Escalar

Código Python

```python
import numpy as np

# Crear un arreglo 1D
arreglo_1d = np.array([1, 2, 3, 4])

# Sumar un escalar al arreglo 1D
resultado = arreglo_1d + 10
print("Suma de un arreglo 1D y un escalar:")
print(resultado)
```

Resultado: Código Python 51, formulación de los autores para la suma de un Arreglo de 1D y un Escalar

Explicación**:** El escalar 10 se extiende a un arreglo de la misma forma que **arreglo_1d** y luego se realiza la suma.

Código Python 54
Suma de un Arreglo 2D y un Arreglo 1D

Código Python

```
# Crear un arreglo 2D
arreglo_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Crear un arreglo 1D
arreglo_1d = np.array([10, 20, 30])

# Sumar el arreglo 2D y el arreglo 1D
resultado = arreglo_2d + arreglo_1d
print("\nSuma de un arreglo 2D y un arreglo 1D:")
print(resultado)
```

Resultado: Código Python 55, formulación de los autores para la suma de un Arreglo 2D y un Arreglo 1D

Explicación**:** El arreglo 1D **arreglo_1d** se extiende a una matriz 2D, replica sus elementos a lo largo de las filas de **arreglo_2d**.

Código Python 55
Multiplicación de un Arreglo 2D y un Arreglo 1D de Diferentes Formas

Código Python

```
# Crear un arreglo 2D
arreglo_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Crear un arreglo 1D con una dimensión que coincide
arreglo_1d_col = np.array([[10], [20]])

# Multiplicar el arreglo 2D y el arreglo 1D
resultado = arreglo_2d * arreglo_1d_col
print("\nMultiplicación de un arreglo 2D y un arreglo 1D con una dimensión
que coincide:")
print(resultado)
```

Resultado: Código Python 56, formulación de los autores para la multiplicación de un Arreglo 2D y un Arreglo 1D de Diferentes Formas

Explicación**:** El arreglo 1D **arreglo_1d_col** se extiende a una matriz 2D, replica sus elementos a lo largo de las columnas de **arreglo_2d**.

Código Python 56
Operación entre Arreglos de Diferentes Dimensiones

Código Python

```
# Crear un arreglo 3D
arreglo_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])

# Crear un arreglo 1D
arreglo_1d = np.array([10, 20, 30])

# Sumar el arreglo 3D y el arreglo 1D
resultado = arreglo_3d + arreglo_1d
print("\nSuma de un arreglo 3D y un arreglo 1D:")
print(resultado)
```

Resultado: Código Python 57, formulación de los autores para la operación entre Arreglos de Diferentes Dimensiones

Explicación**:** El arreglo 1D **arreglo_1d** se extiende para coincidir con las dimensiones del arreglo 3D **arreglo_3d**.

1.7.4 Ejercicios de aplicación

En este ejercicio se aplicarán las funcionalidades de la librería NumPy para crear, manipular y realizar operaciones básicas con arreglos multidimensionales en Python.

Ejercicio de aplicación 1: Uso de la librería NumPy

Trabajas en el departamento de análisis de datos de una fábrica. Se te ha proporcionado un conjunto de datos que contiene la cantidad de unidades producidas por cada máquina en la fábrica durante una semana. Necesitas analizar estos datos

para calcular la eficiencia promedio de producción y determinar qué días y máquinas tuvieron un rendimiento por debajo del promedio.

Código Python 57
Solución ejercicio de aplicación 4: Utilización de arreglos con la liberia NumPy

Código Python Parte 1

```
# Importar la librería NumPy
import numpy as np

# Crear un arreglo NumPy con datos de producción
# En este arreglo, cada fila representa un día diferente y cada columna una máquina diferente.
# Por ejemplo, [40, 35, 50, 45] representa la producción de las 4 máquinas en el primer día.
# [40, 38, 35, 30, 32] representa la producción de la máquina 1 en los 5 días.
datos_produccion = np.array([
    [40, 35, 50, 45],
    [38, 42, 47, 44],
    [35, 40, 45, 43],
    [30, 38, 40, 42],
    [32, 35, 44, 41]
])

# Mostrar el arreglo
print("Datos de producción: \n")
print(datos_produccion)
print("-----------------------------------------------------------------")

# Calcular el promedio de producción por máquina
# Usamos axis=0 para calcular el promedio a lo largo de las columnas (máquinas).
print("Número de máquinas:", datos_produccion.shape[1])
promedio_maquinas = np.mean(datos_produccion, axis=0)
print("Promedio de producción por máquina:", promedio_maquinas)
print("-----------------------------------------------------------------")
```

Código Python Parte 2

```
# Calcular el promedio de producción por día
# Usamos axis=1 para calcular el promedio a lo largo de las filas (días).
print("Número de días:", datos_produccion.shape[0])
promedio_dias = np.mean(datos_produccion, axis=1)
print("Promedio de producción por día:", promedio_dias)
print("----------------------------------------------------------------
")

# Encontrar y mostrar los días y máquinas con producción por debajo del
promedio
# Iteramos sobre cada elemento del arreglo con dos bucles for anidados.
# El primer bucle itera sobre las filas (días) y el segundo sobre las
columnas (máquinas).
# Para cada elemento, comparamos su valor con el promedio correspondiente.
# Si el valor es menor, mostramos un mensaje indicando el día y la máquina.
print("Días y máquinas con producción por debajo del promedio:")
print(" ")
for index_dia in range(datos_produccion.shape[0]): # Iterar sobre las
filas (días), range() genera una lista de números del 0 al 4 (5 días)
    for index_maquina in range(datos_produccion.shape[1]): # Iterar sobre
las columnas (máquinas), range() genera una lista de números del 0 al 3
(4 máquinas)
                if    datos_produccion[index_dia,    index_maquina]    <
promedio_maquinas[index_maquina]: # Comparar con el promedio de la máquina
          # Mostrar un mensaje indicando el día, la máquina, la producción
y el promedio
            print("Día:", index_dia+1,
                  "Máquina:", index_maquina+1,
                              "Producción:",   datos_produccion[index_dia,
index_maquina],
                            f"|  Promedio  máquina  {index_maquina+1}:",
promedio_maquinas[index_maquina])
```

Resultado: https://www.udemy.com/course/python-analisis-de-datos/ (*Python para el análisis de datos*, s. f.)

El ejercicio de aplicación permite utilizar cada tema que se ha descrito en la unidad como:

Creación y Manipulación de Arreglos:

- Uso de **np.array** para crear arreglos.

- Acceso a elementos del arreglo utiliza índices.

Cálculo de Estadísticas:

- Uso de **np.mean** para calcular la media a lo largo de un eje específico.

Shape y Ejes:

- Uso de **shape** para obtener las dimensiones del arreglo.
- Comprensión de los ejes (axis) en operaciones estadísticas (**axis=0** para columnas y **axis=1** para filas).

Bucles y Condiciones:

- Iteración sobre elementos de un arreglo utiliza bucles.
- Uso de condiciones para comparar elementos del arreglo.

Este código abarca varios conceptos esenciales de NumPy, como la creación y manipulación de arreglos, cálculo de estadísticas, comprensión de las dimensiones y ejes de los arreglos, y el uso de bucles y condiciones para procesar datos.

Ejercicio de aplicación 2: Análisis de Ventas

Trabajas como analista de datos en una empresa de ventas. Tienes los datos de ventas mensuales de diferentes regiones y necesitas analizar estos datos para obtener varias estadísticas y comparaciones. Las estadísticas incluyen el promedio, la varianza y el total de ventas, y deseas comparar el desempeño mes a mes para identificar tendencias.

Código Python 58
Análisis de Ventas

Código Python Parte 1

```python
# Importar la librería NumPy
import numpy as np

# Datos de ventas mensuales en diferentes regiones (en miles de dólares)
# Cada fila representa un mes y cada columna una región
ventas = np.array([
    [25, 30, 45],
    [35, 40, 50],
    [30, 35, 60],
    [45, 50, 65]
])

# Mostrar el arreglo
print("Datos de producción: \n")
print(ventas)
print("-----------------------------------------------------------------")

# Calcular el promedio de ventas por región
# Usamos axis=0 para calcular el promedio a lo largo de las columnas (regiones).
print("Número de regiones:", ventas.shape[1])
promedio_region = np.round(np.mean(ventas, axis=0), 2) # Redondeamos el promedio a 2 decimales con np.round
print("Promedio de ventas por región:", promedio_region)
print("-----------------------------------------------------------------")
```

Código Python Parte 2

```
# Calcular la varianza de ventas por mes
# Usamos axis=1 para calcular la varianza a lo largo de las filas (meses)
print("Número de meses:", ventas.shape[0])
varianza_mes = np.round(np.var(ventas, axis=1), 2) # Redondeamos la
varianza a 2 decimales con np.round
print("Varianza de ventas por mes:", varianza_mes)

# Total de ventas por mes
# Usamos np.sum con axis=1 para sumar las ventas de todas las regiones por
mes.
total_mes = np.sum(ventas, axis=1)
print("Total de ventas por mes:", total_mes)
print("-----------------------------------------------------------------
")

# Comparar ventas mes a mes para identificar tendencias
# np.diff calcula la diferencia en ventas entre meses consecutivos.
diferencia_mes = np.diff(ventas, axis=0)
print("Cambio en ventas mes a mes:\n", diferencia_mes)
# Interpretación:
# La primera fila representa la diferencia entre el mes 1 y el mes 2.
# La segunda fila representa la diferencia entre el mes 2 y el mes 3.
# La tercera fila representa la diferencia entre el mes 3 y el mes 4.
print("-----------------------------------------------------------------
")

# Identificar el mes con el mayor aumento en ventas
# np.argmax para encontrar el índice del mayor valor en la suma de las
diferencias.
mes_mayor_aumento = np.argmax(np.sum(diferencia_mes, axis=1)) + 1  # + 1
ya que np.diff reduce la longitud del array en 1
print(f"Mes con el mayor aumento en ventas: Mes {mes_mayor_aumento}")
```

Resultado: https://www.udemy.com/course/python-analisis-de-datos/(*Python para el análisis de datos*, s. f.)

El ejercicio establece una situación de la vida real y se resuelve utilizando los conceptos de arreglos multidimensionales y la librería NumPy de Python, logrando que los lectores pongan en práctica todos los conceptos aprendidos hasta el momento.

1.7.5 Ejercicio para desarrollar: Aplicación de conceptos y uso de la Librería NumPy

Análisis de Rendimiento Académico en una Institución Educativa

Objetivos

- Crear y manipular arreglos multidimensionales con NumPy para representar datos académicos.
- Realizar operaciones matemáticas y estadísticas para analizar el rendimiento académico.
- Aplicar técnicas avanzadas de manipulación de arreglos para extraer y modificar datos.
- Entender y aplicar los principios y reglas del broadcasting para operar con arreglos de diferentes tamaños.

Instrucciones

Esta actividad consiste en analizar un conjunto de datos relacionados con el rendimiento académico de estudiantes en diferentes materias y clases a lo largo de un semestre. Utilizarán NumPy para crear, manipular y realizar operaciones matemáticas y estadísticas en los datos, así como aplicar los principios de broadcasting para comparar y operar con arreglos de diferentes tamaños. Los datos incluirán las calificaciones de los estudiantes en varias materias y clases, así como información estadística relevante.

Desarrollar los arreglos de datos de acuerdo con las siguientes especificaciones

Creación de Arreglos Multidimensionales

- Crear un arreglo 3D que represente las calificaciones de los estudiantes, donde las dimensiones representen [clases, estudiantes, materias].
- Inicializar el arreglo con datos simulados o reales.

Operaciones Matemáticas y Estadísticas

- Calcular la media, mediana, desviación estándar y varianza de las calificaciones por clase, por estudiante y por materia.
- Identificar a los estudiantes con calificaciones por debajo del promedio en cada materia.

Manipulación de Arreglos

- Extraer y analizar los datos de una clase y una materia específicas.
- Reestructurar los arreglos para comparar el rendimiento entre diferentes clases y materias.

Principios y Reglas del Broadcasting

- Utilizar broadcasting para comparar el rendimiento de cada estudiante con el promedio general de la clase.
- Aplicar operaciones aritméticas entre arreglos de diferentes tamaños para normalizar las calificaciones.

Desafíos Propuestos:

Representación e Inicialización de Datos

- Crear un arreglo 3D calificaciones de forma (5, 20, 6) para 5 clases, 20 estudiantes por clase, y 6 materias.

- Inicializar el arreglo con calificaciones aleatorias entre 0 y 100.

Este ejercicio permitirá que los lectores apliquen conceptos avanzados de NumPy en un contexto realista, analizando y manipulando datos académicos. Les proporciona experiencia práctica en la creación y manipulación de arreglos multidimensionales, realización de operaciones matemáticas y estadísticas, y aplicación de principios de broadcasting.

CAPÍTULO III: EJERCICIOS Y RETOS PRÁCTICOS DE PROGRAMACIÓN PARA CIENCIA DE DATOS

La Ciencia de Datos se ha consolidado como una disciplina fundamental en la toma de decisiones informadas y en la resolución de problemas complejos a partir del análisis de grandes volúmenes de datos. A medida que la tecnología avanza, la habilidad para procesar, analizar e interpretar datos de manera efectiva se ha vuelto esencial en sectores como la salud, la economía, la ingeniería y las ciencias sociales. En este capítulo, se presentarán una serie de ejercicios y retos prácticos diseñados para fortalecer las competencias necesarias en programación dentro del contexto de la Ciencia de Datos. Estos desafíos cubrirán aspectos clave como la limpieza de datos, la implementación de algoritmos de aprendizaje automático, la visualización de datos y la optimización de modelos. A través de casos reales y problemas prácticos, el lector podrá afianzar conceptos teóricos y adquirir experiencia práctica, lo que permitirá desarrollar las habilidades necesarias para enfrentar los retos que plantea esta disciplina en constante evolución.

1.8 Ejercicio de sistema de inventario de productos

Descripción:

Desarrolla un sistema que permita gestionar un inventario de productos en una tienda. Cada producto debe tener atributos básicos como nombre, precio y cantidad en stock. El sistema debe incluir métodos para agregar productos, actualizar cantidades y mostrar información detallada del inventario.

Objetivos de Aprendizaje:

- Introducir el concepto de clases como representaciones de objetos del mundo real.
- Comprender cómo encapsular atributos y métodos en una clase.
- Desarrollar habilidades iniciales en la creación y manipulación de objetos.

1.8.1 Especificación técnica detallada:

Crear una clase Producto con los atributos:

- nombre (string): nombre del producto.
- precio (float): precio del producto.
- cantidad (int): cantidad en stock.

Implementar los métodos:

- actualizar_stock(cantidad): Aumenta o disminuye el stock del producto.
- realizar_venta(cantidad): Reduce la cantidad disponible y devuelve un mensaje si el stock es insuficiente.
- mostrar_info(): Muestra los detalles del producto.

Crear una lista de productos y un menú para gestionar las acciones disponibles: agregar productos, listar productos y realizar ventas.

Desafíos propuestos:

- Implementar validaciones para evitar introducir precios negativos o cantidades no válidas.
- Crear una función que calcule el valor total del inventario basado en las cantidades y precios de los productos.
- Extender el sistema para que soporte categorías de productos.

1.9 Ejercicio: Sistema de gestión de estudiantes

Descripción:

Crea un sistema para almacenar y gestionar información de estudiantes en una universidad. Cada estudiante tendrá atributos como nombre, matrícula y promedio general. Los supervisores podrán actualizar los promedios y visualizar los datos de los estudiantes.

Objetivos de Aprendizaje:

- Practicar la creación de clases y objetos en Python.
- Aprender a encapsular datos relacionados en una clase.
- Desarrollar habilidades para manejar listas de objetos.

1.9.1 Especificación técnica detallada:

Crear una clase Estudiante con los atributos:

- nombre (string): nombre del estudiante.
- matricula (string): matrícula del estudiante.
- promedio (float): promedio del estudiante.

Implementar los métodos:

- actualizar_promedio(nuevo_promedio): Actualiza el promedio del estudiante.
- mostrar_info(): Muestra la información completa del estudiante.

Diseñar una lista de estudiantes y un menú con opciones para:

- Registrar un nuevo estudiante.
- Buscar un estudiante por matrícula.
- Mostrar todos los estudiantes registrados.

Desafíos propuestos:

- Implementar una función que permita calcular el promedio general de todos los estudiantes registrados.
- Extender el sistema para que registre materias cursadas y calificaciones individuales.
- Permitir buscar estudiantes con promedios mayores o menores a un valor específico.

1.10 Ejercicio: Sistema básico de clima

Descripción:

Desarrolla un sistema para registrar y mostrar información sobre condiciones climáticas en diferentes ciudades. Los supervisores podrán actualizar la temperatura, humedad y el estado del tiempo de cada ciudad registrada.

Objetivos de Aprendizaje:

- Introducir cómo modelar objetos del mundo real.
- Practicar métodos para manejar y modificar datos de objetos.
- Familiarizarse con la creación de listas de objetos.

1.10.1 Especificación técnica detallada:

Crear una clase Ciudad con los atributos:

- nombre (string): nombre de la ciudad.
- temperatura (float): temperatura actual.
- humedad (float): porcentaje de humedad.
- estado_tiempo (string): estado del tiempo (soleado, nublado, lluvioso, etc.).

Implementar los métodos:

- actualizar_clima(temperatura, humedad, estado_tiempo): Actualiza los valores del clima.
- mostrar_info(): Muestra la información climática de la ciudad.

Crear una lista de ciudades y un menú con opciones para:

- Registrar nuevas ciudades.
- Mostrar información de todas las ciudades.
- Actualizar las condiciones climáticas de una ciudad específica.

Desafíos propuestos:

- Implementar una búsqueda que permita encontrar ciudades con temperaturas mayores o menores a un valor específico.
- Crear un sistema para calcular el promedio de temperaturas de todas las ciudades registradas.
- Extender el sistema para permitir registrar y mostrar pronósticos climáticos para varios días.

1.11 Ejercicio: Sistema de gestión de libros en una biblioteca

Descripción:

Implementa un sistema para registrar libros en una biblioteca. Cada libro tendrá atributos como título, autor, ISBN y estado (disponible o prestado). Los supervisores podrán prestar y devolver libros, así como listar los libros disponibles.

Objetivos de Aprendizaje:

- Entender cómo utilizar clases para modelar sistemas más complejos.
- Practicar métodos específicos para gestionar estados de objetos.
- Trabajar con listas de objetos para manejar múltiples datos.

1.11.1 Especificación técnica detallada:

Crear una clase Libro con los atributos:

- titulo (string): título del libro.
- autor (string): autor del libro.
- isbn (string): código único del libro.
- estado (string): estado del libro (disponible o prestado).

Implementar los métodos:

- prestar(): Cambia el estado del libro a “prestado”.

- devolver(): Cambia el estado del libro a “disponible”.
- mostrar_info(): Muestra los detalles del libro.

Crear una lista de libros y un menú con opciones para:

- Registrar nuevos libros.
- Buscar libros por título o autor.
- Mostrar todos los libros disponibles.

Desafíos propuestos:

- Implementar un sistema que permita registrar el nombre del usuario que prestó el libro.
- Crear una función para buscar libros prestados y calcular cuántos están disponibles.
- Extender el sistema para incluir categorías y permitir búsquedas por categoría.

1.12Ejercicio: Sistema de gestión de vehículos

Descripción:
Implementa un sistema para registrar y gestionar información sobre vehículos en un concesionario. Cada vehículo debe tener atributos como marca, modelo, precio y disponibilidad. Los supervisores podrán marcar vehículos como vendidos y listar los vehículos disponibles.

Objetivos de Aprendizaje:

- Aplicar los conceptos básicos de clases y objetos.
- Practicar el uso de métodos para modificar los estados de los objetos.
- Trabajar con listas de objetos para organizar múltiples datos.

1.12.1 Especificación técnica detallada:

Crear una clase Vehiculo con los atributos:

- marca (string): marca del vehículo.

- modelo (string): modelo del vehículo.
- precio (float): precio del vehículo.
- disponible (boolean): indica si el vehículo está disponible para la venta.

Implementar los métodos:

- vender(): Cambia el estado del vehículo a "no disponible".
- mostrar_info(): Muestra los detalles del vehículo.

Crear una lista de vehículos y un menú con opciones para:

- Registrar nuevos vehículos.
- Buscar vehículos por marca o modelo.
- Mostrar todos los vehículos disponibles.
- Marcar un vehículo como vendido.

Desafíos propuestos:

- Implementar un sistema para calcular el valor total de los vehículos disponibles y vendidos.
- Crear una función para filtrar vehículos por rango de precios.
- Extender el sistema para incluir atributos adicionales como kilometraje y año, y permitir búsquedas avanzadas basadas en estos atributos.

1.13 Ejercicio: Sistema de reservas de salas

Descripción:

Crea un sistema para gestionar la reserva de salas en una empresa. Cada sala tendrá atributos como nombre, capacidad y estado de disponibilidad. Los usuarios podrán reservar o liberar salas según las necesidades, y se podrá buscar salas disponibles por capacidad.

Objetivos de Aprendizaje:

- Desarrollar habilidades en la creación de clases y objetos.

- Diseñar métodos que modifiquen el estado de los objetos.
- Aplicar listas de objetos para manejar múltiples instancias.

1.13.1 Especificación técnica detallada:

Crear una clase Sala con los atributos:

- nombre (string): nombre de la sala.
- capacidad (int): número máximo de personas que pueden usar la sala.
- disponible (boolean): indica si la sala está disponible para reserva.

Implementar los métodos:

- reservar(): Cambia el estado de la sala a "no disponible".
- liberar(): Cambia el estado de la sala a "disponible".
- mostrar_info(): Muestra los detalles de la sala.

Crear una lista de salas y un menú con opciones para:

- Registrar nuevas salas.
- Reservar una sala por nombre.
- Liberar una sala reservada.
- Mostrar todas las salas disponibles.

Desafíos propuestos:

- Implementar una función para buscar salas disponibles con una capacidad mayor o igual a un número específico.
- Extender el sistema para registrar el nombre del usuario que reservó la sala.
- Crear un sistema de prioridad para reservas que permita definir un orden en caso de que dos usuarios soliciten la misma sala.

1.14 Ejercicio: Sistema de gestión de animales en un zoológico

Descripción:

Desarrolla un sistema para gestionar animales en un zoológico. El sistema debe incluir una clase base Animal con atributos comunes, y subclases como Mamifero, Ave y Reptil que extiendan la funcionalidad según sus características específicas.

Objetivos de Aprendizaje:

- Comprender cómo usar herencia para modelar jerarquías de clases.
- Aprender a extender atributos y métodos de una clase base.
- Implementar métodos específicos en subclases.

1.14.1 Especificación técnica detallada:

Crear una clase base Animal con los atributos:

- nombre (string): nombre del animal.
- especie (string): especie del animal.
- edad (int): edad del animal.

Implementar los métodos:

- mostrar_info(): Muestra información básica del animal.

Crear subclases Mamifero, Ave y Reptil que extiendan Animal:

- Agregar atributos específicos (ejemplo: tipo_piel para reptiles, vuela para aves).
- Implementar métodos específicos (ejemplo: volar() para aves, deslizar() para reptiles).

Crear una lista de animales que permita almacenar instancias de diferentes tipos y listar sus características.

Desafíos propuestos:

- Implementar un método en la clase base que sea sobrescrito en las subclases para agregar comportamiento único.
- Extender el sistema para que permita alimentar a los animales según su tipo (ejemplo: mamíferos comen carne, aves comen granos).
- Agregar validaciones para evitar datos inválidos (ejemplo: un ave no puede tener vuela como falso si pertenece a una especie voladora).

1.15 Ejercicio: Sistema de empleados por departamento

Descripción:

Diseña un sistema que gestione empleados en una empresa. El sistema debe tener una clase base Empleado y subclases como Técnico, Gerente y Vendedor, cada una con atributos y métodos específicos.

Objetivos de Aprendizaje:

- Entender cómo la herencia permite modelar diferentes tipos de empleados.
- Aplicar el concepto de métodos sobrescritos.
- Organizar jerarquías de clases de manera lógica y efectiva.

1.15.1 Especificación técnica detallada:

Crear una clase base Empleado con los atributos:

- nombre (string).
- id (string).
- salario (float).

Implementar los métodos:

- calcular_bono(): Devuelve un bono base del 5% del salario.

Crear subclases Técnico, Gerente y Vendedor que sobrescriban calcular_bono():

- Tecnico: Bono del 7%.
- Gerente: Bono del 10%.
- Vendedor: Bono del 5% + comisión por ventas.

Crear un menú para listar empleados y calcular sus bonos de manera dinámica.

Desafíos Propuestos:

- Implementar una función que permita calcular el costo total de los salarios para cada departamento.
- Extender el sistema para agregar una jerarquía en la que los gerentes supervisen técnicos y vendedores.
- Permitir a los usuarios modificar los bonos de manera dinámica según políticas de la empresa.

1.16 Ejercicio: Sistema de vehículos

Descripción:

Crea un sistema que modele una flota de vehículos. El sistema debe incluir una clase base Vehiculo y subclases como Auto, Camion y Moto, cada una con atributos y comportamientos específicos.

Objetivos de Aprendizaje:

- Aprender a usar herencia para extender atributos y métodos.
- Practicar la implementación de métodos específicos en subclases.
- Trabajar con listas de objetos de diferentes tipos en una jerarquía.

1.16.1 Especificación técnica detallada:

Crear una clase base Vehiculo con los atributos:

- marca (string).

- modelo (string).
- precio (float).

Implementar los métodos:

- mostrar_info(): Muestra información del vehículo.

Crear subclases Auto, Camion y Moto que extiendan Vehiculo:

- Auto: Atributo adicional num_puertas.
- Camion: Atributo adicional capacidad_carga.
- Moto: Atributo adicional tipo_moto (deportiva, scooter, etc).

Diseñar un sistema que permita buscar vehículos por tipo y listar sus detalles.

Desafíos propuestos:

- Implementar un método para calcular impuestos basados en el precio y el tipo de vehículo.
- Extender el sistema para incluir un historial de servicios para cada vehículo.
- Agregar validaciones para evitar datos inconsistentes (ejemplo: un camión no puede tener capacidad_carga menor a cero).

1.17 Ejercicio: Sistema de cursos y estudiantes

Descripción:

Desarrolla un sistema para gestionar estudiantes y los cursos en los que están inscritos. El sistema debe tener una clase base Curso y subclases CursoTeorico y CursoPractico que extiendan sus funcionalidades.

Objetivos de Aprendizaje:

- Modelar jerarquías que incluyan relaciones entre objetos.
- Practicar la implementación de métodos sobrescritos.
- Usar herencia para extender funcionalidades de una clase base.

1.17.1 Especificación técnica detallada:

Crear una clase base Curso con los atributos:

- nombre (string).
- codigo (string).
- creditos (int).

Implementar los métodos:

- mostrar_info(): Muestra detalles del curso.

Crear subclases CursoTeorico y CursoPractico:

- CursoTeorico: Incluye un atributo adicional num_clases_teoricas.
- CursoPractico: Incluye un atributo adicional num_practicas.

Crear una lista de cursos y un menú para registrar estudiantes en cursos.

Desafíos propuestos:

- Implementar una función para calcular la carga académica total de un estudiante.
- Extender el sistema para incluir una lista de estudiantes inscritos en cada curso.
- Permitir buscar cursos por tipo (teórico o práctico).

1.18 Ejercicio: Sistema de figuras geométricas

Descripción:

Crea un sistema que modele figuras geométricas. El sistema debe incluir una clase base Figura y subclases como Cuadrado, Circulo y Triangulo con métodos específicos para calcular el área y el perímetro.

Objetivos de Aprendizaje:

- Aplicar herencia para extender funcionalidades de una clase base.
- Practicar la implementación de métodos específicos en subclases.

- Usar la sobrescritura de métodos para personalizar comportamientos.

1.18.1 Especificación técnica detallada:

Crear una clase base Figura con los métodos:

- calcular_area(): Método base sin implementación.
- calcular_perimetro(): Método base sin implementación.

Crear subclases Cuadrado, Circulo y Triangulo que sobrescriban los métodos base:

- Cuadrado: Implementa área y perímetro usando lado.
- Circulo: Implementa área y perímetro usando radio.
- Triangulo: Implementa área y perímetro usando base y altura.

Crear una lista de figuras y un sistema que calcule áreas y perímetros automáticamente.

Desafíos propuestos:

- Implementar validaciones para evitar datos inconsistentes (ejemplo: un círculo no puede tener radio negativo).
- Crear una función para listar figuras por tipo y ordenarlas por área.
- Extender el sistema para incluir figuras tridimensionales como cubos o esferas.

1.19 Ejercicio: Sistema de dispositivos electrónicos

Descripción:

Desarrolla un sistema que modele dispositivos electrónicos. Incluye una clase base Dispositivo y subclases como Laptop, Telefono y Tablet con atributos y métodos específicos.

Objetivos de Aprendizaje:

- Usar herencia para extender atributos y métodos.
- Aprender a sobrescribir métodos en subclases.
- Modelar sistemas con múltiples tipos de objetos.

1.19.1 Especificación técnica detallada:

Crear una clase base Dispositivo con los atributos:

- marca (string).
- modelo (string).
- precio (float).

Implementar los métodos:

- mostrar_info(): Muestra detalles del dispositivo.

Crear subclases Laptop, Telefono y Tablet:

- Laptop: Atributo adicional ram.
- Telefono: Atributo adicional num_camaras.
- Tablet: Atributo adicional soporte_lapiz.

Diseñar un sistema que permita buscar dispositivos por tipo y listar sus detalles.

Desafíos propuestos:

- Crear un método para calcular descuentos en el precio según el tipo de dispositivo.
- Extender el sistema para incluir un historial de reparaciones para cada dispositivo.
- Agregar una función que permita comparar dispositivos por precio o características.

1.20 Ejercicio: Sistema de animales con sonidos

Descripción:

Crea un sistema para modelar animales y los sonidos que hacen. Utiliza una clase base Animal con un método hacer_sonido() que será sobrescrito en las subclases para cada tipo de animal.

Objetivos de Aprendizaje:

- Comprender el concepto de polimorfismo mediante la sobrescritura de métodos.
- Implementar y utilizar métodos que compartan la misma firma pero tengan diferentes comportamientos.
- Trabajar con listas heterogéneas de objetos para demostrar polimorfismo.

1.20.1 Especificación técnica detallada:

Crear una clase base Animal con el método:

- hacer_sonido(): Método base que será sobrescrito.
- Crear subclases Perro, Gato y Vaca que sobrescriban hacer_sonido() para devolver "Ladra", "Maúlla" y "Muge", respectivamente.
- Crear una lista de animales e iterar sobre ella para invocar hacer_sonido() de manera polimórfica.

Desafíos Propuestos:

- Agregar más subclases con diferentes sonidos (ejemplo: Pato, Caballo).
- Extender el sistema para incluir un método adicional moverse() con comportamientos únicos para cada animal.
- Diseñar una función que clasifique animales según su tipo basado en el sonido que hacen.

1.21 Ejercicio: Sistema de figuras geométricas con polimorfismo

Descripción:

Desarrolla un sistema que calcule el área de diferentes figuras geométricas utilizando polimorfismo. Todas las figuras deben compartir una clase base Figura con un método calcular_area().

Objetivos de Aprendizaje:

- Entender cómo usar polimorfismo para implementar cálculos comunes con variaciones según el tipo de objeto.
- Practicar el diseño de jerarquías de clases que utilicen métodos sobrescritos.
- Usar listas heterogéneas para trabajar con objetos de diferentes subclases.

1.21.1 Especificación técnica detallada:

Crear una clase base Figura con un método:

- calcular_area(): Método abstracto que será sobrescrito en las subclases.

Crear subclases Cuadrado, Circulo y Triangulo que implementen calcular_area():

- Cuadrado: Usa lado para calcular el área.
- Circulo: Usa radio para calcular el área.
- Triangulo: Usa base y altura para calcular el área.

Diseñar una función que reciba una lista de figuras y calcule el área de todas las figuras de manera polimórfica.

Desafíos propuestos:

- Implementar un método adicional calcular_perimetro() en las subclases.

- Extender el sistema para incluir figuras tridimensionales como Esfera y Cubo que sobrescriban métodos para calcular volumen.
- Crear un menú interactivo para que el usuario registre figuras y vea sus áreas calculadas.

1.22 Ejercicio: Sistema de vehículos en movimiento

Descripción:

Diseña un sistema que modele vehículos en movimiento. Utiliza una clase base Vehiculo con un método mover() que será sobrescrito en subclases para reflejar diferentes maneras de moverse (ejemplo: ruedas, alas, hélices).

Objetivos de Aprendizaje:

- Aplicar polimorfismo para modelar comportamientos dinámicos en clases derivadas.
- Aprender a sobrescribir métodos para representar acciones únicas en subclases.
- Usar listas de objetos para demostrar el polimorfismo en acción.

1.22.1 Especificación técnica detallada:

Crear una clase base Vehiculo con un método:

- mover(): Método abstracto que será sobrescrito.

Crear subclases Auto, Avion y Barco que sobrescriban mover():

- Auto: Devuelve "Se mueve sobre ruedas".
- Avion: Devuelve "Vuela en el aire".
- Barco: Devuelve "Navega en el agua".

Crear una lista de vehículos y usar polimorfismo para invocar el método mover() en cada uno.

Desafíos propuestos:

- Agregar más subclases (ejemplo: Moto, Helicoptero).
- Implementar un método adicional combustible_usado() que calcule el tipo de combustible según el vehículo.
- Diseñar una simulación que permita al usuario agregar vehículos y ver cómo se mueven.

1.23 Ejercicio: Sistema de gestión de empleados con polimorfismo

Descripción:
Desarrolla un sistema que modele diferentes tipos de empleados en una empresa, cada uno con una forma distinta de calcular su salario. Utiliza polimorfismo para que cada empleado calcule su salario de manera única.

Objetivos de Aprendizaje:

- Aplicar polimorfismo en el cálculo de salarios.
- Practicar el uso de listas de objetos heterogéneos.
- Diseñar clases que sobrescriban métodos de una clase base.

1.23.1 Especificación técnica detallada:

Crear una clase base Empleado con un método:

- calcular_salario(): Método base que será sobrescrito.

Crear subclases EmpleadoFijo, EmpleadoPorHora y EmpleadoPorComision:

- EmpleadoFijo: Salario fijo.
- EmpleadoPorHora: Salario basado en horas trabajadas y tarifa por hora.
- EmpleadoPorComision: Salario basado en ventas realizadas.

Crear una lista de empleados e iterar sobre ella para calcular los salarios de manera polimórfica.

Desafíos Propuestos:

- Extender el sistema para incluir empleados híbridos con múltiples métodos de cálculo.
- Crear un método adicional mostrar_detalles() que incluya el cálculo del salario.
- Diseñar un menú interactivo para agregar empleados y ver sus salarios.

1.24 Ejercicio: Sistema de instrumentos musicales

Descripción:

Crea un sistema que modele diferentes tipos de instrumentos musicales y cómo se tocan. Utiliza polimorfismo para que cada instrumento tenga su forma única de generar sonido.

Objetivos de Aprendizaje:

- Practicar la sobrescritura de métodos para modelar comportamientos únicos.
- Usar listas heterogéneas para trabajar con objetos de diferentes tipos.
- Aplicar polimorfismo para representar acciones comunes con variaciones.

1.24.1 Especificación Técnica Detallada:

Crear una clase base Instrumento con un método:

- tocar(): Método abstracto que será sobrescrito.

Crear subclases Guitarra, Piano y Bateria que sobrescriban tocar():

- Guitarra: Devuelve "Tocando acordes en la guitarra".
- Piano: Devuelve "Tocando notas en el piano".
- Bateria: Devuelve "Golpeando tambores y platillos".

Crear una lista de instrumentos y usar polimorfismo para tocar todos los instrumentos.

Desafíos Propuestos:

- Agregar más subclases (ejemplo: Violin, Flauta).
- Implementar un método adicional afinar() que sea sobrescrito para cada instrumento.
- Diseñar un simulador que permita a los usuarios seleccionar y tocar instrumentos.

1.25 Ejercicio 6: Sistema de pagos con polimorfismo

Descripción:

Desarrolla un sistema que modele diferentes formas de pago (efectivo, tarjeta de crédito, transferencia bancaria) utilizando polimorfismo. Cada forma de pago debe procesarse de manera diferente.

Objetivos de Aprendizaje:

- Aplicar polimorfismo para modelar acciones dinámicas.
- Diseñar clases que sobrescriban métodos base para comportamientos únicos.
- Implementar un sistema de procesamiento de datos usando listas de objetos.

1.25.1 Especificación técnica detallada:

Crear una clase base Pago con un método:

- procesar_pago(): Método base que será sobrescrito.

Crear subclases PagoEfectivo, PagoTarjeta y PagoTransferencia:

- PagoEfectivo: Devuelve "Pago realizado en efectivo".
- PagoTarjeta: Devuelve "Pago realizado con tarjeta de crédito".
- PagoTransferencia: Devuelve "Pago realizado por transferencia bancaria".

Crear una lista de pagos y procesarlos de manera polimórfica.

Desafíos Propuestos:

- Extender el sistema para incluir un método generar_recibo() único para cada forma de pago.
- Agregar validaciones específicas para cada tipo de pago (ejemplo: tarjeta válida).
- Diseñar un menú interactivo para que el usuario seleccione y procese pagos.

1.26 Ejercicio: Sistema de cuentas bancarias

Descripción:

Desarrolla un sistema para gestionar cuentas bancarias utilizando encapsulamiento. La clase CuentaBancaria debe proteger los datos sensibles como el balance, permitiendo accesos controlados mediante métodos específicos.

Objetivos de Aprendizaje:

- Comprender el concepto de encapsulamiento y cómo proteger atributos.
- Practicar el uso de métodos getter y setter para controlar el acceso a datos.
- Diseñar sistemas seguros para trabajar con información sensible.

1.26.1 Especificación técnica detallada:

Crear una clase CuentaBancaria con los atributos privados:

- __titular (string): Nombre del titular de la cuenta.

- __balance (float): Balance de la cuenta.

Implementar métodos:

- depositar(cantidad): Incrementa el balance si la cantidad es positiva.
- retirar(cantidad): Reduce el balance si hay suficientes fondos.
- mostrar_balance(): Devuelve el balance actual.

Implementar validaciones para evitar valores inválidos (ejemplo: depósito negativo, retiros mayores al balance).

Desafíos Propuestos:

- Agregar un método para transferir fondos entre dos cuentas.
- Implementar un límite máximo de retiro diario.
- Diseñar un menú interactivo para que el usuario administre varias cuentas.

1.27 Ejercicio: Sistema de gestión de vehículos con encapsulamiento

Descripción:

Crea un sistema que administre vehículos en una empresa de transporte. Los atributos como el número de matrícula y kilometraje deben estar protegidos mediante encapsulamiento.

Objetivos de Aprendizaje:

- Aplicar el encapsulamiento para proteger datos sensibles.
- Usar métodos getter y setter para modificar atributos controlados.
- Diseñar sistemas robustos que validen datos al ser modificados.

1.27.1 Especificación Técnica Detallada:

Crear una clase Vehiculo con los atributos privados:

- __matricula (string): Número de matrícula del vehículo.
- __kilometraje (float): Kilómetros recorridos.

Implementar métodos:

- actualizar_kilometraje(nuevo_kilometraje): Permite actualizar el kilometraje solo si es mayor al actual.
- mostrar_informacion(): Devuelve los detalles del vehículo.

Diseñar un menú para registrar vehículos y actualizar su información.

Desafíos Propuestos:

- Agregar validaciones para asegurarse de que las matrículas sean únicas.
- Implementar un sistema para calcular el mantenimiento necesario basado en el kilometraje.
- Diseñar una subclase Camion que extienda Vehiculo y añada un atributo encapsulado para la capacidad de carga.

1.28 Ejercicio: Sistema de usuarios con abstracción

Descripción:

Crea un sistema que modele diferentes tipos de usuarios en un sistema (administradores y clientes). Utiliza una clase abstracta para definir métodos que deben ser implementados por cada tipo de usuario.

Objetivos de Aprendizaje:

- Comprender el concepto de abstracción y su uso en clases base.
- Aprender a usar clases y métodos abstractos para forzar la implementación de ciertos comportamientos.
- Diseñar jerarquías de clases usando herencia y abstracción.

1.28.1 Especificación técnica detallada:

Crear una clase abstracta Usuario con el método abstracto:

- mostrar_datos(): Método que cada subclase debe implementar.

Crear subclases Administrador y Cliente que extiendan Usuario:

- Administrador: Incluye un método para gestionar permisos.
- Cliente: Incluye un método para consultar su historial de compras.

Diseñar un sistema que registre usuarios y permita mostrar sus datos.

Desafíos Propuestos:

- Implementar un método adicional login() en la clase abstracta y sobrescribirlo en las subclases.
- Diseñar un sistema que permita buscar usuarios por tipo (administrador o cliente).
- Extender el sistema para incluir un tipo de usuario adicional, como Proveedor.

1.29 Ejercicio: Sistema de pedidos con abstracción y encapsulamiento

Descripción:

Desarrolla un sistema que modele pedidos en una tienda. Utiliza abstracción para definir métodos generales y encapsulamiento para proteger datos sensibles como el precio total.

Objetivos de Aprendizaje:

- Practicar el uso combinado de abstracción y encapsulamiento.
- Diseñar sistemas que manejen datos sensibles de forma segura.
- Aplicar el concepto de métodos abstractos para implementar funcionalidades específicas.

1.29.1 Especificación técnica detallada:

Crear una clase abstracta Pedido con métodos:

- calcular_total(): Método abstracto que será implementado en subclases.
- mostrar_detalles(): Método abstracto para mostrar la información del pedido.

Crear subclases PedidoOnline y PedidoLocal que implementen los métodos abstractos:

- PedidoOnline: Calcula el total incluyendo costos de envío.
- PedidoLocal: Calcula el total sin incluir costos de envío.

Usar encapsulamiento para proteger atributos como __total.

Desafíos Propuestos:

- Agregar validaciones para verificar que los pedidos online incluyan una dirección de envío válida.
- Extender el sistema para permitir descuentos en los pedidos.
- Diseñar un menú interactivo para registrar y mostrar pedidos.

1.30 Ejercicio: Sistema de productos con encapsulamiento

Descripción:

Crea un sistema para gestionar productos en un inventario. Protege atributos como el precio y la cantidad mediante encapsulamiento, permitiendo accesos controlados mediante métodos.

Objetivos de Aprendizaje:

- Aplicar encapsulamiento para proteger datos críticos en un sistema.
- Usar métodos getter y setter para validar y modificar atributos de forma controlada.

- Diseñar sistemas que implementen reglas de negocio para el manejo de datos.

1.30.1 Especificación técnica detallada:

Crear una clase Producto con atributos privados:

- __nombre (string): Nombre del producto.
- __precio (float): Precio del producto.
- __cantidad (int): Cantidad en stock.

Implementar métodos:

- actualizar_precio(nuevo_precio): Permite actualizar el precio si es mayor a cero.
- actualizar_cantidad(cantidad): Incrementa o decrementa el stock según sea necesario.
- mostrar_detalles(): Devuelve la información del producto.

Diseñar un menú para registrar productos y actualizar su información.

Desafíos Propuestos:

- Implementar un método para calcular el valor total del inventario.
- Extender el sistema para incluir una categoría para cada producto.
- Validar que no se puedan realizar ventas si el stock es insuficiente.

1.31 Ejercicio: Sistema de biblioteca con abstracción y encapsulamiento

Descripción:

Desarrolla un sistema para gestionar libros y revistas en una biblioteca. Utiliza abstracción para definir métodos comunes en una clase base y encapsulamiento para proteger datos como el estado del préstamo.

Objetivos de Aprendizaje:

- Usar abstracción para definir métodos genéricos y sobrescribirlos en subclases.
- Aplicar encapsulamiento para proteger y validar datos sensibles.
- Diseñar sistemas con reglas de negocio específicas para el manejo de datos.

1.31.1 Especificación técnica detallada:

Crear una clase abstracta Material con métodos:

- mostrar_detalles(): Método abstracto para mostrar información del material.
- estado_prestamo(): Devuelve si está disponible o prestado.

Crear subclases Libro y Revista que implementen los métodos abstractos:

- Libro: Incluye atributos como autor y numero_paginas.
- Revista: Incluye atributos como numero_edicion y tema.

Proteger atributos como __estado usando encapsulamiento y métodos para modificarlo.

Desafíos propuestos:

- Implementar un sistema que registre préstamos y devoluciones de materiales.
- Extender el sistema para incluir búsquedas avanzadas por autor o tema.
- Validar que no se puedan prestar materiales que ya estén en préstamo.

1.32 Ejercicio: Sistema de gestión de datos de ventas

Descripción:

Desarrolla un sistema que permita gestionar los datos de ventas de una empresa. Cada venta estará asociada a un cliente y a un producto, y se implementará un análisis para calcular métricas como ingresos totales, productos más vendidos y clientes frecuentes.

Objetivos de Aprendizaje:

- Integrar clases, herencia, polimorfismo, encapsulamiento y abstracción en un sistema completo.
- Aplicar conceptos de análisis de datos para procesar grandes volúmenes de información.
- Diseñar sistemas escalables que incluyan validaciones y cálculos.

1.32.1 Especificación técnica detallada:

Crear una clase base Entidad con atributos comunes (id, nombre) y un método abstracto mostrar_info().

- Cliente: Incluye atributos como email y historial_compras.
- Producto: Incluye atributos como precio y stock.
- Venta: Incluye atributos como cliente, producto, y cantidad.

Implementar métodos para:

- Registrar nuevos clientes, productos y ventas.
- Actualizar el stock de los productos al realizar una venta.
- Calcular métricas como ingresos totales, producto más vendido y cliente más frecuente.

Proteger datos sensibles como precio y email usando encapsulamiento.

Desafíos propuestos:

- Extender el sistema para incluir descuentos basados en el cliente o el producto.
- Implementar visualizaciones gráficas (usando librerías como Matplotlib) para mostrar tendencias de ventas.
- Diseñar un menú interactivo que permita analizar datos específicos, como ventas por mes o productos agotados.

1.33 Ejercicio: Sistema de análisis climático

Descripción:

Diseña un sistema que permita registrar y analizar datos climáticos de diferentes ciudades. El sistema debe permitir calcular métricas como temperaturas promedio, máximas y mínimas, y generar informes resumidos.

Objetivos de Aprendizaje:

- Aplicar todos los pilares de POO en un sistema orientado al análisis de datos.
- Diseñar un sistema que integre el manejo de datos heterogéneos.
- Implementar cálculos y validaciones para datos continuos.

1.33.1 Especificación técnica detallada:

Crear una clase base Lugar con atributos comunes (nombre, region) y métodos abstractos.

Crear subclases:

- Ciudad: Incluye atributos como temperaturas (lista de temperaturas diarias) y métodos para calcular métricas (promedio, máxima, mínima).
- EstacionClimatica: Incluye atributos como ciudades (lista de objetos Ciudad) y un método para generar un informe resumen.

Implementar métodos para:

- Registrar nuevas ciudades y sus temperaturas diarias.
- Analizar datos climáticos para obtener métricas como variaciones de temperatura.
- Generar un informe con las ciudades más cálidas y frías.

Proteger atributos como temperaturas para evitar modificaciones directas.

Desafíos Propuestos:

- Extender el sistema para incluir predicciones climáticas basadas en datos históricos.
- Crear gráficos que muestren tendencias de temperatura por región.
- Implementar una función para cargar y guardar datos climáticos desde un archivo CSV.

1.34 Ejercicio: Sistema de análisis de datos académicos

Descripción:

Crea un sistema para gestionar datos académicos de estudiantes y cursos. El sistema debe calcular métricas como el promedio general de los estudiantes, tasas de aprobación por curso y el estudiante con el mejor rendimiento.

Objetivos de Aprendizaje:

- Diseñar sistemas complejos usando POO y análisis de datos.
- Integrar cálculos avanzados y validaciones en el manejo de datos académicos.
- Aplicar visualización de datos para mostrar resultados.

1.34.1 Especificación técnica detallada:

Crear una clase base EntidadAcademica con atributos comunes (id, nombre) y un método abstracto mostrar_info().

Crear subclases:

- Estudiante: Incluye atributos como calificaciones (diccionario de cursos y calificaciones).
- Curso: Incluye atributos como estudiantes (lista de objetos Estudiante) y profesor.

Implementar métodos para:

- Registrar estudiantes y cursos.
- Asignar calificaciones a los estudiantes por curso.

- Calcular métricas como promedio por estudiante, promedio por curso y tasas de aprobación.

Proteger atributos como calificaciones y estudiantes para evitar modificaciones no controladas.

Desafíos Propuestos:

- Extender el sistema para incluir gráficos de barras que muestren tasas de aprobación por curso.
- Implementar validaciones para evitar registrar calificaciones fuera de rango (0-100).
- Crear un menú interactivo que permita buscar estudiantes o cursos y mostrar informes detallados.

1.35 Ejercicio: Sistema de análisis de redes sociales

Descripción:
Desarrolla un sistema que analice datos de usuarios y publicaciones en una red social. El sistema debe permitir calcular métricas como el usuario más activo, la publicación con más interacciones y las tendencias en hashtags.

Objetivos de Aprendizaje:

- Integrar todos los pilares de POO con el análisis de datos textuales y numéricos.
- Diseñar sistemas que combinen datos heterogéneos para análisis.
- Implementar cálculos y visualizaciones avanzadas para datos sociales.

1.35.1 Especificación técnica detallada:

Crear una clase base EntidadRedSocial con atributos comunes (id, nombre) y un método abstracto mostrar_info().

Crear subclases:

- Usuario: Incluye atributos como publicaciones (lista de objetos Publicacion) y seguidores.
- Publicacion: Incluye atributos como contenido, hashtags y interacciones.

Implementar métodos para:

- Registrar usuarios y publicaciones.
- Analizar interacciones para calcular métricas como el usuario más activo y la publicación más popular.
- Identificar los hashtags más utilizados.

Proteger atributos como seguidores y interacciones para evitar manipulaciones directas.

Desafíos Propuestos:

- Extender el sistema para incluir gráficos de línea que muestren la evolución de interacciones por día.
- Implementar una función para analizar sentimientos en los contenidos de las publicaciones.
- Crear un sistema para exportar informes de métricas en formato PDF o CSV.

1.36 Ejercicio 5: Sistema de análisis de transacciones financieras

Descripción:

Desarrolla un sistema que gestione y analice transacciones financieras de una empresa. El sistema debe permitir registrar transacciones, calcular métricas como gastos totales, ingresos totales, y generar un balance financiero.

Objetivos de Aprendizaje:

- Aplicar todos los conceptos de POO (herencia, polimorfismo, abstracción y encapsulamiento) en el contexto de datos financieros.
- Realizar análisis numéricos para generar métricas y visualizaciones.
- Diseñar sistemas robustos que permitan trabajar con grandes volúmenes de datos.

1.36.1 Especificación técnica detallada:

Crear una clase base Transaccion con atributos comunes:

- id (string): Identificador único.
- monto (float): Cantidad de la transacción.
- fecha (date): Fecha de la transacción.

Crear subclases Ingreso y Gasto que extiendan Transaccion:

- Ingreso: Incluye un atributo adicional fuente.
- Gasto: Incluye un atributo adicional categoria.

Implementar métodos para:

- Registrar nuevas transacciones (ingresos y gastos).
- Calcular métricas como ingresos totales, gastos totales, y balance final.
- Filtrar transacciones por categoría, fuente o rango de fechas.

Proteger datos sensibles como el monto utilizando encapsulamiento.

Desafíos Propuestos:

- Extender el sistema para generar reportes mensuales y anuales.
- Implementar gráficos de torta para mostrar la distribución de gastos por categoría.
- Crear un sistema para cargar y guardar transacciones desde y hacia archivos CSV.

1.37 Ejercicio: Sistema de recomendación de películas

Descripción:

Desarrolla un sistema que permita gestionar una base de datos de películas y usuarios, y que implemente un sistema de recomendación basado en calificaciones y géneros preferidos.

Objetivos de Aprendizaje:

- Integrar todos los conceptos de POO en un sistema orientado al análisis y filtrado de datos.
- Aplicar técnicas de agrupación y cálculo para generar recomendaciones personalizadas.
- Diseñar un sistema escalable para manejar grandes volúmenes de datos de usuarios y películas.

1.37.1 Especificación técnica detallada:

Crear una clase base Entidad con atributos comunes (id, nombre) y un método abstracto mostrar_info().

Crear subclases:

- Usuario: Incluye atributos como calificaciones (diccionario de películas y puntuaciones).
- Pelicula: Incluye atributos como género y calificacion_promedio.

Implementar métodos para:

- Registrar usuarios y películas.
- Calcular métricas como la calificación promedio de cada película.
- Filtrar películas por género, rango de calificaciones o popularidad.

- Generar recomendaciones para un usuario basado en las películas que ha calificado y en los géneros preferidos.

Desafíos Propuestos:

- Extender el sistema para incluir un análisis de tendencias de películas populares.
- Implementar visualizaciones como gráficos de barras para mostrar las películas más calificadas.
- Diseñar un sistema para exportar recomendaciones personalizadas en formato PDF.

CONCLUSIONES

"Programación para Ciencia de Datos utilizando Python. Tomo I" constituye una contribución significativa al aprendizaje de la programación y la ciencia de datos desde una perspectiva práctica y accesible. Este texto fue diseñado para servir como una guía detallada que introduce a los lectores en el uso de Python, un lenguaje de programación esencial en el campo del análisis de datos y de alta demanda en el mundo laboral. La obra proporciona un enfoque progresivo y estructurado que permite a los estudiantes y profesionales sin experiencia previa construir una base sólida en conceptos fundamentales, desde la programación orientada a objetos hasta el manejo de datos con NumPy.

Uno de los principales logros de este libro radica en su capacidad para transformar conceptos abstractos de la programación en aplicaciones prácticas y comprensibles. La programación orientada a objetos se presenta de manera detallada, permitiendo a los lectores comprender no solo su funcionamiento, sino también su aplicabilidad en situaciones de la vida real. Los principios de clases, objetos, herencia, polimorfismo y encapsulamiento se explican con ejemplos prácticos que los lectores pueden replicar y adaptar según sus necesidades. Esto no solo facilita el aprendizaje de los aspectos técnicos de la programación, sino que también proporciona una visión integral de cómo estructurar y organizar el código de manera eficiente y escalable. La inclusión de ejercicios prácticos al final de este capítulo permite a los lectores consolidar su comprensión y aplicar los conocimientos adquiridos en proyectos propios, lo que fomenta una experiencia de aprendizaje activa.

El manejo de arreglos multidimensionales es otra área clave abordada en este libro a través de la biblioteca NumPy, una herramienta indispensable en la ciencia de datos. La enseñanza de esta biblioteca se aborda desde sus fundamentos, permitiendo a los lectores familiarizarse con las estructuras de datos más utilizadas en análisis y procesamiento de grandes volúmenes de datos. NumPy se presenta no solo como una

herramienta de manipulación de datos, sino también como una técnica para optimizar el rendimiento y la eficiencia de los cálculos numéricos. En este sentido, los ejercicios propuestos permiten a los lectores experimentar con las diversas funcionalidades de NumPy y aplicar técnicas de optimización, reforzando así su capacidad para trabajar con datos de manera ágil y eficaz.

Además, el libro se caracteriza por su enfoque práctico. La estructura de cada capítulo está diseñada para proporcionar una transición fluida entre teoría y práctica, lo que permite a los lectores aplicar de inmediato los conceptos aprendidos. Esta estrategia facilita el desarrollo de habilidades analíticas y de resolución de problemas, fundamentales en el campo de la ciencia de datos. La inclusión de proyectos integradores al final de cada sección refuerza el aprendizaje al desafiar a los lectores a combinar y aplicar múltiples conceptos de manera coherente. Esto no solo fortalece su capacidad técnica, sino que también fomenta el pensamiento crítico y analítico, habilidades indispensables para enfrentar los desafíos del mundo laboral.

El carácter progresivo del contenido asegura que los lectores obtengan una base sólida en programación y análisis de datos antes de aventurarse en temas más avanzados. Este libro no solo enseña a programar en Python, sino que también prepara a los lectores para entender y utilizar bibliotecas mediante técnicas de análisis de datos que serán útiles en etapas posteriores de su formación académica y profesional. Al establecer una base en conceptos fundamentales, el libro ofrece una plataforma desde la cual los lectores pueden explorar temas avanzados, como el aprendizaje automático y la inteligencia artificial, con mayor confianza y comprensión.

El impacto de esta obra no se limita a la adquisición de habilidades técnicas. Al proporcionar un recurso accesible para el aprendizaje de la ciencia de datos, este libro también fomenta una inclusión tecnológica significativa. La accesibilidad de Python como lenguaje de programación de código abierto, combinado con la claridad de la estructura del libro, democratiza el aprendizaje de la ciencia de datos, haciéndolo accesible para estudiantes y profesionales de diversos contextos y niveles de

experiencia. Esto tiene un valor especial en contextos de educación superior en Latinoamérica, donde el acceso a recursos de aprendizaje de calidad puede ser limitado.

Asimismo, este libro destaca la importancia de la autogestión y la responsabilidad en el aprendizaje. A lo largo de los capítulos, los lectores son motivados a asumir un rol activo en su proceso de aprendizaje, enfrentándose a ejercicios y retos que simulan problemas reales en ciencia de datos. La estructura autodidacta del libro fomenta la disciplina y la perseverancia, valores fundamentales para el aprendizaje continuo y el crecimiento profesional. Al desarrollar proyectos autónomos, los lectores no solo aplican los conocimientos adquiridos, sino que también ganan confianza en su capacidad para resolver problemas y enfrentar desafíos.

Finalmente, la obra sienta un precedente educativo significativo al servir como bibliografía básica en cursos de ciencia de datos, programación y análisis de datos. Su inclusión en el currículo académico de instituciones de educación superior no solo fortalecería la formación de los estudiantes, sino que también enriquecería el contenido de los programas de estudio, adaptándolos a las demandas del mercado laboral actual. Este libro constituye un recurso didáctico que complementa y refuerza el aprendizaje, ofreciendo a los estudiantes una oportunidad para experimentar, explorar y consolidar los conceptos que se enseñan en el aula.

BIBLIOGRAFÍA

Classes (OOP) | Brilliant Math & Science Wiki. (s. f.). Recuperado 13 de abril de 2024, de https://brilliant.org/wiki/classes-oop/

Guía del usuario de NumPy—Manual de NumPy v1.22. (s. f.). Recuperado 10 de mayo de 2024, de https://NumPy.org/doc/1.22/user/index.html#user

Jorge Santiago Nolasco Valenzuela. (2018). *Python Aplicaciones prácticas*. RA-MA Editorial.

Lott, S. F., & Phillips, D. (2021). *Python Object-Oriented Programming: Build robust and maintainable object-oriented Python applications and libraries*. Packt Publishing Ltd.

NumPy documentation—NumPy v1.22 Manual. (s. f.). Recuperado 10 de mayo de 2024, de https://NumPy.org/doc/1.22/

Óscar Ramírez Jiménez. (2021). *Python a fondo*. Marcombo.

Python Data Science Handbook | Python Data Science Handbook. (s. f.). Recuperado 10 de mayo de 2024, de https://jakevdp.github.io/PythonDataScienceHandbook/

Python para el análisis de datos. (s. f.). Udemy. Recuperado 30 de septiembre de 2024, de https://www.udemy.com/course/python-analisis-de-datos/

Rodríguez Guerrero, R., Vanegas, C. A., & Castang Montiel, G. (2020). *Python a su alcance*. Editorial Universidad Distrital Francisco José de Caldas. Editorial UD.

Classes (OOP) | Brilliant Math & Science Wiki. (s. f.). Recuperado 13 de abril de 2024, de https://brilliant.org/wiki/classes-oop/

Guía del usuario de NumPy—Manual de NumPy v1.22. (s. f.). Recuperado 10 de mayo de 2024, de https://NumPy.org/doc/1.22/user/index.html#user

Jorge Santiago Nolasco Valenzuela. (2018). *Python Aplicaciones prácticas*. RA-MA Editorial.

Lott, S. F., & Phillips, D. (2021). *Python Object-Oriented Programming: Build robust and maintainable object-oriented Python applications and libraries*. Packt Publishing Ltd.

NumPy documentation—NumPy v1.22 Manual. (s. f.). Recuperado 10 de mayo de 2024, de https://NumPy.org/doc/1.22/

Óscar Ramírez Jiménez. (2021). *Python a fondo*. Marcombo.

Python Data Science Handbook | Python Data Science Handbook. (s. f.). Recuperado 10 de mayo de 2024, de https://jakevdp.github.io/PythonDataScienceHandbook/

Python para el análisis de datos. (s. f.). Udemy. Recuperado 30 de septiembre de 2024, de https://www.udemy.com/course/python-analisis-de-datos/

Rodríguez Guerrero, R., Vanegas, C. A., & Castang Montiel, G. (2020). *Python a su alcance*. Editorial Universidad Distrital Francisco José de Caldas. Editorial UD.

EL libro: Programación para ciencia de datos utilizando Python, se publicó en el mes de enero de 2025.

ISBN: 978-9942-48-913-5

Editorial InvestiGo
Riobamba – Ecuador
Cel: +593 97 911 9620
investigoeditorial@gmail.com

BIOGRAFÍA DE LOS AUTORES

Alfredo Rodrigo Colcha Ortiz.

Magister en Informática Aplicada, Microsoft Certified Professional de Microsoft (MCP), Ingeniero de Sistemas, Docente de instituciones de educación superior de grado y posgrado Universidad Nacional de Chimborazo (UNACH) y Escuela Superior Politécnica de Chimborazo (ESPOCH), participante en proyectos de investigación, publicaciones científicas ponente en eventos académicos nacionales e internacionales, Director de tesis, méritos en proyectos de Smart Cities, Consultor independiente.

José Luis Jinez Tapia.

Master en Ingeniería de las Telecomunicaciones, Magister en Electricidad Energías Renovables y Eficiencia Energética, Ingeniero en Electrónica y Telecomunicaciones, Docente de instituciones de educación superior Universidad Nacional de Chimborazo (UNACH), Docente investigador, participación en proyectos de investigación y vinculación, Autor de publicaciones científicas, Director de tesis de grado y posgrado.

Wilmer Enrique Mera Herrera.

Magister en Gestión Empresarial, Magister en Matemática Aplicada con mención en Matemática Computacional, Ingeniero Industrial, Tecnólogo en Aviónica, Coordinador de Posgrado Universidad Nacional de Chimborazo (UNACH), Investigador en proyectos de Investigación de la Universidad Nacional de Chimborazo, Autor de publicaciones científicas, Asesor de tesis de posgrado.

PROGRAMACIÓN PARA CIENCIA DE DATOS UTILIZANDO PYTHON

En la era digital actual, el manejo de datos se ha convertido en una habilidad imprescindible para profesionales de todos los campos. Python, reconocido como uno de los lenguajes de programación más versátiles y accesibles, ofrece una poderosa plataforma para explorar, analizar y visualizar datos. Con una curva de aprendizaje amigable, una rica colección de bibliotecas como NumPy y Pandas, y el respaldo de una comunidad global activa, Python es la herramienta ideal tanto para principiantes como para expertos que buscan potenciar sus proyectos y resolver desafíos reales en el ámbito de la ciencia de datos.

Este libro está diseñado para guiar en el mundo de la programación y el análisis de datos, sin necesidad de experiencia previa. A través de capítulos prácticos, podrá aprender a dominar conceptos clave como la Programación Orientada a Objetos y el manejo eficiente de datos multidimensionales. Cada lección combina teoría, ejemplos prácticos y ejercicios desafiantes que permitirán aplicar lo aprendido en contextos profesionales, como el análisis financiero, la visualización de tendencias o la simulación de fenómenos complejos.

Más allá de la técnica, este libro fomenta el pensamiento crítico y analítico, clave para resolver problemas del mundo real. Preparará al lector no solo para dominar Python, sino también para enfrentar retos avanzados como el aprendizaje automático y la inteligencia artificial. Al finalizar, habrá adquirido una base sólida para liderar proyectos innovadores, integrar herramientas de vanguardia y aprovechar al máximo el poder de los datos en cualquier área profesional. ¡Inicie su viaje hacia el dominio del análisis de datos con Python y transforme su carrera!

EDITORIAL
InvestiGO

ISBN: 978-9942-48-913-5

Editorial InvestiGo
Riobamba – Ecuador
Cel: +593 97 911 9620
investigoeditorial@gmail.com