Evaluacion: Programacion funcional, multiprocessing, asyncio y concurrent.futures

¿Qué es la programación funcional?

La base de la programación funcional es la idea matemática de una función, que acepta entradas y devuelve una salida sin cambiar ningún estado. Las funciones pueden asignarse a variables, suministrarse como argumentos a otras funciones y devolverse como valores en la programación funcional, ya que son reconocidas como ciudadanos de primera clase. Este método compositivo facilita la prueba y depuración del código y el razonamiento sobre el comportamiento del programa.

La inmutabilidad, transparencia referencial, funciones de orden superior, recursión y evaluación perezosa son algunos de los conceptos fundamentales de la programación funcional. Estas ideas permiten a los programadores crear código declarativo, componible y fácil de entender.

En lugar de cambiar el estado, la programación funcional se centra en calcular valores. Por lo tanto, las funciones no tienen efectos secundarios.

El paradigma de programación funcional para Python tiene varios beneficios que lo hacen útil para los desarrolladores. Algunos de los beneficios incluyen:

- Código conciso y expresivo: Los desarrolladores pueden escribir código que es claro, expresivo y fácil de leer y modificar utilizando la programación funcional. Las funciones de orden superior de Python proporcionan a los programadores un código base listo para modificar que puede tener fácilmente código nuevo añadido según sea necesario, permitiendo así un crecimiento más rápido.
- Más fácil de razonar: La programación funcional enfatiza las funciones puras, que no tienen efectos secundarios y dan el mismo resultado para la misma entrada. Esto hace que sea más fácil pensar sobre el comportamiento de una función y cómo afecta al resto del programa.
- Paralelismo y concurrencia: Al minimizar estados mutables y efectos secundarios, la programación funcional facilita el paralelismo y la concurrencia. Esto hace que sea más fácil escribir código que puede ejecutarse en paralelo o en múltiples hilos.
- Testabilidad: La programación funcional promueve funciones puras, que son más fáciles de verificar porque no tienen efectos secundarios. Esto facilita la escritura de pruebas unitarias y asegura la exactitud de un programa.
- Mejora en la modularidad y reutilización: La programación funcional enfatiza funciones pequeñas y modulares que pueden reutilizarse en diferentes partes de un programa. Como resultado, el código se vuelve más modular y es más fácil de mantener y refactorizar.
- Interoperabilidad con otros paradigmas: La programación funcional es uno de los varios paradigmas de programación que Python admite. Esto significa que los desarrolladores pueden combinar y contrastar diferentes paradigmas según las necesidades específicas de un programa.

Python proporciona varios conceptos de programación funcional que permiten a los desarrolladores escribir código funcional.

- Funciones de orden superior: Estos son bloques de código que aceptan otros bloques de código como entrada y producen otros bloques de código como salida. Permiten a los programadores crear código

reutilizable y componible.

- Funciones anónimas: Estas funciones no tienen nombre. Se generan utilizando la palabra clave lambda y pueden utilizarse en funciones de orden superior.
- Estructuras de datos inmutables: Estas son estructuras de datos que, una vez construidas, no pueden ser cambiadas, como las tuplas y los frozensets. Las estructuras de datos inmutables permiten a los programadores crear código más comprensible y seguro para los hilos.
- Generadores: Estas son operaciones que permiten pausar y reiniciar. Los generadores facilitan la evaluación perezosa, que es ventajosa para procesar grandes cantidades de datos.
- Decoradores: Estas son funciones que modifican cómo otras funciones se comportan. Los decoradores permiten a los programadores extender la funcionalidad de funciones ya escritas sin cambiar el código fuente.

In [2]:
```python
# Define una función decoradora que añade información de tiempo a una función
import time
def time_it(func):
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} tomó {end - start} segundos en ejecutarse.")
        return result
    return wrapper
# Usa el decorador time_it para medir cuánto tiempo tarda en ejecutarse una función
@time_it
def my_function():
    time.sleep(1)
my_function() # Salida: my_function tomó 1.0000739097595215 segundos en ejecutarse.
```

```
my_function tomó 1.0009305477142334 segundos en ejecutarse.
```

- Funciones de primer orden: Como ciudadanos de primera clase, las funciones de Python pueden asignarse a variables, utilizarse como argumentos para otras funciones y tener valores devueltos a ellas. Esto facilita la creación de composiciones de funciones fuertes y funciones de orden superior.
- Recursión: Utilizando esta estrategia para dividir un problema más grande en subproblemas más pequeños, se puede usar para resolverlo. La característica de recursión de Python permite la creación de código elegante y conciso para algoritmos complejos. Los desarrolladores de Python pueden escribir código más condensado, expresivo y defensible aprovechando estas técnicas de programación funcional. También pueden utilizar todo el potencial de las características de programación funcional incorporadas de Python para producir programas que son más modulares y reutilizables.

Mejores prácticas para la programación funcional en Python

- Usar estructuras de datos inmutables: Las estructuras de datos inmutables, donde una vez creadas no pueden modificarse bajo ninguna circunstancia, como las tuplas y los frozensets, son importantes en la programación funcional porque pueden compartirse de manera segura entre funciones sin riesgo de efectos secundarios no deseados. Las estructuras de datos mutables pueden introducir errores y hacer más difícil razonar sobre tu código. Las estructuras de datos inmutables garantizan seguridad.

In [11]:
```python
# Crear una tupla inmutable
my_tuple = (1, 2, 3)

# Intentar modificar la tupla lanzará un error porque las tuplas son inmutables
try:
```

```
    my_tuple[0] = 0  # Intento de modificar la tupla
except TypeError as e:
    print(f"Error: {e}")  # Mostrar el error esperado

# Crear un frozenset, que es una versión inmutable de un conjunto (set)
my_frozenset = frozenset([4, 5, 6])  # Crear un frozenset con los elementos 4, 5 y 6

# Imprimir la tupla y el frozenset para ver sus contenidos
print("Tupla:", my_tuple)
print("Frozenset:", my_frozenset)
```

```
Error: 'tuple' object does not support item assignment
Tupla: (1, 2, 3)
Frozenset: frozenset({4, 5, 6})
```

- Evitar el estado global: El estado global puede dificultar el razonamiento del código que escribas e introducir errores. En lugar de eso, usa argumentos de función y valores de retorno para pasar datos entre funciones.

In [4]:
```
def calculate_sum(a, b):
    return a + b
def main():
    x = 2
    y = 3
    result = calculate_sum(x, y)
    print(f"La suma de {x} y {y} es {result}")
if __name__ == "__main__":
    main()
```

```
La suma de 2 y 3 es 5
```

- Usar funciones de orden superior: Las funciones de orden superior son funciones que toman otras funciones como entrada o devuelven funciones como salida. Te permiten escribir código que es componible y reutilizable. Python proporciona varias funciones de orden superior integradas, como `map()`, `filter()` y `reduce()`, que se utilizan comúnmente en la programación funcional.

In [5]:
```
# Define una función de orden superior que toma una función y la aplica a una lista de n
def apply_function(numbers, function):
    return [function(number) for number in numbers]
# Define una función para elevar al cuadrado un número
def square(number):
    return number ** 2
# Usa la función apply_function para elevar al cuadrado una lista de números
numbers = [1, 2, 3, 4, 5]
squared_numbers = apply_function(numbers, square)
print(squared_numbers) # Salida: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

- Usar recursión: Mediante el uso del poder de la recursión podemos resolver el problema dividiéndolos en subproblemas más pequeños. Python sí soporta la recursión y ayuda a escribir código elegante y conciso incluso para problemas complejos que existen.

In [6]:
```
# Define una función recursiva para calcular el factorial de un número
# ejemplo clásico de factorial aquí
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

```
# Usa la función para calcular el factorial de 5
result = factorial(5)
print(result) # Salida: 120
```

```
120
```

- Usar funciones lambda: Las funciones anónimas que se pueden utilizar para crear funciones simples de una línea se denominan funciones Lambda en Python. A menudo se utilizan en conjunto con funciones de orden superior, como map(), filter() y reduce(), para escribir código conciso y expresivo.

In [7]:
```
# Define una función lambda para elevar al cuadrado un número
square = lambda x: x ** 2
# Usa la función lambda para elevar al cuadrado una lista de números
numbers = [1, 2, 3, 4, 5]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers) # Salida: [1, 4, 9, 16, 25]
```

```
[1, 4, 9, 16, 25]
```

- Escribir funciones puras: Una función pura es una función que no tiene efectos secundarios y siempre devuelve la misma salida para la misma entrada. En otras palabras, una función pura no modifica ningún estado o variable externo y solo opera sobre sus parámetros de entrada. Las funciones puras son importantes en la programación funcional porque son predecibles y fáciles de razonar, lo que las hace más fáciles de probar, depurar y mantener.

In [8]:
```
def add(a, b):
    return a + b
```

- Usar generadores: Los generadores son funciones que pueden pausarse y reanudarse. Permiten la evaluación perezosa, que puede ser útil para procesar grandes cantidades de datos. Al usar generadores, puedes evitar cargar todos los datos en la memoria a la vez y procesarlos de manera más eficiente en términos de memoria

In [9]:
```
# Define una función generadora para generar la secuencia de Fibonacci
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b
# Usa la función generadora para generar los primeros 10 números en la secuencia de Fibo
fib = fibonacci()
fib_numbers = [next(fib) for _ in range(10)]
print(fib_numbers) # Salida: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

- Probar tu código: Probar es importante en la programación funcional porque puede ayudarte a detectar errores temprano y asegurar que tus funciones se comporten correctamente. Escribe casos de prueba para tus funciones y usa herramientas como pytest para automatizar tus pruebas.

In [10]:
```
def calculate_sum(a, b):
    return a + b
def test_calculate_sum():
    assert calculate_sum(2, 3) == 5
    assert calculate_sum(0, 0) == 0
if __name__ == "__main__":
    test_calculate_sum()
```

Errores comunes en la programación funcional en Python y cómo evitarlos

La programación funcional es un poderoso paradigma de programación que enfatiza el uso de funciones puras, datos inmutables, un estilo de programación declarativo y hace énfasis en las pruebas. Sin embargo, incluso los desarrolladores experimentados pueden cometer algunos errores comunes de programación funcional en Python. Es importante evitar los errores más comunes. Discutiremos algunos de estos errores y cómo evitarlos.

- Modificar estructuras de datos mutables: El uso de estructuras de datos inmutables es uno de los principales principios de la programación funcional. Sin embargo, Python ofrece una serie de estructuras de datos mutables que pueden ser cambiadas mientras aún están en uso, como las listas y los diccionarios. La modificación de una estructura de datos mutable puede resultar en un comportamiento inesperado y hacer que el código sea difícil de entender. Usa estructuras de datos inmutables o duplica estructuras de datos mutables antes de hacer cambios en ellas para evitar cometer este error.

- Usar variables globales: El código que usa variables globales puede ser difícil de interpretar y probar. Evita utilizar variables globales completamente cuando uses la programación funcional. En su lugar, transmite datos entre funciones utilizando argumentos de función y valores de retorno.

- Usar bucles en lugar de funciones de orden superior: Aunque se utilizan con frecuencia en la programación imperativa, los bucles pueden hacer que el código sea más difícil de leer y mantener. Cuando se manipulan colecciones de datos en la programación funcional, es preferible usar funciones de orden superior como map, filter y reduce. Estas funciones ofrecen un medio declarativo de comunicar la intención del código y pueden facilitar el razonamiento sobre él.

- No aprovechar la evaluación perezosa: Un método conocido como evaluación perezosa pospone la evaluación de una expresión hasta que su valor es realmente necesario. En Python, se pueden usar generadores e iteradores para implementar la evaluación perezosa. La falta de uso de la evaluación perezosa puede resultar en un mayor uso de la memoria y un rendimiento lento. Usa generadores e iteradores cuando trabajes con grandes conjuntos de datos para evitar cometer este error.

- No usar la recursión: La recursión es un concepto fundamental en la programación funcional y una herramienta potente para la resolución de problemas de manera declarativa. Pero si la profundidad de la recursión es demasiado grande, Python tiene un límite de recursión predeterminado que se puede alcanzar rápidamente. Usa la recursión de cola o cambia los algoritmos recursivos a iterativos para evitar cometer este error.

Ventajas de la programación funcional en Python

- Previsibilidad: Es crucial tener un código previsible, ya que facilita las pruebas y la depuración. Cuando una función incluye efectos secundarios, puede ser más difícil predecir cómo se comportará en ciertas circunstancias. Las funciones puras son esenciales para la programación funcional porque hacen que sea fácil predecir el resultado de una función dado un cierto insumo. Ventaja probada para el desarrollador.

- Modularidad: La programación funcional facilita la evitación más simple de errores causados por estados mutables y efectos secundarios mediante el uso de funciones puras e inmutabilidad.

- Concurrencia y paralelismo: Debido a la facilidad de crear aplicaciones multi-hilo o multi-proceso, las funciones pueden ejecutarse simultáneamente sin correr el riesgo de colisionar. Hacer que no haya problema para situaciones de carrera y problemas de sincronización no es una preocupación porque las funciones no dependen de ningún estado externo. Además, la programación funcional promueve el uso de estructuras de datos inmutables y alienta la escritura de funciones puras, lo que facilita la escritura de código paralelo y concurrente.

- Optimización y refactorización del código: Dado que las funciones tienen salidas predecibles, el compilador o intérprete puede hacer suposiciones sobre cómo se comportará la función y optimizar en consecuencia. Esto hace que la programación funcional sea una buena opción para aplicaciones que necesitan ser altamente eficientes.

- Legibilidad, reutilización y mantenibilidad: Los pequeños funciones reutilizables pueden ser utilizadas por los desarrolladores para construir programas más complejos gracias a la programación funcional. Además, promueve el uso de funciones y desalienta el uso de efectos secundarios, haciendo que el código sea más fácil de leer y entender. Esto reduce la complejidad del código y lo hace más fácil de mantener.

- Testabilidad: Con el uso de funciones puras, la programación funcional facilita la prueba del código, ya que asegura que la salida de una función solo dependa de su entrada.

- Evitación de errores: Al usar los métodos mencionados anteriormente cuando se combinan, ayuda a evitar errores fácilmente para que el desarrollador detecte cualquier error y lo resuelva.

Ejercicios

Presenta tus respuestas en tu repositorio personal. Estos ejercicios proporcionan un reto significativo que pone a prueba tus habilidades en programación funcional, así como tu capacidad para integrar y optimizar la computación paralela y concurrente en aplicaciones reales y complejas.

Ejercicio 1: Sistema de procesamiento de imágenes en tiempo real

Desarrollar un sistema que procese imágenes en tiempo real para detectar y clasificar objetos, utilizando programación funcional para el manejo de las transformaciones de imágenes y concurrent.futures para procesar múltiples flujos de imágenes simultáneamente.

Descripción:

- Implementa funciones puras para cada paso del procesamiento de imágenes, como filtrado, detección de bordes, y clasificación de objetos.
- Utiliza concurrent.futures.ThreadPoolExecutor para paralelizar el procesamiento de imágenes provenientes de múltiples fuentes en tiempo real.
- Emplea asyncio para manejar asincrónicamente la entrada/salida de imágenes y la integración con sistemas de almacenamiento o bases de datos. Asegurar que todas las operaciones sobre los datos de las imágenes sean inmutables para evitar efectos secundarios no deseados.

Adicional:

- Integra el sistema con una interfaz web en tiempo real donde los usuarios puedan cargar imágenes y recibir resultados inmediatamente.

Sugerencias

1 .Primero, definiremos algunas funciones puras para realizar operaciones básicas de procesamiento de imágenes, como la conversión a escala de grises y la detección de bordes. Usaremos Pillow (PIL) para las operaciones de imagen:

In [ ]:
```
from PIL import Image, ImageFilter

def convert_to_grayscale(image):
    """Convierte la imagen a escala de grises."""
    return image.convert('L')

def apply_edge_detection(image):
    """Aplica detección de bordes a la imagen."""
    return image.filter(ImageFilter.FIND_EDGES)
```

2 . Función decoradora para paralelizar el procesamiento

A continuación, usaremos un decorador para medir el tiempo que tarda cada operación y paralelizar el procesamiento utilizando ThreadPoolExecutor de concurrent.futures:

In [ ]:
```
import time
from concurrent.futures import ThreadPoolExecutor
from functools import wraps

def time_it(func):
    """Decorador que mide el tiempo de ejecución de una función."""
    @wraps(func)
    def wrapper(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()
        print(f"{func.__name__} took {end - start:.2f} seconds to run.")
        return result
    return wrapper

def parallelize_image_processing(function):
    """Decorador que paraleliza el procesamiento de imágenes."""
    @wraps(function)
    def wrapper(images):
        with ThreadPoolExecutor(max_workers=5) as executor:
            results = list(executor.map(function, images))
        return results
    return wrapper
```

3 . Procesamiento asíncrono y concurrente de imágenes

Finalmente, integramos asyncio para manejar el flujo de imágenes de entrada y salida, y aplicamos los decoradores a nuestras funciones de procesamiento:

In [ ]:
```
import asyncio

@time_it
@parallelize_image_processing
def process_images(images):
    """Procesa una lista de imágenes aplicando las funciones de procesamiento."""
    processed_images = [convert_to_grayscale(img) for img in images]
    processed_images = [apply_edge_detection(img) for img in processed_images]
    return processed_images

async def main():
```

```
    # Simulando la carga de imágenes
    images = [Image.open(f'image_{i}.jpg') for i in range(10)]  # Asegúrate de tener imá
    processed_images = process_images(images)
    # Aquí podrías guardar las imágenes procesadas o enviarlas a otro servicio

if __name__ == "__main__":
    asyncio.run(main())
```

Ejercicio 2: Simulación de sistema de reservas con alta concurrencia

Crea una simulación de un sistema de reservas (como para hoteles o vuelos) que pueda manejar un alto volumen de peticiones concurrentes sin conflictos de datos.

Descripción:

- Diseña funciones puras para manejar la lógica de reservas, cancelaciones, y modificaciones de reservas.
- Utiliza asyncio para manejar múltiples solicitudes de clientes de manera asincrónica.
- Emplea multiprocessing para distribuir la carga de trabajo a través de múltiples procesos y aprovechar múltiples núcleos de CPU, especialmente para tareas que requieren intensos cálculos o procesamiento de datos.
- Implementar mecanismos de sincronización y control de concurrencia para asegurar la consistencia y la integridad de los datos en un entorno de múltiples usuarios.

Adicional:

- Simula un escenario de "Black Friday" donde se espera un pico de demanda y evaluar el rendimiento y la escalabilidad del sistema.

Sugerencias:

1 .Definición de funciones puras para lógica de reservas

Primero, definimos algunas funciones puras que manipulan el estado de las reservas sin producir efectos secundarios:

In [ ]:
```
def add_reservation(reservations, reservation):
    """Agrega una nueva reserva a la lista de reservas de manera inmutable."""
    return reservations + [reservation]

def cancel_reservation(reservations, reservation_id):
    """Cancela una reserva por ID, inmutablemente."""
    return [res for res in reservations if res['id'] != reservation_id]

def update_reservation(reservations, reservation_id, new_details):
    """Actualiza una reserva por ID, inmutablemente."""
    return [res if res['id'] != reservation_id else {**res, **new_details} for res in re
```

2 . Manejo de concurrencia con concurrent.futures

Usaremos ThreadPoolExecutor para simular el procesamiento paralelo de solicitudes de reserva:

In [ ]:
```
from concurrent.futures import ThreadPoolExecutor
import copy

def process_booking_requests(requests):
    """Procesa una lista de solicitudes de reserva concurrentemente."""
    with ThreadPoolExecutor(max_workers=5) as executor:
```

```
        results = list(executor.map(handle_request, requests))
    return results

def handle_request(request):
    """Maneja una solicitud individual simulando cierta lógica y tiempo de procesamiento
    # Simulación de procesamiento: modificar según la lógica de negocio
    time.sleep(random.uniform(0.1, 0.5))  # Simular tiempo de procesamiento
    return f"Processed request {request['id']} with status: {request['status']}"
```

3 . Integración con asyncio para Asincronía

Integraremos asyncio para simular un sistema que maneje solicitudes de reserva de manera asincrónica:

In [ ]:
```
import asyncio

async def manage_reservations(requests):
    """Gestiona reservas asincrónicamente."""
    loop = asyncio.get_running_loop()
    future = loop.run_in_executor(None, process_booking_requests, requests)
    result = await future
    print(result)

async def simulate_requests():
    """Simula la llegada de solicitudes de reserva."""
    requests = [{'id': i, 'status': 'new'} for i in range(10)]  # Simular 10 solicitudes
    await manage_reservations(requests)

if __name__ == "__main__":
    asyncio.run(simulate_requests())
```

Ejercicio 3: Análisis y visualización de datos de tráfico en tiempo real

Construye una aplicación que recoja, procese y visualice datos de tráfico en tiempo real para una ciudad, utilizando programación funcional para el análisis de datos y asyncio junto con concurrent.futures para el procesamiento concurrente.

Descripción:

- Recoge datos de tráfico de múltiples fuentes, como cámaras de tráfico y sensores en carreteras.
- Implementa funciones puras para calcular métricas de tráfico, como velocidad media, densidad de tráfico, y tiempos de viaje estimados.
- Usa asyncio para procesar datos de tráfico de manera asincrónica.
- Aplica concurrent.futures.ProcessPoolExecutor para realizar análisis de datos pesados en paralelo.
- Desarrollar una interfaz de usuario que muestre en tiempo real los datos de tráfico y las métricas en un mapa interactivo.

Adicional:

- Predice patrones de tráfico y congestiones utilizando modelos de machine learning sobre los datos procesados.

Sugerencias:

1 . Definición de funciones puras para análisis de datos

Primero, vamos a definir algunas funciones puras para calcular métricas comunes en datos de tráfico, como la velocidad promedio y el volumen de tráfico:

In [ ]:
```python
def calculate_average_speed(data):
    """Calcula la velocidad promedio a partir de datos de velocidad recogidos."""
    total_speed = sum(d['speed'] for d in data)
    count = len(data)
    return total_speed / count if count else 0

def calculate_traffic_volume(data):
    """Calcula el volumen de tráfico a partir de datos de conteo de vehículos."""
    return sum(d['vehicles'] for d in data)
```

2 . Uso de concurrent.futures para paralelizar el procesamiento de datos

Vamos a utilizar concurrent.futures para procesar datos de tráfico provenientes de múltiples ubicaciones en paralelo, lo que puede ser crucial durante las horas pico:

In [ ]:
```python
from concurrent.futures import ProcessPoolExecutor

def process_traffic_data(locations_data):
    """Procesa datos de múltiples ubicaciones utilizando procesamiento en paralelo."""
    with ProcessPoolExecutor() as executor:
        results = list(executor.map(process_single_location, locations_data))
    return results

def process_single_location(location_data):
    """Procesa los datos de tráfico de una única ubicación."""
    average_speed = calculate_average_speed(location_data)
    traffic_volume = calculate_traffic_volume(location_data)
    return {'location': location_data['location'], 'average_speed': average_speed, 'traf
```

3 . Integración con asyncio para manejo asíncrono

Incorporamos asyncio para simular la recolección y procesamiento asíncrono de datos de tráfico, así como para actualizar la interfaz de usuario:

In [ ]:
```python
import asyncio

async def update_traffic_data():
    """Actualiza periódicamente los datos de tráfico y la visualización."""
    while True:
        locations_data = fetch_traffic_data()  # Esta función debe ser implementada para
        processed_data = await asyncio.get_event_loop().run_in_executor(None, process_tr
        update_visualization(processed_data)  # Esta función debe actualizar la interfaz
        await asyncio.sleep(10)  # Actualiza cada 10 segundos

def fetch_traffic_data():
    """Simula la recolección de datos de tráfico de múltiples sensores."""
    # Esta función debe implementarse para interactuar con sensores o APIs
    return [{'location': 'Location A', 'speed': [40, 50, 55], 'vehicles': [100, 120, 130
             {'location': 'Location B', 'speed': [30, 35, 40], 'vehicles': [90, 110, 115]

def update_visualization(data):
    """Actualiza una interfaz de usuario o un dashboard con los últimos datos procesados
    # Esta función debe ser implementada para mostrar datos actualizados
    print("Updated visualization with:", data)

if __name__ == "__main__":
    asyncio.run(update_traffic_data())
```

Ejercicio 4: Sistema de análisis de sentimiento en tiempo real para redes sociales

Desarrolla un sistema que monitoree y analice en tiempo real los sentimientos de los usuarios en redes sociales, utilizando principios de programación funcional para procesar los textos y asyncio para manejar múltiples flujos de datos.

Descripción:

- Crea funciones puras para limpiar y pre-procesar los textos recolectados de las redes sociales.
- Implementa análisis de sentimiento utilizando una librería de procesamiento de lenguaje natural, asegurando que las funciones sean inmutables y sin efectos secundarios.
- Utiliza asyncio para recoger datos simultáneamente de varias plataformas de redes sociales.
- Emplea concurrent.futures para procesar y analizar los datos de sentimiento en paralelo, mejorando la capacidad de respuesta del sistema.

Sugerencias:

1 . Primero, definiremos funciones puras para realizar el pre-procesamiento de los textos extraídos de las redes sociales, como la eliminación de stopwords, tokenización y normalización:

In [ ]:
```
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

stop_words = set(stopwords.words('english'))

def clean_text(text):
    """Limpia el texto eliminando caracteres especiales y convirtiéndolo a minúsculas."""
    text = re.sub(r'\W', ' ', text)
    text = text.lower()
    return text

def remove_stopwords(text):
    """Elimina las stopwords de un texto."""
    words = word_tokenize(text)
    filtered_words = [word for word in words if word not in stop_words]
    return ' '.join(filtered_words)

def preprocess_text(text):
    """Combina todas las operaciones de preprocesamiento de texto."""
    text = clean_text(text)
    text = remove_stopwords(text)
    return text
```

2 . Función para análisis de sentimiento

Usaremos una librería externa, como TextBlob, para realizar análisis de sentimiento, y encapsularemos esta operación en una función pura:

In [ ]:
```
from textblob import TextBlob

def analyze_sentiment(text):
    """Analiza el sentimiento de un texto dado y devuelve el resultado."""
    analysis = TextBlob(text)
    return analysis.sentiment
```

3 . Uso de concurrent.futures para paralelizar el análisis

Utilizaremos concurrent.futures para procesar y analizar los datos de sentimiento en paralelo:

In [ ]:
```python
from concurrent.futures import ThreadPoolExecutor

def analyze_texts_concurrently(texts):
    """Analiza una lista de textos concurrentemente."""
    with ThreadPoolExecutor(max_workers=10) as executor:
        results = list(executor.map(preprocess_and_analyze, texts))
    return results

def preprocess_and_analyze(text):
    """Preprocesa y analiza el sentimiento de un texto."""
    preprocessed_text = preprocess_text(text)
    sentiment = analyze_sentiment(preprocessed_text)
    return sentiment
```

4 . Integración con asyncio para manejo asíncrono

Incorporaremos asyncio para gestionar asincrónicamente la recolección y procesamiento de datos:

In [ ]:
```python
import asyncio

async def collect_and_process_data(stream_data):
    """Asíncronamente recolecta y procesa datos de un flujo."""
    processed_data = await asyncio.get_event_loop().run_in_executor(None, analyze_texts_
    print("Sentiment Analysis Results:", processed_data)

async def simulate_streaming_data():
    """Simula la llegada de datos de texto de un flujo en tiempo real."""
    sample_data = [
        "I love this product!",
        "This is the worst service ever.",
        "I'm not sure if I like this.",
        "Absolutely fantastic!"
    ]
    await collect_and_process_data(sample_data)

if __name__ == "__main__":
    asyncio.run(simulate_streaming_data())
```

Ejercicio 5: Plataforma de análisis de datos genómicos distribuidos

Construye una plataforma para analizar grandes volúmenes de datos genómicos de manera eficiente, utilizando programación funcional para las transformaciones y cálculos, y multiprocessing para el procesamiento paralelo.

Descripción:

- Implementa funciones puras para diversas operaciones genómicas, como el alineamiento de secuencias, la identificación de variantes, y el cálculo de frecuencias genéticas.
- Utiliza multiprocessing.Pool para distribuir el procesamiento de datos a múltiples núcleos de procesador, permitiendo que grandes conjuntos de datos se analicen rápidamente.
- Integra asyncio para manejar eficientemente las entradas/salidas, especialmente para cargar y guardar grandes conjuntos de datos genómicos.
- Implementa un sistema de CI/CD que automatice las pruebas, valide la integridad del código, y orqueste la implementación de la infraestructura necesaria en un entorno de computación en la nube.

Sugerencias:

1 . Funciones puras para el procesamiento genómico

Empezamos definiendo algunas funciones puras para tareas comunes en el análisis genómico, como el filtrado de variantes genéticas y el cálculo de frecuencias alélicas:

In [ ]:
```python
def filter_variants(variants, min_depth=10, min_quality=20):
    """Filtra variantes genéticas basadas en profundidad y calidad."""
    return [variant for variant in variants if variant['depth'] >= min_depth and variant

def calculate_allele_frequencies(variants):
    """Calcula las frecuencias alélicas de un conjunto de variantes genéticas."""
    allele_counts = {}
    for variant in variants:
        alleles = variant['alleles']
        for allele in alleles:
            if allele in allele_counts:
                allele_counts[allele] += 1
            else:
                allele_counts[allele] = 1
    total_alleles = sum(allele_counts.values())
    return {allele: count / total_alleles for allele, count in allele_counts.items()}
```

2 . Uso de multiprocessing para procesamiento paralelo de datos

Para manejar el procesamiento de grandes volúmenes de datos, utilizaremos el módulo multiprocessing para distribuir la carga de trabajo entre múltiples procesos:

In [ ]:
```python
from multiprocessing import Pool

def process_genomic_data(data):
    """Procesa datos genómicos en paralelo utilizando múltiples procesos."""
    with Pool(processes=4) as pool:
        results = pool.map(process_sample, data)
    return results

def process_sample(sample):
    """Procesa un único conjunto de datos genómicos."""
    filtered_variants = filter_variants(sample['variants'])
    allele_frequencies = calculate_allele_frequencies(filtered_variants)
    return {'sample_id': sample['id'], 'allele_frequencies': allele_frequencies}
```

3 . Integración con asyncio para entrada/salida asincrónica

Integraremos asyncio para realizar operaciones de E/S de manera eficiente, especialmente útil cuando se trata de leer y escribir grandes archivos de datos genómicos:

In [ ]:
```python
import asyncio

async def load_genomic_data(file_path):
    """Carga datos genómicos de forma asíncrona."""
    # Simulación: en un caso real, se leerían los datos de un archivo
    return [
        {'id': 'sample1', 'variants': [{'depth': 15, 'quality': 30, 'alleles': ['A', 'T'
        {'id': 'sample2', 'variants': [{'depth': 12, 'quality': 22, 'alleles': ['T', 'T'
    ]

async def analyze_genomic_data(file_path):
    """Analiza datos genómicos utilizando funciones asincrónicas y paralelismo."""
    data = await load_genomic_data(file_path)
    results = await asyncio.get_event_loop().run_in_executor(None, process_genomic_data,
    for result in results:
        print(result)
```

```
if __name__ == "__main__":
    asyncio.run(analyze_genomic_data('path_to_genomic_data.txt'))
```

Ejercicio 6: Simulador de mercados financieros en tiempo real

Desarrolla un simulador de mercados financieros que pueda procesar y analizar datos bursátiles en tiempo real, usando programación funcional para las operaciones de cálculo y asyncio para manejar datos de múltiples fuentes.

Descripción:

- Diseña funciones puras para calcular indicadores financieros, como medias móviles, RSI, y bandas de Bollinger.
- Emplea asyncio para recibir datos de mercado en tiempo real de múltiples intercambios y fuentes de información.
- Aplica concurrent.futures.ThreadPoolExecutor para realizar análisis técnico en paralelo y generar señales de trading.

Desafío:

- Crear un pipeline de CI/CD para probar automáticamente la lógica del simulador, realizar backtesting periódico, y desplegar actualizaciones del simulador en un entorno de nube seguro y escalable.

Sugerencias:

1 . Funciones Puras para análisis financiero

Empezamos definiendo algunas funciones puras para realizar cálculos financieros comunes, como el cálculo de la media móvil y el índice de fuerza relativa (RSI):

In [ ]:
```
def calculate_moving_average(prices, window_size=20):
    """Calcula la media móvil simple de los precios."""
    if len(prices) < window_size:
        return None  # No hay suficientes datos para calcular la media móvil
    return sum(prices[-window_size:]) / window_size

def calculate_rsi(prices, periods=14):
    """Calcula el índice de fuerza relativa (RSI) para una lista de precios."""
    if len(prices) < periods:
        return None  # No hay suficientes datos para calcular el RSI

    gains = [max(0, prices[i] - prices[i - 1]) for i in range(1, len(prices))]
    losses = [max(0, prices[i - 1] - prices[i]) for i in range(1, len(prices))]

    average_gain = sum(gains[-periods:]) / periods
    average_loss = sum(losses[-periods:]) / periods

    if average_loss == 0:
        return 100  # Evitar división por cero
    rs = average_gain / average_loss
    rsi = 100 - (100 / (1 + rs))
    return rsi
```

2 . Uso de concurrent.futures para paralelizar el análisis

Utilizaremos ThreadPoolExecutor para procesar y analizar los datos financieros en paralelo:

In [ ]:
```
from concurrent.futures import ThreadPoolExecutor
```

```python
def parallel_analyze_data(stock_data):
    """Analiza datos bursátiles en paralelo."""
    with ThreadPoolExecutor(max_workers=10) as executor:
        results = list(executor.map(analyze_stock, stock_data))
    return results

def analyze_stock(data):
    """Analiza los datos de un solo stock."""
    moving_average = calculate_moving_average(data['prices'])
    rsi = calculate_rsi(data['prices'])
    return {'stock': data['stock'], 'moving_average': moving_average, 'RSI': rsi}
```

3 . Integración con asyncio para manejo asíncrono y streaming

Integraremos asyncio para simular la recepción de datos bursátiles en tiempo real y para manejar los datos asincrónicamente:

In [ ]:
```python
import asyncio

async def stream_stock_data():
    """Simula la recepción de datos bursátiles en tiempo real."""
    example_data = [
        {'stock': 'AAPL', 'prices': [150, 151, 152, 153, 154]},
        {'stock': 'GOOGL', 'prices': [1200, 1202, 1204, 1206, 1208]}
    ]
    while True:
        await asyncio.sleep(1)  # Simular la recepción de datos cada segundo
        processed_data = await asyncio.get_event_loop().run_in_executor(None, parallel_a
        print("Processed Data:", processed_data)

if __name__ == "__main__":
    asyncio.run(stream_stock_data())
```

In [ ]:
```python
## Tus respuestas.
```