Curso de Python Avanzado

Scotiabank | **Belatrix**

Instructor: Alan Badillo Salas

Bienvenida

Bienvenidos al curso de **Python Avanzado** para Scotiabank, brindado por Belatrix.

En este curso aprenderás a desarrollar una programación avanzada con Python.

El curso está dirigido a profesionales relacionados al área de desarrollo, soporte y análisis de datos.

Se requieren conocimientos previos de Python Intermedio para poder cubrir satisfactoriamente este curso.

Temario

Módulo 1: Programación Funcional en Python

1. Introducción a la programación funcional
2. Funciones de orden superior
3. Lambdas y expresiones generadoras

Módulo 2: Diseño de Patrones Avanzados

1. Patrones de diseño comunes en Python
2. Aplicación práctica de patrones en el desarrollo de software

Módulo 3: Manipulación Eficiente de Datos

1. Uso avanzado de NumPy y Pandas
2. Operaciones vectorizadas y broadcasting

Módulo 4: Concurrencia y Paralelismo en Python

1. Hilos y procesos en Python
2. Multiprocessing y Asyncio

Módulo 5: Optimización de Código

1. Estrategias para mejorar el rendimiento
2. Perfilado de código y herramientas de optimización

Módulo 6: Seguridad en Desarrollo Python

1. Principios básicos de seguridad
2. Mejores prácticas de codificación segura

Módulo 7: Despliegue y Escalabilidad

1. Configuración de entornos de producción
2. Estrategias para escalabilidad horizontal y vertical

Módulo 4: Concurrencia y Paralelismo en Python

1. Hilos y procesos en Python
2. Multiprocessing y Asyncio

Los hilos y procesos en Python son esenciales para entender la programación concurrente y paralela, aspectos fundamentales en el desarrollo de aplicaciones eficientes y rápidas.

Hilos (Threads)

Los hilos son una forma de ejecutar múltiples tareas o subprocesos simultáneamente, permitiendo que una aplicación realice múltiples operaciones al mismo tiempo. En Python, el módulo `threading` es una de las maneras de implementar hilos.

Características de los Hilos:

- **Comparten memoria**: Los hilos de un mismo proceso comparten el mismo espacio de memoria, lo que facilita la comunicación entre ellos pero también requiere la gestión de acceso concurrente a recursos compartidos.

- **GIL (Global Interpreter Lock)**: Python tiene un mecanismo llamado GIL que permite que solo un hilo se ejecute en el intérprete en cualquier momento dado. Esto puede ser un limitante cuando se busca obtener beneficios de múltiples núcleos de CPU en operaciones intensivas de CPU.

Ejemplo de uso de `threading`:

```
import threading

def funcion_para_hilo():
    print("Hilo en ejecución")

# Crear un hilo
hilo = threading.Thread(target=funcion_para_hilo)
hilo.start()

# Esperar a que el hilo termine
hilo.join()
print("Hilo finalizado")
```

Procesos

Los procesos son instancias completamente independientes del intérprete de Python. Cada proceso tiene su propio espacio de memoria y su propio intérprete. Esto los hace más robustos en términos de aislamiento y seguridad, pero más costosos en términos de recursos y comunicación.

Características de los Procesos:

- **Memoria independiente**: Cada proceso tiene su propio espacio de memoria.
- **No afectados por el GIL**: Multiprocesamiento en Python permite aprovechar múltiples CPU y núcleos porque cada proceso tiene su propio intérprete de Python y, por lo tanto, su propio GIL.

Ejemplo de uso de `multiprocessing`:

```
from multiprocessing import Process

def funcion_para_proceso():
    print("Proceso en ejecución")

# Crear un proceso
proceso = Process(target=funcion_para_proceso)
proceso.start()

# Esperar a que el proceso termine
proceso.join()
print("Proceso finalizado")
```

Cuándo usar Hilos vs Procesos

- **Uso de Hilos**: Cuando la tarea implica operaciones de E/S o espera activa, y se puede beneficiar de la compartición fácil de información.
- **Uso de Procesos**: Cuando la tarea es intensiva en CPU y se desea evitar el GIL para mejorar el rendimiento utilizando múltiples núcleos de CPU.

En resumen, la elección entre hilos y procesos depende del tipo de tarea que necesitas realizar y de los recursos del sistema. En el contexto de Python, es crucial comprender estos conceptos para diseñar aplicaciones que sean tanto eficientes como efectivas.

Ejemplo del uso de hilos para simular una venta

Vamos a ver cómo implementar un hilo en Python que utilice parámetros para simular una venta de cliente. En este ejemplo, vamos a usar el módulo `threading` para crear un hilo que maneje la información de una venta, como el nombre del cliente y el monto de la venta.

Ejemplo: Simulación de Venta de Cliente con Hilos

Primero, vamos a definir una función que represente la acción de procesar una venta. Esta función recibirá el nombre del cliente y el monto de la venta como parámetros. Luego, crearemos un hilo para cada venta simulada, pasando los argumentos necesarios a la función.

Aquí tienes el código:

```
import threading
import time

def procesar_venta(nombre_cliente, monto_venta):
    print(f"Procesando venta para {nombre_cliente} por un monto de ${monto_venta}")
    # Simulamos algún procesamiento que toma tiempo
    time.sleep(2)
    print(f"Venta procesada para {nombre_cliente}: ${monto_venta}")

# Lista de ventas a procesar
```

```
ventas = [
    ("Alice", 200),
    ("Bob", 150),
    ("Charlie", 300)
]

hilos = []

# Crear un hilo para cada venta
for nombre, monto in ventas:
    hilo = threading.Thread(target=procesar_venta, args=(nombre, monto))
    hilos.append(hilo)
    hilo.start()

# Esperar a que todos los hilos terminen
for hilo in hilos:
    hilo.join()

print("Todas las ventas han sido procesadas.")
```

Explicación del Código:

1. **Definición de la Función:** `procesar_venta` toma dos parámetros: `nombre_cliente` y `monto_venta`. Esta función simplemente imprime un mensaje para simular el procesamiento de una venta.
2. **Simulación de Procesamiento:** Usamos `time.sleep(2)` para simular que el procesamiento de la venta toma algo de tiempo. Esto ayuda a visualizar cómo los hilos operan en paralelo.
3. **Creación de Hilos:** Iteramos sobre la lista de `ventas` y creamos un hilo para cada una. Pasamos `nombre` y `monto` a la función `procesar_venta` a través del argumento `args`.
4. **Inicio de Hilos:** Cada hilo se inicia con `hilo.start()`.
5. **Espera de Hilos:** `hilo.join()` asegura que el programa principal espere a que todos los hilos terminen antes de imprimir que todas las ventas han sido procesadas.

Este enfoque es útil en escenarios reales donde, por ejemplo, podrías necesitar procesar múltiples transacciones de clientes de manera concurrente, optimizando así los tiempos de respuesta en sistemas que manejan un alto volumen de operaciones.

Multiprocessing

En Python, `multiprocessing` es un módulo que permite crear procesos, cada uno con su propio intérprete de Python y espacio de memoria. Esto significa que puedes realizar computación paralela en Python, superando el Global Interpreter Lock (GIL) que limita la ejecución simultánea de múltiples hilos en el mismo intérprete.

Características de `multiprocessing`:

- **Paralelismo real**: A diferencia de los hilos, que son limitados por el GIL, `multiprocessing` permite que cada proceso ejecute tareas en paralelo en diferentes núcleos del procesador.
- **Seguridad de memoria**: Dado que cada proceso tiene su propio espacio de memoria, los procesos no comparten datos a menos que explícitamente se configure para hacerlo (por ejemplo, usando `multiprocessing.Queue` o `multiprocessing.Pipe`).

Uso típico:

`multiprocessing` es especialmente útil para tareas que son intensivas en CPU, como procesamiento de imágenes, operaciones matemáticas pesadas, etc., donde el paralelismo puede significar una mejora significativa en el rendimiento.

Ejemplo de `multiprocessing`:

```
from multiprocessing import Process

def tarea_proceso(num):
    print(f"Proceso {num} ejecutándose")

procesos = []
for i in range(5):  # Crear 5 procesos
    p = Process(target=tarea_proceso, args=(i,))
    procesos.append(p)
    p.start()

for p in procesos:
    p.join()  # Esperar a que todos los procesos terminen

print("Todos los procesos han finalizado.")
```

Asyncio

`asyncio` es un módulo de Python que proporciona un marco para escribir código concurrente utilizando la sintaxis `async` / `await`. Es utilizado principalmente para escribir código asincrónico para operaciones basadas en E/S (como accesos a la web, bases de datos y archivos).

Características de `asyncio`:

- **Asincronía**: Permite que el código se ejecute de manera no bloqueante mediante el uso de corutinas que se suspenden y reanudan en puntos de espera.
- **Gestión de eventos**: Usa un bucle de eventos para manejar todas las operaciones asincrónicas. El bucle de eventos es el núcleo de cada programa asyncio, proporcionando una manera de programar y gestionar un alto nivel de concurrencia.

Uso típico:

`asyncio` es ideal para aplicaciones de red, servidores web, y otras operaciones que son I/O-bound más que CPU-bound, donde la asincronía puede reducir el tiempo de espera y aumentar la eficiencia.

Ejemplo de `asyncio`:

```
import asyncio

async def tarea_asincrona(num):
    print(f"Comenzando tarea {num}")
    await asyncio.sleep(1)  # Simular I/O con sleep
    print(f"Tarea {num} completada")

async def main():
    # Ejecutar tres tareas asincrónicas
    await asyncio.gather(
        tarea_asincrona(1),
        tarea_asincrona(2),
        tarea_asincrona(3)
    )

# Ejecutar el bucle de eventos
asyncio.run(main())
```

Comparación entre `multiprocessing` y `asyncio`

- **Cuándo usar `multiprocessing`**: Cuando necesitas superar el GIL para realizar computación intensiva en CPU que puede beneficiarse de múltiples núcleos.
- **Cuándo usar `asyncio`**: Para aplicaciones que realizan muchas operaciones de E/S y donde la capacidad de manejar muchas conexiones simultáneas es crucial.

Ambas tecnologías tienen su lugar en el ecosistema de Python y se pueden combinar para aprovechar sus respectivas ventajas en aplicaciones complejas.

Ejemplo de la descarga en paralelo con Threading

```
import requests
import time
from threading import Thread

def tarea_hilo(num):
    print(f"Hilo {num} ejecutándose")

    # URL del archivo que quieres descargar
    url = 'https://archive.ics.uci.edu/static/public/360/air+quality.zip'

    # Realizar la petición GET para obtener el contenido del archivo
    respuesta = requests.get(url)

    # Verificar si la petición fue exitosa (código de estado 200)
    if respuesta.status_code == 200:
        # Escribir el contenido descargado en un archivo local
        with open(f'source_{num}.zip', 'wb') as archivo:
            archivo.write(respuesta.content)
        print("¡Descarga exitosa!")
    else:
        print("Error al descargar el archivo:", respuesta.status_code)
    print(f"Hilo {num} terminado")

start = time.time()
hilos = []
for i in range(5):  # Crear 5 procesos
    t = Thread(target=tarea_hilo, args=(i,))
    hilos.append(t)
    t.start()

for t in hilos:
    t.join()  # Esperar a que todos los procesos terminen

end = time.time()

delta = end - start

print(f"Todos los hilos han finalizado en {delta} s")
```

```
Hilo 0 ejecutándose
Hilo 1 ejecutándose
Hilo 2 ejecutándose
Hilo 3 ejecutándose
Hilo 4 ejecutándose
¡Descarga exitosa!
Hilo 0 terminado
¡Descarga exitosa!
Hilo 2 terminado
¡Descarga exitosa!
Hilo 4 terminado
¡Descarga exitosa!
Hilo 3 terminado
¡Descarga exitosa!
Hilo 1 terminado
Todos los hilos han finalizado en 3.634617328643799 s
```

Ejemplo de la descarga en paralelo con Multiprocessing

```
```

```
import requests
import time
from multiprocessing import Process

def tarea_proceso(num):
    print(f"Proceso {num} ejecutándose")

    # URL del archivo que quieres descargar
    url = 'https://archive.ics.uci.edu/static/public/360/air+quality.zip'

    # Realizar la petición GET para obtener el contenido del archivo
    respuesta = requests.get(url)

    # Verificar si la petición fue exitosa (código de estado 200)
    if respuesta.status_code == 200:
        # Escribir el contenido descargado en un archivo local
        with open(f'source_{num}.zip', 'wb') as archivo:
            archivo.write(respuesta.content)
        print("¡Descarga exitosa!")
    else:
        print("Error al descargar el archivo:", respuesta.status_code)
    print(f"Proceso {num} terminado")

start = time.time()
procesos = []
for i in range(5):  # Crear 5 procesos
    p = Process(target=tarea_proceso, args=(i,))
    procesos.append(p)
    p.start()

for p in procesos:
    p.join()  # Esperar a que todos los procesos terminen

end = time.time()

delta = end - start

print(f"Todos los procesos han finalizado en {delta} s")
```

```
Proceso 0 ejecutándose
Proceso 1 ejecutándose
Proceso 2 ejecutándose
Proceso 3 ejecutándose
Proceso 4 ejecutándose
¡Descarga exitosa!
Proceso 2 terminado
¡Descarga exitosa!
Proceso 0 terminado
¡Descarga exitosa!
Proceso 4 terminado
¡Descarga exitosa!
Proceso 3 terminado¡Descarga exitosa!

Proceso 1 terminado
Todos los procesos han finalizado en 2.4124584197998047 s
```

Ejemplo de la descarga en secuencial con Asyncio

```
import requests
import asyncio
import time

async def tarea_asincrona(num):
    print(f"Tarea {num} ejecutándose")

    # URL del archivo que quieres descargar
    url = 'https://archive.ics.uci.edu/static/public/360/air+quality.zip'

    # Realizar la petición GET para obtener el contenido del archivo
    respuesta = requests.get(url)

    # Verificar si la petición fue exitosa (código de estado 200)
    if respuesta.status_code == 200:
        # Escribir el contenido descargado en un archivo local
        with open(f'source_{num}.zip', 'wb') as archivo:
            archivo.write(respuesta.content)
        print("¡Descarga exitosa!")
    else:
        print("Error al descargar el archivo:", respuesta.status_code)
    print(f"Tarea {num} terminada")

async def main():
    start = time.time()
    print("Ejecutando tareas")

    # Ejecutar tareas asincrónicas
    await asyncio.gather(
      tarea_asincrona(1),
      tarea_asincrona(2),
      tarea_asincrona(3),
      tarea_asincrona(4),
      tarea_asincrona(5),
    )

    end = time.time()

    delta = end - start

    print(f"Todas los tareas han finalizado en {delta} s")

# Ejecutar el bucle de eventos
# asyncio.run(main()) # <<< FUERA DE COLAB

await main()
```

```
Ejecutando tareas
Tarea 1 ejecutándose
¡Descarga exitosa!
Tarea 1 terminada
Tarea 2 ejecutándose
¡Descarga exitosa!
Tarea 2 terminada
Tarea 3 ejecutándose
¡Descarga exitosa!
Tarea 3 terminada
Tarea 4 ejecutándose
¡Descarga exitosa!
Tarea 4 terminada
Tarea 5 ejecutándose
¡Descarga exitosa!
Tarea 5 terminada
Todas los tareas han finalizado en 8.887385368347168 s
```

Ejemplo de la descarga en paralelo con Asyncio

```
import asyncio
import time
import aiohttp

async def tarea_asincrona(num):
    print(f"Tarea {num} ejecutándose")

    url = 'https://archive.ics.uci.edu/static/public/360/air+quality.zip'

    async with aiohttp.ClientSession() as session:
        async with session.get(url) as respuesta:
            if respuesta.status == 200:
                # Escribir el contenido descargado en un archivo local
                contenido = await respuesta.read()
                with open(f'source_{num}.zip', 'wb') as archivo:
                    archivo.write(contenido)
                print("¡Descarga exitosa!")
            else:
                print(f"Error al descargar el archivo {num}: {respuesta.status}")

    print(f"Tarea {num} terminada")

async def main():
    start = time.time()
    print("Ejecutando tareas")

    # Ejecutar tareas asincrónicas
    await asyncio.gather(
        tarea_asincrona(1),
        tarea_asincrona(2),
        tarea_asincrona(3),
        tarea_asincrona(4),
        tarea_asincrona(5),
    )

    end = time.time()
```

Ejemplo sin bloqueos en asyncio

```
import
```