Concurrencia en hilos.

Concurrencia.

En el contexto de la programación, la **concurrencia** se refiere a la **capacidad de un programa para realizar múltiples tareas o procesos simultáneamente** en una misma máquina o en varias máquinas en una red.

La concurrencia se logra mediante el uso de técnicas y herramientas como hilos, procesos, subprocesos, programación asíncrona, entre otros, que permiten que los recursos del sistema sean compartidos de manera eficiente y se eviten los cuellos de botella que pueden surgir cuando se realizan múltiples tareas de forma secuencial.

La concurrencia es una característica importante en los sistemas modernos que deben manejar grandes volúmenes de datos y procesar múltiples solicitudes simultáneamente, y es esencial en aplicaciones en tiempo real, sistemas operativos, servidores web y bases de datos distribuidas.

Técnicas de concurrencia.

Existen varios métodos de realizar concurrencia en un programa. Las técnicas más importantes son:

- **Hilos**. Un hilo (**thread**) es una unidad básica de ejecución dentro de un proceso. Un proceso puede tener múltiples hilos que se ejecutan en paralelo y **comparten el mismo espacio de memoria**. Los hilos pueden ser utilizados para realizar varias tareas en un solo proceso y mejorar la eficiencia de la concurrencia.

- **Procesos**. Un proceso es una instancia de un programa en ejecución, que tiene su propio espacio de memoria y recursos del sistema. Los procesos se utilizan a menudo para lograr la separación y la seguridad de los datos y recursos del sistema, y pueden ser ejecutados en paralelo en una máquina o en varias máquinas en una red. El sistema operativo se encarga de gestionarlos.

- **Subproceso**. Un subproceso es una forma de crear un nuevo proceso dependiente de un programa existente. Un subproceso puede ser utilizado para realizar tareas adicionales en paralelo con el proceso principal, y puede ser utilizado para lograr la concurrencia en sistemas multi-core o multi-procesador. Cuando el proceso principal termina, este termina.

- **Ejecución asíncrona**. La ejecución asíncrona es una técnica que permite ejecución de varias tareas en un solo hilo o proceso. Consiste en parar y retomar recurrentemente la ejecución de las tareas para repartir el tiempo de ejecución entre todas. La ejecución asíncrona es útil cuando se desea mejorar la capacidad de respuesta de un programa, ya que permite que la tarea principal continúe ejecutándose sin tener que esperar a que la tarea secundaria se complete. En Python, la ejecución asíncrona se logra mediante el uso de la biblioteca **asyncio**.

Ejecución por hilos (`threading`).

Para crear hilos en Python utilizamos la librería `threading`. Dentro de ella existe la clase `Thread` que representa la ejecución de un hilo del programa. En el constructor debemos proveerle de una función que contendrá la actividad del hilo.

```
import threading

def hola_hilo():
    "Saluda desde un hilo"
    print("¡Estoy en un hilo 🤩!")

# Crear un objeto Thread
hilo = threading.Thread(target=hola_hilo)

# Iniciar el hilo
print("Arrancamos hilo!")
hilo.start()

# Esperar a que el hilo termine
hilo.join()

print("Hilo finalizado")
```

En este ejemplo, creamos una función llamada `hola_hilo` que simplemente imprime un mensaje en la terminal. Luego, creamos un objeto `Thread` utilizando la función `Thread` de la biblioteca `threading` y le asignamos la función `hola_hilo` como su objetivo (`target`).

Después, iniciamos el hilo con el método `start()` del objeto de hilo. El hilo comenzará a ejecutarse y llamará a la función `hola_hilo` en segundo plano.

Finalmente, usamos el método `join()` para esperar a que el hilo termine antes de imprimir "Hilo finalizado" en la terminal. Esto asegura que el programa principal no finalice antes de que el hilo termine de ejecutarse.

En esta ejecución ha habido dos hilos involucrados: el hilo del programa principal y el de la función `hola_hilo`.

> **¡Importante!** Una vez finalizado un hilo, este no puede volver a ser lanzado.

Veamos un ejemplo con dos hilos.

```
import threading
import time
import random

def print_numbers():
    "Imprime los números del 1 al 10 en un tiempo indeterminado."
    for i in range(1, 11):
```

```
            time.sleep(random.randint(1,10)/10) # duerme tiempo aleatorio.
            print(i)

    def print_letters():
        "Imprime las 10 primeras letras en un tiempo indeterminado."
        for letter in 'abcdefghij':
            time.sleep(random.randint(1,10)/10)
            print(letter)

    t1 = threading.Thread(target=print_numbers)
    t2 = threading.Thread(target=print_letters)

    t1.start()
    t2.start()

    t1.join()
    t2.join()

    print("Hecho")
```

En este ejemplo, hemos creado dos funciones: `print_numbers()` y `print_letters()`, cada una de las cuales imprime una serie de números o letras en la consola con una pausa aleatoria entre elementos. Luego, creamos dos objetos de hilo (`t1` y `t2`) utilizando la clase `Thread`, y les asignamos las funciones `print_numbers` y `print_letters` respectivamente.

Después, iniciamos los subprocesos con los métodos `start()` de cada hilo. Para asegurarnos de que ambos subprocesos finalicen antes de que finalice el programa, **usamos el método `join()` para bloquear el programa principal hasta que los subprocesos terminen**.

Por último, imprimimos "Hecho" en la terminal para indicar que los hilos han finalizado correctamente.

Al ejecutar este código, deberías ver que los números y las letras se imprimen en la consola de forma concurrente, lo que indica que los dos hilos funcionan en paralelo.

> La gestión de los hilos puede llegar a ser muy complicada en programas que requieren de alto rendimiento. En este curso de va a tratr la creación y gestión básicas los hilos.

Si queremos pasar argumentos a los hilos simplemente debemos pasarselos en una tupla o diccionario:

```
import threading
import time

def calcula_fibo(n: int):
    # Calcula los enésimos números la serie de Fibonacci y los muestra por
pantalla.
    fibo = [1, 1]
    print('#1', fibo[0])
    print('#2', fibo[1])
    for i in range(2, n):
        fibo.append(fibo[-2]+fibo[-1])
        time.sleep(0.2)
```

```
        print(f'#{i+1}', fibo[-1])

hilo = threading.Thread(target=calcula_fibo, args=(100,))
hilo.start()
hilo.join()
```

Hilos como clases.

Una forma más interesante de crear hilos es mediante la creación de subclases de `Thread`. Mediante el uso de herencia obtendremos un clase con todas las funcionalidades de `Thread` pero con los atributos adicionales que necesitemos. Solo tenemos que sobrecargar (es decir, volver a definir) el método `__init__()` y un método llamado `run()` que se corresponde con la actividad del hilo.

```
import threading
import random
import time

class Spammer(threading.Thread):
    "Representa un hilo que spammea mensajes por consola."

    def __init__(self, mensaje: str, nombre_hilo: str|None = None):
        self.mensaje = mensaje
        self.stop = False
        super().__init__(name=nombre_hilo)

    def run(self) -> None:
        "Representa la actividad del hilo."
        while not self.stop:
            print(self.mensaje)
            tiempo_espera = random.randint(1,5)
            time.sleep(tiempo_espera)

# Creamos los hilos.
sp1 = Spammer('¡Rebajas en bitcoin!')
sp2 = Spammer('¡Vota! ¡Mariano para presidente!')
sp3 = Spammer('¡Pierde peso comiendo hamburguesas! Haz click aquí.')

# Iniciamos los hilos.
sp1.start()
sp2.start()
sp3.start()

# Esperamos 30 segundos
time.sleep(30)

# Modificamos el miembro "stop" para finalizar
sp1.stop = True
sp2.stop = True
sp3.stop = True

# Esperamos a que los hilos finalicen
```

```
sp1.join()
sp2.join()
sp3.join()

print('Fin del programa.')
```

En este ejemplo se define la clase `Spammer` que hereda de `Thread` sus métodos y atributos. La clase representa un hilo que imprime un mensaje en la consola de forma repetida hasta que se le indica que pare. La clase tiene dos atributos: `mensaje` que contiene el mensaje que se imprimirá y `stop` que se utiliza para indicar al hilo que debe detenerse.

El método `__init__()` inicializa el objeto `Spammer` con un mensaje y un nombre de hilo opcional. El método `run()` es el método principal de la clase que define la actividad del hilo. El método `run()` imprime el mensaje almacenado en el atributo `mensaje` y luego espera un tiempo aleatorio entre 1 y 5 segundos antes de imprimir el mensaje nuevamente. El ciclo se repetirá hasta que se establezca el atributo `stop` como True.

Luego, se crean tres objetos `Spammer` con diferentes mensajes, y se inician los hilos con el método `start()`.

Finalmente, el programa espera a que los hilos terminen con el método `join()` y luego imprime "Fin del programa." en la terminal.

Crear una clase derivada en vez de hacer instancias de `Thread` tiene las siguientes ventajas:

- **Permite definir mejor la actividad**. Mediante métodos auxiliares y atributos de clase tenemos más recursos para definir la actividad del hilo.
- **Da un punto de entrada para modificar la ejecución del hilo.** Mediante atributos podemos modificar el comportamiento del hilo.
- **Permite recolectar la salida de forma segura**. Mediante el uso de atributos podemos recoger el resultado de la actividad del hilo.

> **¡Recuerda! Los hilos comparten memoria entre ellos en el mismo proceso**. Esto puede ser problemático: cuando dos hilos acceden al mismo objeto pueden generarse situaciones indeseadas. La solución a esto pasa por sincronizar la actividad de los hilos. La sincronización se explica con detalle en el documento S7.2.

Grupo de hilos.

Los hilos no tienen porqué estar siempre referenciados en una varible. Podemos crear instancias de `Thread` o nuestras subclases e introducirlos para agruparlos en, por ejemplo, en una lista:

```
pool = [threading.Thread(target=funcion) for _ in range(10)]
```

En el ejemplo de arriba se crean 10 hilos con el target `funcion`. Si queremos lanzarlos todos:

```

```

```
for thread in pool:
    thread.start()
```

Y para esperar a que acaben:

```
for thread in pool:
    thread.join()
```

Ejercicio de clase.

Crea un programa que simule una carrera de caballos RPG.

- El programa debe crear varios hilos, **cada uno representando a un caballo**, y cada hilo debe ejecutar una función que haga mover al caballo una distancia aleatoria en cada iteración. Los caballos se moverán una distancia `x` metros cada `t` tiempo. Las fórmulas de ambas variables son:
  - x = random.gauss(fuerza, suerte)
  - t = random.gauss(resistencia, suerte)
- Los parámetros `fuerza`, `resistencia` y `suerte` son atributos de cada caballo.
- Se debe implementar una clase `Hipodromo` con al menos los atributos/métodos:
  - `caballos`: una lista con los **caballos** que correran.
  - `metros_pista`: lo larga que es la pista.
  - `ready()`: prepara los caballos y muestra un informe.
  - `start()`: empieza la carrera y espera a que terminen los caballos. Si no ha hecho antes `ready`, debe dar fallo.
- El programa debe imprimir en la consola el progreso de la carrera en tiempo real, mostrando la posición actual de cada caballo en cada iteración.
- El programa debe detener la carrera cuando todos los caballos lleguen a la meta, y debe imprimir el resultado final de la carrera indicando la clasificación y su tiempo de carrera.

> La función `time.sleep()` simula el tiempo que tarda cada caballo en moverse.