**Instituto Tecnológico de Colima**

**Maestría en Sistemas Computacionales**

**Unidad II Concurrencia de Hilos**

***Práctica Concurrencia en Python***

**Introducción**

Python incluye herramientas sofisticadas para el manejo de operaciones concurrentes a través de la utilización de procesos e hilos.

El módulo **threading** incluye un API de alto nivel orientada a objetos para el trabajo con concurrencia en Python. Los objetos **Thread** corren concurrentemente dentro del mismo proceso y comparten la memoria. La utilización de hilos es una manera sencilla para escalar tareas que están más vinculadas con operaciones de I/O que de CPU.

Ejercicio No. 1 Objetos Thread

La manera más sencilla para utilizar un objeto **Thread**, es instanciar éste con una función objetivo y llamar el método **start()** para permitir que éste comience a trabajar. Realice en Python el siguiente script y conteste a las preguntas posteriores a esta sección:

**import** threading  
  
  
**def** trabajador():  
 *"""funcion del hilo"""* print(**'Trabajador'**)  
  
  
hilos = []  
**for** i **in** range(5):  
 h = threading.Thread(target=trabajador)  
 hilos.append(h)  
 h.start()

Es útil poder generar un hilo y pasarle argumentos que le indiquen qué hacer; para esto, cualquier tipo de objeto puede ser pasado al hilo. Por ejemplo, el siguiente código pasa un número que luego imprime el hilo.

**import** threading  
  
  
**def** trabajador(num):  
 *"""funcion del hilo"""* print(**'Trabajador: %s'** % num)  
  
  
hilos = []  
**for** i **in** range(5):  
 h = threading.Thread(target=trabajador, args=(i,))  
 hilos.append(h)  
 h.start()

¿Cuál es la diferencia entre los dos scripts anteriores?

La diferencia que existe en los scripts es que en el primero solo se muestra el nombre del trabajador, mientras que en el segundo se muestra el nombre y su número en el que se encuentra.

¿Qué pasa si retira la “,” de los argumentos (args) que se pasan al método trabajador cuándo se crea la instancia del hilo, explique por qué y fundamente su respuesta?

Al quitar la “,” de los argumentos (args) aparece el siguiente error, a lo que se entiende y se observa es que el hilo no puede correr correctamente ya que al ser un entero no puede realizar correctamente el for, para que corra correctamente el hilo debe ser un iterable.

¿Cuál es la ventaja en crear un hilo pasando como argumento el método que ejecutará la funcionalidad del hilo y crear un hilo a través de la creación de una subclase de la clase Thread que utiliza el método run() para ejecutar la funcionalidad del hilo?

Que en una se comparte el mismo espacio de datos que el hilo principal y se tiene acceso a la misma información., al crear una creación de una subclase de la clase Thread que utiliza el método run() se obtiene como resultado la posibilidad de tener dos programas para un mismo código.

Ejercicio No. 2 Determinando el hilo actual.

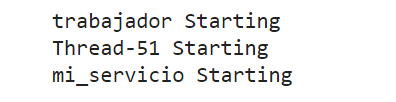
Usar argumentos para identificar y nombrar un hilo es innecesario, ya que cada instancia **Thread** tiene un nombre con un valor por defecto que puede ser cambiado cuando el hilo es creado. Nombrar hilos es útil en procesos servidor en dónde múltiples hilos de servicio manejan diferentes operaciones.

**import** threading  
**import** time  
  
  
**def** trabajador():  
 print(threading.current\_thread().getName(), **'Starting'**)  
 time.sleep(0.2)  
 print(threading.current\_thread().getName(), **'Exiting'**)

**def** mi\_servicio():  
 print(threading.current\_thread().getName(), **'Starting'**)  
 time.sleep(0.3)  
 print(threading.current\_thread().getName(), **'Exiting'**)  
  
  
s = threading.Thread(name=**'mi\_servicio'**, target=mi\_servicio)  
h = threading.Thread(name=**'trabajador'**, target=trabajador)  
h2 = threading.Thread(target=trabajador)  
  
h.start()  
h2.start()  
s.start()

¿Cuál es el resultado del script anterior? y ¿Por qué aun cuando se declaró el nombre del hilo 2 aparece Thread-1?. Fundamente su respuesta.

En base a lo observado primero no se le asigna un name al hilo dos por lo que en base a la documentación el Thread lo que está regresando es el objeto actual o el proceso que se está ejecutando, ya que al hacer más pruebas el número aumenta en 1 entonces se deduce que se imprime lo siguiente Thread + el número del proceso que se ejecutó o está ejecutando.



Muchos programas no usan el **print** para depuración; por lo cual, el módulo **logging** soporta incluir el nombre del hilo en cada mensaje log usando el **%(threadName)s** formateador de código. Incluir el nombre de los hilos en los mensajes log hace posible rastrear hacia atrás el origen de los mensajes.

**import** threading  
**import** time  
**import** logging  
  
  
**def** trabajador():  
 logging.debug(**"Starting"**)  
 time.sleep(0.2)  
 logging.debug(**"Exiting"**)  
  
  
**def** mi\_servicio():  
 logging.debug(**"Starting"**)  
 time.sleep(0.3)  
 logging.debug(**"Exiting"**)  
  
  
logging.basicConfig(  
 level=logging.DEBUG,  
 format=**'[%(levelname)s] (%(threadName)-10s) %(message)s'**,  
)  
  
s = threading.Thread(name=**'mi\_servicio'**, target=mi\_servicio)  
h = threading.Thread(name=**'trabajador'**, target=trabajador)  
h2 = threading.Thread(target=trabajador)  
  
h.start()  
h2.start()  
s.start()

¿Cómo funciona el formateador definido en logging.basicConfig?.

Nota: puede recurrir a la siguiente liga para explicar la respuesta a esta pregunta <https://docs.python.org/2.4/lib/node357.html>

Los formateadores son responsables de convertir un LogRecord en una cadena que puede ser interpretada por un ser humano o un sistema externo, además de permitir especificar una cadena de formato en base de no existir se utiliza el valor predeterminado del mensaje. Logginng se utiliza para un conjunto de funciones convenientes para un sencillo registro.

Ejercicio No. 3 Hilos tipo demonio

Hasta ahora todos los programas han implícitamente esperado a terminar hasta que todos los hilos han completado su trabajo. Algunas veces, sin embargo, los programas generan un hilo como un ***demonio*** que corre sin bloquear el programa principal de su salida. Los hilos tipo demonio, son útiles para servicios puede que no haya una manera fácil de interrumpir el hilo, o donde dejar que el hilo muera en medio de su trabajo no conduce a la pérdida o corrupción de datos. Para marcar el hilo como un demonio, se debe pasar el parámetro **daemon = True** cuando se construye o llamar su método **set\_daemon()** con **True**.

**import** threading  
**import** time  
**import** logging  
  
  
**def** demonio():  
 logging.debug(**'Starting'**)  
 time.sleep(0.2)  
 logging.debug(**'Exiting'**)  
  
  
**def** no\_demonio():  
 logging.debug(**'Starting'**)  
 logging.debug(**'Exiting'**)

logging.basicConfig(  
 level=logging.DEBUG,  
 format=**'(%(threadName)-10s) %(message)s'**,  
)  
  
d = threading.Thread(name=**'daemon'**, target=demonio, daemon=**True**)  
h = threading.Thread(name=**'non-daemon'**, target=no\_demonio)

d.start()  
h.start()

¿Por qué la salida del script anterior no incluye el mensaje “Exiting” del hilo demonio?

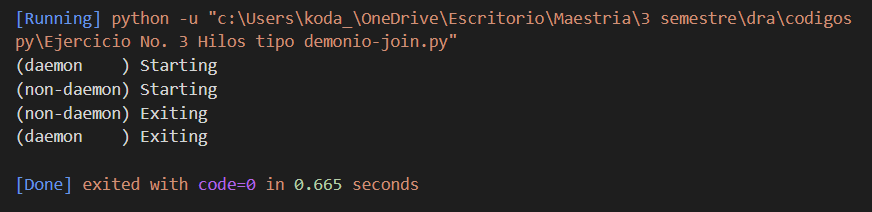
El motivo por el cual no se muestra el resultado se debe a que en al momento de definir la función del demonio se incluye un time.sleep de dos segundos, durante esa condicionante se debería de ejecutar la impresión del “Exiting” pero cómo entra en el rango de la condicionante no se muestra, además de que en el hilo se está marcando como un demonio lo cual igual influye al momento de imprimir los resultados.

Modifique el script anterior para utilizar el método **join().**

**import** threading  
**import** time  
**import** logging  
  
  
**def** demonio():  
 logging.debug(**'Starting'**)  
 time.sleep(0.2)  
 logging.debug(**'Exiting'**)  
  
  
**def** no\_demonio():  
 logging.debug(**'Starting'**)  
 logging.debug(**'Exiting'**)  
  
  
logging.basicConfig(  
 level=logging.DEBUG,  
 format=**'(%(threadName)-10s) %(message)s'**,  
)  
  
d = threading.Thread(name=**'daemon'**, target=demonio, daemon=**True**)  
h = threading.Thread(name=**'non-daemon'**, target=no\_demonio)  
d.start()  
h.start()  
  
d.join()  
h.join()

¿Qué diferencia hay en la salida del script anterior?. Fundamente su respuesta.

La diferencia que existe entre los dos es que en este las salidas primero imprime el inicio del dominio después continúa con las salidas del no demonio las cuales son el inicio y la salida y por último objeto es la salida del demonio.

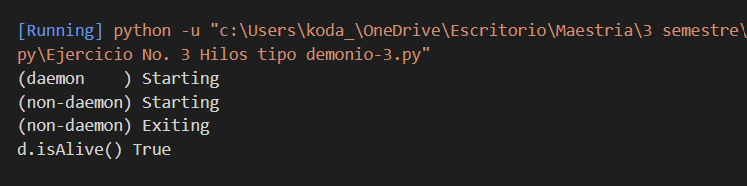


Alternativamente, un valor flotante puede ser pasado para representar el número de segundos para esperar a que el hilo se convierta en inactivo

**import** threading  
**import** time  
**import** logging  
  
  
**def** demonio():  
 logging.debug(**'Starting'**)  
 time.sleep(0.2)  
 logging.debug(**'Exiting'**)  
  
  
**def** no\_demonio():  
 logging.debug(**'Starting'**)  
 logging.debug(**'Exiting'**)  
  
  
logging.basicConfig(  
 level=logging.DEBUG,  
 format=**'(%(threadName)-10s) %(message)s'**,  
)  
  
d = threading.Thread(name=**'daemon'**, target=demonio, daemon=**True**)  
h = threading.Thread(name=**'non-daemon'**, target=no\_demonio)  
d.start()  
h.start()  
  
d.join(0.1)  
print(**'d.isAlive()'**, d.is\_alive())  
h.join()

¿Cuál es el resultado del proceso anterior?, ¿Qué sucedió y por qué?.

En este proceso se imprime como salida el inicio del demonio, como siguiente el inicio y el término del demonio posteriormente imprime la salida d.isAlive() True y ahi termina, ya no imprime lo que es la finalización del demonio. Lo último que imprime hace referencia a que el método isAlive() trabaja en paralelo con las etapas del ciclo de vida de un subproceso, prácticamente este imprime si el hilo en que se está trabajando está vivo o no.



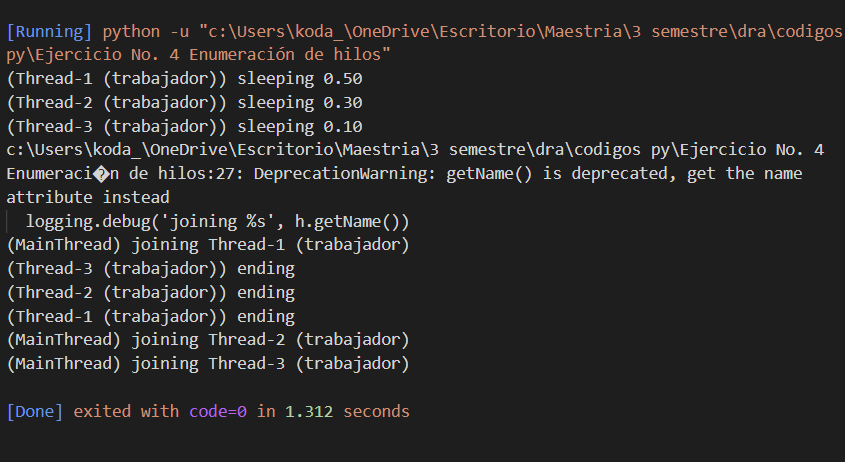
Ejercicio No. 4 Enumeración de hilos.

No es necesario mantener un manejador explicito para todos los hilos de tipo demonio para asegurar que han terminado después de que el proceso principal sale; para esto, el método enumerate() regresa un lista de instancias activas de Thread.

**import** logging  
  
**def** trabajador():  
 *"""funcion del trabajador"""* pausa = random.randint(1, 5) / 10  
 logging.debug(**'sleeping %0.2f'**, pausa)  
 time.sleep(pausa)  
 logging.debug(**'ending'**)  
  
  
logging.basicConfig(  
 level=logging.DEBUG,  
 format=**'(%(threadName)-10s) %(message)s'**,  
)  
  
**for** i **in** range(3):  
 h = threading.Thread(target=trabajador, daemon=**True**)  
 h.start()  
  
hilo\_principal = threading.main\_thread()  
**for** h **in** threading.enumerate():  
 **if** h **is** hilo\_principal:  
 **continue** logging.debug(**'joining %s'**, h.getName())  
 h.join()

¿Explique brevemente el funcionamiento del script anterior? y ¿por qué se introduce un **continue** al comprobar sí es el hilo principal?

El resultado del ejercicio anterior es el siguiente:



En la imagen anterior primero se puede observar que lo primero que se imprime es el número de proceso o hilo que se está trabajando además del nombre de la variable definida que es trabajador y el tiempo que se duerme o en el que el proceso está detenido. Lo siguiente que se muestra es el número de hilos que se están ejecutando. La siguiente salida que se muestra es la impresión del for que da como resultado las 3 iteraciones que se le piden además de la clase que se crea y por último se imprime que el proceso ha terminado.

Por último se realiza un for donde se imprime el subproceso que se hace pero enumerado, posteriormente se pone un if y la indicación continue que al momento que se cumple con el if la iteración actual del bucle se interrumpe y el proceso que se cumple vuelve a la parte superior del bucle y después se termina el for que se inició en un principio.

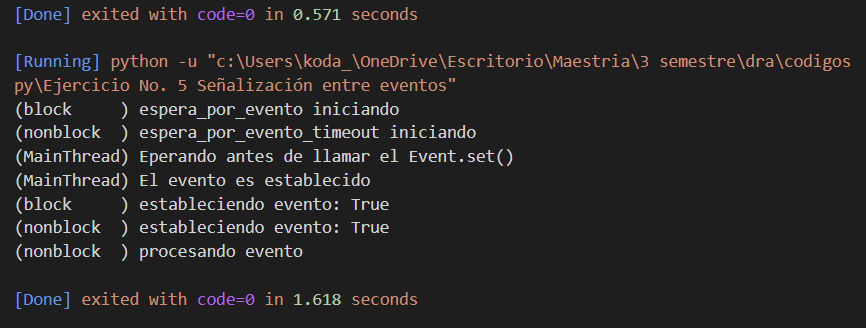
Ejercicio No. 5 Señalización entre eventos.

Aun cuando el punto de utilizar múltiples hilos es correr operaciones separadas concurrentemente, algunas veces es importante ser capaz de sincronizar las operaciones en uno o más hilos. Los objetos **Event** son una manera sencilla de comunicarse de manera segura entre procesos. Un **Event** maneja una señal interna que los llamantes pueden controlar con los métodos **set()** y **clear()**. Otros hilos pueden utilizar **wait()** para pausar hasta que la bandera es fijada, bloqueando efectivamente el progreso hasta que a aquellos hilos se les permite continuar.

**import** logging  
**import** threading  
**import** time  
  
  
**def** espera\_por\_evento(e):  
 *"""Wait for the event to be set before doing anything"""* logging.debug(**'espera\_por\_evento iniciando'**)  
 estableciendo\_evento = e.wait()  
 logging.debug(**'estableciendo evento: %s'**, estableciendo\_evento)  
  
  
**def** espera\_por\_evento\_timeout(e, t):  
 *"""Wait t seconds and then timeout"""* **while not** e.is\_set():  
 logging.debug(**'espera\_por\_evento\_timeout iniciando'**)  
 estableciendo\_evento = e.wait(t)  
 logging.debug(**'estableciendo evento: %s'**, estableciendo\_evento)  
 **if** estableciendo\_evento:  
 logging.debug(**'procesando evento'**)  
 **else**:  
 logging.debug(**'haciendo otro trabajo'**)  
  
  
logging.basicConfig(  
 level=logging.DEBUG,  
 format=**'(%(threadName)-10s) %(message)s'**,  
)  
  
e = threading.Event()  
  
t1 = threading.Thread(name=**'block'**,target=espera\_por\_evento,args=(e,),)  
t1.start()  
  
t2 = threading.Thread(name=**'nonblock'**,target=espera\_por\_evento\_timeout,args=(e, 2),)  
t2.start()  
  
logging.debug(**'Eperando antes de llamar el Event.set()'**)  
time.sleep(0.3)  
e.set()  
logging.debug(**'El evento es establecido'**)

¿Cuál es el funcionamiento del método wait() en el script anterior? ¿Cuál es el resultado del script?, ¿Cuál es la diferencia entre espera\_por\_evento\_timeout() y epsera\_por\_evento?

El método wait se emplea para detener la ejecución de los hilos hasta que la propiedad alcance el valor de verdadero, entonces el método devuelve el valor que tenga la propiedad. El resultado del script anterior es el siguiente.



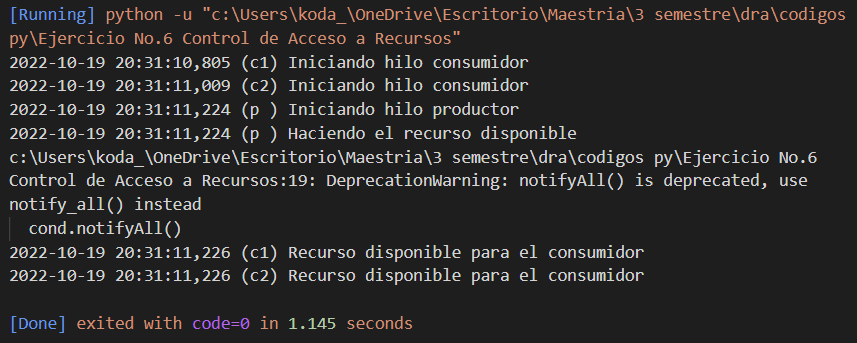
La diferencia entre espera\_por\_evento\_timeout() y epsera\_por\_evento es que uno tiene un while el cual es una sentencia de control de flujo que hace que un bloque de instrucciones se ejecuten de forma continua siempre que se cumpla una condición determinada, las cuales están definidas posteriormente por un if y un else.

**Ejercicio No. Control de Acceso a Recursos.**

En adición a los eventos otra manera de sincronizar hilos es a través del uso de condiciones a través de un objeto **Condition**. Debido a que el objeto **Condition** usa un **Lock**, este puede ser ligado a un recurso compartido, permitiendo que múltiples hilos esperen por el recurso a ser actualizado. En el siguiente ejemplo el hilo **consumidor()** espera que se establezca la condición antes de continuar. El hilo **productor()** es responsable de establecer la condición y notificar a los otros hilos que ellos pueden continuar.

**import** logging  
**import** threading  
**import** time  
  
  
**def** consumidor(cond):  
 *"""espera por la condición y utilizar el recurso"""* logging.debug(**'Iniciando hilo consumidor'**)  
 **with** cond:  
 cond.wait()  
 logging.debug(**'Recurso disponible para el consumidor'**)  
  
  
**def** productor(cond):  
 *"""coloca el recurso a ser utilizado por el consumidor"""* logging.debug(**'Iniciando hilo productor'**)  
 **with** cond:  
 logging.debug(**'Haciendo el recurso disponible'**)  
 cond.notifyAll()  
  
  
logging.basicConfig(  
 level=logging.DEBUG,  
 format=**'%(asctime)s (%(threadName)-2s) %(message)s'**,  
)  
  
condicion = threading.Condition()  
c1 = threading.Thread(name=**'c1'**, target=consumidor,args=(condicion,))  
c2 = threading.Thread(name=**'c2'**, target=consumidor,args=(condicion,))  
p = threading.Thread(name=**'p'**, target=productor,args=(condicion,))  
  
c1.start()  
time.sleep(0.2)  
c2.start()  
time.sleep(0.2)  
p.start()

¿Cuál es el resultado del script anterior? ¿Explique el funcionamiento del with ya que este adquiere el Lock asociado al Condition?. Fundamente su respuesta.



En adición a la sincronización de operaciones de los hilos, es importante ser capaz de controlar el acceso a recursos compartidos para prevenir la corrupción o pérdida de los datos. Los tipos de datos de Python como listas, diccionarios son atómicos por lo cual un solo hilo a la vez puede acceder a ellos. Otros tipos de datos simples implementados en Python como enteros o flotantes no tiene esta protección. Para proteger el acceso simultáneo a un objeto se utiliza el objeto **Lock** , con la finalidad de lograr la ***exclusión mutua***.

**import** logging  
**import** random  
**import** threading  
**import** time  
  
  
**class** Contador:  
 **def** \_\_init\_\_(self, start=0):  
 self.lock = threading.Lock()  
 self.valor = start  
  
 **def** incrementar(self):  
 logging.debug(**'Esperando por lock'**)  
 self.lock.acquire()  
  
 **try**:  
 logging.debug(**'Lock adquirido'**)  
 self.valor = self.valor + 1  
 **finally**:  
 logging.debug(**'Lock liberado'**)  
 self.lock.release()  
  
  
**def** trabajador(c):  
 **for** i **in** range(2):  
 pausa = random.random()  
 logging.debug(**'Durmiendo %0.02f'**, pausa)  
 time.sleep(pausa)  
 c.incrementar()  
 logging.debug(**'Done'**)  
  
  
logging.basicConfig(  
 level=logging.DEBUG,  
 format=**'(%(threadName)-10s) %(message)s'**,  
)  
  
contador = Contador()  
**for** i **in** range(2):  
 t = threading.Thread(target=trabajador, args=(contador,))  
 t.start()  
  
logging.debug(**'Esperando por hilos trabajador '**)  
main\_thread = threading.main\_thread()  
  
**for** t **in** threading.enumerate():  
 **if** t **is not** main\_thread:  
 t.join()  
logging.debug(**'Contador: %d'**, contador.valor)

¿Cómo funciona el script anterior?, ¿Qué realiza el hilo trabajador? y ¿Qué función realiza la clase Counter?.

El ejercicio anterior funciona con el inicio de la función trabajador la cual realiza un for con un rango de dos trabajadores posteriormente el hilo se queda en espera de los hilos trabajador, después las salidas que se muestran son las del hilo trabajador 1 que espera, adquiere y libera el lock que tienen, después de eso se pone en dormir el hilo, el trabajador numero 2 hace el mismo proceso y por ultimo se corre el hilo trabajador el cual realiza un proceso anterior a las salidas de antes pero con la diferencia de que en lugar de dormir solo imprime la palabra done. La clase Counter hace un for en un rango de dos con el objetivo trabajador, pero con los argumentos del contador.

