SEGURIDAD, PROCESOS Y ZÓCALOS

AA2. PROGRAMACIÓN DE HILOS

Boj Cobos, Daniel
DAM
05/2023

AA2. PROGRAMACIÓN DE HILOS

Teoría previa

Antes de pasar a la parte práctica de la actividad, es importante presentar los aspectos clave que necesitamos dominar a nivel teórico.

La programación multihilo permite aprovechar los procesadores modernos y preparar nuestros programas para que ejecuten varias tareas de forma simultánea aprovechando los diversos ***threads*** de un procesador actual.

Multiproceso y multihilo

- Multiproceso es la ejecución de diversos procesos de forma simultánea.
- Multihilo es la ejecución de diversas tarreas, o hilos, dentro de un mismo proceso.

Hay que puntualizar que los procesadores actuales suelen ser de varios núcleos, con lo que permiten el multiproceso, y que **cada núcleo es multihilo**.

Tipos de hilo

- **ULT** → User Level Threads: Suceden en el espacio del usuario en un único proceso.
- **KLT** → Kernel Lebel Threads: Suceden en el espacio del *kernel* que es el encargado de gestionarlos y planificarlos dentro del procesador.

Diferencias

Procesos	Hilos
Necesitan más recursos	Son procesos ligeros
No comparten memoria	Comparten memoria
No necesitan sincronización de memoria	Necesitan sincronización de memoria
Pueden existir individualmente	Siempre están ligados a un proceso
Si un proceso desaparece, todos sus hilos mueren con él	Cuando expira un hilo, su pila se puede recuperar, porque cada hilo tiene su propia pila
Pueden existir en diferentes CPU	Deben permanecer en la misma CPU

Conceptos clave

Los procesos e hilos se pueden distribuir en distintas CPU o, dentro de la misma, en distintos núcleos, ya que solo compartirán la memoria, en el primer caso, o la CPU y el caché en el segundo.

La comunicación entre procesos se define a través de un estado inicial y unos resultados de procesamiento.

Siempre existe un proceso principal que coordina al resto.

Programación multihilo en Python

Thread

Permiten ejecutar múltiples operaciones concurrentes en el mismo espacio de proceso.

- Para poderlo usar hay que importar la librería ***threading***:

import threading

- Mediante la clase **thread** creamos un hilo y mediante el método ***start*** lo iniciamos.

```
import threading

def worker:
        print("esto es un thread")
        return

for i in range(3):
        t = threading.Thread(target = worker)
        t.start
```

Instrucciones y métodos

- **threading.Thread** → Crea un hilo y se le asigna una función.
- **hilo.start** → Inicia la ejecución del hilo.
- **hilo.is_alive()** → Retorna el estado del **thread** mediante un booleano y saber si se ha ejecutado o no. Es decir, tras ejecutar la intrucción start, el valor del **thread** pasa a ser true.
- **hilo.sleep(n)** → Indica que se realice una pausa del tiempo indicado a través del argumento.
- Como toda función, las funciones ejecutadas por un **thread** pueden recibir argumentos, para ello tenemos que usar el *def args*

```
t = threading.Start(target = my_function, args = (arg1, arg2,..., argn))
```

- Podemos encadenar la creación de varios argumentos.

Non-Daemon

Por defecto, los *threads* son del tipo non-daemon, **si el proceso finaliza, se espera a que todos sus *threads* hayan concluido**.

Es decir, si el *thread* principal finaliza, espera a que finalicen el resto de *thread* para cerrarse.

```
import threading
import time

def Worker_A:
        print("Starting A")
        time.sleep(2)
        print("Finishing A")

def Worker_B:
        print("Starting B")
        print("Finishing B")

a = threading.Thread(target = Worker_A)
b = threading.Thread(targe = Worker_B)
a.start()
b.start()
```

A no finaliza hasta que no acaba de ejecutarse B.

Daemon

Si declaramos un *thread* como tipo Daemon, cuando el flujo del programa finaliza y se abortan los *threads* de este tipo aunque estén en curso.

```
a = threading.Thread(target = Worker_A)
a.daemon = True
b = threading.Thread(targe = Worker_B)
a.start()
b.start()
```

A no finaliza hasta que no se cierra el programa. El programa no devuelve el texto de finalización.

Ejecución secuencial de hilos

El método join **permite ejecutar un *thread* de manera secuencial** y evitar que se lancen de forma concurrente.

```
a = threading.Thread(target = Worker_A)
a.start()
a.join()
```

```
b = threading.Thread(targe = Worker_B)
b.start()
```

Variables compartidas

Una variable compartida por varios multiprocesos puede ver su valor alterado dependiendo de los tiempos y orden de ejecución de los hilos o procesos. Así, **un valor puede variar en diferentes ejecuciones del script.**

Para mantener el valor de la variable bloqueado para un solo *thread* hay que usar los comando *lock* y *release*.

- lock → Bloquea los valores de una variable hasta la finalización del proceso o hilo.
- release → Libera la variable para que pueda cambiar su estado en otros procesos o hilos.

```
lock.acquire()
lock.release()
```

Análisis del programa sin modificar

Ahora que se han presentado los conocimientos básicos para entender cómo funcionan los procesos multihilo y cómo los podemos usar en nuestro código; pasamos a analizar el programa de ejemplo antes de realizar cualquier modificación.

Visión general

El análisis a vista general de programa nos deja claro qué es lo que hace, cómo lo hace y cuál va a ser el problema al pasar a una codificación multihilo.

- **Propósito:** El programa imita un proceso secuencial mediante una serie de funciones.
- **Flujo de ejecución:** Cada una de las funciones es llamada de forma secuencial siguiendo el orden de flujo esperado, pero vemos que en ellos bloques de codificación de cada una se utilizan funciones para simular un tiempo de ejecución en concreto.
- **Comportamiento imperativo:** Aplicando el programa sin modificaciones, como sigue un principio imperativo, aunque cada función tenga un tiempo de ejecución concreto, no se ejecutará hasta que no haya finalizado la llamada de la primera. Con ello, si ejecutamos el programa tal cuál, sin modificar, parece que funciona bien, pero **no aplica la programación multihilo**, que es nuestro objetivo, con lo que las tareas no se ejecutarán de forma simultánea si no una a una.
- **Comportamiento multihilo:** Cuando refactoricemos el programa para que implemente una ejecución multihilo nos encontraremos el problema, **las funciones se ejecutarán simultáneamente y se resolverán de forma desordenada**, así que los resultados no aparecerá por pantalla en el orden esperado. En nuestra solución, no solo deberemos codificar las funciones para que se ejecutan de forma simultánea, si no para que lo hagan de forma secuencial y ordenada.

- **¿Por qué?** Es muy probable que, en escenarios reales, nos topemos en situaciones en las que necesitamos que, aunque haya tareas ejecutándose de manera simultánea, unas modifiquen el estado de nuestro programa antes que otras o en un orden establecido. Este programa simula de forma gráfica un caso de este tipo. Ante estas necesidades, podemos optar por la secuencialización de nuestro flujo de tareas síncronas o por bloquear variables para que no puedan ser modificadas por una función paralela. En este caso, lo más idóneo será controlar la secuencialización de nuestras tareas.
- **Variable de control de tiempo de ejecución:** Nos sirve para poder observar, según los cambios de código que vayamos aplicando, si las funciones se están ejecutando de forma simultánea o si estamos aplicando algún tipo de secuenciación.

Análisis punto a punto

```
# Importamos las librerías necesarias para el funcionamiento del programa
# Importamos la librería threading para poder crear hilos
import threading
# La librería time nos permite trabajar con tipos de datos relacionados con el tiempo
import time
# La librería datetime nos permite trabajar con fechas y horas
import datetime

# En el primer bloque del programa definimos las funciones que vamos a utilizar, en
este caso, las funciones que van a ejecutar los hilos en las futuras modificaciones
del programa. Vemos que las funciones están ordenadas de forma secuencial, pero que su
duración significa que se van a ejecutar de forma desordenada cuando se ejecuten los
hilos.

# Esta función es la primera en ejecutarse y deberá finalizar antes de poder continuar
con la siguiente. Simula la búsqueda de la llave.
def BuscarLlave():
    # Vemos como la función se ejecuta en 0.9 segundos, analizando el resto de
funciones, entendemos que el tiempo de ejecución de esta función es el que más tarda
en ejecutarse y que, dentro de una ejecución por hilos, finalizará la última, cuando
debería ser la primera.
    time.sleep(0.9)
    # Mostramos el resultado de la función por pantalla.
    print('1.Busco la llave.')

# Esta función es la segunda en ejecutarse y deberá finalizar antes de poder continuar
con la siguiente. Simula el momento en el que se encuentra la llave.
def EncontrarLlave():
    # Vemos como la función se ejecuta en 0,7 segundos, esto significa que acabaría de
ejecutarse antes que la función anterior, cuando debería ser al revés.
    time.sleep(0.7)
    # Mostramos el resultado por pantalla.
    print('2.Ya he encontrado la llave.')

# Esta función es la tercera en ejecutarse y deberá finalizar antes de poder continuar
con la siguiente. Simula el momento en el que se abre la puerta.
def AbrirPuerta():
```

```
    # Vemos como la función se ejecuta en 0,3 segundos, esto significa que la función
acabaría de ejecutarse antes de iniciarse la función siguiente, con lo que respetaría
la secuendia en ese sentido, pero también, acabaría de ejecutarse antes que la función
anterior, cuando debería ser al revés, con lo que también se ejecuta de forma
desordenada y será necesario secuenciar de forma correcta su ejecución.
    time.sleep(0.3)
    print('3.Abro la puerta.')

# Esta función es la cuarta en ejecutarse y deberá finalizar antes de poder continuar
con la siguiente. Simula el momento en el que se cierra la puerta.
def CerrarPuerta():
    # La última función tiene un tiempo de ejecución medio, esto significa que se
ejecutará de forma desordenada con respecto a las funciones anteriores y su resultado
aparecerá en un momento no deseado en el flujo del programa. Es la última función y
debería ser la que tuviera un tiempo de ejecución más largo, como no es así, deberemos
manejar el flujo del programa para que se ejecute de forma secuencial.
    time.sleep(0.5)
    # Mostramos el resultado por pantalla.
    print('4.Cierro la puerta.')

# Usamos una variable para almacenar el tiempo de inicio de la ejecución del programa.
tiempo_ini = datetime.datetime.now()

# Ejecutamos de forma secuencial nuestras funciones, como no estamos usando la
programación multihilo, seguirá un proceso imperativo e irá ejecutando las funciones
de forma secuencial, sin tener en cuenta el tiempo de ejecución de cada una de ellas,
por lo que aparecerán ordenadas. Cada función muestra por pantalla un resultado fijo,
así que podremos seguir en "tiempo real" si la ejecución del programa se produce de
forma ordenada o no.
# Vemos que la ejecución del programa se produce de forma secuencial, pero que el
tiempo de ejecución de cada función no se respeta, por lo que el resultado cuando
apliquemos la programación multihilo no será el esperado.
#proceos 1
BuscarLlave()
#proceso 2
EncontrarLlave()
#proceso 3
AbrirPuerta()
#proceso 4
CerrarPuerta()

# Usamos una nueva variable para capturar el tiempo de finalización de la ejecución
del programa.
tiempo_fin = datetime.datetime.now()

# Mostramos por pantalla el tiempo total de ejecución del programa.
print('Tiempo total de ejecuciÃ³n:',str(tiempo_fin.second - tiempo_ini.second))
```

Tiempo de ejecución

Como vamos a ejecutar el programa con un flujo imperativo, las funciones se ejecutarán en orden secuencial, esperando cada una a finalizar su tiempo de proceso adjudicado, pero no realizándose la llamada a la siguiente hasta que haya finalizado la actual.

Análisis de las modificaciones del programa

El objetivo de esta actividad es modificar el programa para que se comporte siguiendo los patrones de **programación multihilo**, con lo que sea capaz de manejar dentro de su flujo la ejecución de distintas tareas de forma simultánea, teniendo en cuenta, además, las necesidades de nuestro código para que se mantenga el flujo secuencial necesario y las tareas esperen a los resultados de otras tareas siempre que se necesite.

Conceptos clave

Threads

La primera modificación que realizamos sobre nuestro código es reconvertir la llamada a las funciones para que se ejecuten como procesos multihilo, o sea, de manera simultánea. Para ello usaremos los métodos que implementa la librería *threading*.

El primero de estos será *thread*, este permite ejecutar operaciones de nuestro programa como operaciones concurrentes del mismo proceso. Es decir, el proceso, que es nuestro programa, ejecutará varias de sus operaciones de forma simultánea.

Interpolándolo a un caso real, por ejemplo, podríamos necesitar que nuestro programa iniciase la descarga de un archivo formado por varios trozos de menos tamaño, para aligerar el uso de la red. Podríamos, así, declarar varios hilos para que descargaran de forma simultánea varios de estos hilos, tendríamos que controlar la ejecución de estos hilos de forma en que todos podrían ejecutarse en paralelo, pero, para el último, tendríamos que usar una ejecución secuencial para no iniciar la acción de ensamblaje del archivo. Este ejemplo es algo más complejo, pero es una forma sencilla de aproximarse a un caso real.

Además, como parámetros pasamos la función que queremos ejecutar en el hilo. **Usamos la clase *thread* para declarar un nuevo hilo de ejecución**:

```
#proceos 1
    t_Buscar_Lave = threading.Thread(target = Buscar_Llave)
```

Start

El método *start* se usa para iniciar el hilo. Se invoca sobre la variable que hemos creado con la clase *thread*.

```
t_Buscar_Lave.start()
```

Join

El método *join* permite ejecutar un hilo de forma secuencial, así, evitamos que los procesos se lancen de forma concurrente y les indicamos que un proceso debe de esperar a que acabe el anterior para que pueda iniciar su ejecución. Así, **al usar *join* estamos indicando explícitamente que, aunque se ejecuten de forma simultánea, el hilo siguiente debe esperar a que el hilo con el método *join* termine.**

```
t_Buscar_Lave.join()
```

Implementación extra

Se han añadido algunos aspectos extra al código:

Queue

Podemos usar la clase *queue* para usar colas y pasarle variables de entrada y salida a nuestras funciones que se ejecutan en un hilo. Así vemos otra manera de modificar variables externas al ámbito de la función, además del uso de las variables globales.

```
def Cerrar_Puerta(res: queue.Queue, door: bool):
res = queue.Queue()
t_Cerrar_Puerta = threading.Thread(target = Cerrar_Puerta, args = (res, Is_Open))
```

Flags

Se usan un par de variables a modo de flags de control para que sea algo más interactivo y divertido el código, así realizamos un seguimiento de ¨dónde se encuentra la llave y cómo está la puerta¨ que impide que un método se ejecute si el **estado** no es el correcto, es decir, las funciones no podrán ejecutarse si no hemos codificado el programa de modo que el flujo de ejecución se ordene del modo esperado.

Nueva función

Se añade una función más para seguir con la lógica del ejemplo y practicar el uso del temporizador.

Análisis punto a punto

```
""" Este programa demuestra cómo se pueden ejecutar tareas en multihilo con Python.
"""
# Importamos las librerías necesarias
# Threading permite manejar los hilos implementando por detrás _thread.
import threading
# Time nos permite controlar el tiempo durante el que se va a ejecutar un hilo.
import time
# Datetime nos permite controlar el tiempo de ejecución de los hilos.
import datetime
# Queue nos permite manejar las colas de los hilos.
import queue
```

```
def Buscar_Llave():
    """ Esta función representa la tarea de buscar la llave. """
    # La primera tarea que se ejecutará tiene el tiempo de ejecución más alto, esto
significa que deberemos manejar
    # la ejecución de los hilos para ser capaces de ordenar la ejecución del proceso
de forma correcta. Así, deberemos
    # asegurarnos de que el resto de procesos esperen a que este proceso finalice
antes de iniciarse, para ello usaremos
    # join()
    time.sleep(0.9)
    # Imprimimos por pantalla que estamos buscando la llave.
    print('1.Busco la llave.')

def Encontrar_Llave(res: queue.Queue, key: bool):
    """ Esta función representa la tarea de encontrar la llave. Solo podemos encontrar
la llave si aún no la tenemos en
    nuestro bolsillo. """
    # Esta tarea un tiempo de ejecución menor a la anterior, por lo que tenemos que
asegurarnos de que se ejecuta antes la
    # tarea anterior para comenzar esta, además, tiene un tiempo de ejecución mayor
que la siguiente, por lo que la siguiente
    # tarea no debe finalizar antes que esta.
    time.sleep(0.7)
    if not key:
        # Imprimimos por pantalla que hemos encontrado la llave.
        print('2.Ya he encontrado la llave.')
        # Añadimos un valor a la cola para indicar que hemos encontrado la llave.
Luego capturaremos la cola en el programa
        # main y podremos cambiar los valores de los booleanos de control. La funcion
se comportará así como una función
        # pura con un valor de return.
        res.put(True)
        # Salimos de la función para que no se ejecute el resto de código.
        return
    # Si ya tenemos la llave, no es necesario encontrala de nuevo.
    print('2.No puedo encontrar la llave porque ya la tengo!.')

def Abrir_Puerta(res: queue.Queue, key: bool):
    """ Esta función representa la tarea de abrir la puerta. Solo podemos abrir la
puerta si tenemos la llave.
    Para poder abrir la puerta, tenemos que tener la llave. """
    # Esta tarea tiene un tiempo de ejecución menor que la anterior, por lo que
deberemos asegurarnos de que la anterior ha
    # finalizado su ejecución antes de iniciarla, y tiene un tiempo de ejecución menor
que la siguiente, por lo que acabará de
    # ejecutarse antes, aún así, mantendremos el control secuencia de ejecución de los
hilos para poder asignar correctamente
    # la variable con la que trabajamos antes de iniciar la siguiente tarea.
    time.sleep(0.3)
    # Comprobamos que tengamos la llave para poder abrir la puerta.
    if key:
        # Imprimimos por pantalla que hemos abierto la puerta.
```

```
        print('3.Abro la puerta.')
        # Añadimos un valor a la cola para indicar que hemos abierto la puerta. Luego
capturaremos la cola en el programa
        # main y podremos cambiar los valores de los booleanos de control. La funcion
se comportará así como una función
        # pura con un valor de return.
        res.put(True)
        # Salimos de la función para que no se ejecute el resto de código.
        return
    # Si no tenemos la llave, no podemos abrir la puerta.
    print('3. No puedo abrir la puerta, no tengo la llave!')
    # Añadimos un valor a la cola para indicar que no hemos abierto la puerta. Luego
capturaremos la cola en el programa
    # main y podremos cambiar los valores de los booleanos de control. La funcion se
comportará así como una función
    # pura con un valor de return.
    res.put(False)

def Cerrar_Puerta(res: queue.Queue, door: bool):
    """ Esta función representa la tarea de cerrar la puerta. Solo podemos cerrar la
puerta si la hemos abierto. """
    # Esta tarea tiene un tiempo de ejecución mayor a la anterior, por lo que tardaría
más en ejecutarse que la anterior,
    # y así veríamos finalizado el proceso de abrir la puerta antes del de cerrar la
puerta.
    # Aún así, como hemos incluido el trabajo con las variables de control,
necesitamos que la tarea anteriror finalice
    # para tener seteada la variable de forma correcta, por lo tanto, mantenemos el
control secuencial de las tareas.
    time.sleep(0.5)
    # Comprobamos que hayamos abierto la puerta para poder cerrarla.
    if door:
        # Imprimimos por pantalla que hemos cerrado la puerta.
        print('4.Cierro la puerta.')
        # Añadimos un valor a la cola para indicar que hemos cerrado la puerta. Luego
capturaremos la cola en el programa
        # main y podremos cambiar los valores de los booleanos de control. La funcion
se comportará así como una función
        # pura con un valor de return.
        res.put(False)
        # Salimos de la función para que no se ejecute el resto de código.
        return
    # Si no hemos abierto la puerta, no podemos cerrarla.
    print('4.No puedo cerrar la puerta porque no la he abierto todavía!.')
    # Añadimos un valor a la cola para indicar que no hemos cerrado la puerta. Luego
capturaremos la cola en el programa
    # main y podremos cambiar los valores de los booleanos de control. La funcion se
comportará así como una función
    # pura con un valor de return.
    res.put(True)

def Guardar_Llave(res : queue.Queue, key: bool):
    """ Esta función representa la tarea de guardar la llave. Solo podemos guardar la
llave si la hemos encontrado. """
```

```
    time.sleep(0.1)
    if key:
        print('5.Guardo la llave.')
        res.put(False)
        return
    print('5.No puedo guardar la llave porque no la he encontrado todavía!.')
    res.put(True)

def __main__():
    # Variables para que sea más interactivo y divertido
    Is_Key = False
    Is_Open = False
    # Cola para guardar los resultados del hilo
    res = queue.Queue()

    tiempo_ini = datetime.datetime.now()

    #proceos 1
    t_Buscar_Lave = threading.Thread(target = Buscar_Llave)
    t_Buscar_Lave.start()
    t_Buscar_Lave.join()
    #proceso 2
    t_Encontrar_Llave = threading.Thread(target = Encontrar_Llave, args = (res,
Is_Key))
    t_Encontrar_Llave.start()
    t_Encontrar_Llave.join()
    Is_Key = res.get()
    # EncontrarLlave()
    #proceso 3
    t_Abrir_Puerta = threading.Thread(target = Abrir_Puerta, args = (res, Is_Key))
    t_Abrir_Puerta.start()
    t_Abrir_Puerta.join()
    Is_Open = res.get()
    # AbrirPuerta()
    #proceso 4
    t_Cerrar_Puerta = threading.Thread(target = Cerrar_Puerta, args = (res, Is_Open))
    t_Cerrar_Puerta.start()
    # Como se trata de la última tarea, no es necesario hacer un join, ya que no hay
más tareas que ejecutar.

    Is_Open = res.get()
    # CerrarPuerta()
    #proceso 5
    t_Guardar_Llave = threading.Thread(target = Guardar_Llave, args = (res, Is_Key))
    t_Guardar_Llave.start()
    t_Guardar_Llave.join()
    Is_Key = res.get()

    if not Is_Key and not Is_Open:
        print('La puerta está cerrada y la llave en nuestro bolsillo.')
    else:
        print('No hemos podido abrir esa puerta!')
    tiempo_fin = datetime.datetime.now()
    print('Tiempo total de ejecución:',str(tiempo_fin.second - tiempo_ini.second))
```

```
# Ejecutar el programa principal mediante una función main
if __name__ == '__main__':
    __main__()
```

Tiempo de ejecución

En este caso, hemos preparado el programa para crear todas las funciones como hilos que se ejecutarán de forma simultánea. Debido a los temporizadores que hemos incluido en las funciones, estás finalizarán su ejecución en un orden totalmente distinto al necesario para que el flujo principal tenga sentido. Para que los hilos se ejecuten de forma ordenada, secuencia, usamos el método *join*. Así, volvemos a una situación como la que encontramos en la aproximación imperativa donde cada función esperará a que la función anterior finalice antes de ejecutarse. De este modo, mantendremos el orden de ejecución de las funciones según su orden de declaración. Para la última función no es necesario preocuparse porque ya no tenemos que ejecutar nada más.

Notas sobre el uso de *thread* y *join*

Puede parecer que cualquier forma de declarar nuestros hilos es válida, pero debemos tener cuidado.
Mientras que con esta opción nuestro hilo se ejecutará de forma secuencial:

```
# Ejecución de los hilos de forma secuencial
a.start()
a.join()
b.start()
b.join()
```

Con este otro código se ejecutaría en paralelo:

```
# Ejecución de los hilos de forma concurrente
a.start()
b.start()
a.join()
b.join()
```

Tiempo total de ejecución

Caso imperative e hilo secuencial

Tanto el caso del programa original, ejecutándose de modo imperativo, como en el caso del manejo de los hilos de forma secuencial, veremos que el tiempo total de ejecución es la suma del tiempo de ejecución de todos los métodos.

Caso hilo simultáneo

En el punto intermedio del diseño de nuestro programa, podríamos encontrarnos con un código con los hilos creados, pero sin usar los métodos *join*, la creación e hilos con llamadas a las funciones podría ser algo así:

```
#proceos 1
t_b = threading.Thread(target=BuscarLlave)
t_b.start()
#proceso 2
t_e = threading.Thread(target=EncontrarLlave)
t_e.start()
#proceso 3
t_a = threading.Thread(target=AbrirPuerta)
t_a.start()
#proceso 4
t_c = threading.Thread(target=CerrarPuerta)
t_c.start()
```

En este caso tendríamos un retorno desordenado y un tiempo de ejecución de 0 segundos, ya que el proceso mayor es de 0.9 segundos.

Recursos

threading — Paralelismo basado en hilos — documentación de Python - 3.8.16. (s. f.). https://docs.python.org/es/3.8/library/threading.htm l

ProgrammingKnowledge. (2018, 19 noviembre). *Python Thread Tutorial For Beginners 1 - Introduction to multithreading in Python* [Vídeo]. YouTube. https://www.youtube.com/watch?v=M04E_Wr6dG4

Ultimate python de cero a experto. (s. f.). Hola Mundo. https://academia.holamundo.io/courses/ultimate-python