Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

viernes, 30 de diciembre de 2016

Threading: programación con hilos (y II)

Temporizadores

Un **temporizador** (**Timer**) es un tipo de hilo especial que permite ajustar el comienzo de su ejecución con un tiempo de espera. Además, mientras transcurre este tiempo de espera es posible cancelar su ejecución.

Un temporizador es un objeto de la subclase **Timer** que deriva de la clase **Thread** y como sucede con sus ancestros admite el paso de argumentos:

class threading.Timer(intervalo, función, args=None, kwargs=None)

En el siguiente ejemplo se crean dos temporizadores (**hilo1** y **hilo2**) con un tiempo de espera de 0.2 y 0.5 segundos, respectivamente. Cuando se cumple el tiempo de espera de **hilo1** comienza su ejecución. Sin embargo, **hilo2** es cancelado antes de concluir su tiempo de espera. El programa termina cuando **hilo1** finaliza su trabajo.

```
import threading
import time

def retrasado():
    nom_hilo = threading.current_thread().getName()
    contador = 1
    while contador <=10:
        print(nom_hilo, 'ejecuta su trabajo', contador)
        time.sleep(0.1)
        contador+=1
    print(nom_hilo, 'ha terminado su trabajo')

hilo1 = threading.Timer(0.2, retrasado)
hilo1.setName('hilo1')
hilo2 = threading.Timer(0.5, retrasado)
hilo2.setName('hilo2')

hilo1.start()
hilo2.start()
print('hilo1 espera 0.2 segundos')
print('hilo2 espera 0.5 segundos')

time.sleep(0.3)
print('hilo2 va a ser cancelado')
hilo2.cancel()
print('hilo2 fue cancelado antes de iniciar su ejecución')
```

Sincronizar hilos con objetos Event

Buscar

Buscar

Python para impacientes

Python
IPython
EasyGUI
Tkinter
JupyterLab
Numpy

Anexos

Guía urgente de MySQL
Guía rápida de SQLite3

Entradas + populares

Dar color a las salidas en la consola
En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

Instalación de Python, paso a paso
Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6 . El propósito de esta entrada es mostrar, pas...

Añadir, consultar, modificar y suprimir elementos en Numpy
Acceder a los elementos de un array. [], [,], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

Variables de control en Tkinter
Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

Cálculo con arrays Numpy
Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebid...

Tkinter: interfaces gráficas en Python
Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario (GUI) pero Tkinter es fácil d...

Operaciones con fechas y horas. Calendarios
Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

Convertir, copiar, ordenar, unir y dividir arrays Numpy
Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

Tkinter: Tipos de ventanas
Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

Threading: programación con hilos (I)
En programación, la técnica que permite que una aplicación ejecute

A veces, es necesario que varios hilos se puedan comunicar entre si para sincronizar sus trabajos. Uno de los mecanismos disponibles se basa en los objetos **Event** por su capacidad de anunciar a uno o más hilos (que esperan) que se ha producido un suceso para que puedan proseguir su ejecución. Para ello, utiliza el valor de una propiedad que es visible por todos los hilos. Los valores posibles de esta propiedad son **True** o **False** y pueden ser asignados con los métodos **set()** y **clear()**, respectivamente.

Por otro lado, el método **wait()** se emplea para detener la ejecución de uno o más hilos hasta que la propiedad alcance el valor **True**. Dicho método devuelve el valor que tenga la propiedad y cuenta con un argumento opcional para fijar un tiempo de espera. Otra opción para obtener el estado de un evento es mediante el método **is_set()**.

En el ejemplo siguiente se inician dos hilos que permanecen a la espera hasta la obtención de 25 números pares (los números son generados con la función **randint()** del módulo **random**). Cuando se tienen todos los números los dos hilos continúan su ejecución de manera sincronizada.

```
import threading, random

def gen_pares():
    num_pares = 0
    print('Números:', end=' ')
    while num_pares < 25:
        numero = random.randint(1, 10)
        resto = numero % 2
        if resto == 0:
            num_pares +=1
            print(numero, end=' ')
    print()

def contar():
    contar = 0
    nom_hilo = threading.current_thread().getName()
    print(nom_hilo, "en espera")
    estado = evento.wait()
    while contar < 25:
        contar+=1
        print(nom_hilo, ':', contar)

evento = threading.Event()
hilo1 = threading.Thread(name='h1', target=contar)
hilo2 = threading.Thread(name='h2', target=contar)
hilo1.start()
hilo2.start()

print('Obteniendo 25 números pares...')
gen_pares()
print('Ya se han obtenido')
evento.set()
```

A continuación, otro ejemplo en el que dos hilos alternan su ejecución en función al valor del objeto **Event**. Dicho valor cambia cuando se cumple un número de ciclos, que es diferente en cada hilo. El programa implementa el funcionamiento de dos contadores: uno avanza rápidamente y el otro retrocede lentamente porque se incluye un argumento con un tiempo de retardo. En el momento que es imposible pasar el testigo al otro hilo (porque cumplió su cometido) el hilo que queda activo concluye el suyo.

```
import threading

def avanza(evento):
    ciclo = 0
    valor = 0
    while valor < 20:
        estado = evento.wait()
        if estado:
            ciclo+=1
            valor+=1
            print('avanza', valor)
            if ciclo == 10 and hilo2.isAlive():
                evento.clear()
                ciclo = 0
    print('avanza: ha finalizado')

def retrocede(evento, tiempo):
    ciclo = 0
    valor = 21
    while valor > 1:
        estado = evento.wait(tiempo)
```

simultáneamente varias operaciones en el mismo espacio de proceso se...

Archivo

diciembre 2016 (2)

python.org

pypi.org

Sitios

- ActivePython
- Anaconda
- Bpython
- Django
- Flask
- Ipython
- IronPython
- Matplotlib
- MicroPython
- Numpy
- Pandas
- Pillow
- PortablePython
- PyBrain
- PyCharm
- PyDev
- PyGame
- Pypi
- PyPy
- Pyramid
- Python.org
- PyTorch
- SciPy.org
- Spyder
- Tensorflow
- TurboGears

```
        if not estado:
            ciclo+=1
            valor-=1
            print('retrocede', valor)
            if ciclo == 5 and hilo1.isAlive():
                evento.set()
                ciclo = 0
    print('retrocede: ha finalizado')

evento = threading.Event()
hilo1 = threading.Thread(target=avanza,
                         args=(evento,),)
hilo2 = threading.Thread(target=retrocede,
                         args=(evento, 0.5),)
hilo1.start()
hilo2.start()
```

Control del acceso a los recursos. Bloqueos

Además de sincronizar el funcionamiento de varios subprocesos también es importante controlar el acceso de los hilos a los recursos compartidos (variables, listas, diccionarios, etc.) para evitar la corrupción o pérdida de datos. En determinadas circunstancias estas estructuras de datos requieren protegerse con bloqueos contra el acceso simultáneo de varios hilos que intentan modificar su valores. Esto raramente va a suceder si varios hilos tratan de actualizar una sola variable. Sin embargo, el problema se puede dar al actualizar el valor de una variable que utiliza los datos de otras variables (intermedias) que son leídas y modificadas varias veces en el proceso por varios hilos.

Los objetos **Lock** permiten gestionar los bloqueos que evitan que los hilos modifiquen variables compartidas al mismo tiempo. El método **acquire()** permite que un hilo bloquee a otros hilos en un punto del programa, donde se leen y actualizan datos, hasta que dicho hilo libere el bloqueo con el método **release()**. En el momento que se produzca el desbloqueo otro hilo (o el mismo) podrá bloquear de nuevo.

En el ejemplo que sigue se inician dos hilos que actualizan la variable global **total** donde se van acumulando números que son múltiplos de 5. Antes y después de cada actualización se produce un bloqueo y un desbloqueo, respectivamente:

```
import threading

total = 0

def acumula5():
    global total
    contador = 0
    hilo_actual = threading.current_thread().getName()
    while contador < 20:
        print('Esperando para bloquear', hilo_actual)
        bloquea.acquire()
        try:
            contador = contador + 1
            total = total + 5
            print('Bloqueado por', hilo_actual, contador)
            print('Total', total)

        finally:
            print('Liberado bloqueo por', hilo_actual)
            bloquea.release()

bloquea = threading.Lock()
hilo1 = threading.Thread(name='h1', target=acumula5)
hilo2 = threading.Thread(name='h2', target=acumula5)
hilo1.start()
hilo2.start()
```

Para conocer si otro hilo ha adquirido el bloqueo sin mantener al resto de subprocesos detenidos hay que asignar al argumento **blocking** de **acquire()** el valor **False**. De esta forma se pueden realizar otros trabajos mientras se espera a tener éxito en un bloqueo y controlar el número de reintentos realizados. El método **locked()** se puede utilizar para verificar si un bloqueo se mantiene en un momento dado:

```
import threading

def acumula5():
    global total
    contador = 0
    hilo_actual = threading.current_thread().getName()
```

```
    num_intentos = 0
    while contador < 20:
        lo_conseguí = bloquea.acquire(blocking=False)
        try:
            if lo_conseguí:
                contador = contador + 1
                total = total + 5
                print('Bloqueado por', hilo_actual, contador)
                print('Total', total)
            else:
                num_intentos+=1
                print('Número de intentos de bloqueo',
                      num_intentos,
                      'hilo',
                      hilo_actual,
                      bloquea.locked())
                print('Hacer otro trabajo')

        finally:
            if lo_conseguí:
                print('Liberado bloqueo por', hilo_actual)
                bloquea.release()

total = 0
bloquea = threading.Lock()
hilo1 = threading.Thread(name='h1', target=acumula5)
hilo2 = threading.Thread(name='h2', target=acumula5)
hilo1.start()
hilo2.start()
```

Los objetos **RLock** son parecidos a los objetos **Lock** con la diferencia de que permiten que un bloqueo pueda ser adquirido por el mismo hilo varias veces.

Para concluir este apartado, hacer mención al uso de la declaración **with** que evita tener que adquirir y liberar explícitamente cada bloqueo. En el ejemplo siguiente las dos funciones que llaman los hilos son equivalentes:

```
import threading

def con_with(bloqueo):
    with bloqueo:
        print('Bloqueo adquirido con with')
        print('Procesando...')

def sin_with(bloqueo):
    bloqueo.acquire()
    try:
        print('Bloqueo adquirido directamente')
        print('procesando...')
    finally:
        bloqueo.release()

bloqueo = threading.Lock()
hilo1 = threading.Thread(target=con_with, args=(bloqueo,))
hilo2 = threading.Thread(target=sin_with, args=(bloqueo,))
```

Sincronizar hilos con objetos Condition

Los objetos **Condition** se utilizan también para sincronizar la ejecución de varios hilos. En este caso los bloqueos suelen estar vinculados con unas operaciones que se tienen que realizar antes que otras.

En el siguiente ejemplo un hilo espera -llamando al método **wait()**- a que otro hilo genere mil números aleatorios que son añadidos a una lista. Una vez que se han obtenido los números el hecho es notificado con **notifyAll()** al hilo que espera. Finalmente, el hilo detenido continua su ejecución mostrando el número de elementos generados y la suma de todos ellos, con la función **fsum()** del módulo **math**.

```
import threading, random, math

def funcion1(condicion):
    global lista
    print(threading.current_thread().name,
          'esperando a que se generen los números')
    with condicion:
        condicion.wait()
        print('Elementos:', len(lista))
```

```
        print('Suma total:', math.fsum(lista))

def funcion2(condicion):
    global lista
    print(threading.current_thread().name,
          'generando números')
    with condicion:
        for numeros in range(1, 1001):
            entero = random.randint(1,100)
            lista.append(entero)
        print('Ya hay 1000 números')
        condicion.notifyAll()

lista = []
condicion = threading.Condition()
hilo1 = threading.Thread(name='hilo1', target=funcion1,
                         args=(condicion,))
hilo2 = threading.Thread(name='hilo2', target=funcion2,
                         args=(condicion,))

hilo1.start()
hilo2.start()
```

Sincronizar hilos con objetos Barrier

Los objetos **barrera** (**Barrier**) son otro mecanismo de sincronización de hilos. Como su propio nombre sugiere actúan como una verdadera barrera que mantiene los hilos bloqueados en un punto del programa hasta que todos hayan alcanzado ese punto.

En el siguiente ejemplo se inician cinco hilos que obtienen un número aleatorio y permanecen bloqueados en el punto donde se encuentra el método **wait()** a la espera de que el último hilo haya obtenido su número. Después, continúan todos mostrando el factorial del número obtenido en cada caso.

```
import threading, random, math

def funcion1(barrera):
    nom_hilo = threading.current_thread().name
    print(nom_hilo,
          'Esperando con',
          barrera.n_waiting,
          'hilos más')

    numero = random.randint(1,10)
    ident = barrera.wait()
    print(nom_hilo,
          'Ejecutando después de la espera',
          ident)
    print('factorial de',
          numero,
          'es',
          math.factorial(numero))

NUM_HILOS = 5
barrera = threading.Barrier(NUM_HILOS)
hilos = [threading.Thread(name='hilo-%s' % i,
                          target=funcion1,
                          args=(barrera,),
                          ) for i in range(NUM_HILOS)]

for hilo in hilos:
    print(hilo.name, 'Comenzando ejecución')
    hilo.start()
```

Existe la posibilidad de enviar un aviso de cancelación a todos los hilos que esperan con el método **abort()** del objeto **Barrier**. Esta acción genera una excepción de tipo **threading.BrokenBarrierError** que se debe capturar y tratar convenientemente:

```
try:
 ident = barrera.wait()
except threading.BrokenBarrierError:
 print(nom_hilo, 'Cancelando')
else:
 print('Ejecutando después de la espera', ident)
```

Limitar el acceso concurrente con semáforos

Un objeto **Semaphore** es un instrumento de bloqueo avanzado que utiliza un contador interno para controlar el número de hilos que pueden acceder de forma concurrente a una parte del código. Si el número de hilos que intentan acceder supera, en un momento dado, al valor establecido se producirá un bloqueo que será liberado en la medida que los hilos no bloqueados vayan completando las operaciones previstas.

Realmente actúa como un semáforo en la entrada de un aparcamiento público: dejando pasar vehículos mientras existen plazas disponibles y cerrando el acceso hasta que no quede libre al menos una plaza.

Obviamente, este dispositivo se utiliza para restringir el acceso a recursos con capacidad limitada.

En el siguiente ejemplo se generan cinco hilos para simular una descarga simultánea de archivos. Las descargas podrán ser concurrentes hasta un máximo de 3 (es el valor que tiene la constante **NUM_DESCARGAS_SIM** que se utiliza para declarar el objeto **Semaphore**).

```
import threading
import time

def descargando(semaforo):
    global activas
    nombre = threading.current_thread().getName()
    print('Esperando para descargar:', nombre)
    with semaforo:
        activas.append(nombre)
        print('Descargas activas', activas)
        print('...Descargando...', nombre)
        time.sleep(0.1)
        activas.remove(nombre)
        print('Descarga finalizada', nombre)

NUM_DESCARGAS_SIM = 3
activas = []
semaforo = threading.Semaphore(NUM_DESCARGAS_SIM)
for indice in range(1,6):
    hilo = threading.Thread(target=descargando,
                            name='D' + str(indice),
                            args=(semaforo,),)
    hilo.start()
```

Ir al índice del tutorial de Python

Publicado por Pherkad en 17:35

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua