viernes, 16 de diciembre de 2016

# Threading: programación con hilos (I)

En programación, la técnica que permite que una aplicación ejecute simultáneamente varias operaciones en el mismo espacio de proceso se llama **Threading**. A cada flujo de ejecución que se origina durante el procesamiento se le denomina **hilo** o **subproceso**, pudiendo realizar o no una misma tarea. En Python, el módulo **threading** hace posible la programación con hilos.

Existe infinidad de situaciones en las que el uso de hilos puede ser de mucha utilidad: una aplicación con la capacidad de realizar varias descargas en paralelo, que puede abrir o guardar un documento de tamaño considerable mientras se edita otro, que permite lanzar varias operaciones de búsqueda al mismo tiempo, que puede chequear el funcionamiento de un conjunto de sistemas simultáneamente, etc.

Ejecutar varios hilos o subprocesos es similar a ejecutar varios programas diferentes al mismo tiempo, pero con algunas ventajas añadidas:

- Los hilos en ejecución de un proceso comparten el mismo espacio de datos que el hilo principal y pueden, por tanto, tener acceso a la misma información o comunicarse entre sí más fácilmente que si estuvieran en procesos separados.
- Ejecutar un proceso de varios hilos suele requerir menos recursos de memoria que ejecutar lo equivalente en procesos separados.
- Permite simplificar el diseño de las aplicaciones que necesitan ejecutar varias operaciones concurrentemente.

Para cada hilo de un proceso existe un puntero que realiza el seguimiento de las instrucciones que se ejecutan en cada momento. Además, la ejecución de un hilo se puede detener temporalmente o de manera indefinida. En general, un proceso sigue en ejecución cuando al menos uno de sus hilos permanece activo, es decir, cuando el último hilo concluye su cometido, termina el proceso, liberándose en ese momento todos los recursos utilizados.

## Objetos Thread: los hilos

En Python un objeto **Thread** representa una determinada operación que se ejecuta como un subproceso independiente, es decir, representa a un hilo. Hay dos formas de definir un hilo: la primera, consiste en pasar al método constructor un objeto invocable, como una función, que es llamada cuando se inicia la ejecución del hilo y, la segunda, radica en crear una subclase de **Thread** en la que se reescribe el método **run()** y/o el constructor **__init__()**.

En el siguiente ejemplo se crean dos hilos que llaman a la función **contar**. En dicha función se utiliza la variable **contador** para contar hasta cien. Los objetos **Thread** (los hilos) utilizan el argumento **target** para establecer el nombre de la función a la que hay que llamar. Una vez que los hilos se han creado se inician con el método **start()**. A todos los hilos se les asigna, automáticamente, un nombre en el momento de la creación que se puede conocer con el método **getName()** y, también, un identificador único (en el momento que son iniciados) que se puede obtener accediendo al valor del atributo **ident**:

**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...

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...

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

El módulo random
El módulo random de la librería estándar de Python incluye un conjunto de funciones que permiten obtener de distintos modos números a...

Threading: programación con hilos (I)
En programación, la técnica que permite que una aplicación ejecute simultáneamente varias operaciones en el mismo espacio de proceso se...

Cadenas, listas, tuplas, diccionarios y conjuntos (set)
Las cadenas , listas y tuplas son distintos tipos de secuencias . Una secuencia es un

```
import threading

def contar():
    '''Contar hasta cien'''
    contador = 0
    while contador<100:
        contador+=1
        print('Hilo:',
              threading.current_thread().getName(),
              'con identificador:',
              threading.current_thread().ident,
              'Contador:', contador)

hilo1 = threading.Thread(target=contar)
hilo2 = threading.Thread(target=contar)
hilo1.start()
hilo2.start()
```

A continuación, una versión mejorada del ejemplo anterior que utiliza la constante **NUM_HILOS** para establecer el número de hilos que han de iniciarse. Los hilos se crean e inician implementando un bucle basado en **range()**. En este caso el nombre de cada hilo se construye con el valor de la variable **num_hilo** que es asignado al atributo **name**. Existe otra posibilidad de asignar un nombre a un hilo con el método **hilo.setName(nombre)**; y de acceder a su nombre mediante **hilo.name**:

```
import threading

def contar():
    contador = 0
    while contador<100:
        contador+=1
        print('Hilo:',
              threading.current_thread().getName(),
              'con identificador:',
              threading.current_thread().ident,
              'Contador:', contador)

NUM_HILOS = 3

for num_hilo in range(NUM_HILOS):
    hilo = threading.Thread(name='hilo%s' %num_hilo,
                            target=contar)
    hilo.start()
```

## Hilos con argumentos

Para ajustar el comportamiento de los programas que usan hilos nada mejor que tener la posibilidad de pasar valores a los hilos. Para eso están los argumentos **args** y **kwargs** en el constructor.

En el siguiente ejemplo se utilizan estos argumentos para pasar una variable con el número de hilo que se ejecuta en un momento dado y un diccionario con tres valores que ajustan el funcionamiento del contador en todos los hilos:

```
import threading

def contar(num_hilo, **datos):
    contador = datos['inicio']
    incremento = datos['incremento']
    limite = datos['limite']
    while contador<=limite:
        print('hilo:', num_hilo, 'contador:', contador)
        contador+=incremento

for num_hilo in range(3):
    hilo = threading.Thread(target=contar,
                            args=(num_hilo,),
                            kwargs={'inicio':0,
                                    'incremento':1,
                                    'limite':10})
    hilo.start()
```

## Hilos que funcionan durante un tiempo

tipo de objeto que almacena datos y que permite ...

**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

Otra opción interesante consiste en limitar el funcionamiento de los hilos a un tiempo determinado.

En el ejemplo siguiente se inician 5 hilos que funcionan durante 1 segundo. Mientras transcurre el tiempo cada hilo incrementa un contador hasta que se alcanza el tiempo límite. El módulo **time** se utiliza para obtener el momento inicial y calcular el tiempo límite de ejecución.

Al concluir el tiempo de cada hilo el valor máximo contado se va añadiendo a un diccionario que se muestra cuando el último hilo activo está finalizando. Para conocer cuándo está finalizando el último hilo se utiliza la función **threading.active_count()** que devuelve el número de hilos que aún quedan activos, incluyendo al hilo principal (que se corresponde con el hilo que inicia el propio programa), es decir, cuando el último hilo **Thread** esté terminando quedarán activos 2 hilos. Para satisfacer la curiosidad al final se muestra una lista con información de estos hilos obtenida con la función **threading.enumerate()**.

La variable **vmax_hilos** contiene los valores máximos del contador de cada hilo. Esta variable se inicializa al comienzo del programa y se declara después como **global** dentro de la función (Ver variables locales y glboales). Esto se hace para lograr mantener "vivos" los valores máximos que se añaden al diccionario al concluir cada hilo. Si no se declara como global sólo permanecerá el último valor agregado.

```
import threading, time

vmax_hilos = {}

def contar(segundos):
    """Contar hasta un límite de tiempo"""
    global vmax_hilos
    contador = 0
    inicial = time.time()
    limite = inicial + segundos
    nombre = threading.current_thread().getName()
    while inicial<=limite:
        contador+=1
        inicial = time.time()
        print(nombre, contador)

    vmax_hilos[nombre] = contador
    if threading.active_count()==2:
        print(vmax_hilos)
        print(threading.enumerate())

segundos = 1
for num_hilo in range(5):
    hilo = threading.Thread(name='hilo%s' %num_hilo,
                            target=contar,
                            args=(segundos,))
    hilo.start()
```

## Demonios

Existen dos modos diferentes de finalizar un programa basado en hilos. En el primer modo el hilo principal del programa espera a que todos los hilos creados con **Thread** terminan su trabajo. Ese es el caso de todos los ejemplos mostrados hasta ahora.

En el segundo modo, el hilo principal del programa puede finalizar aunque uno o más hilos hijos no hayan terminado su tarea; teniendo en cuenta que cuando finalice el hilo principal también lo harán estos hilos especiales llamados **demonios**. Si existen hilos no-demonios el hilo principal esperará a que estos concluyan su trabajo. Los **demonios** son útiles para programas que realizan operaciones de monitorización o de chequeo de recursos, servicios, aplicaciones, etc.

Para declarar un hilo como **demonio** se asigna **True** al argumento **daemon** al crear el objeto **Thread**, o bien, se establece dicho valor con posterioridad con el método **set_daemon()**.

El ejemplo que sigue utiliza dos hilos: un hilo escribe en un archivo y el otro hilo (el demonio) chequea el tamaño del archivo cada cierto tiempo. Cuando el hilo encargado de escribir termina, todo el programa llega a su fin a pesar de que el contador del demonio no ha alcanzado el valor límite.

```
import time, os, threading

def chequear(nombre):
    '''Chequea tamaño de archivo'''
    contador = 0
    tam = 0
    while contador<100:
        contador+=1
        if os.path.exists(nombre):
```

```
            estado = os.stat(nombre)
            tam = estado.st_size

        print(threading.current_thread().getName(),
              contador,
              tam,
              'bytes')

        time.sleep(0.1)

def escribir(nombre):
    '''Escribe en archivo'''
    contador = 1
    while contador<=10:
        with open(nombre, 'a') as archivo:
            archivo.write('1')
            print(threading.current_thread().getName(),
                  contador)
            time.sleep(0.3)
            contador+=1

nombre = 'archivo.txt'
if os.path.exists(nombre):
    os.remove(nombre)

hilo1 = threading.Thread(name='chequear',
                         target=chequear,
                         args=(nombre,),
                         daemon=True)

hilo2 = threading.Thread(name='escribir',
                         target=escribir,
                         args=(nombre,))
hilo1.start()
hilo2.start()
```

Para hacer que el hilo principal espere a que el hilo demonio complete su trabajo, utilizar el método **join()** con dicho hilo. El método **isAlive()** también es útil parca conocer si un hilo está o no activo. En el ejemplo retorna **False** porque el demonio ya ha terminado:

```
import time, os, threading

def chequear(nombre):
    '''Chequea tamaño de archivo'''
    contador = 0
    tam = 0
    while contador<100:
        contador+=1
        if os.path.exists(nombre):
            estado = os.stat(nombre)
            tam = estado.st_size

        print(threading.current_thread().getName(),
              contador,
              tam,
              'bytes')

        time.sleep(0.1)

def escribir(nombre):
    '''Escribe en archivo'''
    contador = 1
    while contador<=10:
        with open(nombre, 'a') as archivo:
            archivo.write('1')
            print(threading.current_thread().getName(),
                  contador)
            time.sleep(0.3)
            contador+=1

nombre = 'archivo.txt'
if os.path.exists(nombre):
    os.remove(nombre)

hilo1 = threading.Thread(name='chequear',
                         target=chequear,
                         args=(nombre,),
                         daemon=True)

hilo2 = threading.Thread(name='escribir',
```

```
                        target=escribir,
                        args=(nombre,))
hilo1.start()
hilo2.start()

hilo1.join()
print(hilo1.isAlive())
```

### Controlar la ejecución de varios demonios

Cuando un programa utiliza un número indeterminado de demonios y se pretende que el hilo principal espere a que todos terminen su ejecución, utilizaremos **join()** con cada demonio. Para hacer el seguimiento de los hilos activos se puede emplear **enumerate()** pero teniendo en cuenta que dentro de la lista que devuelve se incluye el hilo principal. Con este hilo hay que tener cuidado porque no acepta ciertas operaciones, por ejemplo, no se puede obtener su nombre con **getName()** o utilizar el método **join()**.

En el ejemplo se emplea la función **threading.main_thread()** para identificar al hilo principal. Después, se recorren todos los hilos activos para ejecutar **join()**, excluyendo al principal.

```
import threading

def contar(numero):
    contador = 0
    while contador<10:
        contador+=1
        print(numero, threading.get_ident(), contador)

for numero in range(1, 11):
    hilo = threading.Thread(target=contar,
                            args=(numero,),
                            daemon=True)
    hilo.start()

# Obtiene hilo principal

hilo_ppal = threading.main_thread()

# Recorre hilos activos para controlar estado de su ejecución

for hilo in threading.enumerate():

    # Si el hilo es hilo_ppal continua al siguiente hilo activo

    if hilo is hilo_ppal:
        continue

    # Se obtiene información hilo actual y núm. hilos activos

    print(hilo.getName(),
          hilo.ident,
          hilo.isDaemon(),
          threading.active_count())

    # El programa esperará a que este hilo finalice:

    hilo.join()
```

### Crear una subclase Thread y redefinir sus métodos

Cuando comienza la ejecución de un hilo se invoca, automáticamente, al método subyacente **run()** que es el que llama a la función pasada al constructor. Para crear una subclase **Thread** es necesario reescribir como mínimo el método **run()** con la nueva funcionalidad.

```
import threading

class MiHilo(threading.Thread):
    def run(self):
        contador = 1
        while contador <= 10:
            print('ejecutando',
                  threading.current_thread().getName(),
                  contador)
            contador+=1

for numero in range(10):
```

```
    hilo = MiHilo()
    hilo.start()
```

La situación se hace algo más compleja si se quieren pasar valores utilizando los argumentos **args** y/o **kwargs** porque es necesario reescribir el método **__init__()**. Por defecto, el constructor Thread utiliza variables privadas para estos argumentos. En el siguiente ejemplo se declara una subclase con dos argumentos.

```
import threading

class MiHilo(threading.Thread):
    def __init__(self, group=None, target=None, name=None,
                 args=(), kwargs=None, *, daemon=None):
        super().__init__(group=group, target=target, name=name,
                         daemon=daemon)
        self.arg1 = args[0]
        self.arg2 = args[1]

    def run(self):
        contador = 1
        while contador <= 10:
            print('ejecutando...',
                  'contador', contador,
                  'argumento1', self.arg1,
                  'argumento2', self.arg2)
            contador+=1

for numero in range(10):
    hilo = MiHilo(args=(numero,numero*numero), daemon=False)
    hilo.start()
```

Ir al índice del tutorial de Python

Publicado por Pherkad en 5:57

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua