Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

viernes, 17 de marzo de 2017

Programar eventos para ejecutar tareas

El módulo sched

El módulo **sched** implementa un programador o planificador de eventos para ejecutar tareas después de que transcurra un tiempo específico, o bien, para ejecutar en un momento determinado (por ejemplo, en una fecha y hora concretas).

El módulo **sched** se basa en la clase **scheduler** que tiene métodos para declarar un programador; definir, ejecutar y cancelar eventos; y para consultar información sobre eventos pendientes de ejecución.

Declarar un programador

Para declarar un programador se utiliza el método constructor **scheduler()** que tiene como primer argumento una función para obtener la hora actual -que por defecto es **monotonic()**- y una función para introducir los tiempos de espera -que de forma predeterminada es **sleep()**-. Las dos funciones son del módulo time:

sched.scheduler(*timefunc*=time.monotonic, *delayfunc*=time.sleep)

Se pueden utilizar otras funciones para los argumentos de **scheduler()** pero las dos deben trabajar con la misma escala temporal.

```
# Declarar un programador con los valores por defecto:

programador1 = sched.scheduler()
```

Programar eventos y poner en marcha el programador

Para programar los eventos se emplean dos métodos: **enterabs()** y **enter()**. Mientras que el primero define los eventos de un programador para que se ejecuten en un momento específico, el segundo establece la ejecución para después de un tiempo de espera.

```
# Definir un evento para ejecutar en un momento específico

scheduler.enterabs(time, priority, action, argument=(), kwargs={})

# Definir un evento para ejecutar después de un tiempo de espera

scheduler.enter(delay, priority, action, argument=(), kwargs={})
```

Los dos métodos tienen los siguientes argumentos:

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

- **time/delay**: en **entertabs()** define el momento de ejecución del evento y en **enter()** el tiempo de espera. El valor expresado debe ser compatible con el valor devuelto por la función indicada como primer argumento del constructor **scheduler()**.
- **priority**: establece la prioridad de ejecución cuando dos o más eventos tienen el mismo momento de ejecución. Cuanto más bajo es el número mayor prioridad tiene.
- **action**: es la función a ejecutar.
- **argument**: normalmente, es una tupla con los valores de los argumentos de la función de **action**. Opcional.
- **kwargs**: es un diccionario que se puede pasar como argumento de la función de **action**. Opcional.

Los métodos **enterabs()** y **enter()** devuelven un valor que se puede utilizar para cancelar con posterioridad un evento con el método **cancel()**, siempre que el evento no haya iniciado su ejecución o, dicho de otro modo, cuando el evento esté en cola esperando a ejecutarse. El método **empty()** se utiliza para conocer si quedan eventos pendientes. La clase **scheduler** también cuenta con el atributo **queue**, muy útil, para consultar información sobre los eventos pendientes de ejecución.

Después de definir los eventos hay que poner en marcha el programador con el método **run()**. Este método hace que el programador espere (valiéndose de la función de retardo especificada en el método constructor) a que los eventos se vayan ejecutando, uno a uno, en el tiempo establecido y hasta que no quede ninguno pendiente. El método **run()** espera la ejecución porque tiene el argumento **blocking** con el valor **True**, por defecto. Si el valor de este argumento es **False** se ejecutarán sólo los eventos que vencen inmediatamente (por ejemplo, eventos con tiempo de espera en **0**), devolviendo **run()**, si existe, el momento límite de la siguiente ejecución programada. Este argumento está disponible a partir de Python 3.3.

Programar eventos para ejecutar en un momento determinado

En el siguiente ejemplo se declara un programador con dos eventos para ejecutar una tarea 1 segundo después de poner en marcha el programador y la misma tarea cinco segundos después. El programador se crea con la función **time.time()** que devuelve el tiempo expresado en segundos. (Cualquier fecha-hora se puede expresar en segundos y viceversa. Ver: El módulo time)

```
import sched
import time

# Función que es llamada por los eventos
def abrir_cerrar(estado):
    print('TIEMPO:', int(time.time()))
    if estado:
        print("Abriendo compuertas...")
    else:
        print("Cerrando compuertas...")

# Declarar el programador
programador = sched.scheduler(time.time, time.sleep)

# Asignar el tiempo de comienzo (en segundos)
comienzo = int(time.time())
t1 = comienzo + 1   # Tiempo para abrir compuertas (1")
t2 = t1 + 4         # Tiempo para cerrar compuertas (5")
print('PROGRAMADOR INICIADO:', comienzo)

# Definir los dos eventos indicando: momento de ejecución,
# prioridad, función a la que se llama y el valor que se
# pasa al argumento de la función:
programador.enterabs(t1, 1, abrir_cerrar, (1,))
programador.enterabs(t2, 1, abrir_cerrar, (0,))

# Poner en marcha el programador.
# El programa permanece a la espera hasta que se
# ejecuten los dos eventos. La ejecución se producirá
# cuando se alcance el momento definido.
programador.run()
print('PROGRAMADOR FINALIZADO:', int(time.time()))

'''
Salida:

PROGRAMADOR INICIADO: 1489659568
TIEMPO: 1489659569
Abriendo compuertas...
TIEMPO: 1489659573
Cerrando compuertas...
PROGRAMADOR FINALIZADO: 1489659573
'''
```

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

Archivo

marzo 2017 (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

Programar eventos con posibilidad de cancelación

A continuación, un ejemplo que declara un programador con dos eventos de las mismas características que en el ejemplo anterior. En este caso se utiliza el valor de retorno del método **enterabs()** para cancelar el segundo evento cuando el número aleatorio devuelto por la función **randint()** es impar.

```
import sched
import time
import random

# Función que es llamada por los eventos
def abrir_cerrar(estado, mantener):
    print('TIEMPO:', int(time.time()))
    if mantener and not programador.empty():
        programador.cancel(ev2)

    if estado:
        print("Abriendo compuertas...")
    else:
        print("Cerrando compuertas...")

# Obtener un número aleatorio entre 1 y 5
# Si el valor es impar: no cerrar compuertas
valor = random.randint(1, 5)
if valor % 2:
    print("No mantener compuertas abiertas")
    mantener_abiertas = False
else:
    print("Mantener compuertas abiertas")
    mantener_abiertas = True

# Declarar el programador
programador = sched.scheduler(time.time, time.sleep)

# Asignar el tiempo de comienzo (en segundos)
comienzo = int(time.time())
t1 = comienzo + 1   # Tiempo para abrir compuertas
t2 = t1 + 4         # Tiempo para cerrar compuertas
print('PROGRAMADOR INICIADO:', comienzo)

# Definir los dos eventos indicando: momento de ejecución,
# prioridad, función a la que se llama y valores que se
# pasan a los argumentos de la función:
ev1 = programador.enterabs(t1, 1, abrir_cerrar,
                           (1, mantener_abiertas))
ev2 = programador.enterabs(t2, 1, abrir_cerrar,
                           (0, mantener_abiertas))

# Poner en marcha el programador.
# El programa permanece a la espera hasta que se
# ejecuten los dos eventos. La ejecución se producirá
# cuando se alcance el momento definido.
programador.run()
print('PROGRAMADOR FINALIZADO:', int(time.time()))

'''
Salidas posibles:

1) Si el número devuelto por randint() es par

No mantener compuertas abiertas
PROGRAMADOR INICIADO: 1489659900
TIEMPO: 1489659901
Abriendo compuertas...
TIEMPO: 1489659905
Cerrando compuertas...
PROGRAMADOR FINALIZADO: 1489659905

2) Si el número devuelto por randint() es impar

Mantener compuertas abiertas
PROGRAMADOR INICIADO: 1489659972
TIEMPO: 1489659973
Abriendo compuertas...
PROGRAMADOR FINALIZADO: 1489659973

'''
```

Programar eventos para ejecutar después de un tiempo de espera

Seguidamente otro ejemplo con un programador con tres eventos para ejecutar la misma tarea después de esperar 1, 2 y 3 segundos, respectivamente. El programador se crea con **time.monotonic()** que es la función que utiliza por defecto el constructor, que devuelve el tiempo expresado en segundos. En este caso el primer argumento de la función **enter()** representa el valor del tiempo de espera que utiliza la función **sleep()** en el constructor.

```
import sched
import time

def tarea(nombre, comienzo):
    ahora = time.time()
    diferencia = int(ahora - comienzo)
    print('MOMENTO :', int(ahora),
          'Diferencia:', diferencia, 'segundos',
          'Tarea:', nombre)

programador = sched.scheduler()
comienzo = time.time()
print('COMIENZO:', comienzo)

programador.enter(1, 1, tarea, ('TAREA_1', comienzo))
programador.enter(2, 1, tarea, ('TAREA_2', comienzo))
programador.enter(3, 1, tarea, ('TAREA_3', comienzo))
programador.run()

print('FINAL   :', time.time())

'''
Salida:

COMIENZO: 1489579919.1849022
MOMENTO : 1489579920 Diferencia: 1 segundos Tarea: TAREA_1
MOMENTO : 1489579921 Diferencia: 2 segundos Tarea: TAREA_2
MOMENTO : 1489579922 Diferencia: 3 segundos Tarea: TAREA_3
FINAL   : 1489579922.1852293
'''
```

Programar eventos para ejecutar considerando la prioridad

En el siguiente ejemplo se programan tres eventos para ejecutar la misma tarea. Uno de ellos, la ejecuta después de 1 segundo de espera y los otros dos eventos como tienen el mismo tiempo de espera (3 segundos) utilizan el argumento de la prioridad para establecer el orden de ejecución. En este caso el tercero que tiene una prioridad más alta se ejecuta antes que el segundo.

El programador se crea con la función **time.time()** y la función **tiempo()** se utiliza para 'humanizar' la salida del tiempo que se expresa en esta ocasión con el siguiente formato de fecha/hora: '%d-%m-%Y %H:%M:%S'.

```
import sched
import time

def tiempo():
    formato = '%d-%m-%Y %H:%M:%S'

    # Obtiene fecha/hora local como tupla struct_time
    st_tiempo = time.localtime()

    # Convierte fecha/hora a segundos
    tiempo = time.mktime(st_tiempo)

    # Convierte fecha/hora a cadena
    str_tiempo = time.strftime(formato, st_tiempo)
    return tiempo, str_tiempo

def tarea(nombre, comienzo):
    ahora, str_ahora = tiempo()
    diferencia = int(ahora - comienzo)
    print('MOMENTO :', str_ahora,
          'Diferencia:', diferencia, 'segundos',
          'Tarea:', nombre)

programador = sched.scheduler(time.time, time.sleep)
comienzo, str_comienzo = tiempo()
```

```
print('COMIENZO:', str_comienzo)

programador.enter(1, 1, tarea, ('TAREA_1', comienzo))
programador.enter(3, 2, tarea, ('TAREA_2', comienzo))
programador.enter(3, 1, tarea, ('TAREA_3', comienzo))
programador.run()

final, str_final = tiempo()
print('FINAL   :', str_final)

'''
Salida:

COMIENZO: 15-03-2017 11:36:07
MOMENTO : 15-03-2017 11:36:08 Diferencia: 1 segundos Tarea: TAREA_1
MOMENTO : 15-03-2017 11:36:10 Diferencia: 3 segundos Tarea: TAREA_3
MOMENTO : 15-03-2017 11:36:10 Diferencia: 3 segundos Tarea: TAREA_2
FINAL   : 15-03-2017 11:36:10
'''
```

Programar eventos para ejecutar controlando el bloqueo

Para finalizar, un ejemplo en el que se utiliza el argumento **blocking** del método **run()** para bloquear el programa para que espere a que finalice la ejecución de todos los eventos o, bien, para que se ejecuten sólo los eventos de ejecución inmediata (son aquellos que tienen el valor **0** en el tiempo de espera.

En este caso se utiliza también la función **randint()** para generar un número aleatorio. Si el número obtenido es par **blocking** toma el valor **True** y el programa espera la ejecución de todos los eventos. Si el número es impar sólo se ejecutan los eventos inmediatos.

El programador se crea con la función **time.time()** y la función **tiempo()** se utiliza para 'humanizar' la salida del tiempo que se expresa en esta ocasión con el siguiente formato de fecha/hora: '%d-%m-%Y %H:%M:%S'.

```
import sched
import time
import random

def tiempo():
    formato = '%d-%m-%Y %H:%M:%S'

    # Obtiene fecha/hora local como tupla struct_time
    st_tiempo = time.localtime()

    # Convierte fecha/hora a segundos
    tiempo = time.mktime(st_tiempo)

    # Convierte fecha/hora a cadena
    str_tiempo = time.strftime(formato, st_tiempo)
    return tiempo, str_tiempo

def tarea(nombre, comienzo):
    ahora, str_ahora = tiempo()
    diferencia = int(ahora - comienzo)
    print('MOMENTO :', str_ahora,
          'Diferencia:', diferencia, 'segundos',
          'Tarea:', nombre)

# Obtener un número aleatorio entre 1 y 5
# Si el valor es impar: ejecutar todos los eventos
valor = random.randint(1, 5)
if valor % 2:
    print("Ejecutar todos los eventos")
    bloquear = True
else:
    print("Ejecutar sólo eventos inmediatos")
    bloquear = False

programador = sched.scheduler(time.time, time.sleep)
comienzo, str_comienzo = tiempo()
print('COMIENZO:', str_comienzo)

programador.enter(0, 1, tarea, ('TAREA_1', comienzo))
programador.enter(0, 1, tarea, ('TAREA_2', comienzo))
programador.enter(3, 1, tarea, ('TAREA_3', comienzo))
programador.run(blocking=bloquear)

final, str_final = tiempo()
```

```
print('FINAL   :', str_final)

'''
Salidas posibles:

1) Si el número devuelto por randint() es impar:

Ejecutar todos los eventos
COMIENZO: 16-03-2017 12:13:44
MOMENTO : 16-03-2017 12:13:44 Diferencia: 0 segundos Tarea: TAREA_1
MOMENTO : 16-03-2017 12:13:44 Diferencia: 0 segundos Tarea: TAREA_2
MOMENTO : 16-03-2017 12:13:47 Diferencia: 3 segundos Tarea: TAREA_3
FINAL   : 16-03-2017 12:13:47

2) Si el número devuelto por randint() es par:

Ejecutar sólo eventos inmediatos
COMIENZO: 16-03-2017 12:14:11
MOMENTO : 16-03-2017 12:14:11 Diferencia: 0 segundos Tarea: TAREA_1
MOMENTO : 16-03-2017 12:14:11 Diferencia: 0 segundos Tarea: TAREA_2
FINAL   : 16-03-2017 12:14:11
'''
```

Ir al índice del tutorial de Python

Publicado por Pherkad en 5:14

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua