Python | Different ways to kill a Thread

Last Updated: 09-09-2020

In general, killing threads abruptly is considered a bad programming practice. Killing a thread abruptly might leave a critical resource that must be closed properly, open. But you might want to kill a thread once some specific time period has passed or some interrupt has been generated. There are the various methods by which you can kill a thread in python.

- Raising exceptions in a python thread
- Set/Reset stop flag
- Using traces to kill threads
- Using the multiprocessing module to kill threads
- Killing Python thread by setting it as daemon
- Using a hidden function `_stop()`

Raising exceptions in a python thread :
This method uses the function PyThreadState_SetAsyncExc() to raise an exception in the a thread. For Example,

```
# Python program raising
# exceptions in a python
# thread

import threading
import ctypes
import time

class thread_with_exception(threading.Thread):
    def __init__(self, name):
        threading.Thread.__init__(self)
        self.name = name

    def run(self):

        # target function of the thread class
        try:
            while True:
                print('running ' + self.name)
        finally:
            print('ended')

    def get_id(self):

        # returns id of the respective thread
        if hasattr(self, '_thread_id'):
            return self._thread_id
        for id, thread in threading._active.items():
            if thread is self:
                return id

    def raise_exception(self):
        thread_id = self.get_id()
        res = ctypes.pythonapi.PyThreadState_SetAsyncExc(thread_id,
              ctypes.py_object(SystemExit))
        if res > 1:
            ctypes.pythonapi.PyThreadState_SetAsyncExc(thread_id, 0)
            print('Exception raise failure')

t1 = thread_with_exception('Thread 1')
t1.start()
time.sleep(2)
t1.raise_exception()
t1.join()
```

When we run the code above in a machine and you will notice, as soon as the function `raise_exception()` is called, the target function `run()` ends. This is because as soon as an exception is raised, program control jumps out of the `try` block and `run()` function is terminated. After that `join()` function can be called to kill the thread. In the absence of the function `run_exception()`, the target function `run()` keeps running forever and `join()` function is never called to kill the thread.

Set/Reset stop flag :
In order to kill a threads, we can declare a stop flag and this flag will be check occasionally by the thread. For Example

```
# Python program showing
# how to kill threads
# using set/reset stop
# flag

import threading
import time

def run():
    while True:
        print('thread running')
        global stop_threads
        if stop_threads:
            break

stop_threads = False
t1 = threading.Thread(target = run)
t1.start()
time.sleep(1)
stop_threads = True
t1.join()
print('thread killed')
```

In the above code, as soon as the global variable `stop_threads` is set, the target function `run()` ends and the thread `t1` can be killed by using `t1.join()`. But one may refrain from using global variable due to certain reasons. For those situations, function objects can be passed to provide a similar functionality as shown below.

```
# Python program killing
# threads using stop
# flag

import threading
import time

def run(stop):
    while True:
        print('thread running')
        if stop():
                break

def main():
        stop_threads = False
        t1 = threading.Thread(target = run, args =(lambda : stop_thread
        t1.start()
        time.sleep(1)
        stop_threads = True
        t1.join()
        print('thread killed')
main()
```

The function object passed in the above code always returns the value of the local variable `stop_threads`. This value is checked in the function `run()`, and as soon as `stop_threads` is reset, the `run()` function ends and the thread can be killed.

Using traces to kill threads :
This methods works by installing **traces** in each thread. Each trace terminates itself on the detection of some stimulus or flag, thus instantly killing the associated thread. For Example

```
# Python program using
# traces to kill threads

import sys
import trace
import threading
import time
class thread_with_trace(threading.Thread):
  def __init__(self, *args, **keywords):
    threading.Thread.__init__(self, *args, **keywords)
    self.killed = False

  def start(self):
    self.__run_backup = self.run
    self.run = self.__run
    threading.Thread.start(self)

  def __run(self):
    sys.settrace(self.globaltrace)
    self.__run_backup()
    self.run = self.__run_backup

  def globaltrace(self, frame, event, arg):
    if event == 'call':
      return self.localtrace
    else:
      return None

  def localtrace(self, frame, event, arg):
    if self.killed:
      if event == 'line':
        raise SystemExit()
    return self.localtrace

  def kill(self):
    self.killed = True

def func():
  while True:
    print('thread running')

t1 = thread_with_trace(target = func)
t1.start()
time.sleep(2)
t1.kill()
t1.join()
if not t1.isAlive():
  print('thread killed')
```

In this code, `start()` is slightly modified to set the system trace function using settrace(). The local trace function is defined such that, whenever the kill flag (`killed`) of the respective thread is set, a SystemExit exception is raised upon the excution of the next line of code, which end the execution of the target function `func`. Now the thread can be killed with `join()`.

Using the multiprocessing module to kill threads :
The multiprocessing module of Python allows you to spawn processes in the similar way you spawn threads using the threading module. The interface of the multithreading module is similar to that of the threading module. For Example, in a given code we created three threads(processes) which count from 1 to 9.

```
# Python program creating
# three threads

import threading
import time

# counts from 1 to 9
def func(number):
    for i in range(1, 10):
        time.sleep(0.01)
        print('Thread ' + str(number) + ': prints ' + str(number*i))

# creates 3 threads
for i in range(0, 3):
    thread = threading.Thread(target=func, args=(i,))
    thread.start()
```

The functionality of the above code can also be implemented by using the multiprocessing module in a similar manner, with very few changes. See the code given below.

```
# Python program creating
# thread using multiprocessing
# module

import multiprocessing
import time

def func(number):
    for i in range(1, 10):
        time.sleep(0.01)
        print('Processing ' + str(number) + ': prints ' + str(number*i))

for i in range(0, 3):
    process = multiprocessing.Process(target=func, args=(i,))
    process.start()
```

Though the interface of the two modules is similar, the two modules have very different implementations. All the threads share global variables, whereas processes are completely separate from each other. Hence, killing processes is much safer as compared to killing threads. The `Process` class is provided a method, terminate(), to kill a process. Now, getting back to the initial problem. Suppose in the above code, we want to kill all the processes after 0.03s have passed. This functionality is achieved using the multiprocessing module in the following code.

```
# Python program killing
# a thread using multiprocessing
# module

import multiprocessing
import time

def func(number):
    for i in range(1, 10):
        time.sleep(0.01)
        print('Processing ' + str(number) + ': prints ' + str(number*i))

# list of all processes, so that they can be killed afterwards
all_processes = []

for i in range(0, 3):
    process = multiprocessing.Process(target=func, args=(i,))
    process.start()
    all_processes.append(process)

# kill all processes after 0.03s
time.sleep(0.03)
for process in all_processes:
    process.terminate()
```

Though the two modules have different implementations. This functionality provided by the multiprocessing module in the above code is similar to killing threads. Hence, the multiprocessing module can be used as a simple **alternative** whenever we are required to implement the killing of threads in Python.

Killing Python thread by setting it as daemon :
Daemon threads are those threads which are killed when the main program exits. For Example

```
import threading
import time
import sys

def func():
    while True:
        time.sleep(0.5)
        print("Thread alive, and it won't die on program termination")

t1 = threading.Thread(target=func)
t1.start()
time.sleep(2)
sys.exit()
```

Notice that, thread `t1` stays alive and prevents the main program to exit via `sys.exit()`. In Python, any alive non-daemon thread blocks the main program to exit. Whereas, daemon threads themselves are killed as soon as the main program exits. In other words, as soon as the main program exits, all the daemon threads are killed. To declare a thread as daemon, we set the keyword argument, `daemon` as `True`. For Example in the given code it demonstrates the property of daemon threads.

```
# Python program killing
# thread using daemon

import threading
import time
import sys

def func():
    while True:
        time.sleep(0.5)
        print('Thread alive, but it will die on program termination')

t1 = threading.Thread(target=func)
t1.daemon = True
t1.start()
time.sleep(2)
sys.exit()
```

Notice that, as soon as the main program exits, the thread `t1` is killed. This method proves to be extremely useful in cases where program termination can be used to trigger the killing of threads. Note that in Python, the main program terminates as soon as all the non-daemon threads are dead, irrespective of the number of daemon threads alive. Hence, the resources held by these daemon threads, such as open files, database transactions, etc. may not be released properly. The initial thread of control in a python program is not a daemon thread. Killing a thread forcibly is not recommended unless it is known for sure, that doing so will not cause any leaks or deadlocks.

Using a hidden function `_stop()` :
In order to kill a thread, we use hidden function `_stop()` this function is not documented but might disappear in the next version of python.

```
# Python program killing
# a thread using ._stop()
# function

import time
import threading

class MyThread(threading.Thread):

    # Thread class with a _stop() method.
    # The thread itself has to check
    # regularly for the stopped() condition.

    def __init__(self, *args, **kwargs):
        super(MyThread, self).__init__(*args, **kwargs)
        self._stop = threading.Event()

    # function using _stop function
    def stop(self):
        self._stop.set()

    def stopped(self):
        return self._stop.isSet()

    def run(self):
        while True:
            if self.stopped():
                return
            print("Hello, world!")
            time.sleep(1)

t1 = MyThread()

t1.start()
time.sleep(5)
t1.stop()
t1.join()
```

Note: Above methods might not work in some situation or another, because python does not provide any direct method to kill threads.

Recommended Posts:
Python | os.kill() method
Kill a Process by name using Python
kill command in Linux with Examples
Handling a thread's exception in the caller thread in Python
What is thread safe or non-thread safe in PHP ?
Python | Ways to split a string in different ways
Different way to create a thread in Python
Different ways to access Instance Variable in Python
Reverse string in Python (5 different ways)
Print lists in Python (4 Different Ways)
Python | Multiply all numbers in the list (4 different ways)
Extending a list in Python (5 different ways)
Different ways to clear a list in Python
Different ways to Invert the Binary bits in Python
Different ways to convert a Python dictionary to a NumPy array
PyQt5 – Different padding size at different edge of Label
PyQt5 - Setting different toolTip to different item of ComboBox
Thread-based parallelism in Python
Check if a Thread has started in Python
Start and stop a thread in Python

Manthanchauhan
Check out this Author's contributed articles.

Improved By : nandavardhanthupalli

Article Tags : Python Python-multithreading

Previous
Create a new column in Pandas DataFrame based on the existing columns

Next
Python | Printing list vertically

bwarne · 5 months ago
The first method of killing a thread needs a small update. As of Python 3.8.3, the first parameter of PyThreadState_SetAsyncExc is an unsigned long so you'll need to wrap the thread id in ctypes.c_long. eg. thread_id = ctypes.c_long(self.get_id())
Reply · Share ›

Tobias Krabel · a year ago
Thanks for the article!
Unfortunately, your first way of killing a thread doesn't work for me (Python 3.6.8). Could you tell me what python version you are using?
Reply · Share ›

Yan · a year ago
Excellent!
Reply · Share ›

egigoka · a year ago
Wow! It's very cool!
Reply · Share ›