Multithreading All Versions

Threads allow Python programs to handle multiple functions at once as opposed to running a sequence of commands individually. This topic explains the principles behind threading and demonstrates its usage.

Examples

Basics of multithreading

Using the `threading` module, a new thread of execution may be started by creating a new `threading.Thread` and assigning it a function to execute:

```
import threading

def foo():
  print "Hello threading!"

my_thread = threading.Thread(target=foo)
```

The `target` parameter references the function (or callable object) to be run. The thread will not begin execution until `start` is called on the `Thread` object.

```
my_thread.start() # prints 'Hello threading!'
```

Now that `my_thread` has run and terminated, calling `start` again will produce a `RuntimeError` .

Create a Custom Thread Class

Using `threading.Thread` class we can subclass new custom Thread class. we must override `run` method in a subclass.

```
from threading import Thread
import time

class Sleepy(Thread):

    def run(self):
        time.sleep(5)
        print("Hello form Thread")

if __name__ == "__main__":
    t = Sleepy()
    t.start()      # start method automatic call Thread class run method.
    # print 'The main program continues to run in foreground.'
    t.join()
    print("The main program continues to run in the foreground.")
```

Communicating between threads

There are multiple threads in your code and you need to safely communicate between them.

You can use a `Queue` from the `queue` library.

```
from queue import Queue
from threading import Thread

# create a data producer
def producer(output_queue):
    while True:
        data = data_computation()

        output_queue.put(data)

# create a consumer
def consumer(input_queue):
    while True:
        # retrieve data (blocking)
        data = input_queue.get()

        # do something with the data

        # indicate data has been consumed
        input_queue.task_done()
```

Creating producer and consumer threads with a shared queue

```
q = Queue()
```

```
t1 = Thread(target=consumer, args=(q,))
t2 = Thread(target=producer, args=(q,))
t1.start()
t2.start()
```

A very basic example on threading

```
import time
from threading import Thread

def worker(i):
    print("sleeping 10 sec from thread {}".format(i))
    time.sleep(10)
    print("finished sleeping from thread {}".format(i))

for i in range(5):
    t = Thread(target=worker, args=(i,))
    t.start()
```

Initially prints:

```
sleeping 10 sec from thread 0
sleeping 10 sec from thread 1
sleeping 10 sec from thread 2
sleeping 10 sec from thread 3
sleeping 10 sec from thread 4
```

and then 10 seconds later:

```
finished sleeping from thread 0
finished sleeping from thread 1
finished sleeping from thread 2
finished sleeping from thread 3
finished sleeping from thread 4
```

Creating a worker pool

Using `threading` & `queue` :

```
from socket import socket, AF_INET, SOCK_STREAM
from threading import Thread
from queue import Queue

def echo_server(addr, nworkers):
    print('Echo server running at', addr)
    # Launch the client workers
    q = Queue()
    for n in range(nworkers):
        t = Thread(target=echo_client, args=(q,))
        t.daemon = True
        t.start()

    # Run the server
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        q.put((client_sock, client_addr))

echo_server(('',15000), 128)
```

Using `concurrent.futures.Threadpoolexecutor` :

```
from socket import AF_INET, SOCK_STREAM, socket
from concurrent.futures import ThreadPoolExecutor

def echo_server(addr):
    print('Echo server running at', addr)
    pool = ThreadPoolExecutor(128)
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(addr)
    sock.listen(5)
    while True:
        client_sock, client_addr = sock.accept()
        pool.submit(echo_client, client_sock, client_addr)

echo_server(('',15000))
```

```
```


Daemon threads

If you want a thread to run during execution, but not to keep Python alive once the other threads have terminated, you can set the `Thread` 's `daemon` attribute to `True` .

```
my_thread = threading.Thread(target=foo)
my_thread.setDaemon(True)
```

Once set to Daemon, this thread will cease execution once *all non-daemon threads* have terminated. The default value is `False` , so there's no need to `setDaemon(False)` in most cases.

Waiting for threads to finish before continuing

In order to pause execution while child threads complete, you can call the `Thread.join` method. For example:

```
# prepare threads
foo_thread = threading.Thread(target=foo)
bar_thread = threading.Thread(target=bar)
threads = [foo_thread, bar_thread]

# start threads
for t in threads:
  t.start()

# continue main execution...

# wait for each thread to finish
for t in threads:
  t.join()
```

Notice that this will `join` the threads in sequence. If the order of `join` s matters, you must ensure the threads are ordered properly in the list.

Syntax

Parameters

Remarks