Producer Consumer problem

We will solve **Producer Consumer problem** in Python using Python threads. This problem is nowhere as hard as they make it sound in colleges.

Why care about Producer Consumer problem:

- Will help you understand more about concurrency and different concepts of concurrency.
- The concept of Producer Consumer problem is used to some extent in implementing a message queue. And you will surely need message queue at some point of time.

While we use threads, you will learn about the following thread topics:

- **Condition** in threads.
- **wait()** method available on Condition instances.
- **notify()** method available on Condition instances.

I will assume you are comfortable with basics of Threads, race condition and how to prevent race condition i.e using locks.

Quoting Wikipedia:

```
The producer's job is to generate a piece of data, put it into the buffer and start again.
At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time
```

The catch here is "At the same time". So, producer and consumer need to run concurrently. Hence we need separate threads for Producer and Consumer.

```
from threading import Thread

class ProducerThread(Thread):
    def run(self):
        pass

class ConsumerThread(Thread):
    def run(self):
        pass
```

Quoting Wikipedia again:

```
The problem describes two processes, the producer and the consumer, who share a common,
fixed-size buffer used as a queue.
```

So we keep one variable which will be global and will be modified by both Producer and Consumer threads. Producer produces data and adds it to the queue. Consumer consumes data from the queue i.e removes it from the queue.

```
queue = []
```

In first iteration, we will not put fixed-size constraint on queue. We will make it fixed-size once our basic program works.

###Initial buggy program:

```
from threading import Thread, Lock
import time
import random

queue = []
lock = Lock()

class ProducerThread(Thread):
    def run(self):
        nums = range(5) #Will create the list [0, 1, 2, 3, 4]
        global queue
        while True:
            num = random.choice(nums) #Selects a random number from list [0, 1, 2, 3, 4]
            lock.acquire()
            queue.append(num)
            print "Produced", num
            lock.release()
            time.sleep(random.random())


class ConsumerThread(Thread):
    def run(self):
        global queue
        while True:
            lock.acquire()
            if not queue:
                print "Nothing in queue, but consumer will try to consume"
            num = queue.pop(0)
            print "Consumed", num
            lock.release()
            time.sleep(random.random())


ProducerThread().start()
ConsumerThread().start()
```

Run it few times and notice the result. Your program might not end after raising **IndexError**. Use **Ctrl+Z** to terminate.

Sample output:

```
Produced 3
Consumed 3
Produced 4
Consumed 4
Produced 1
Consumed 1
Nothing in queue, but consumer will try to consume
Exception in thread Thread-2:
Traceback (most recent call last):
  File "/usr/lib/python2.7/threading.py", line 551, in __bootstrap_inner
```

```
    self.run()
  File "producer_consumer.py", line 31, in run
    num = queue.pop(0)
IndexError: pop from empty list
```

####Explanation:

- We started one producer thread(hereafter referred as producer) and one consumer thread(hereafter referred as consumer).
- Producer keeps on adding to the queue and consumer keeps on removing from the queue.
- Since queue is a shared variable, we keep it inside lock to avoid race condition.
- At some point, consumer has consumed everything and producer is still sleeping. Consumer tries to consume more but since queue is empty, an **IndexError** is raised.
- But on every execution, before IndexError is raised you will see the print statement telling "Nothing in queue, but consumer will try to consume", which explains why you are getting the error.

We found this implementaion as the wrong behaviour.

####What is the correct behaviour?

When there was nothing in the queue, consumer should have stopped running and waited instead of trying to consume from the queue. And once producer adds something to the queue, there should be a way for it to notify the consumer telling it has added something to queue. So, consumer can again consume from the queue. And thus IndexError will never be raised.

About Condition

- Condition object allows one or more threads to wait until notified by another thread.

And this is exactly what we want. We want consumer to wait when the queue is empty and resume only when it gets notified by the producer. Producer should notify only after it adds something to the queue. So after notification from producer, we can be sure that queue is not empty and hence no error can crop if consumer consumes.

- Condition is always associated with a lock.
- A condition has acquire() and release() methods that call the corresponding methods of the associated lock.

Condition provides acquire() and release() which calls lock's acquire() and release() internally, and so we can replace lock instances with condition instances and our lock behaviour will keep working properly.

Consumer needs to wait using a condition instance and producer needs to notify the consumer using the condition instance too. So, they must use the same condition instance for the wait and notify functionality to work properly.

Let’s rewrite our Consumer and Producer code:

```
from threading import Condition

condition = Condition()

class ConsumerThread(Thread):
    def run(self):
        global queue
        while True:
            condition.acquire()
            if not queue:
                print "Nothing in queue, consumer is waiting"
                condition.wait()
                print "Producer added something to queue and notified the consumer"
            num = queue.pop(0)
            print "Consumed", num
            condition.release()
            time.sleep(random.random())
```

Let’s rewrite Producer code:

```
class ProducerThread(Thread):
    def run(self):
        nums = range(5)
        global queue
        while True:
            condition.acquire()
            num = random.choice(nums)
            queue.append(num)
            print "Produced", num
            condition.notify()
            condition.release()
            time.sleep(random.random())
```

Sample output:

```
Produced 3
Consumed 3
Produced 1
Consumed 1
Produced 4
Consumed 4
Produced 3
Consumed 3
Nothing in queue, consumer is waiting
Produced 2
Producer added something to queue and notified the consumer
Consumed 2
Nothing in queue, consumer is waiting
Produced 2
Producer added something to queue and notified the consumer
```

```
Consumed 2
Nothing in queue, consumer is waiting
Produced 3
Producer added something to queue and notified the consumer
Consumed 3
Produced 4
Consumed 4
Produced 1
Consumed 1
```

####Explanation:

- For consumer, we check if the queue is empty before consuming.
- If yes then call **wait()** on condition instance.
- wait() blocks the consumer and also releases the lock associated with the condition. This lock was held by consumer, so basically consumer loses hold of the lock.
- Now unless consumer is notified, it will not run.
- Producer can acquire the lock because lock was released by consumer.
- Producer puts data in queue and calls notify() on the condition instance.
- Once notify() call is made on condition, consumer wakes up. But waking up doesn't mean it starts executing.
- notify() does not release the lock. Even after notify(), lock is still held by producer.
- Producer explicitly releases the lock by using condition.release().
- And consumer starts running again. Now it will find data in queue and no IndexError will be raised.

###Adding a max size on the queue

Producer should not put data in the queue if the queue is full.

It can be accomplished in the following way:

- Before putting data in queue, producer should check if the queue is full.
- If not, producer can continue as usual.
- If the queue is full, producer must wait. So call **wait()** on condition instance to accomplish this.
- This gives a chance to consumer to run. Consumer will consume data from queue which will create space in queue.
- And then consumer should notify the producer.
- Once consumer releases the lock, producer can acquire the lock and can add data to queue.

Final program looks like:

```
from threading import Thread, Condition
import time
import random

queue = []
```

```
MAX_NUM = 10
condition = Condition()

class ProducerThread(Thread):
    def run(self):
        nums = range(5)
        global queue
        while True:
            condition.acquire()
            if len(queue) == MAX_NUM:
                print "Queue full, producer is waiting"
                condition.wait()
                print "Space in queue, Consumer notified the producer"
            num = random.choice(nums)
            queue.append(num)
            print "Produced", num
            condition.notify()
            condition.release()
            time.sleep(random.random())


class ConsumerThread(Thread):
    def run(self):
        global queue
        while True:
            condition.acquire()
            if not queue:
                print "Nothing in queue, consumer is waiting"
                condition.wait()
                print "Producer added something to queue and notified the consumer"
            num = queue.pop(0)
            print "Consumed", num
            condition.notify()
            condition.release()
            time.sleep(random.random())


ProducerThread().start()
ConsumerThread().start()
```

Sample output:

```
Produced 0
Consumed 0
Produced 0
Produced 4
Consumed 0
Consumed 4
Nothing in queue, consumer is waiting
Produced 4
Producer added something to queue and notified the consumer
Consumed 4
Produced 3
Produced 2
Consumed 3
```