Part 7: Managing Thread Results

Part 7: Managing Thread Results

We need a thread-safe way of storing results from multiple threads of execution. That is provided by the Queue module.

Queues allow multiple producers and multiple consumers to exchange data safely.

Size of the queue is managed with the maxsize kwarg.

It will block consumers if empty and block producers if full.

If maxsize is less than or equal to zero, the queue size is infinite.

```
from Queue import Queue
q = Queue(maxsize=10)
q.put(37337)
block = True
timeout = 2
print(q.get(block, timeout))
```

- http://docs.python.org/3/library/threading.html
- http://docs.python.org/3/library/queue.html

Queues (queue)

- Easier to use than many of the above.
- Do not need locks.
- Have signaling.

Common use: producer/consumer patterns

```
from Queue import Queue
data_q = Queue()

Producer thread:
for item in produce_items():
    data_q.put(item)

Consumer thread:
while True:
    item = q.get()
    consume_item(item)
```

Scheduling (sched)

- Schedules based on time, either absolute or delay.
- Low level, so it has many of the traps of the threading synchronization primitives.

Timed events (threading.timer)

Run a function at some time in the future:

```
import threading
import time

def called_once():
    """
    this function is designed to be called once in the future
    """
    print("I just got called! It's now: {}".format(time.asctime()))

# setting it up to be called
t = Timer(interval=3, function=called_once)
t.start()

# you can cancel it if you want:
t.cancel()
```

See simple_timer.py in your repository.

Other Queue types

`Queue.LifoQueue`

- Last In, First Out

`Queue.PriorityQueue`

- Lowest valued entries are retrieved first

One pattern for `PriorityQueue` is to insert entries of form data by inserting the tuple:

`(priority_number, data)`

Threading example with a queue

See integrate_main.py in your repository.

```
#!/usr/bin/env python

import threading
import queue
import time

from integrate import f, integrate_numpy

def timer(func):
    def wrapper(*arg, **kw):
        ''''
        Secondary source: https://stackoverflow.com/questions/1622943/timeit-versus-timing-
decorator
        Primary source: http://www.daniweb.com/code/snippet368.html
        '''
        t1 = time.time()
        res = func(*arg, **kw)
        t2 = time.time()
        total_time = t2 - t1
        print(f"Total time: {total_time} seconds")
        return res
    return wrapper

@timer
def threading_integrate(f, a, b, N, thread_count=2):
    """break work into N chunks"""
    N_chunk = int(float(N) / thread_count)
    dx = float(b - a) / thread_count
```

```
    results = queue.Queue()

    def worker(*args):
        results.put(integrate_numpy(*args))

    for i in range(thread_count):
        x0 = dx * i
        x1 = x0 + dx
        thread = threading.Thread(target=worker, args=(f, x0, x1, N_chunk))
        thread.start()
        print("Thread %s started" % thread.name)

    return sum((results.get() for i in range(thread_count)))


if __name__ == "__main__":

    # parameters of the integration
    a = 0.0
    b = 10.0
    N = 10**8
    thread_count = 8

    print("Numerical solution with N=%(N)d : %(x)f" %
          {'N': N, 'x': threading_integrate(f, a, b, N, thread_count=thread_count)})
```

‹ Previous Next ›

Threading on a CPU bound problem

Try running the code in integrate_main.py in your repository.

It has a couple of tunable parameters:

```
a = 0.0  # the start of the integration
b = 10.0  # the end point of the integration
```