queue — Thread-Safe FIFO Implementation

Purpose: Provides a thread-safe FIFO implementation

The queue module provides a first-in, first-out (FIFO) data structure suitable for multi-threaded programming. It can be used to pass messages or other data between producer and consumer threads safely. Locking is handled for the caller, so many threads can work with the same Queue instance safely and easily. The size of a Queue (the number of elements it contains) may be restricted to throttle memory usage or processing.

Note

This discussion assumes you already understand the general nature of a queue. If you do not, you may want to read some of the references before continuing.

Basic FIFO Queue

The Queue class implements a basic first-in, first-out container. Elements are added to one “end” of the sequence using put(), and removed from the other end using get().

```
# queue_fifo.py

import queue

q = queue.Queue()

for i in range(5):
    q.put(i)

while not q.empty():
    print(q.get(), end=' ')
print()
```

This example uses a single thread to illustrate that elements are removed from the queue in the same order in which they are inserted.

```
$ python3 queue_fifo.py

0 1 2 3 4
```

LIFO Queue

In contrast to the standard FIFO implementation of Queue, the LifoQueue uses last-in, first-out ordering (normally associated with a stack data structure).

```
# queue_lifo.py

import queue

q = queue.LifoQueue()

for i in range(5):
    q.put(i)

while not q.empty():
    print(q.get(), end=' ')
print()
```

The item most recently put into the queue is removed by get.

```
$ python3 queue_lifo.py
```

```
$ python3 queue_lifo.py

4 3 2 1 0
```

Priority Queue

Sometimes the processing order of the items in a queue needs to be based on characteristics of those items, rather than just the order they are created or added to the queue. For example, print jobs from the payroll department may take precedence over a code listing that a developer wants to print. `PriorityQueue` uses the sort order of the contents of the queue to decide which item to retrieve.

```
# queue_priority.py
```

```
import functools
import queue
import threading


@functools.total_ordering
class Job:

    def __init__(self, priority, description):
        self.priority = priority
        self.description = description
        print('New job:', description)
        return

    def __eq__(self, other):
        try:
            return self.priority == other.priority
        except AttributeError:
            return NotImplemented

    def __lt__(self, other):
        try:
            return self.priority < other.priority
        except AttributeError:
            return NotImplemented


q = queue.PriorityQueue()

q.put(Job(3, 'Mid-level job'))
q.put(Job(10, 'Low-level job'))
q.put(Job(1, 'Important job'))


def process_job(q):
    while True:
        next_job = q.get()
        print('Processing job:', next_job.description)
        q.task_done()


workers = [
    threading.Thread(target=process_job, args=(q,)),
    threading.Thread(target=process_job, args=(q,)),
]
for w in workers:
    w.setDaemon(True)
    w.start()

q.join()
```

This example has multiple threads consuming the jobs, which are processed based on the priority of items in the queue at the time `get()` was called. The order of processing for items added to the queue while the consumer threads are running depends on thread context switching.

```
$ python3 queue_priority.py
```

```
New job: Mid-level job
New job: Low-level job
New job: Important job
Processing job: Important job
Processing job: Mid-level job
Processing job: Low-level job
```

Building a Threaded Podcast Client

The source code for the podcasting client in this section demonstrates how to use the Queue class with multiple threads. The program reads one or more RSS feeds, queues up the enclosures for the five most recent episodes from each feed to be downloaded, and processes several downloads in parallel using threads. It does not have enough error handling for production use, but the skeleton implementation illustrates the use of the queue module.

First, some operating parameters are established. Usually, these would come from user inputs (e.g., preferences or a database). The example uses hard-coded values for the number of threads and list of URLs to fetch.

```
# fetch_podcasts.py

from queue import Queue
import threading
import time
import urllib
from urllib.parse import urlparse

import feedparser

# Set up some global variables
num_fetch_threads = 2
enclosure_queue = Queue()

# A real app wouldn't use hard-coded data...
feed_urls = [
    'http://talkpython.fm/episodes/rss',
]


def message(s):
    print('{}: {}'.format(threading.current_thread().name, s))
```

The function `download_enclosures()` runs in the worker thread and processes the downloads using `urllib`.

```
def download_enclosures(q):
    """This is the worker thread function.
    It processes items in the queue one after
    another.  These daemon threads go into an
    infinite loop, and exit only when
    the main thread ends.
    """
    while True:
        message('looking for the next enclosure')
        url = q.get()
        filename = url.rpartition('/')[-1]
        message('downloading {}'.format(filename))
        response = urllib.request.urlopen(url)
        data = response.read()
        # Save the downloaded file to the current directory
        message('writing to {}'.format(filename))
        with open(filename, 'wb') as outfile:
            outfile.write(data)
        q.task_done()
```

Once the target function for the threads is defined, the worker threads can be started. When `download_enclosures()` processes the statement `url = q.get()`, it blocks and waits until the queue has something to return. That means it is safe to start the threads before there is anything in the queue.

```
# Set up some threads to fetch the enclosures
for i in range(num_fetch_threads):
    worker = threading.Thread(
```

```
        target=download_enclosures,
        args=(enclosure_queue,),
        name='worker-{}'.format(i),
    )
    worker.setDaemon(True)
    worker.start()
```

The next step is to retrieve the feed contents using the `feedparser` module and enqueue the URLs of the enclosures. As soon as the first URL is added to the queue, one of the worker threads picks it up and starts downloading it. The loop continues to add items until the feed is exhausted, and the worker threads take turns dequeuing URLs to download them.

```
# Download the feed(s) and put the enclosure URLs into
# the queue.
for url in feed_urls:
    response = feedparser.parse(url, agent='fetch_podcasts.py')
    for entry in response['entries'][:5]:
        for enclosure in entry.get('enclosures', []):
            parsed_url = urlparse(enclosure['url'])
            message('queuing {}'.format(
                parsed_url.path.rpartition('/')[-1]))
            enclosure_queue.put(enclosure['url'])
```

The only thing left to do is wait for the queue to empty out again, using `join()`.

```
# Now wait for the queue to be empty, indicating that we have
# processed all of the downloads.
message('*** main thread waiting')
enclosure_queue.join()
message('*** done')
```

Running the sample script produces output similar to the following.

```
$ python3 fetch_podcasts.py

worker-0: looking for the next enclosure
worker-1: looking for the next enclosure
MainThread: queuing turbogears-and-the-future-of-python-web-frameworks.mp3
MainThread: queuing continuum-scientific-python-and-the-business-of-open-source.mp3
MainThread: queuing openstack-cloud-computing-built-on-python.mp3
MainThread: queuing pypy.js-pypy-python-in-your-browser.mp3
MainThread: queuing machine-learning-with-python-and-scikit-learn.mp3
MainThread: *** main thread waiting
worker-0: downloading turbogears-and-the-future-of-python-web-frameworks.mp3
worker-1: downloading continuum-scientific-python-and-the-business-of-open-source.mp3
worker-0: looking for the next enclosure
worker-0: downloading openstack-cloud-computing-built-on-python.mp3
worker-1: looking for the next enclosure
worker-1: downloading pypy.js-pypy-python-in-your-browser.mp3
worker-0: looking for the next enclosure
worker-0: downloading machine-learning-with-python-and-scikit-learn.mp3
worker-1: looking for the next enclosure
worker-0: looking for the next enclosure
MainThread: *** done
```

The actual output will depend on the contents of the RSS feed used.

See also

- Standard library documentation for queue
- deque — Double-Ended Queue from collections
- Queue data structures – Wikipedia article explaining queues.
- FIFO – Wikipedia article explaining first-in, first-out, data structures.
- feedparser module – A module for parsing RSS and Atom feeds, created by Mark Pilgrim and maintained by Kurt McKee.

bisect — Maintain Lists in Sorted Order

struct — Binary Data Structures

Quick Links
Basic FIFO Queue
LIFO Queue
Priority Queue
Building a Threaded Podcast Client

This page was last updated 2017-01-28.

Navigation
bisect — Maintain Lists in Sorted Order
struct — Binary Data Structures

The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?