RateLimitQueue Documentation

Release 0.1.0

John Paton

Aug 21, 2018

Contents

A rate limited wrapper for Python's thread safe queues.

Some external APIs have rate limits that allow faster-than-consecutive queries, e.g. if the rate limit is very high or the API response is quite slow. To make the most of the API, the best option is often to make API calls from multiple threads. Then you can put the requests or URLs to call in a `queue.Queue` and have the threads consume the URLs as they make the calls. However, you still have to make sure that the total calls from all your threads don't exceed the rate limit, which requires some nontrivial coordination.

The `ratelimitqueue` package extends the three built-in Python queues from from `queue` package - `Queue`, `LifeQueue`, and `PriorityQueue` - with configurable, rate limited counterparts. Specifically, the `get()` method is rate limited across all threads so that workers can safely consume from the queue in an unlimited loop, and putting the items in the queue doesn't need to require blocking the main thread.

Contents

CHAPTER 1

ratelimitqueue package

1.1 Package Contents

class `ratelimitqueue.`**`RateLimitQueue`**(*maxsize=0*, *calls=1*, *per=1.0*, *fuzz=0*)

A thread safe queue with a given maximum size and rate limit.

If *maxsize* is <= 0, the queue size is infinite (see *queue.Queue*).

The rate limit is described as *calls per* time window, with *per* measured in seconds. The default rate limit is 1 call per second. If *per* is <= 0, the rate limit is infinite.

To avoid immediately getting the maximum allowed items at startup, an extra randomized wait period can be configured with *fuzz*. This will cause the RateLimitQueue to wait between 0 and *fuzz* seconds before getting the object in the queue. Fuzzing only occurs if there is no rate limit waiting to be done.

Parameters

- **`maxsize`** (*`int, optional, default 0`*) – The number of slots in the queue, <=0 for infinite.
- **`calls`** (*`int, optional, default 1`*) – The number of call per time unit *per*. Must be at least 1.
- **`per`** (*`float, optional, default 1.0`*) – The time window for tracking calls, in seconds, <=0 for infinite rate limit.
- **`fuzz`** (*`float, options, default 0`*) – The maximum length (in seconds) of fuzzed extra sleep, <=0 for no fuzzing

Examples

Basic usage:

```
>>> rlq = RateLimitQueue()
>>> rlq.put(1)
>>> rlq.put(2)
>>> rlq.get()
1
>>> rlq.get()
2
```

A rate limit of 3 calls per 5 seconds:

```
>>> rlq = RateLimitQueue(calls=3, per=5)
```

A queue with the default 1 call per second, with a maximum size of 3:

```
>>> rlq = RateLimitQueue(3)
```

A queue of infinite size and rate limit, equivalent to queue.Queue():

```
>>> rlq = RateLimitQueue(per=0)
```

A queue with wait time fuzzing up to 1 second so that the queue cannot be filled immediately directly after instantiation:

```
>>> rlq = RateLimitQueue(fuzz=1)
```

`empty`()

Return True if the queue is empty, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() == 0 as a direct substitute, but be aware that either approach risks a race condition where a queue can grow before the result of empty() or qsize() can be used.

To create code that needs to wait for all queued tasks to be completed, the preferred technique is to use the join() method.

`full`()

Return True if the queue is full, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() >= n as a direct substitute, but be aware that either approach risks a race condition where a queue can shrink before the result of full() or qsize() can be used.

`get`(*block=True*, *timeout=None*)

Get an item from the queue.

If optional args *block* is True and *timeout* is None (the default), block if necessary until a free slot is available and the rate limit has not been reached. If *timeout* is a non-negative number, it blocks at most *timeout* seconds and raises the RateLimitException if the required rate limit waiting time is shorter than the given timeout, or the Empty exception if no item was available within that time.

Otherwise (*block* is False), get an item on the queue if an item is immediately available and the rate limit has not been hit. Else raise the RateLimitException if waiting on the rate limit, or Empty exception if there is no item available in the queue. Timeout is ignored in this case.

Parameters

- **`block`** (`bool, optional, default True`) – Whether to block until an item can be gotten from the queue

- **timeout** (*float, optional, default None*) – The maximum amount of time to block for

get_nowait()
Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

join()
Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

put(*item*, *block=True*, *timeout=None*)
Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

put_nowait(*item*)
Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

qsize()
Return the approximate size of the queue (not reliable!).

task_done()
Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

class ratelimitqueue.**RateLimitLifoQueue**(*maxsize=0*, *calls=1*, *per=1.0*, *fuzz=0*)
A thread safe LIFO queue with a given maximum size and rate limit.

If *maxsize* is <= 0, the queue size is infinite (see *queue.LifoQueue*).

The rate limit is described as *calls per* time window, with *per* measured in seconds. The default rate limit is 1 call per second. If *per* is <= 0, the rate limit is infinite.

To avoid immediately filling the whole queue at startup, an extra randomized wait period can be configured with *fuzz*. This will cause the RateLimitQueue to wait between 0 and *fuzz* seconds before putting the object in the queue. Fuzzing only occurs if there is no rate limit waiting to be done.

Parameters

- **maxsize** (*int, optional, default 0*) – The number of slots in the queue, <=0 for infinite.
- **calls** (*int, optional, default 1*) – The number of call per time unit *per*. Must be at least 1.

- **per** (*float, optional, default 1.0*) – The time window for tracking calls, in seconds, <=0 for infinite rate limit.
- **fuzz** (*float, options, default 0*) – The maximum length (in seconds) of fuzzed extra sleep, <=0 for no fuzzing

Examples

Basic usage:

```
>>> rlq = RateLimitLifoQueue()
>>> rlq.put(1)
>>> rlq.put(2)
>>> rlq.get()
2
>>> rlq.get()
1
```

A rate limit of 3 calls per 5 seconds:

```
>>> rlq = RateLimitLifoQueue(calls=3, per=5)
```

A queue with the default 1 call per second, with a maximum size of 3:

```
>>> rlq = RateLimitLifoQueue(3)
```

A queue of infinite size and rate limit, equivalent to queue.Queue():

```
>>> rlq = RateLimitLifoQueue(per=0)
```

A queue with wait time fuzzing up to 1 second so that the queue cannot be filled immediately directly after instantiation:

```
>>> rlq = RateLimitLifoQueue(fuzz=1)
```

empty()

Return True if the queue is empty, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() == 0 as a direct substitute, but be aware that either approach risks a race condition where a queue can grow before the result of empty() or qsize() can be used.

To create code that needs to wait for all queued tasks to be completed, the preferred technique is to use the join() method.

full()

Return True if the queue is full, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() >= n as a direct substitute, but be aware that either approach risks a race condition where a queue can shrink before the result of full() or qsize() can be used.

get(*block=True*, *timeout=None*)

Get an item from the queue.

If optional args *block* is True and *timeout* is None (the default), block if necessary until a free slot is available and the rate limit has not been reached. If *timeout* is a non-negative number, it blocks at most *timeout* seconds and raises the RateLimitException if the required rate limit waiting time is shorter than the given timeout, or the Empty exception if no item was available within that time.

Otherwise (*block* is False), get an item on the queue if an item is immediately available and the rate limit has not been hit. Else raise the RateLimitException if waiting on the rate limit, or Empty exception if there is no item available in the queue. Timeout is ignored in this case.

Parameters

- **`block`** (*`bool, optional, default True`*) – Whether to block until an item can be gotten from the queue
- **`timeout`** (*`float, optional, default None`*) – The maximum amount of time to block for

`get_nowait`()

Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

`join`()

Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

`put`(*item*, *block=True*, *timeout=None*)

Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

`put_nowait`(*item*)

Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

`qsize`()

Return the approximate size of the queue (not reliable!).

`task_done`()

Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

`class` `ratelimitqueue.`**`RateLimitPriorityQueue`**(*maxsize=0*, *calls=1*, *per=1.0*, *fuzz=0*)

A thread safe priority queue with a given maximum size and rate limit.

Prioritized items should be tuples of form (priority, item), with priority lowest first. Priority determines the order of items returned by get().

If *maxsize* is <= 0, the queue size is infinite (see *queue.LifoQueue*).

The rate limit is described as *calls per* time window, with *per* measured in seconds. The default rate limit is 1 call per second. If *per* is <= 0, the rate limit is infinite.

To avoid immediately filling the whole queue at startup, an extra randomized wait period can be configured with *fuzz*. This will cause the RateLimitQueue to wait between 0 and *fuzz* seconds before putting the object in the queue. Fuzzing only occurs if there is no rate limit waiting to be done.

Parameters

- **`maxsize`** (*`int, optional, default 0`*) – The number of slots in the queue, <=0 for infinite.
- **`calls`** (*`int, optional, default 1`*) – The number of call per time unit *per*. Must be at least 1.
- **`per`** (*`float, optional, default 1.0`*) – The time window for tracking calls, in seconds, <=0 for infinite rate limit.
- **`fuzz`** (*`float, options, default 0`*) – The maximum length (in seconds) of fuzzed extra sleep, <=0 for no fuzzing

Examples

Basic usage:

```
>>> rlq = RateLimitPriorityQueue()
>>> rlq.put((2, 'second'))
>>> rlq.put((1, 'first'))
>>> rlq.get()
(1, 'first')
>>> rlq.get()
(2, 'second')
```

A rate limit of 3 calls per 5 seconds:

```
>>> rlq = RateLimitPriorityQueue(calls=3, per=5)
```

A queue with the default 1 call per second, with a maximum size of 3:

```
>>> rlq = RateLimitPriorityQueue(3)
```

A queue of infinite size and rate limit, equivalent to queue.Queue():

```
>>> rlq = RateLimitPriorityQueue(per=0)
```

A queue with wait time fuzzing up to 1 second so that the queue cannot be filled immediately directly after instantiation:

```
>>> rlq = RateLimitPriorityQueue(fuzz=1)
```

`empty``()`

Return True if the queue is empty, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() == 0 as a direct substitute, but be aware that either approach risks a race condition where a queue can grow before the result of empty() or qsize() can be used.

To create code that needs to wait for all queued tasks to be completed, the preferred technique is to use the join() method.

full()

Return True if the queue is full, False otherwise (not reliable!).

This method is likely to be removed at some point. Use qsize() >= n as a direct substitute, but be aware that either approach risks a race condition where a queue can shrink before the result of full() or qsize() can be used.

get(*block=True*, *timeout=None*)

Get an item from the queue.

If optional args *block* is True and *timeout* is None (the default), block if necessary until a free slot is available and the rate limit has not been reached. If *timeout* is a non-negative number, it blocks at most *timeout* seconds and raises the RateLimitException if the required rate limit waiting time is shorter than the given timeout, or the Empty exception if no item was available within that time.

Otherwise (*block* is False), get an item on the queue if an item is immediately available and the rate limit has not been hit. Else raise the RateLimitException if waiting on the rate limit, or Empty exception if there is no item available in the queue. Timeout is ignored in this case.

Parameters

- **block** (*bool, optional, default True*) – Whether to block until an item can be gotten from the queue
- **timeout** (*float, optional, default None*) – The maximum amount of time to block for

get_nowait()

Remove and return an item from the queue without blocking.

Only get an item if one is immediately available. Otherwise raise the Empty exception.

join()

Blocks until all items in the Queue have been gotten and processed.

The count of unfinished tasks goes up whenever an item is added to the queue. The count goes down whenever a consumer thread calls task_done() to indicate the item was retrieved and all work on it is complete.

When the count of unfinished tasks drops to zero, join() unblocks.

put(*item*, *block=True*, *timeout=None*)

Put an item into the queue.

If optional args 'block' is true and 'timeout' is None (the default), block if necessary until a free slot is available. If 'timeout' is a non-negative number, it blocks at most 'timeout' seconds and raises the Full exception if no free slot was available within that time. Otherwise ('block' is false), put an item on the queue if a free slot is immediately available, else raise the Full exception ('timeout' is ignored in that case).

put_nowait(*item*)

Put an item into the queue without blocking.

Only enqueue the item if a free slot is immediately available. Otherwise raise the Full exception.

qsize()

Return the approximate size of the queue (not reliable!).

task_done()

Indicate that a formerly enqueued task is complete.

Used by Queue consumer threads. For each get() used to fetch a task, a subsequent call to task_done() tells the queue that the processing on the task is complete.

If a join() is currently blocking, it will resume when all items have been processed (meaning that a task_done() call was received for every item that had been put() into the queue).

Raises a ValueError if called more times than there were items placed in the queue.

1.2 Submodules

1.2.1 ratelimitqueue.exceptions module

exception `ratelimitqueue.exceptions.`**`RateLimitException`**

Bases: `Exception`

1.2.2 ratelimitqueue.ratelimitqueue module

class `ratelimitqueue.ratelimitqueue.`**`RateLimitGetMixin`**

Adds rate limiting to another class' *get()* method.

Assumes that the class being extended has properties *per* (float), *fuzz* (float), and *_call_log* (queue.Queue), else will raise AttributeError on call of put().

`get`(*block=True*, *timeout=None*)

Get an item from the queue.

If optional args *block* is True and *timeout* is None (the default), block if necessary until a free slot is available and the rate limit has not been reached. If *timeout* is a non-negative number, it blocks at most *timeout* seconds and raises the RateLimitException if the required rate limit waiting time is shorter than the given timeout, or the Empty exception if no item was available within that time.

Otherwise (*block* is False), get an item on the queue if an item is immediately available and the rate limit has not been hit. Else raise the RateLimitException if waiting on the rate limit, or Empty exception if there is no item available in the queue. Timeout is ignored in this case.

Parameters

- **`block`** (*`bool, optional, default True`*) – Whether to block until an item can be gotten from the queue
- **`timeout`** (*`float, optional, default None`*) – The maximum amount of time to block for

CHAPTER 2

Installation

To install `ratelimitqueue`, clone the repository and install with `pip`:

```
pip install ratelimitqueue
```

Python Module Index

Index