**http://john16blog.blogspot.ru/2012/05/python-queue.html**

[**Python: модуль Queue. Перевод документации. Примеры работы с очередями**](http://john16blog.blogspot.ru/2012/05/python-queue.html)

Модуль Queue реализует механизм очереди. Он особенно полезен в многопоточном программировании, когда необходимо безопасно передавать информацию между потоками. Класс Queue этого модуля реализует всю необходимую семантику блокировки.

Типы очередей

Модуль предоставляет реализации трех типов очередей, единственная разница которых это порядок получаемых значений.

class Queue.**Queue**(*maxsize*)

Класс реализующий очередь FIFO (First Input First Output - первым вошел, первым вышел). *maxsize* - параметр типа integer, который устанавливает предел для числа элементов, которые могут быть помещены в очередь. Вставка новых элементов блокируется, как только этот размер был достигнут, до тех пор пока элементы не будут удалены из очереди. Если значение параметра равно или меньше нуля, то очередь будет бесконечной.

class Queue.**LifoQueue**(*maxsize*)

Класс реализующий очередь LIFO, или по другому "стэк"(Last Input First Output - последним вошел, первым вышел). Параметр *maxsize* аналогичен параметру в классе Queue.Queue.

class Queue.**PriorityQueue**(*maxsize*)

Класс реализующий очередь с приоритетами. Параметр *maxsize* аналогичен параметру в классе Queue.Queue.   
  
Элементы, добавляемые в подобную очередь должны представлять из себя кортеж типа (значение приоритета, данные). Первыми из очереди забираются элементы с меньшим приоритетом, полученным с помощью функции sorted().

Исключения

В модуле определены следующие исключения:

exception Queue.**Empty**

Возбуждается когда вызывается не блокирующий метод get() (или get\_nowait()) пустого объекта Queue.

exception Queue.**Full**

Возбуждается когда вызывается не блокирующий метод put() (или put\_nowait()) заполненного объекта Queue.

Методы объектов

Классы Queue, LifoQueue, PriorityQueue предоставляют следующие методы:   
  
Queue.**qsize**()

Возвращаем аппроксимированный размер очереди. Однако, qsize()>0 не гарантирует, что последующий get() не будет блокирован и также qsize()<maxsize не будет гарантировать что put() не будет блокирован.

Queue.**empty**()

Возвращает True если очередь пуста, False в противном случае. Однако если метод возвращает True не гарантируется что следующий вызов put() не будет блокирован. Также если возвращает False не гарантируется что последующий get() не будет блокирован.

Queue.**full**()

Возвращает True если очередь заполнена, False в противном случае. Однако если метод возвращает True не гарантируется что следующий вызов get() не будет блокирован. Также если возвращает False не гарантируется что последующий put() не будет блокирован.

Queue.**put**(*item, [block[, timeout]]*)

Помещает объект item в очередь. Если block=True и timeout не задан (None по умолчанию), то при необходимости произойдет блокировка до тех пор пока в очереди не будет доступного места. Если timeout это положительное число, то произойдет блокировка на заданное число секунд и будет возбужденно исключение Queue.Full если в течении этого времени не освободится место в очереди. В другом случае (если block=False), метод помещает объект item в очередь если свободное место доступно немедленно, иначе генерирует исключение Queue.Full (параметр timeout в этом случае игнорируется).

Queue.**put\_nowait**(*item*)

Эквивалент вызову put(item, False)

Queue.**get**(*[block, [timeout]]*)

Удалить и возвратить элемент из очереди. Если block=True и timeout не задан (None по умолчанию), произойдет блокировка до тех пор пока элемент не будет доступен. Если timeout это положительное число, то произойдет блокировка на заданное число секунд и будет возбужденно исключение Queue.Empty если в течении этого времени элемент не будет дсотупен. В другом случае (если block=False), метод возвращает элемент очереди немедленно, иначе генерирует исключение Queue.Empty (параметр timeout в этом случае игнорируется).

Queue.**get\_nowait**()

Эквивалент вызову get(False).

Следующие два метода предлагаются для поддержки и отслеживания были ли ставшие в очередь задачи полностью обработаны потребительскими потоками.

Queue.**task\_done**()

Сообщает что ранее полученное задание из очереди выполнено. Используется потребительскими потоками. Для каждого вызова get() последующий вызов task\_done() сообщает объекту очереди что обработка задания завершена.   
  
Если количество вызовов task\_done() превзойдет количество вызовов put(), то будет возбуждено исключение ValueError.

Queue.**join**()

Вызов блокировки до тех пор, пока все элементы из очереди не будут полученны и обработаны.   
  
Количество не выполненных заданий возрастает тогда, когда добавляются элементы в очередь, а уменьшается когда потребительский поток вызывает task\_done().   
  
Если join() находится в заблокировано состоянии, то это означает, что количество вызванных put() не соответствует количеству вызванных task\_done() и задания из очереди не обработаны. Блокировка снимется тогда когда количество вызовов сравняется.

Примеры

Все приведенные ниже примеры являются демонстрациями использования технологии очередей, в коде отстутвуют необходимые для реального приложения обработки ошибок и т.п. избыточные для примеров конструкции.

Пример 1

Основа примера была взята из официальной документации.

# -\*- coding: utf-8 -\*-  
from Queue import Queue  
from threading import Thread  
  
# количество потоков обслуживающих очередь  
num\_worker\_threads=2  
  
def do\_work(item):  
    """  
    Функция иммитирующая полезную работу  
    """  
  
    s=str(item)  
    print s[::-1]  
  
def worker():  
    """  
    Основной код здесь  
    """  
  
    while True:  
        # Получаем задание из очереди  
        item = q.get()  
        do\_work(item)  
        # Сообщаем о выполненном задании  
        q.task\_done()  
  
def source():  
    """  
    Функция генерирующая данные для очереди  
    """  
  
    for i in xrange(100, 105):  
        yield i  
      
# Создаем FIFO очередь  
q = Queue()  
# Создаем и запускаем потоки, которые будут обслуживать очередь  
for i in range(num\_worker\_threads):  
    t = Thread(target=worker)  
    t.setDaemon(True)  
    t.start()  
  
# Заполняем очередь заданиями   
for item in source():  
    q.put(item)  
  
# Ставим блокировку до тех пор пока не будут выполнены все задания  
q.join()

Результат работы будет выглядеть следующим образом:

001  
101  
201  
301  
401

Пример 2

Данный пример иллюстрирует работу очереди бесконечной длины, в процессе обработки заданий которой существует возможность возникновения ошибки, которую необходимо разрешить. Для этого при возбуждении исключительной ситуации, поток обрабатывающий задание вновь помещает его в очередь, а сам принимается за обработку следующего задания из очереди. Пример снабжен богатым выводом отладочной информации для того что бы детально проследить за ходом выполнения кода программы (в тексте не приведен, выполните код самостоятельно для ознакомления с выводом).

# -\*- coding: utf-8 -\*-  
import Queue, threading, time, random, datetime, sys  
  
# переменная для имитации разовой ошибки  
err=False  
  
class Worker(threading.Thread):  
    """  
    Класс потока который будет брать задачи из очереди и выполнять их до успешного   
    окончания или до исчерпания лимита попыток  
    """  
    def \_\_init\_\_(self, queue, output):  
        # Обязательно инициализируем супер класс (класс родитель)  
        super(Worker,self).\_\_init\_\_()  
        # Устанавливаем поток в роли демона, это необходимо что бы по окончании выполнения  
        # метода run() поток корректно завершил работу,а не остался висеть в ожидании  
        self.setDaemon(True)  
        # экземпляр класса содержит в себе очередь что бы при выполнении потока иметь к ней доступ  
        self.queue=queue  
        self.output=output  
      
    def run(self):  
        """  
        Основной код выполнения потока должен находиться здесь  
        """  
        while True:  
            try:  
                # переменная для иммитации единичной ошибки во время выполнения потока  
                global err  
                # фиксируем время начала работы потока  
                start=datetime.datetime.now().strftime('%H:%M:%S')  
                # запрашиваем из очереди объект  
                target=self.queue.get(block=False)  
                print '%s get target: %s'%(self.getName(), target)  
                  
                # эмулируем однократно возникающую ошибку  
                if ((target==2) and (not err)):  
                    err=True  
                    raise Exception('test error')  
                  
                # делаем видимость занятости потока  
                # путем усыпления его на случайную величину  
                sleep\_time=random.randint(0,10)  
                time.sleep(sleep\_time)  
                print '%s %s target: %s sleep %ss'%(start, self.getName(), target, sleep\_time)  
                # сообщаем о том что задача для полученного объекта из очереди выполнена  
                self.output.put(target, block=False)  
                self.queue.task\_done()  
            # После того как очередь опустеет будет сгенерировано исключение  
            except Queue.Empty:  
                sys.stderr.write('%s get Queue.EMPTY exception\r\n'%self.getName())  
                break  
            # если при выполнении потока будет сгенерировано исключение об ошибке,  
            # то оно будет обработано ниже  
            except Exception, e:  
                self.queue.task\_done()  
                # выводим на экран имя потока и инфо об ошибке  
                sys.stderr.write('%s get %s exception\r\n'%(self.getName(), e))  
                # Предполагаем раз объект из очереди не был корреткно обработан,  
                # то добавляем его в очередь  
                self.queue.put(target, block=False)  
  
class Test(object):  
    def \_\_init\_\_(self, data, number\_threads):  
        # создаем экземпля класса очереди Queue  
        self.queue=Queue.Queue()  
        self.output=Queue.Queue()  
        # заполняем очередь  
        for item in data:  
            self.queue.put(item)  
        # определяем количество потоков которые будут обслуживать очередь  
        self.NUMBER\_THREADS=number\_threads  
        # список экземпляров класса потока, в последствии можно  
        # обратиться к нему что бы получать сведения о состоянии потоков  
        self.threads=[]  
          
    def execute(self):  
        # создаем экземпляра классов потоков и запускаем их  
        for i in xrange(self.NUMBER\_THREADS):  
            self.threads.append(Worker(self.queue, self.output))  
            self.threads[-1].start()  
          
        # Блокируем выполнение кода до тех пор пока не будут выполнены все   
        # элементы очереди. Это означает что сколкьо раз были добавлены элементы  
        # очереди, то столько же раз должен быть вызван task\_done().  
        self.queue.join()  
  
t=datetime.datetime.now()  
test=Test(range(100), 20)  
test.execute()  
print 'the end in %s'%(datetime.datetime.now()-t)  
# вывод debug информации   
print len(list(test.output.\_\_dict\_\_['queue']))  
print sorted(list(test.output.\_\_dict\_\_['queue']))

Пример 3

Данный пример симулирует процесс выдачи зарплаты сотрудникам предприятия. Группа работников предприятия стоят в очереди за зарплатой. Особенность в том что ЗП выдают приоритетно занимаемой должности.

# -\*- coding: utf-8 -\*-  
import Queue, threading, time, random, datetime  
  
workers=[(0, u'Директор'),  
         (1, u'Бухгалтер 1'),  
         (1, u'Бухгалтер 2'),  
         (50, u'Начальник IT отдела'),  
         (60, u'Главный программист'),  
         (70, u'Программист'),  
         (75, u'Дизайнер'),  
         (99, u'Уборщик')]  
  
# Перемешиваем список сотрудников в случайном порядке  
random.shuffle(workers)  
  
def worker(queue):  
    """  
    основной код находится здесь.  
    """  
    while True:  
        # получаем задание из очереди в виде кортежа (приоритет, должность)  
        job = queue.get()  
        # Выводим на экран информацию о начале обслуживания сотрудника  
        print u'[+]%s начал обслуживаться в %s'%(job[1], datetime.datetime.now().strftime('%H:%M:%S'))  
        # Шутки ради, предполагаем что приоритет явно связан с размером зарплаты и исходя из этого, затраченному времени на выдачу денег.  
        time.sleep((100-job[0])/10)  
        # Выводим информацию об окончании обслуживания сотрудника  
        print u'[-]%s кончил обслуживаться в %s'%(job[1], datetime.datetime.now().strftime('%H:%M:%S'))  
        # Сообщаем что задание выполнено  
        queue.task\_done()  
  
# Создаем приоритетную очередь и наполняем ее заданиями  
q = Queue.PriorityQueue()  
for item in workers:  
    q.put(item)  
  
# Создаем 2 потока которые будут обслуживать очередь.  
for i in range(2):  
    # В окнструктор поткоа передаем функцию которую будет выполнять поток при вызове метода start(), а также последовательность аргументов которые будут переданы функции worker.  
    t = threading.Thread(target=worker, args=(q,))  
    t.setDaemon(True)  
    t.start()  
  
# Блокируем выполнение программы до выполнения всех заданий в очереди  
q.join()

Результат будет примерно таким:

[+]Директор начал обслуживаться в 12:38:20  
[+]Бухгалтер 1 начал обслуживаться в 12:38:20  
[-]Бухгалтер 1 кончил обслуживаться в 12:38:29  
[+]Бухгалтер 2 начал обслуживаться в 12:38:29  
[-]Директор кончил обслуживаться в 12:38:30  
[+]Начальник IT отдела начал обслуживаться в 12:38:30  
[-]Начальник IT отдела кончил обслуживаться в 12:38:35  
[+]Главный программист начал обслуживаться в 12:38:35  
[-]Бухгалтер 2 кончил обслуживаться в 12:38:38  
[+]Программист начал обслуживаться в 12:38:38  
[-]Главный программист кончил обслуживаться в 12:38:39  
[+]Дизайнер начал обслуживаться в 12:38:39  
[-]Программист кончил обслуживаться в 12:38:41  
[-]Дизайнер кончил обслуживаться в 12:38:41  
[+]Уборщик начал обслуживаться в 12:38:41  
[-]Уборщик кончил обслуживаться в 12:38:41

Пример 4

В этом примере симулируется ситуация когда перед выдачей зарплаты в очереди всего 3 сотрудника, а остальные подходят позднее. Очередь по прежнему приоритетная. Этот пример показателен тем, что Уборщик, изначально находясь в очереди, тем не менее получит зарплату самым последним - так как всегда в очереди будет кто то с значением приоритета ниже чем у него.

# -\*- coding: utf-8 -\*-  
import Queue, threading, time  
  
# Список работников  
workers=[(99, u'Уборщик'),  
         (75, u'Дизайнер'),  
         (70, u'Программист'),  
         (50, u'Начальник IT отдела'),  
         (0, u'Директор'),  
         (60, u'Главный программист')]  
  
def worker(queue):  
    """  
    основной код здесь  
    """  
    while True:  
        # получаем задание из очереди  
        job = queue.get()  
        # выводим сообщением о начале обслуживания сотруника и о тех кто находится в очереди  
        print u'Обслуживаю %s, в очереди %s'%(job[1],  
                                             ','.join([x[1] for x in queue.\_\_dict\_\_['queue']]))  
        # Иммитируем занятость  
        time.sleep(1)  
        # Сообщаем о выполнении задачи  
        queue.task\_done()  
  
# Создаем приоритетную очередь и заполняем ее первыми тремя сотрудниками из общего списка  
q = Queue.PriorityQueue()  
for item in workers[:3]:  
    q.put(item)  
  
# Создаем и запускаем потоки обслуживающие очередь  
for i in range(1):  
    t = threading.Thread(target=worker, args=(q,))  
    t.setDaemon(True)  
    t.start()  
  
# постепенно добавляем в очередь сотрудников   
for item in workers[3:]:  
    q.put(item)  
    time.sleep(1)  
  
# Блокируем выполнение программы до выполнения всех заданий в очереди      
q.join()

В результате вывод будет следующим:

Обслуживаю Программист, в очереди Дизайнер,Уборщик  
Обслуживаю Директор, в очереди Начальник IT отдела,Уборщик,Дизайнер  
Обслуживаю Начальник IT отдела, в очереди Главный программист,Уборщик,Дизайнер  
Обслуживаю Главный программист, в очереди Дизайнер,Уборщик  
Обслуживаю Дизайнер, в очереди Уборщик  
Обслуживаю Уборщик, в очереди

Пример 5

Данный пример иллюстрирует распространенную ситуацию когда нужно распараллелить обработку элементов массива, при этом получить результат в котором положение элементов будут соответствовать изначальному т.е. элементы позиционно не перемешаются.

# -\*- coding: utf-8 -\*-  
import Queue, threading, time, random, datetime  
  
# подгатавливаем список данных для очереди  
data=range(100,1000,125)  
  
class Worker(threading.Thread):  
    # Аттрибут класса, счетчик, являющийся индексом приоритетов выходной очереди  
    counter=-1  
      
    def \_\_init\_\_(self, queue\_in, queue\_out):  
        super(Worker, self).\_\_init\_\_()  
          
        self.setDaemon(True)  
          
        self.\_\_queue\_in=queue\_in  
        self.\_\_queue\_out=queue\_out  
          
    def run(self):  
        """  
        Основной код здесь.  
        """  
          
        while True:  
            # Получаем задание из входящей очереди   
            job=self.\_\_queue\_in.get()  
              
            # Увеличиваем счетчик  
            Worker.counter+=1  
            # Присваиваем значение счетчика, т.о. если представить элементы очереди в виде списка,  
            # то в переменной num хранится индекс элемента списка, взятого в текущей момент из очереди.   
            num=Worker.counter  
              
            print u'%s %s получил %s'%(datetime.datetime.now().strftime('%H:%M:%S'),  
                                     self.getName(),  
                                     job)  
            # Иммитируем занятость путем усыпания на случайное значение секунд.  
            # В результате порядок обрабатываемых задач очереди перемешается  
            time.sleep(random.randint(0,5))  
            print u'%s %s выполнил %s'%(datetime.datetime.now().strftime('%H:%M:%S'),  
                                     self.getName(),  
                                     job)  
            # Помещаем кортеж из индекса и задания в результирующую очередь  
            self.\_\_queue\_out.put((num,job))  
            # Сообщаем входящей очереди что задача выполнена  
            self.\_\_queue\_in.task\_done()  
  
# Создаем входящую FIFO очередь  
queue=Queue.Queue()  
# Создаем результирующую приоритетную очередь  
result=Queue.PriorityQueue()  
  
# Заполняем входящую очередь данными  
for i in data:  
    queue.put(i)  
  
# Создаем и запускаем потоки  
for i in xrange(2):  
    w=Worker(queue, result)  
    w.start()  
  
# Блокируем дальенейшее выполнение программы до тех пор пока потоки не обслужат все эелементы очереди  
queue.join()  
  
# Формируем список как результат обработки изначального списка  
out=[]  
while not result.empty():  
    out.append(result.get()[1])  
  
print data  
print out

В результате будет получен примерно такой вывод:

10:29:12 Thread-1 получил 100  
10:29:12 Thread-2 получил 225  
10:29:12 Thread-2 выполнил 225  
10:29:12 Thread-2 получил 350  
10:29:15 Thread-1 выполнил 100  
10:29:15 Thread-1 получил 475  
10:29:16 Thread-1 выполнил 475  
10:29:16 Thread-1 получил 600  
10:29:17 Thread-2 выполнил 350  
10:29:17 Thread-2 получил 725  
10:29:18 Thread-1 выполнил 600  
10:29:18 Thread-1 получил 850  
10:29:20 Thread-2 выполнил 725  
10:29:20 Thread-2 получил 975  
10:29:22 Thread-1 выполнил 850  
10:29:24 Thread-2 выполнил 975  
[100, 225, 350, 475, 600, 725, 850, 975]  
[100, 225, 350, 475, 600, 725, 850, 975]

Пример 6

Следующий пример демонстрирует ситуацию когда потоки обрабатывающие очередь работают параллельно с основным потоком выполнения программы. В примере смоделиована ситуация когда с некоторым промежутком приходят 2 сигнала, заполняющую очередь заданиями. Потоки начинают обрабатывать задания, а после вновь переходят в режим ожидания задач.

# -\*- coding: utf-8 -\*-  
import Queue, threading, time  
  
class Worker(threading.Thread):  
    """  
    Класс, обслуживающий задачи из очереди.  
    """  
      
    def \_\_init\_\_(self, queue):  
        super(Worker, self).\_\_init\_\_()  
          
        self.\_\_queue=queue  
          
        # Переменная, указывающая о необходимости завершения работы потока  
        self.need\_exit=False  
          
        self.setDaemon(True)  
        self.start()  
      
    def run(self):  
        """  
        Основной код здесь  
        """  
          
        # переменная, отображающая состояние работы основного кода потока  
        state='free'  
        # метод run() циклически выполняется до тех пор, пока атрибуту экземпляра класса need\_exists не будет присвоено значение True   
        while not self.need\_exit:  
            try:  
                # получаем задание из очереди, причем не используем блокировку и устанавливаем таймаут 1 секунда.  
                # Это означает, что если в течениеи 1 секунды все запросы на получение задания из очереди провалятся,  
                # то будет сгенерировано исключение Queue.Empty, указывающее, что очередь пуста.  
                job=self.\_\_queue.get(block=False, timeout=1)  
                  
                # Если было получено задание из очереди, то меняется статус работы на busy  
                state='busy'  
                # Выводим информацию о выполняемой задаче и содержимое очереди  
                print u'%s get job %s, in queue %s'%(self.getName(),  
                                                     str(job),  
                                                     ','.join(map(str,self.\_\_queue.\_\_dict\_\_['queue'])))  
                # Имитируем занятость путем засыпания на 5 секунд  
                time.sleep(5)  
            except Queue.Empty:  
                # Чтобы не засорять вывод ненужной информацией выводим состояние работы только после его смены  
                if state!='free':  
                    print u'%s is free'%self.getName()  
                # Меняем статус работы на free  
                state='free'  
                # засыпаем на долю секунды, что бы не загружать процессор  
                time.sleep(0.1)  
  
  
class Employer(threading.Thread):  
    """  
    Класс, выдающий задания очереди.  
    """  
      
    def \_\_init\_\_(self, queue):  
        super(Employer, self).\_\_init\_\_()  
          
        self.\_\_queue=queue  
          
        self.setDaemon(True)  
        self.start()  
          
    def execute(self, start, stop):  
          
        # Переменная, отображающая состояние очереди  
        state='full'  
          
        # В роли заданий для очереди будут выступать последовательность чисел заданного диапазона  
        for i in xrange(start, stop):  
            while True:  
                try:  
                    # Помещаем задание в очередь, при этом не используем блокировку и устанавливаем таймаут операции в 1 сек.  
                    # Это означает, что если в течение 1 секунды все попытки поместить задание в очередь окажутся неудачными,  
                    # то будет сгенерировано исключение Queue.Full, указывающее что очередь полная.  
                    self.\_\_queue.put(i, block=False, timeout=1)  
                    # Если предыдущая операция завершилась успешно,то меняем состояние работы на 'avaiable'  
                    state='available'  
                    print 'send job', i  
                      
                    # Делаем небольшой перерыв между отправкой следующего задания в очередь  
                    time.sleep(1)  
                    # Выходим из цикла while и переходим на следующую итерацию цикла for  
                    break  
                except Queue.Full:  
                    # Чтобы не засорять вывод ненужной информацией выводим состояние очереди только после его смены  
                    if state!='full':  
                        print u'Queue is full, in queue %s'%','.join(map(str,self.\_\_queue.\_\_dict\_\_['queue']))  
                    # Меняем состояние очереди на full  
                    state='full'  
                    # Делаем задержку перед очередной попыткой отправить задание в очередь  
                    time.sleep(1)  
      
    def run(self):  
        # Реализуем первую партию заданий  
        self.execute(0,5)  
        # Выжидаем 10 секунд что бы потоки обслужили всю очередь  
        time.sleep(10)  
        # Реализуем вторую партию заданий  
        self.execute(5,10)  
  
# Создаем очередь с заданной длиной 2, это означает что одновременно в очереди могут находиться не более 2ух заданий  
queue=Queue.Queue(2)  
  
# Создаем два потока обслуживающих очередь  
workers=[Worker(queue) for x in xrange(2)]  
  
# Создаем "работодателя"  
employer=Employer(queue)  
  
# Данный цикл предназначен для того что бы завершить работу по прерыванию Ctrl+C  
while True:  
    try:  
        time.sleep(0.1)  
    except KeyboardInterrupt:  
        exit()

В результате мы увидим подобный вывод:

send job 0  
Thread-1 get job 0, in queue   
send job 1  
Thread-2 get job 1, in queue   
send job 2  
send job 3  
Queue is full, in queue 2,3  
Thread-1 get job 2, in queue 3  
send job 4  
Thread-2 get job 3, in queue 4  
Thread-1 get job 4, in queue   
Thread-2 is free  
Thread-1 is free  
send job 5  
Thread-1 get job 5, in queue   
send job 6  
Thread-2 get job 6, in queue   
send job 7  
send job 8  
Queue is full, in queue 7,8  
Thread-1 get job 7, in queue 8  
send job 9  
Thread-2 get job 8, in queue 9  
Thread-1 get job 9, in queue   
Thread-2 is free  
Thread-1 is free

Пример 7

Этот пример является неким расширенным вариантом предыдущего. Он работает по принипу клиент-сервер.   
Класс Worker реализует объект потока который ожидает появления задач в очереди для их обслуживания.   
Класс Listener реализует объект потока который прослушивает TCP порт на предмет входящих подключений, сообщения от которых несут в себе задания для очереди вида. Сообщение должно содержать в себе число и текстовое сообщение разделенное пробелом.   
Основной поток выполнения программы представляет собой примитивную командную строку, в которой можно включить и отключить работу потоков обработки очереди, а также получить текущее количество активных потоков программы.   
  
Серверная часть:

# -\*- coding: utf-8 -\*-  
import Queue, threading, time, datetime, socket  
  
class Worker(threading.Thread):  
    """  
    Класс потока, который будет обслуживать задачи из очереди.  
    """  
      
    # Атрибут класса Worker, счетчик порождаемых экземпляров  
    count\_of\_instance=0  
      
    def \_\_init\_\_(self, queue):  
        super(Worker, self).\_\_init\_\_()  
        # Задаем наглядное имя потоку, на основе счетчика экземпляров. Обратите внимание, что нумерация начинается с нуля.  
        self.setName('Worker-%s'%Worker.count\_of\_instance)  
        # Увеличиваем счетчик порожденных экземпляров класса  
        Worker.count\_of\_instance+=1  
          
        self.\_\_queue=queue  
        # Переменная указывающая о необходимости завершения работы потока  
        self.need\_exit=False  
          
        self.setDaemon(True)  
        self.start()  
      
    def run(self):  
        """  
        Основной код здесь  
        """  
        # метод run() циклически выполняется до тех пор, пока атрибуту экземпляра класса need\_exists не будет присвоено значение True   
        while not self.need\_exit:  
            try:  
                # Получаем задачу из очереди, при этом не используем блокировку, это сделано  
                # что бы при пустой очереди генерировалось исключение, которое далее мы бы обрабатывали  
                # по своему усмотрению  
                job=self.\_\_queue.get\_nowait()  
                  
                print u'%s в %s получил задание "%s" в очереди заданий: %s'%(self.getName(),  
                                                        job[0].strftime('%H:%M:%S'),  
                                                        job[2],  
                                                        self.\_\_queue.\_\_dict\_\_['queue'].\_\_len\_\_())  
                time.sleep(job[1])  
                print u'%s в %s выполнил задание "%s" в очереди заданий: %s'%(self.getName(),  
                                                        job[0].strftime('%H:%M:%S'),  
                                                        job[2],  
                                                        self.\_\_queue.\_\_dict\_\_['queue'].\_\_len\_\_())  
                # сообщаем что задание выполнено  
                self.\_\_queue.task\_done()  
            except Queue.Empty:  
                # После получения исключения о том что очередь пуста, необходимо дожидаться пока задание вновь не попадет в очередь.  
                # Для этого мы засыпаем на долю секунды что бы не загружать понапрасну процессор и переходим на новую итерацию цикла  
                time.sleep(0.1)   
  
class Listener(threading.Thread):  
    """  
    Класс потока, который будет получать данные по сети и формировать их в задания для очереди.  
    """  
      
    def \_\_init\_\_(self, port, queue):  
        super(Listener, self).\_\_init\_\_()  
        self.\_\_queue=queue  
          
        # Создаем сокет TCP/IP4  
        self.socket=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
        # Привязываем сокет к заданному посту на локальном узле  
        self.socket.bind(('localhost', port))  
        # Запускаем поток на выполнение  
        self.start()  
      
    def run(self):  
        # Сокет начинает прослушивать порт на наличии соединений  
        self.socket.listen(5)  
          
        # Обработка входящих соединений происходит в бесконечно цикле  
        while True:  
            data=[]  
            # Ожидаем входящего подключения  
            conn, addr = self.socket.accept()  
              
            # В цикле с заданной длинной буфера считываем данные из сокета и помещаем фрагменты в список data  
            while True:  
                tmp=conn.recv(1024)  
                # Если tmp пуст, значит, прием данных завершен и выходим из цикла  
                if not tmp:  
                    break  
                data.append(tmp)  
            # После получения всех данных соединение нам больше не потребуется, поэтому закрываем его  
            conn.close()  
              
            # Извлекаем из полученных данных число секунд и сообщение  
            data=(''.join(data)).split(' ')  
            sleep\_sec=float(data[0])  
            message=' '.join(data[1:])  
              
            # Помещаем задачу в очередь, при этом используем метод put() т.е. используем блокировку  
            self.\_\_queue.put((datetime.datetime.now(),  
                             sleep\_sec,  
                             message))  
              
class Main(object):  
    """  
    Основной класс, реализующий функционал программы.  
    """  
    def \_\_init\_\_(self, threads\_count=2):  
        # Создаем очередь  
        self.queue=Queue.Queue()  
        self.threads\_count=threads\_count  
          
        # создаем экземпляр класса Listener  
        self.listener=Listener(6000, self.queue)  
          
        self.threads=[]  
        # запускаем потоки обслуживающие очередь  
        self.start\_threads()  
          
    def start\_threads(self):  
        """  
        Метод создает список объектов потока.  
        """  
        if self.threads:  
            self.stop\_threads()  
        self.threads=[Worker(self.queue) for x in xrange(self.threads\_count)]  
  
    def stop\_threads(self):  
        """  
        Метод завершает работу потоков.  
        """  
        for item in self.threads:  
            item.need\_exit=True  
      
    def status(self):  
        return u'Количество активных потоков: %s'%threading.activeCount()  
  
    def main(self):  
        """  
        Основной код программы представляет собой примитивную командную строку.  
        """  
          
        while True:  
            try:  
                s=raw\_input('-->')  
                if s=='start':  
                    self.start\_threads()  
                elif s=='stop':  
                    self.stop\_threads()  
                elif s=='status':  
                    print self.status()  
                elif s=='exit':  
                    raise KeyboardInterrupt  
                else:  
                    print u'Неизвестная команда: %s'%s  
            except KeyboardInterrupt:  
                exit()  
          
          
m=Main(2)  
m.main()

Клиентская часть:

# -\*- coding: utf-8 -\*-   
import socket, time  
  
HOST = 'localhost'      
PORT = 6000               
for item in ('10 10 seconds',  
            '2 2 seconds',  
            '1 1 seconds',  
            '5 5 seconds',  
            '8 8 seconds',  
            '7 7 seconds'):      
    s=socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  
    s.connect((HOST, PORT))  
    s.send(item)  
    print 'send',item  
    s.close()  
    time.sleep(1)

Демонстрация работы программы:

-->Worker-0 в 12:45:27 получил задание "10 seconds" в очереди заданий: 0  
Worker-1 в 12:45:28 получил задание "2 seconds" в очереди заданий: 0  
Worker-1 в 12:45:28 выполнил задание "2 seconds" в очереди заданий: 1  
Worker-1 в 12:45:29 получил задание "1 seconds" в очереди заданий: 0  
Worker-1 в 12:45:29 выполнил задание "1 seconds" в очереди заданий: 1  
Worker-1 в 12:45:30 получил задание "5 seconds" в очереди заданий: 0  
Worker-1 в 12:45:30 выполнил задание "5 seconds" в очереди заданий: 2  
Worker-1 в 12:45:31 получил задание "8 seconds" в очереди заданий: 1  
Worker-0 в 12:45:27 выполнил задание "10 seconds" в очереди заданий: 1  
Worker-0 в 12:45:32 получил задание "7 seconds" в очереди заданий: 0  
Worker-1 в 12:45:31 выполнил задание "8 seconds" в очереди заданий: 0  
Worker-0 в 12:45:32 выполнил задание "7 seconds" в очереди заданий: 0  
status  
Количество активных потоков: 4  
-->stop  
-->status  
Количество активных потоков: 2  
-->start  
-->status  
Количество активных потоков: 4  
-->exit