Simple Message Queue using the Pika Python client
(From https://www.rabbitmq.com/tutorials/tutorial-one-python.html)

A producer (sender) that sends a single message and a consumer (receiver) that receives messages and prints them out. It's a "Hello World" of messaging.

P - producer
C-consumer.

The box in the middle is a queue - a message buffer that RabbitMQ keeps on behalf of the consumer.

Producer sends messages to the "hello" queue. The consumer receives messages from that queue.

**Sending**

send.py will send a single message to the queue. The first thing we need to do is to establish a connection with RabbitMQ server.

```python
collection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = collection.channel()
```

We're connected now, to a broker on the local machine - hence the localhost. If we wanted to connect to a broker on a different machine we'd simply specify its name or IP address here.

Next, before sending we need to make sure the recipient queue exists. If we send a message to non-existing location, RabbitMQ will just drop the message. Let's create a hello queue to which the message will be delivered:

```python
channel.queue_declare(queue='hello')
```
At this point we’re ready to send a message. Our first message will just contain a string *Hello World!* and we want to send it to our *hello* queue.

In RabbitMQ a message can never be sent directly to the queue, it always needs to go through an *exchange*. Here we use a default exchange, which we identify by the empty string ("") and the queue name needs to be specified in the *routing_key* parameter:

```python
channel.basic_publish(exchange='', routing_key='hello', body='Hello World!')
print(" [x] Sent 'Hello World!'")
connection.close()
```

**Receiving**

Our second program `receive.py` will receive messages from the queue and print them on the screen.

Again, first we need to connect to RabbitMQ server. The code responsible for connecting to Rabbit is the same as previously.

The next step, just like before, is to make sure that the queue exists.

```python
channel.queue_declare(queue='hello')
```

Receiving messages from the queue is more complex. It works by subscribing a *callback* function to a queue. Whenever we receive a message, this *callback* function is called by the Pika library. In our case this function will print on the screen the contents of the message.

```python
def callback(ch, method, properties, body):    print(" [x] Received %r" % body)
```

Next, we need to tell RabbitMQ that this particular callback function should receive messages from our *hello* queue:

```python
channel.basic_consume(callback, queue='hello', no_ack=True)
```

In order to make sure a message is never lost, RabbitMQ supports message acknowledgments. An ack(nowledgement) is sent back by the consumer to tell RabbitMQ that a particular message had been received, processed and that RabbitMQ is free to delete it.
If a consumer dies (its channel is closed, connection is closed, or TCP connection is lost) without sending an ack, RabbitMQ will understand that a message wasn't processed fully and will re-queue it. If there are other consumers online at the same time, it will then quickly redeliver it to another consumer. That way you can be sure that no message is lost, even if the workers occasionally die.

And finally, we enter a never-ending loop that waits for data and runs callbacks whenever necessary.

```python
print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()
```

**send.py**

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

connection = pika.BlockingConnection(pika.ConnectionParameters(
    host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

channel.basic_publish(exchange='',
                      routing_key='hello',
                      body='Hello World!')

print(" [x] Sent 'Hello World!'")
connection.close()
```

**receive.py**

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

connection = pika.BlockingConnection(pika.ConnectionParameters(
    host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='hello')

def callback(ch, method, properties, body):
    print(" [x] Received \%s" % body)

connection.close()
```
channel.basic_consume(callback,  
    queue='hello',  
    no_ack=True)

print(' [*] Waiting for messages. To exit press CTRL+C')
channel.start_consuming()

**Work Queues**

The main idea behind Work Queues (aka: Task Queues) is to avoid doing a resource-intensive task immediately and having to wait for it to complete. Instead we schedule the task to be done later. We encapsulate a task as a message and send it to the queue. A worker process running in the background will pop the tasks and eventually execute the job. When you run many workers the tasks will be shared between them.

In the previous part of this tutorial we sent a message containing "Hello World!". Now we'll be sending strings that stand for complex tasks. We don't have a real-world task, like images to be resized or pdf files to be rendered, so let's fake it by just pretending we're busy - by using the time.sleep() function. We'll take the number of dots in the string as its complexity; every dot will account for one second of "work". For example, a fake task described by Hello... will take three seconds.

We will slightly modify the send.py code from our previous example, to allow arbitrary messages to be sent from the command line. This program will schedule tasks to our work queue, so let's name it new_task.py:

```python
import sys
message = ' '.join(sys.argv[1:]) or "Hello World!"
channel.basic_publish(exchange='',
    routing_key='hello',
    body=message)
print(" [x] Sent %r" % message)
```
Our old receive.py script also requires some changes: it needs to fake a second of work for every dot in the message body. It will pop messages from the queue and perform the task, so let's call it worker.py:

```python
import time
def callback(ch, method, properties, body):
    print("\[x\] Received %r \% body")
    time.sleep(body.count(b'.'))
    print("\[x\] Done")
```

Round-robin dispatching

One of the advantages of using a Task Queue is the ability to easily parallelise work. If we are building up a backlog of work, we can just add more workers and that way, scale easily.

First, let's try to run two worker.py scripts at the same time. They will both get messages from the queue, but how exactly? Let's see.

You need three consoles open. Two will run the worker.py script. These consoles will be our two consumers - C1 and C2.

```
# shell 1
python worker.py
# => [*] Waiting for messages. To exit press CTRL+C

# shell 2
python worker.py
# => [*] Waiting for messages. To exit press CTRL+C
```

In the third one we'll publish new tasks. Once you've started the consumers you can publish a few messages:

```
# shell 3
python new_task.py First message.
python new_task.py Second message..
python new_task.py Third message...
python new_task.py Fourth message....
```
Let's see what is delivered to our workers:

# shell 1

python worker.py

# => [*] Waiting for messages. To exit press CTRL+C
# => [x] Received 'First message.'
# => [x] Received 'Third message...'
# => [x] Received 'Fifth message.....'

# shell 2

python worker.py

# => [*] Waiting for messages. To exit press CTRL+C
# => [x] Received 'Second message..
# => [x] Received 'Fourth message....'

By default, RabbitMQ will send each message to the next consumer, in sequence. On average, every consumer will get the same number of messages. This way of distributing messages is called round-robin. Try this out with three or more workers.

new_task.py

#!/usr/bin/env python
import pika
import sys

connection = pika.BlockingConnection(pika.ConnectionParameters(host='localhost'))
channel = connection.channel()

message = ' '.join(sys.argv[1:]) or "Hello World!"
channel.queue_declare(queue='task_queue', durable=True)

channel.basic_publish(exchange='', routing_key='task_queue', body=message, properties=pika.BasicProperties(delivery_mode = 2, # make message persistent ))

print(" [x] Sent %r" % message)
connection.close()
worker.py
#!/usr/bin/env python
import pika
import time

connection = pika.BlockingConnection(pika.ConnectionParameters(
    host='localhost'))
channel = connection.channel()

channel.queue_declare(queue='task_queue', durable=True)
print(' [*] Waiting for messages. To exit press CTRL+C')

def callback(ch, method, properties, body):
    print(" [x] Received %r" % body)
    time.sleep(body.count(b'.'))
    print(" [x] Done")
    ch.basic_ack(delivery_tag = method.delivery_tag)

channel.basic_qos(prefetch_count=1)
channel.basic_consume(callback, queue='task_queue')

channel.start_consuming()

Publish/Subscribe

The assumption behind a work queue is that each task is delivered to exactly one worker. In this part we'll do something completely different -- we'll deliver a message to multiple consumers. This pattern is known as "publish/subscribe".

The core idea in the messaging model in RabbitMQ is that the producer never sends any messages directly to a queue. Actually, quite often the producer doesn't even know if a message will be delivered to any queue at all.

Instead, the producer can only send messages to an exchange. An exchange is a very simple thing. On one side it receives messages from producers and the other side it pushes them to queues. The exchange must know exactly what to do with a message it receives. Should it be appended to a particular queue? Should it be appended to many queues? Or should it get discarded. The rules for that are defined by the exchange type.
There are a few exchange types available: direct, topic, headers and fanout. We'll focus on the last one -- the fanout. Let's create an exchange of that type, and call it `logs`:

```
channel.exchange_declare(exchange='logs', exchange_type='fanout')
```

The fanout exchange is very simple. As you can probably guess from the name, it just broadcasts all the messages it receives to all the queues it knows. And that's exactly what we need for our logger.

Now, we can publish to our named exchange instead:

```
channel.basic_publish(exchange='logs', routing_key=' ', body=message)
```

**Bindings**

We've already created a fanout exchange and a queue. Now we need to tell the exchange to send messages to our queue. That relationship between exchange and a queue is called a `binding`.

```
channel.queue_bind(exchange='logs', queue=result.method.queue)
```

From now on the `logs` exchange will append messages to our queue.
emit_log.py
#!/usr/bin/env python
import pika
import sys

connection = pika.BlockingConnection(pika.ConnectionParameters(
    host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='logs',
    exchange_type='fanout')

message = ' '.join(sys.argv[1:]) or "info: Hello World!"
channel.basic_publish(exchange='logs',
    routing_key='',
    body=message)
print("[x] Sent %r" % message)
connection.close()

receive_logs.py
#!/usr/bin/env python
import pika

connection = pika.BlockingConnection(pika.ConnectionParameters(
    host='localhost'))
channel = connection.channel()

channel.exchange_declare(exchange='logs', exchange_type='fanout')
result = channel.queue_declare(exclusive=True)
queue_name = result.method.queue
channel.queue_bind(exchange='logs',
    queue=queue_name)

print(' [*] Waiting for logs. To exit press CTRL+C')

def callback(ch, method, properties, body):
    print("[x] %r" % body)

channel.basic_consume(callback, queue=queue_name, no_ack=True)

channel.start_consuming()