Q Challenge

Your challenge is to implement a simple message delivery service called `q`. This service accepts (enqueues) messages through a single interface. It applies a set of transformation rules to the data, then chooses which output queue should get the message based on a series of dispatching rules. It also delivers the parts of multi-message sequences in order.

Basic Requirements

You must use Python 3 to complete this challenge. If you're unfamiliar with Python 3, websites like http://getpython3.com/ can help you get started. This "cheat sheet" may be helpful.

Please use the directory/package structure we've given you, and make sure that `q/solution.py` includes a `get_message_service` function that we can use to get a clean instance of your solution. We've provided some very basic tests in `basic_tests.py`. Of course, they're nowhere near complete; but, if you can run that file from `q`'s parent directory, your solution should run just fine when we test it.

We encourage you to use standard libraries when they are available; there's no need to, for instance, write your own implementation of a queue. If you need to use packages that are not included in the Python standard library, you must include a `requirements.txt` referencing publicly available `pip` packages with your solution. We do not anticipate that you will need anything but the standard library.

Input

The message service must accept messages when the following method is called: `enqueue(msg)`.

Output

The message service must provide the ability to get the next message on each output queue.

You must provide a single method `next(queue_number)` that returns the next message for the queue with the number `queue_number`. `next` should immediately throw an exception if nothing is available on the queue.

What to Submit

Please send back a tarball or zip of the `q/` folder.

You can optionally include any other attachments, but we can't promise to evaluate them and certainly won't penalize you if you just give us the `q/` folder.

Message Processing Rules

Messages are standard JSON. You may assume that values are either strings or numeric types. Do not assume anything about the messages other than what is specified in this document.

Upon receiving a message, you must apply transformation rules first, then apply dispatching rules.

Transformation Rules

You must implement the following transformations on input messages. These rules must be applied in order, using the transformed output in later steps. Multiple rules may apply to a single tuple.

- You must string-reverse any string value in the message that contains the string `Qadium`.
- For instance, `{"company": "Qadium, Inc.", "agent": "007"}` changes to `{"company": ".cnI, muidaQ", "agent": "007"}`.
- You must replace any integer values with the value produced by computing the bitwise negation of that integer's value.
- For instance, `{"value": 512}` changes to `{"value": -513}`
- You must add a field `hash` to any message that has a field `_hash`. The value of `_hash` will be the name of another field; the value of your new field `hash` must contain the base64-encoded SHA-256 digest of the UTF-8-encoded value of that field. You may assume that the value you're given to hash is a string.

Transformation rules, except the hash rule, must ignore the values of "private" fields whose names begin with an underscore (`_`).

Dispatch Rules

There are five output queues, numbered 0 through 4.

You must implement the following "dispatch" rules to decide which queue gets a message. These rules must be applied in order; the first rule that matches is the one you should use.

- If a message contains the key `_special`, send it to queue 0.
- If a message contains a `hash` field, send it to queue 1.
- If a message has a value that includes `muidaQ` (`Qadium` in reverse), send it to queue 2.
- If a message has an integer value, send it to queue 3.
- Otherwise, send the message to queue 4.

Dispatch rules must ignore the *values* of "private" fields whose names begin with an underscore (`_`). (Of course, rules that test the presence of *keys* that begin with `_` still apply.)

Sequences

Certain messages may be parts of a sequence. Such messages include some special fields:

- `_sequence`: an opaque string identifier for the sequence this message is part of
- `_part`: an integer indicating which message this is in the sequence, starting at 0

Sequences must be outputted in order. Dispatch rules are to be applied based on the first message in a sequence (message 0) only, while transformation rules must be applied to all messages.

The output queue must enqueue messages from a sequence as soon as it can; don't try to wait to output all messages of a sequence at a time. The output queue must return messages within a sequence in the correct order by part number (message 0 before message 1, before message 2 . . .).

What are we looking for?

This challenge should give you a chance to show off your programming and system design skills. We hope you'll finish it in no more than a few hours. When we evaluate your submission, we'll first consider whether your solution behaves correctly on the inputs we gave you. We also have some other tests up our sleeves, so be sure to make your solution work exactly as we've specified.

Beyond correctness, we'll assess other criteria, including:

- Did you make good design choices? Is your design easily extensible?
- Do you use standard functions and libraries appropriately?
- Did you document your code in a way that helps others clearly understand how it works?
- Did you choose reasonable data structures?
- Would we be happy to maintain your code?