Python Algorithm Tutorial for Speed Bump

Table of Contents

Introduction 2
Implementation 3
Basic setup 3
Speed bump 5
Running the Algorithm 8

Speed Bump

Introduction

When writing an algorithm for RIT using a programming language, students often encounter an issue related to 'speed'. This is mainly caused by the discrepancy between how long it takes for the programming language to cycle through the algorithm codes, and how 'often' the RIT market information is updated through API. Most of the time, as mentioned briefly in other tutorial documents, students can simply use a 'time-delay' (i.e. 'sleep') function to resolve this issue. In other words, this 'sleep' function is implemented so that it makes the programming language to pause at a certain point when cycling through the algorithm codes in order to ensure that the execution is done and completed properly, before moving onto the next algorithm codes.

This 'sleep' function effectively works as a 'speed bump' for an algorithm especially for a successful order execution. While students can simply try using a 'fixed' value for the speed bump (i.e. try using 0.1, 0.2, or 0.5 seconds), this tutorial is designed for those who would like to improve their speed bump logic in order to create a speed bump that adjusts dynamically through the case depending on the execution time of orders ("endogenous speed bump") and other case variables for an optimal outcome. However, please also note that this tutorial document only provides an example of how one may approach and resolve this issue. In other words, students should not take this tutorial as a 'solution' since there are many ways of dealing with this time discrepancy which is also often different on case-by-case basis.

Students are expected to have followed the RIT REST API User Guide to set up the necessary Python environment and completed a python algorithm.

Overview

Let's start with a simple example to purchase 20,000 shares when the 'rate limit' is set to 5 and the maximum volume per order is set to 1,000 shares. The 'rate limit' is the maximum number of orders one is allowed to submit per second. The 'rate limit' may have been provided to you already, or you may 'observe' the 'rate limit' value by submitting a large numbers of orders (and cancelling them immediately) in order to push the REST API limit which triggers the 'rate limit' message which will also be demonstrated below.

In order to not exceed the 'rate limit', we will implement a speed bump in order to 'slow down' the execution of our algorithm. Instead of assigning a single value to the speed bump, we would like to create an endogenous speed bump which is a speed bump that changes dynamically throughout the case depending on the order execution time as well as other case variables.

Each time we submit an order we will first calculate the 'transaction time'. The 'transaction time' is how long it takes an order to get submitted successfully to the market. We then calculate our speed bump by determining how long of a speed bump is needed between each order for us to submit the maximum orders per second given our 'transaction time'.

In order to calculate our speed bump, let's create a simple formula to represent this scenario in which we are allowed to submit 5 orders every second.

$$1\ (second) = (t + sb) + (t + sb) + (t + sb) + (t + sb) + (t + sb)$$

Where,

t is the transaction time of a single order, and
sb is our speed bump time per order

Once we re-arrange the formula and simplify it for sb, we get the following:

$1/5 = t + sb$, or
$sb = -t + 1/5$

This means that the addition of transaction time and speed bump (time) of a single order *should be* 0.2 second given that we are only allowed to submit 5 orders per second in this case. This also means that, once we 'measure' the transaction time, we can calculate the speed bump value.

Once we calculate the speed bump value, instead of using it directly to delay our algorithm, we will calculate the 'average' speed bump using a total value of speed bump and a total number of orders submitted. Then, we will use this 'average' value of speed bump. This is because the most 'recent' speed bump may not be entirely reliable in case the submission was completed too quickly or too slowly for the most recent order, resulting in an extreme value of speed bump.

Implementation

Basic setup

Similar to previous tutorials we will first import the 'requests' package as well as the 'signal' and 'time' packages in order to create some helpful boilerplate code to handle exceptions and CTRL+C commands to stop the algorithm. Then we will save the API KEY for easy access.

```python
# This is a python example algorithm using REST API for Speed Bump

import time
import signal
import requests
from time import sleep

# this class definition allows us to print error messages and stop the program when needed
class ApiException(Exception):
    pass

# this signal handler allows for a graceful shutdown when CTRL+C is pressed
def signal_handler(signum, frame):
    global shutdown
    signal.signal(signal.SIGINT, signal.SIG_DFL)
    shutdown = True

# set your API key to authenticate to the RIT client
API_KEY = {'X-API-Key': 'Insert your API KEY here'}
shutdown = False

```

Before we move onto writing an algorithm to demonstrate a speed bump, let's start with a simple order submission to observe the issue of rate limiting. First, we can define a few key variables so that we can use them as reference later.

```
21
22  # maximum orders per second
23  ORDER_LIMIT = 5
24  # maximum size per order
25  MAX_SIZE = 1000
26  # Target total volume
27  TOTAL_VOLUME = 20000
28  # order counter
29  COUNT = int(TOTAL_VOLUME/MAX_SIZE)
30  # starting number of orders
31  number_of_orders = 0
32
```

"COUNT" refers to the # of orders to achieve the total target volume. Since we are only allowed to submit an order with a volume of 1,000 shares and we want to achieve a total volume of 20,000 shares, we need to submit 20 orders (=20,000 shares / 1,000 shares).

The "number_of_orders" variable refers to the # of submitted orders. Because we do not submit any orders at the beginning, its starting value is set to 0.

Lets now setup the basic main() method shown below that will submit orders in order to buy a total volume of 20,000 shares. The "Number of orders" variable is defined as "global" so that we can refer to it later under the 'while' loop. The logic here is simple: keep submitting an order to buy 1,000 shares of Algo until the maximum # of orders are reached.

```
33  def main():
34      global number_of_orders
35      with requests.Session() as s:
36          s.headers.update(API_KEY)
37
38          # while the number of submitted order is less than the total volume, do the following
39          while number_of_orders < COUNT:
40
41              # buy 1000 shares
42              resp = s.post('http://localhost:9999/v1/orders', params = {'ticker': 'ALGO',
43              'type': 'LIMIT', 'quantity': MAX_SIZE, 'price': 20, 'action': 'BUY'})
44              number_of_orders = number_of_orders + 1
45
46              # the order has been sucessfully submitted
47              if(resp.ok):
48                  continue
49
50              # went over trading limit
51              else:
52                  print(resp.json())
53
54  if __name__ == '__main__':
55
56      signal.signal(signal.SIGINT, signal_handler)
57      main()
```

We first use the "number_of_orders" variables to check how many orders are submitted. If it is less than the maximum # of orders, then the algorithm will keep submitting an order. Furthermore, each time we submit a buy order we send a `GET` request to http://localhost\v1\orders with the query parameter equal to the ticker, type of order, price, and action.

Then, once each order is submitted, we increase the "number_of_orders" variable by one, in order to count the # of orders submitted. The algorithm will continue until the 20 orders are submitted.

However, not surprisingly, you will notice that your algorithm will only submit 5 orders successfully, and encounter the following "rate-limiting" error messages for the 15 orders that were submitted unsuccessfully.

```
{'code': 'TOO_MANY_REQUESTS', 'message': 'Rate limit of 5 commands/second
exceeded for ALGO.', 'wait': 65}
```

As you may know, this is simply because the case only allows for an order submission of 5 orders per second and we are submitting 20 orders at the same time. Therefore, we need to have a better control for this order submission logic.

Speed bump

In order to solve this issue, we will now implement the speed bump discussed earlier.

One simple solution may be to "pause" the algorithm for a certain time period. However, as discussed under the "Overview" section, we will try to implement a logic such that the speed bump value is dynamically updated and adjusted according to the current order execution time and the case environment.

First, we want to define another variable under other case variables as shown below.

```
27  TOTAL_VOLUME = 20000
28  # starting order counter value
29  COUNT = int(TOTAL_VOLUME/MAX_SIZE)
30  # starting number of orders
31  number_of_orders = 0
32  # starting total speed bumps
33  total_speedbumps = 0
34
```

The new variable, "total_speedbumps", refers to the total aggregate value of speed bump that we used. Later, this variable will be used to calculate the 'average' speed bump value which will be applied to our order execution.

In order to calculate the speed bump discussed earlier, let's now create a method that calculate the endogenous speed bump.

```
35  def speedbump(transaction_time):
36
37      # this is done so we can use our global variables defined above
38      global total_speedbumps
39      global number_of_orders
40
41      # calculate the speed bump of the current order
42      order_speedbump = -transaction_time + 1/ORDER_LIMIT
43
44      # add it to the total aggregated value of speed bump
45      total_speedbumps = total_speedbumps + order_speedbump
46
47      # increase the number of orders (taken from the previous main method)
48      number_of_orders = number_of_orders + 1
49
50      # sleep for the speed bump value calculated based on the average
51      sleep(total_speedbumps/number_of_orders)
52
```

First, in order to keep using and referring to the two variables, "total_speedbumps" and "number_of_orders", outside of this method, it is important to make them 'global' variables. This ensures that we are changing the global variables, not local variables only defined within the method.

Then, we calculate the speed bump of the current order. Then, it is added to the aggregated value of speed bumps, which is then divided by the total number of orders submitted so far. In other words, this method computes an 'average' value of speed bump and applies it to the algorithm to pause after an order submission before moving onto the next logic.

One may simply decide to use the 'current' speed bump calculated based on the most recent order and it may work well in this case or any other cases where the order submission logic is relatively simple. However, it may not be optimal to do so especially when the submission of the most recent order was completed too quickly or too slowly, as the speed bump may be an extreme value which may not be reliable.

In order to call our speed bump method, we need to calculate our transaction time and then call it from the main method.

```
52
53  def main():
54      with requests.Session() as s:
55          s.headers.update(API_KEY)
56
57          # while the number of submitted order is less than the total volume, do the following
58          while number_of_orders < COUNT:
59
60              # get time before buying our order
61              start = time.time()
62              # buy 1000 shares
63              resp = s.post('http://localhost:9999/v1/orders', params = {'ticker': 'ALGO',
64              'type': 'LIMIT', 'quantity': MAX_SIZE, 'price': 20, 'action': 'BUY'})
65
66              # the order has been sucessfully submitted
67              if(resp.ok):
68                  # calculate the transaction time
69                  transaction_time = time.time() - start
70                  # calculate our speedbump
71                  speedbump(transaction_time)
72
73              # went over trading limit
74              else:
75                  print(resp.json())
76
77  if __name__ == '__main__':
78
79      signal.signal(signal.SIGINT, signal_handler)
80      main()
```

The main method is updated mainly in order to calculate the transaction time by call the time method. The "start" time represents the time before the order is submitted. Then, we will use the time method again to record the time after the order submission. Essentially, we define a transaction time as the difference between the time right before and after the order submission by using the time methods. Then, this transaction time is used in our speed bump method in order to calculate the speed bump value for this order.

This logic is repeated until 20 orders are successfully submitted.

Running the Algorithm

Here's how the complete algorithm should look like:

```python
# This is a python example algorithm using REST API for Speed Bump

import time
import signal
import requests
from time import sleep

# this class definition allows us to print error messages and stop the program when needed
class ApiException(Exception):
    pass

# this signal handler allows for a graceful shutdown when CTRL+C is pressed
def signal_handler(signum, frame):
    global shutdown
    signal.signal(signal.SIGINT, signal.SIG_DFL)
    shutdown = True

# set your API key to authenticate to the RIT client
API_KEY = {'X-API-Key': 'Insert your API KEY here'}
shutdown = False

# maximum orders per second
ORDER_LIMIT = 5
# maximum size per order
MAX_SIZE = 1000
# Target total volume
TOTAL_VOLUME = 20000
# starting order counter value
COUNT = int(TOTAL_VOLUME/MAX_SIZE)
# starting number of orders
number_of_orders = 0
# starting total speed bumps
total_speedbumps = 0

def speedbump(transaction_time):

    # this is done so we can use our global variables defined above
    global total_speedbumps
    global number_of_orders

    # calculate the speed bump of the current order
    order_speedbump = -transaction_time + 1/ORDER_LIMIT

    # add it to the total aggregated value of speed bump
    total_speedbumps = total_speedbumps + order_speedbump

    # increase the number of orders (taken from the previous main method)
    number_of_orders = number_of_orders + 1

    # sleep for the speed bump value calculated based on the average
    sleep(total_speedbumps/number_of_orders)

```

```
52
53  def main():
54      with requests.Session() as s:
55          s.headers.update(API_KEY)
56
57          # while the number of submitted order is less than the total volume, do the following
58          while number_of_orders < COUNT:
59
60              # get time before buying our order
61              start = time.time()
62              # buy 1000 shares
63              resp = s.post('http://localhost:9999/v1/orders', params = {'ticker': 'ALGO',
64              'type': 'LIMIT', 'quantity': MAX_SIZE, 'price': 20, 'action': 'BUY'})
65
66              # the order has been sucessfully submitted
67              if(resp.ok):
68                  # calculate the transaction time
69                  transaction_time = time.time() - start
70                  # calculate our speedbump
71                  speedbump(transaction_time)
72
73              # went over trading limit
74              else:
75                  print(resp.json())
76
77  if __name__ == '__main__':
78
79      signal.signal(signal.SIGINT, signal_handler)
80      main()
```

In order to run the algorithm, ensure that the RIT client is connected and the REST API is enabled. Then, from the working directory, enter `python <FILENAME>.py` into the prompt. To stop the algorithm before the case is finished, press `CTRL+C`. If the file name has any space in it, please enter `python "<FILENAME>.py"`

Note: if students make changes to the algorithm's code while it is running in the prompt, those changes will not be reflected in what is running. Students will have to stop and restart the algorithm.