**MiningInfoBot**

**(V1.0)**

**By SoulXHades**

Contents

[Introduction 3](#_Toc502854432)

[Set-up 3](#_Toc502854433)

[**Firewall ports requirement** 3](#_Toc502854434)

[**Text editor** 3](#_Toc502854435)

[**Python 3** 3](#_Toc502854436)

[Windows: 3](#_Toc502854437)

[GNU/Linux: 4](#_Toc502854438)

[**pip** 4](#_Toc502854439)

[Windows: 4](#_Toc502854440)

[GNU/Linux: 4](#_Toc502854441)

[**Telepot** 4](#_Toc502854442)

[**sshpass** 4](#_Toc502854443)

[**Open ssh** 5](#_Toc502854444)

[**Time difference** 5](#_Toc502854445)

[**“data” directory** 5](#_Toc502854446)

[**API Bot Token** 5](#_Toc502854447)

[Flow chart 6](#_Toc502854448)

[**Overview of the structure** 6](#_Toc502854449)

[**miningInfoBot.py - main function** 6](#_Toc502854450)

[**miningInfoBot.py - handle()** 7](#_Toc502854451)

[**miningInfoBot.py – handle\_commands()** 8](#_Toc502854452)

[**miningInfoBot.py – handle\_markupkeyboard()** 8](#_Toc502854453)

[**miningInfoBot.py – checkRigUpStatus()** 9](#_Toc502854454)

[Code Explanations 10](#_Toc502854455)

[**miningInfoRigServer.sh** 10](#_Toc502854456)

[**miningInfoBot.py** 13](#_Toc502854457)

# Change Log

## **Major Changes**

1. Switched from miningInfoRigServer.sh (put in Rig Servers) to miningInfo\_RigServer.py (put in Centralized Server) to reduce workload. miningInfoRigServer.sh is to be placed in all Rig Servers. However, that can be too tedious as whenever a new Rig is set up or change of IP Addresses. miningInfoRigServer.sh sends data from Rig Server to Centralized Server. miningInfo\_RigServer.py gets data from Rig Server.
2. Set epoch time of SSH using Centralized Server’s local time instead of Rig Server’s local time as Rig Server’s local time might not be the same as Centralized Server’s local time. This remove the unnecessity of setting local time of Rig Server one by one.

## **Bug Fixes**

1. Due to mining of different cryptocurrency, the number of nanopools web link maybe different. This returns in the number of lines in the data text file to be different, thus different in array positions. Index error occurred. This bug is now fixed by putting only stuff we want into the data text file before reading it.

## **Cosmestic Changes and Cleanup**

1. Add more emoji icon into Rig Status in Telegram Chat to allow users to easily notice key status and improve design appearance.

# Introduction

\*\*You may access this telegram bot @SXMiningBot.

This is a bot created to eliminate the access of ethos panel and nanopool website by providing convenient for customers or users to access data easily. The whole project consist of two files, miningInfo\_RigServer(get Rig Server status) and miningInfoBot.py(bot’s script).

The miningInfo\_RigServer.py is a Python script that is written to get real time data from the Rig Server to the Centralized Server for us to access the data easily. These data is then processed by miningInfoBot.py.

miningInfoBot.py is a Python file that will process the text file(data) sent by the miningInfoRigServer.sh and allow users to access it easily through Telegram.

# Set-up

You need:

1. Firewall ports requirement
2. Text editor (sublime, notepad++, vim, nano, etc)
3. Python 3 and above
4. pip or pip3
5. Telepot module
6. Paramiko module
7. Pyodbc module
8. open ssh server
9. “data” directory
10. API Bot Token

## **Firewall ports requirement**

For the centralized server, port 22 (ssh and sftp) and port 443 (outbound to api.telegram.org) is needed to be open.

For Rig Server, port 22 (ssh and sftp) is needed to be open.

For Database Server, port 1433 (SQL) is needed to be open. (For future development)

## **Text editor**

Install any text editor in your OS(can be Windows, GNU/Linux, etc) as Python files are cross-platform. For the project, we used sublime text editor to code the Python script and vim to code the bash script.

## **Python 3**

### Windows:

Install Python in your OS. Only Windows do not comes with Python. This project uses Python 3.6.4. You can download it from the official site here <https://www.python.org/downloads/> .

Set Python in environmental Variable for easy use in terminal. This only applies to Windows OS as Unix and Unix-liked system has been done for you by default. To do this, open up your search bar, then input “View Advanced System Settings”. Navigate in this order View Advanced System Settings > Advanced system settings > Advanced (tab) > Environment Variable > User variables for [your PC name] > Path (double click to edit). Double click a new line to add a new line to the path. Input the location of where you installed Python 3 then click OK.

### GNU/Linux:

Python 2 is the default Python in Unix and Unix-liked system if no default was set. To set Python 3 as the default Python in terminal:

root@ubuntu:~# alias python = python3

## **pip**

### Windows:

If pip is not install, you can install from this link <https://pypi.python.org/pypi/pip> .

### GNU/Linux:

For GNU/Linux, you can install pip3 by:

root@ubuntu:~# sudo apt-get -y install python3-pip

## **Telepot**

Telepot is Python module that will handles the API of the telegram bot such as incoming request and out going messages/packages. Install Telepot by:

root@ubuntu:~# sudo pip3 install telepot

## **Paramiko**

Install sshpass from apt-get by inputting “sudo apt-get install sshpass”. The purpose of this will be explained in the bash script section. Install sshpass from apt-get by:

root@ubuntu:~# sudo pip3 install paramiko

## **Pyodbc**

Pyodbc is a Python library to access Microsoft SQL database. Install Pyodbc by:

root@ubuntu:~# sudo –H pip3 install pyodbc

## **Open ssh**

Install open ssh in the Rig Server for ssh uses if ssh server daemon does not exist. Input “sudo apt-get install openssh-server”.

root@ubuntu:~# sudo apt-get install openssh-server

## **“data” directory**

Create a “data” directory in the directory that you have placed in miningInfoBot.py in. This “data” directory is used to store text file that contains the data of the Rig Servers.

root@ubuntu:~# mkrdir data

## **API Bot Token**

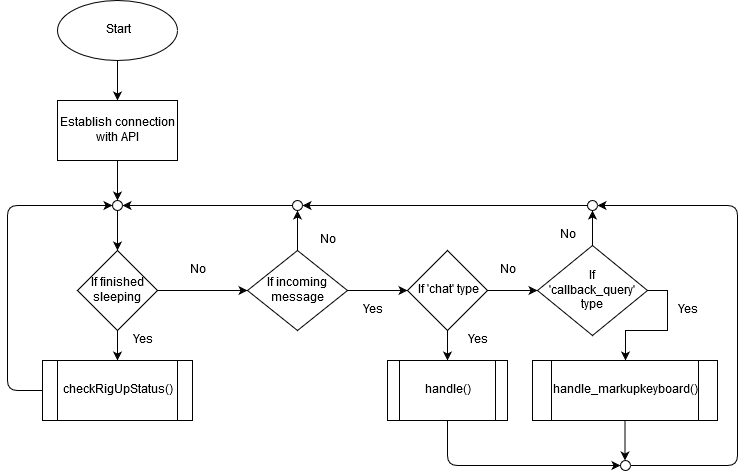
Request a Telegram Bot token. Search for @BotFather in telegram and request a new bot from him. Input “/newbot” and follow the steps given by Bot Father. After creating the bot, you will receive a token from Bot Father. Keep the token safe. We will be using it later.

# Flow chart

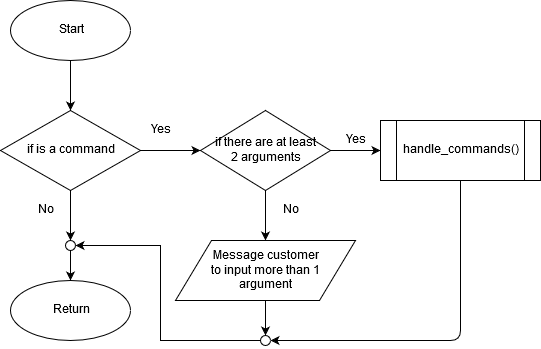
## **Overview of the structure**



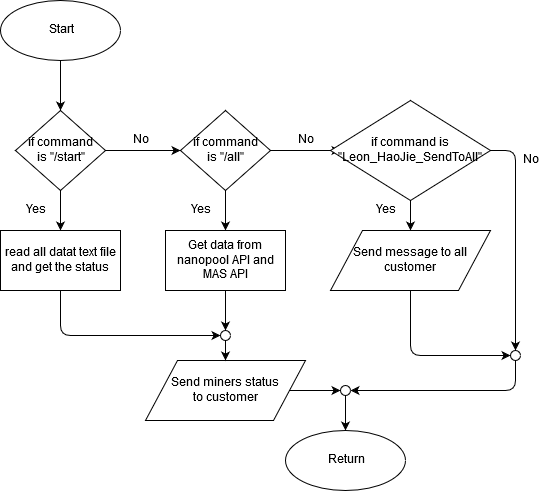
## **miningInfoBot.py - main function**



## **miningInfoBot.py - handle()**



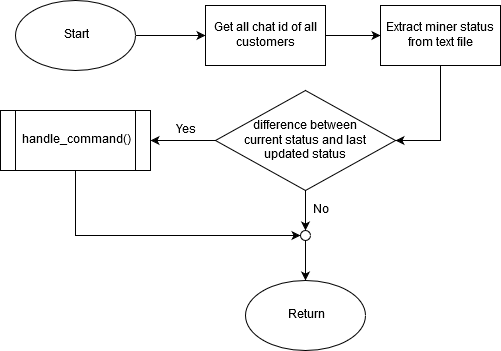
## **miningInfoBot.py – handle\_commands()**



## **miningInfoBot.py – handle\_markupkeyboard()**



## **miningInfoBot.py – checkRigUpStatus()**



# Code Explanations

## **miningInfoRigServer.sh**

1 #!/bin/bash

2 #developed by SoulXHades aka. SoulX

3

4 username="shwn"

5 rignum=1

6 sleeptime=30

7 serveruser="soulx"

8 serverip="192.168.1.17"

9 serverpass="6369408ahjj37"

10 workingdict="/home/soulx/Documents/PythonProject/data"

11

12 echo "Launching..."

13 echo "Sleeptime is set to ${sleeptime}"

14

15 #Newline

16 echo ""

The miningInfoRigServer.sh is a bash script that is used to extract data from the Rig Server and then pass the data into a centralized server. This section will be the explanation of the syntax and the rationale of each line of codes. This script is to be run in all Rig Server.

Line 4 to 10 of the script in this picture are declaration of variables. These allows reusability and for future users to edit the script easily such as changing or username or server’s IP address, etc.

Line 12 and 13 of the script in this picture is to display on console for debugging or other purposes. ‘echo’ is the same as ‘print’ or ‘printf’(but included ‘\n’).

17 while [ 1 ]

18 do

19 echo "Reading stats"

20 #write current time in 24-hours format to text file

21 command date +"%s" > ${username}\_${rignum}\_statsServer.txt

22 #write stats to text file

23 command stats >> ${username}\_${rignum}\_statsServer.txt

24

25 echo -e "\e[96mSending data to server\e[39m"

26 sshpass -p ${serverpass} sftp ${serveruser}@${serverip}:${workingdict} << 'EOS'

27 put shwn\_1\_statsServer.txt

28 bye

29 EOS

30 #echo -e "\e[92mSent to server successfully!\e[39m"

31

32 #Newline

33 echo ""

34

35 #edit your time interval to send the data

36 #default is 1 min

37 sleep ${sleeptime}

38 done

The next section of the script is the infinite while loop. This is to ensure data will keep transmitting to the centralize server until the process of this script is terminated by the user or admin.

In line 17, ‘while [1]’ is similar to ‘while (1)’ in other programming languages to define a infinite while loop then follow by a ‘do’ to write the script of the do loop inside.

In line 21, ‘command date +”%s” > ${username}\_${rignum}\_statsServer.txt will use the date command from the terminal with %s format (time in seconds since epoch) and overwrite the content of the text file with it. The name declared in username variable and rignum variable. ${<variable>} is to use the content in the variable. The username must set according to what the admin gives the customer and it must be unique. The rig num will be starting from 1. It is basically the rig number. It has to start from 1 else the Python Script will fail. Let’s say user have 3 rigs. The 1st rig’s rignum will be 1, then 2nd rig is 2, 3rd rig is 3. If the text file does not exist, it will create a text file with that name format then write the date command’s output into it. The purpose of it using seconds from epoch will be explained in the miningInfoBot.py file.

In line 23, the output of ‘stats’ (info of the rig) command will be written into the text file. “stats” command will output data such as hash rate, temperature, number of GPUs running, etc. It will not overwrite the content but continue from it. Newline in the text file is auto created when writing to the text file.

In line 25, ‘echo –e “\e[96m”’ is to change the color of the words output in the terminal.

In line 26, sshpass is used to solve the password authentication issue when connecting through sftp. Better alternatives like SSH RSA key will be used in future. Inside the EOS will be the commands when use sftp. The commands are to put the data of the Rig text file into the centralize server then end the sftp connection. \**Take note that the username and rig number has to change here as well as sftp command do not allow us to use bash script’s variable.*

In line 37, the sleep command will causes the script to halt at that line in the script until a given period of time is up. It is to control the timing interval between how long will the rig server send the data to the centralize server again. The ‘sleeptime’ variable’s value once again, can be changed at the top of the script.

Important:

1. The variables in the bash script
2. \*ssh into server from Rig to solve host key issue
3. chmod the bash script

The variables in the bash scripts are at the top section of the script. Change what you need especially the username and the rig number. Remember to change the text file name in the EOS section in the sftp (highlighted in red).

26 sshpass -p ${serverpass} sftp ${serveruser}@${serverip}:${workingdict} << 'EOS'

27 put shwn\_1\_statsServer.txt

28 bye

29 EOS

\*Due to ssh host key issue. You have to use the rig server to ssh into the destined server to solve the host key issue. ssh from the terminal into the server will solve this issue. Remember to do this step before running the bash script.

Remember to use ‘chmod’ command in the terminal to change the permission of the script to be able to run the script. Finally, to run the bash script discretely as a process in the background without outputting any text, input this:

root@ethos:~# chmod 770 miningInfoRigServer

root@ethos:~# ./miningInfoRigServer &>/dev/null &

## **miningInfoBot.py**

This is a Python script that will run in the centralized server for the Telegram bot to work. When the user requests data of the bot, it will extract data from the text file (sent from the rig server by the miningInfoRigServer bash script and return the data to the user. It also have other capabilities such as detecting if the rig is up or down, the total hashrate, ETH balance from nanopool’s API. It also can return the earn rate in USD as well as SGD which the exchange rate is obtained from MAS’s API.

335 bot = telepot.Bot("432196079:AAGRT5Ym2dm1h8fWq9b6RUfWdEkWQc2ra9c")

336

337 MessageLoop(bot, {'chat': handle, 'callback\_query': handle\_markupkeyboard}).run\_as\_thread()

338 print("listening...")

339

340 while 1:

341 time.sleep(300)

342 checkRigUpStatus()

Line 335 onwards are the code in the main function like int main().

In line 335, *bot* instance is created and your API token is used here as the argument. A connection will be establish between the customer’s telegram chat and the server running this Python script.

In line 337, MessageLoop() will wait for incoming requests(messages) from users. The run\_as\_thread() will make the MessageLoop() run async. Normal text from the user will be send to the handle function. Return markup(in chat buttons)’s data will be send to the handle\_markupkeyboard function.

In line 340, the infinite WHILE loop is introduced to keep the script running as the MessageLoop() is now running async.

Between line 341 and 342, sleep() is introduced to let the program wait for 300seconds before calling the checkRigUpStatus() to check if the rigs are up or down.

We will firstly dive into the handle() that handles normal incoming text messages from users.

314 #handles normal message (not sent from inline keyboards)

315 def handle(msg):

316 #print(msg) #debug

317 chat\_id = msg['chat']['id']

318 query\_message = msg['text']

319 chat\_message = ""

320

321 #commands will start with "/"

322 if query\_message.startswith("/"):

323 #remove "/" from user's message

324 query\_message = query\_message[1:].split()

325

326 #incase user sent only commands means length of 1

327 if len(query\_message) > 1:

328 handle\_commands(chat\_id, query\_message[0], “ ”.join(query\_message[1:]))

329 else:

330 bot.sendMessage(chat\_id, "Input either: \n/start <username> \nor \n/all <wallet key> \n \nemail to: hadessoulx@gmail.com for bugs")

331

332 return

The *msg* parameter receives a json-liked format that contains dictionary in dictionary.

In line 317, the chat id from the user is extracted from the *msg* variable.

In line 318, the message sent from the user is extracted.

In line 322, the *query\_message* (message from the user) is checked if it starts with “/” in the string. If it is true, it means it is a command sent from the user. The command will instruct the bot what the user requested, be it status of Rig Server, or help, etc.

In line 324, slicing is used to remove “/” and then split the message from user into two arrays. The 1st array will contain the command while the 2nd array will contain additional message from the user. The 2nd message could be username, wallet key, etc.

We will now look at handle\_commands() called from the handle().

69 #handles commands

70 def handle\_commands(chat\_id, commands, message\_cilent):

71 markup\_contents = []

72 count = 1

73 dicts = "data/"

*markup\_contents* is used to create content for in-chat buttons which are made of array in array.

*count* is used as a counter to count the numbers of Rig Servers owned by the customer.

*dicts* contains the directory that the text file data of the Rig Servers are stored.

In this section, I will explain the code that will run if the command received from the user is “start”.

75 #will only print miner numbers but not contents(stats of the miners)

76 if commands == "start":

77 #store user's username which link to chat id for future use

78 userNameDict[chat\_id] = message\_cilent

79 #reset minerUpStatus content

80 minerUpStatus[chat\_id] = ["empty"]

81

82 while 1:

83 try:

84 #send stats of rig to cilent

85 aFile = open(dicts + message\_cilent + "\_" + str(count) + "\_" + "statsServer.txt")

86

87 #read stats of the selected miner

88 allDetails = aFile.readlines()

89

90 #if file time difference is less than 5mins(300sec), means it's still up (using epoch as time)

91 if abs(int(time.time()) - int(allDetails[0])) < 300:

92 #blue circle picture

93 minerUpStatus[chat\_id].append("🔵")

94 else:

95 #red circle picture

96 minerUpStatus[chat\_id].append("🔴")

The “start” section is used to send the customers the status of the Rig Server locally.

In line 78, *message\_cilent*(contains username) sent from the user is stored in a dictionary for future use. *chat\_id* is used as a key for easy reference to username.

In line 80, *minerUpStatus* is a dictionary that will store 🔵 or 🔴 of that customer’s Rig Server to show if the customer’s Rig Servers are up or down. Since the *count* variable starts from 1, the 1st array in *minerUpStatus*, array 0, must be created so that the data will not be -1.

Infinite *while* loop is created until an exception is thrown when no such file exist. Remember in the bash script, we stated that the Rig Number must be unique? Over here, the *count* variable is used to search for all the data text file own by the user. Once it increments to a number that the data text file does not exist, exception is thrown to exit the *while* loop. In line 85 shows the usage of count to search for the customer’s Rig Servers data. Example if customer’s username is “john” and he has two Rigs, he will have two data text file.

john\_1\_statsServer.txt and john\_2\_statsServer.txt

If no exceptions are thrown when reading the file, line 88 onwards will be executed.

In line 88, the whole text file is being read and will store as array in *allDetails*.

In line 91, the difference between the time of the data text file being created and the current time is being compared. If the difference is less than 5mins (300 seconds), 🔵 will be added to the *minerUpStatus* array, else 🔴. 🔵 is used to indicate that the miner is active.

Continue:

98 #exception thrown if file doesn't exist

99 except IOError:

100 break

101

102 except Exception as ex:

103 print(ex) #debugging

104

105 #pop out from RAM to save RAM space since no longer in used

106 aFile.close()

107

108 break

109

110 else:

111 #return <username>\_<miner no.>

112 markup\_contents.append([InlineKeyboardButton(text="miner" + str(count) + " " + str(minerUpStatus[chat\_id][count]), callback\_data=message\_cilent + "\_" + str(count))])

113

114 #pop out from RAM to save RAM space since no longer in used

115 aFile.close()

116

117 count += 1

118

119

120 #create Markup from

121 minerInfoButtons = InlineKeyboardMarkup(inline\_keyboard=markup\_contents)

122

123 #for future edit or delete of message

124 markupDict[chat\_id] = bot.sendMessage(chat\_id, text=message\_cilent + " status", reply\_markup=minerInfoButtons)

This chuck of code continues where we left off after the *try:* section.

In line 99, IOError exception will be thrown if the file does not exist. This means that the Python script has finished scanning all the data text file own by the customer.

In line 102, it will be triggered if any other unexpected exception is thrown. Print statement of that exception’s error will be printed on the command line for debugging purpose.

If no exception is thrown in the *try:* section, the interpreter will goes to the *else:* section.

In line 112, buttons for drop down list of miner(s) will be created. Remember I mentioned before markup buttons use array in array. The *count* variable is used here again to create the miner rig number. The *callback\_data* is the data that will be send back to the centralized server when the customer presses the markup buttons.

In line 121, the finalized array in array is used to create the markup button.

In line 124, markup buttons of the miners with the status of the miner is being sent to the user. The markupDict[chat\_id] will store the sent message for future edit of the same message. This is useful to edit the message and buttons to create a drop down list look later on.



In this section, I will explain the code that will run if the command received from the user is “all”.

127 #web scrapping from https://etn.nanopool.org/api

128 if commands == "all":

129 #message\_cilent will contain the wallet address

130 #get general wallet info

131 r = requests.get("https://api.nanopool.org/v1/etn/user/" + message\_cilent)

132 #print(r.json()) #debug

133 nanopool\_data = r.json()

134

135 #get earning by hash rate from https://etn.nanopool.org/api

136 r = requests.get("https://api.nanopool.org/v1/etn/approximated\_earnings/" + nanopool\_data['data']['hashrate'])

137 currentEarning = r.json()

138 r = requests.get("https://api.nanopool.org/v1/etn/approximated\_earnings/" + nanopool\_data['data']['avgHashrate']['h1'])

139 avg1hourEarning = r.json()

140 r = requests.get("https://api.nanopool.org/v1/etn/approximated\_earnings/" + nanopool\_data['data']['avgHashrate']['h6'])

141 avg6hourEarning = r.json()

142 r = requests.get("https://api.nanopool.org/v1/etn/approximated\_earnings/" + nanopool\_data['data']['avgHashrate']['h12'])

143 avg12hourEarning = r.json()

144

145 #get currency exchange rate from http://www.mas.gov.sg/Statistics/APIs/API-Documentation.aspx

146 r = requests.get("https://eservices.mas.gov.sg/api/action/datastore/search.json?resource\_id=95932927-c8bc-4e7a-b484-68a66a24edfe&filters[end\_of\_day]=" + time.strftime("%Y-%m-%d"))

147 currency\_converter = r.json()

The “all” section will sent the data from nanopool such as total hash and balance of ETN as well as money to be earned per month, back to the customer.

If the customer sends the command “all”, the message from the customer, *message\_cilent* variable, will contain the wallet key.

From line 131 to 143, extraction of data from the nanopool’s API is done and convert into json format.

In line 146 and line 147, the real time data of the currency exchange rate between USD and SGD is extracted from MAS to convert the earnings.

Continue:

149 #sending data from nanopool to user

150 bot.sendMessage(chat\_id, text="[Current]" + \

151 "\nHashrate: " + nanopool\_data['data']['hashrate'] + "H/s" + \

152 "\nUSD: $" + str(currentEarning['data']['month']['dollars']) + "/month" + \

153 "\nSGD: $" + str((float(currency\_converter['result']['records'][0]['usd\_sgd']) \* (currentEarning['data']['month']['dollars']) \* 10\*\*12 )/10\*\*12) + "/month" + \

154 "\n" + \

155 "\n[Average for last 1 hour]" + \

156 "\nHashrate: " + nanopool\_data['data']['avgHashrate']['h1'] + "H/s" + \

157 "\nUSD: $" + str(avg1hourEarning['data']['month']['dollars']) + "/month" + \

158 "\nSGD: $" + str((float(currency\_converter['result']['records'][0]['usd\_sgd']) \* (avg1hourEarning['data']['month']['dollars']) \* 10\*\*12 )/10\*\*12) + "/month" + \

159 "\n" + \

160 "\n[Average for last 6 hours]" + \

161 "\nHashrate: " + nanopool\_data['data']['avgHashrate']['h6'] + "H/s" + \

162 "\nUSD: $" + str(avg6hourEarning['data']['month']['dollars']) + "/month" + \

163 "\nSGD: $" + str((float(currency\_converter['result']['records'][0]['usd\_sgd']) \* (avg6hourEarning['data']['month']['dollars']) \* 10\*\*12 )/10\*\*12) + "/month" + \

164 "\n" + \

165 "\n[Average for last 24 hours]" + \

166 "\nHashrate: " + nanopool\_data['data']['avgHashrate']['h24'] + "H/s" + \

167 "\nUSD: $" + str(avg12hourEarning['data']['month']['dollars']) + "/month" + \

168 "\nSGD: $" + str((float(currency\_converter['result']['records'][0]['usd\_sgd']) \* (avg12hourEarning['data']['month']['dollars']) \* 10\*\*12 )/10\*\*12) + "/month" + \

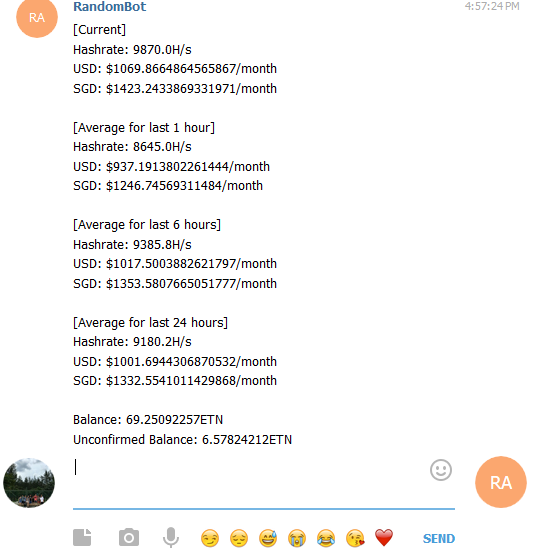
169 "\n" + \

170 "\nBalance: " + nanopool\_data['data']['balance'] + "ETN" + \

171 "\nUnconfirmed Balance: " + nanopool\_data['data']['unconfirmed\_balance'] + "ETN")

From line 150 to 171, the extracted data from nanopool API and MAS API is sent to the user.

In line 153, 158, 163, 168, “10\*\*12” is used to multiple the average earning and multiple to the currency converter before dividing by “10\*\*12” to eliminate precision error when calculating floating points.



In this section, I will explain the code that will run if the command received from the user is “Leon\_HaoJie\_SendToAll”.

174 if commands == "Leon\_HaoJie\_SendToAll":

175 for chat\_ids in userNameDict:

176 bot.sendMessage(chat\_ids, message\_cilent)

This is useful for admin to broadcast message to all customers.

In line 175, chat ids are extracted from temporary database to send to all messages. Remember earlier on we stored chat id and username of customers in “start” section? One of the usefulness of it is being used here.

We will now dive into handle\_markupkeyboard() where it is being called when receive call back data from those markup buttons.

179 #handles messages sent from inline keyboards

180 def handle\_markupkeyboard(msg):

181 #print(msg) #debug

182 query\_id, chat\_id, query\_data = telepot.glance(msg, flavor='callback\_query')

183 #incase used in group chat

184 chat\_id = msg['message']['chat']['id']

185 #remember markup returns <username>\_<miner no.>? So split them now

186 markupReturnData = msg['data'].split("\_")

187 markup\_contents = []

188 markup\_contents\_2ndHalf = []

189 minerUpStatus[chat\_id] = ["empty"]

190 count = 1

191 message\_server = ""

192 closeList = False

193 dicts = "data/"

194

195 if len(markupReturnData) == 1:

196 markupReturnData = msg['data'].split("+")

197 #set close list to TRUE

198 closeList = True

In line 182, *query\_id* is used to return acknowledgement packet else the customer’s telegram will be waiting for acknowledgement packet. The acknowledgement packet will be sent later at the end of the handle\_markupkeyboard().

From line 182 to 193 is the declaration of variables and data extraction from *msg* parameter. This *msg* parameter is the same as in handle().

In line 195, *markupReturnData* variable content’s length is being checked. If the data from *msg* contains “\_”, it will be successfully split. *markupReturnData* will have a length of 2, else length of 1. If the return data contains “\_”, it means it is a closed markup button that is being clicked. “+” means a dropped down button is being clicked. This is useful to identify whether to close or open the list from the button. The *closeList* variable will keep track of where to close the list or not.

continue:

201 while 1:

202 try:

203 #send stats of rig to cilent

204 aFile = open(dicts + userNameDict[chat\_id] + "\_" + str(count) + "\_" + "statsServer.txt")

205

206 #read stats of the selected miner

207 allDetails = aFile.readlines()

208

209 #if file time difference is less than 5mins(300sec), means it's still up (using epoch as time)

210 if abs(int(time.time()) - int(allDetails[0])) < 300:

211 #blue circle picture

212 minerUpStatus[chat\_id].append("🔵")

213 else:

214 #red circle picture

215 minerUpStatus[chat\_id].append("🔴")

216

217 # only read stats of the miner selected by the user

218 if str(count) != markupReturnData[1] or closeList:

219 aFile.close()

220 count += 1

221 continue

Line 201 to line 215 are the same as what I explained previously. However, this part is repeated so that less computation power is need as both methods use of the data in the text file are different. More computation power and time is needed here.

In line 218, it is useful to prevent the interpreter from reading continuing the code after line 221 as there is no need to read other data into the RAM from the text file (buffer). This is also useful to close the list if *closeList* is True.

Continue:

224 #if file does not exist

225 except IOError:

226 #exception thrown if file doesn't exist

227 break

228

229 #if cannot find username in dict (RAM's database)

230 except KeyError:

231 bot.sendMessage(chat\_id, "Error searching for username. \nPlease input \n/start <username> \nagain.")

232 return

233

234 except Exception as ex:

235 print(ex)

236 break

Line 225 to line 236 is the same as line 99. Hence, I will not explain again.

In line 230, KeyError exception is only thrown if the dictionary cannot find username in RAM’s database. This usually happens if the Python’s script is stopped and rerun. This causes the temporary data in RAM to be erased hence no data is found.

Continue:

238 else:

239 #clear it each markup must have text so example mine2 markup will send miner1's stats as text

240 message\_server = ""

241

242 #No. GPUs

243 allDetails[46] = allDetails[46].split()

244 message\_server += "\nNo. of GPUs = " + allDetails[46][1]

245

246 #Temp

247 allDetails[31] = allDetails[31].split()

248 message\_server += "\nTemp = " + allDetails[31][1] + "degrees\n"

249

250 #Status

251 allDetails[43] = allDetails[43].split()

252 message\_server += "\nMining Status = " + ((" ".join(allDetails[43][1:])) if allDetails[43][1] == "overheat:" else (allDetails[43][3] + " " + allDetails[43][4]))

253

254 #Hash rate

255 allDetails[49] = allDetails[49].split()

256 message\_server += "\nHash rate = " + allDetails[49][1]

257

258

259 #pop out from RAM to save RAM space since no longer inused

260 aFile.close()

261

262 count += 1

If exception is not being thrown or *continue* was not called, the *else*: section will be executed. In this section, other data of the Rig Server status is being read from the text file (buffer). The comments are written to each reading indicate what is being read.

Continue:

265 #create buttons of miner1 to the miner number that user selected to see the stats

266 #condition assignment is used so that it won't stop at int(markupReturnData[1]) if closeList is True

267 for i in range(1, count if closeList else int(markupReturnData[1])+1):

268 markup\_contents.append([InlineKeyboardButton(text="miner" + str(i) + " " + str(minerUpStatus[chat\_id][i]), callback\_data=markupReturnData[0] + "\_" + str(i))])

269

270 if i == int(markupReturnData[1]):

271 if not closeList:

272 if len(markup\_contents) != 0:

273 markup\_contents.pop()

274 markup\_contents.append([InlineKeyboardButton(text="miner" + str(i) + " " + str(minerUpStatus[chat\_id][i]), callback\_data=markupReturnData[0] + "+" + str(i))])

275

276 for i in range(int(markupReturnData[1])+1, count):

277

278 markup\_contents\_2ndHalf.append([InlineKeyboardButton(text="miner" + str(i) + " " + str(minerUpStatus[chat\_id][i]), callback\_data=markupReturnData[0] + "\_" + str(i))])

279

280

281 #create Markup from

282 minerInfoButtons = InlineKeyboardMarkup(inline\_keyboard=markup\_contents)

In this section, markup buttons are being created dynamically depending on the numbers of miners own by the customer. There for two *for* loops here to create two section of buttons. This only applies if there is a drop down list to be opened.

This is one section by itself. Only the 1st *for* loop is used.



Figure 1

This is two section. Top red square is the 1st section (created by 1st *for* loop), bottom red square is the 2nd section (created by 2nd *for* loop).



Figure 2

The reason why two *for* loops are needed is because for every markup buttons sent, there must be a text message sent on top of the button. A collection of buttons can have one text like the 1st example. However, for the 2nd example, there are text messages in-between the two buttons. Therefore, a 2nd *for* loop is need when there is a drop down list to be displayed when is in-between buttons.

In line 267, a conditional assignment is used in the *for* loop condition. When a button is clicked and the fake-list is not open, a return data with “\_” is in the string (refers to page 25). Remember the 2nd array after the split() will contain the rig number? This will allow the program to know when should the *for* loop loop until. The 2nd section will be created by the 2nd for loop. Remember that if there return data string from markup buttons contain “+”, the *closeList* will be True (refers to page 25)? When a fake-list is closed, there will only be one section (refers to figure 1). However, *markupReturnData[1]* contains the clicked return data rig number. This will causes the *for* loop to stop at the markup button that the customer pressed and attempt to create the 2nd section. This is a grave mistake as it is supposed to close the fake-list and not open the fake-list. Therefore, the conditional assignment will check if *closeList* is True. If True, the *for* loop will generate all the buttons available like figure 1, thus only the 1st *for* loop is used, and 2nd *for* loop will not be used as the initial condition will not even satisfy.

In line 282, the 1st section of the buttons are created using the array created in the 1st *for* loop.

Continue:

284 #useful when server restarted and user clicked on previous inLine buttons

285 try:

286 markupDict[chat\_id] = bot.editMessageText(telepot.message\_identifier(markupDict[chat\_id]), text=markupReturnData[0] + " status", reply\_markup=minerInfoButtons)

287 except Exception as ex:

288 #print(ex)

289 markupDict[chat\_id] = bot.sendMessage(chat\_id, text=markupReturnData[0] + " status", reply\_markup=minerInfoButtons)

290

291 minerInfoButtons = InlineKeyboardMarkup(inline\_keyboard=markup\_contents\_2ndHalf)

292

293

294 #Don't send it there are no 2nd half of the buttons after the drop down list

295 if len(message\_server) != 0:

296 #useful when server restarted and user clicked on previous inLine buttons

297 #markupDict[str(chat\_id)+str("\_1")] instead of markupDict[chat\_id] as is to store identifier of 2nd half of the message/buttons

298 try:

299 markupDict[str(chat\_id)+str("\_1")] = bot.editMessageText(telepot.message\_identifier(markupDict[str(chat\_id)+str("\_1")]), text=message\_server, reply\_markup=minerInfoButtons)

300 except:

301 markupDict[str(chat\_id)+str("\_1")] = bot.sendMessage(str(chat\_id)+str("\_1"), text=message\_server, reply\_markup=minerInfoButtons)

302 else:

303 #try to delete dropdownlist if user selected to close it

304 try:

305 bot.deleteMessage(telepot.message\_identifier(markupDict[str(chat\_id)+str("\_1")]))

306 except:

307 pass

308

309

310 # answer callback query or else telegram will forever wait on this

311 bot.answerCallbackQuery(query\_id)

In line 295, if there *message\_server* variable is not an empty string, the content in the IF statement will be executed. The *message\_server* variable contains the rig status obtained from the data text file. If this variable is not empty, it means that the user selected to open the fake-list, else it will be empty as the closeList variable will stop the code from reading the rig status from the data text file (refer to page 25).

In the IF statement, this will send the 2nd half of the message and markup buttons(if have) to the customer. “\_1” extension is used to store in *markupDict* dictionary as a key to store the 2nd half of the message for future edits.

In the ELSE statement, it will try to delete the 2nd half of the message if it exist (when *closeList* is True, there is no need for 2nd half of the message). This allows the messages to be less messy and create a menu drop down list looked.

In line 311, it returns an acknowledge packet to customer, or else the customer’s telegram will be waiting for this packet.

Finally, we will look at the checkRigUpStatus() which is run after every sleep cycle in the WHILE loop.

16 #check if Rig is up

17 def checkRigUpStatus():

18 #for comparison if there are changes in status

19 minerUpStatus\_check = {}

20 count = 1

21 dicts = "data/"

22

23 for chat\_id, username in userNameDict.items():

24 #minerUpStatus[chat\_id][0] is ["empty"] so must set the same

25 minerUpStatus\_check[chat\_id] = ["empty"]

26 count = 1

27

28 while 1:

29 try:

30 #send stats of rig to cilent

31 aFile = open(dicts + username + "\_" + str(count) + "\_" + "statsServer.txt")

32

33 #read stats of the selected miner

34 allDetails = aFile.readlines()

35

36 #if file time difference is less than 5mins(300sec), means it's still up (using epoch as time)

37 if abs(int(time.time()) - int(allDetails[0])) < 300:

38 #blue circle picture

39 minerUpStatus\_check[chat\_id].append("🔵")

40 else:

41 #red circle picture

42 minerUpStatus\_check[chat\_id].append("🔴")

Continue:

43 except IOError:

44 #exit while loop to stop checking

45 break

46

47 except Exception as ex:

48 print(ex) #debugging

49

50 #pop out from RAM to save RAM space since no longer inused

51 aFile.close()

52

53 break

54

55 else:

56 #pop out from RAM to save RAM space since no longer inused

57 aFile.close()

58

59 count += 1

60

61 if (len(minerUpStatus[chat\_id]) == len(minerUpStatus\_check[chat\_id])) and (minerUpStatus[chat\_id] == minerUpStatus\_check[chat\_id]):

62 continue

63 else:

64 #send customer an updated info

65 handle\_commands(chat\_id, "start", username)

This function is run every few minutes (given the time in the sleep() in WHILE loop). This function will check the status of the Rig Servers, whether it is up or down. If there are difference in the up or down status from the last time the customer requested (send a “/start” command or click on the markup buttons) to check the status of the Rig Server, a new list of markup buttons will be sent to the customer to update the customer of the new Rig Servers status.

In line 23, all the chat id are extracted from the *userNameDict* dictionary to check all the Rig Servers of all customers.

In line 25, I have explained before a nop is created in array 0 position as count starts from 1.

From line 28 to line 59, it is the same as the checking of rig status in handle\_commands()’s IF statement for “/start”. However, no data is sent to the customer in this function. The Rig up or down status is output to a different dictionary (*minerUpStatus\_check*[]) then compare with preciously stored rig status in line 61. If there is a difference, ELSE statement will be used and handle\_commands() will be called in line 65.