



# 암호화폐 트레이딩 교육

Day 2 (2/2)

실시간 시세처리

Kangwuk Heo  
calvin.heo@gmail.com



---

## Contents

실시간 시세처리 3

SECTION 1: 웹소켓 .....4

---

## Overview

암호화폐 트레이딩 교육 과정에서, 실습을 통해서, 트레이딩이 무엇인지, 트레이딩을 어떤 방식으로 구성하고 진행하는지를 배울 수 있습니다.

## 실시간 시세처리

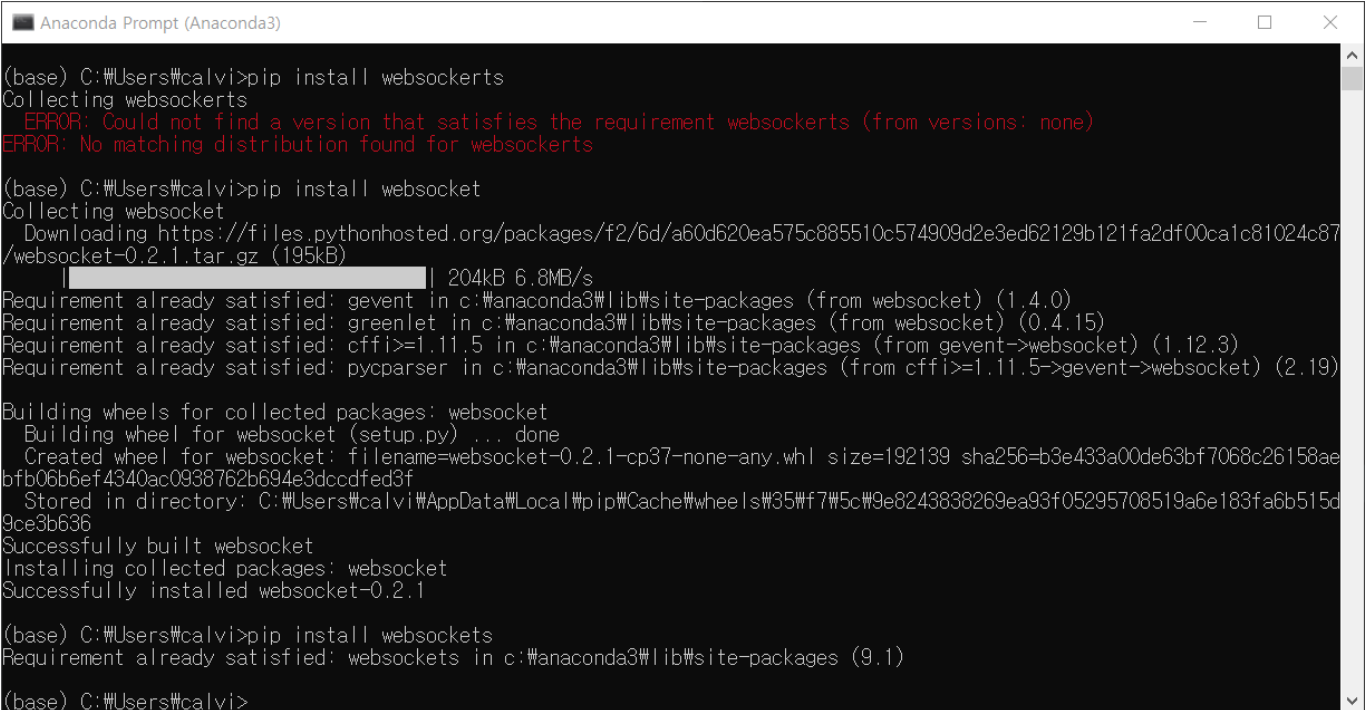
**Section 1** 은 웹소켓(websocket)을 이용한 구독(subscribe) 방식으로 가상화폐 거래소의 실시간 데이터를 처리하는 방법을 알아본다.

## Section 1: 웹소켓

웹소켓(Websocket) 방식은 클라이언트와 서버사이에서 실시간 양방향 통신을 위한 기술입니다. 대부분의 가상화폐 거래소는 이러한 웹소켓 방식으로 실시간 데이터를 처리하고 있습니다.

Section 1 에서 웹소켓 기술에 대해서 알아보고, 가상화폐 거래소와의 연계를 통해 데이터를 처리해 보는 방법을 알아보도록 하겠습니다.

1. 아나콘다(Anaconda) 메뉴에서, Anaconda Prompt 아이콘을 클릭하여, 보여지는 Anaconda Prompt 창에서 websockets 모듈을 설치합니다.



```
(base) C:\Users\calvi>pip install websockerts
Collecting websockerts
  ERROR: Could not find a version that satisfies the requirement websockerts (from versions: none)
ERROR: No matching distribution found for websockerts

(base) C:\Users\calvi>pip install websocket
Collecting websocket
  Downloading https://files.pythonhosted.org/packages/f2/6d/a60d620ea575c885510c574909d2e3ed62129b121fa2df00ca1c81024c87/websocket-0.2.1.tar.gz (195kB)
    | 204kB 6.8MB/s
Requirement already satisfied: gevent in c:\anaconda3\lib\site-packages (from websocket) (1.4.0)
Requirement already satisfied: greenlet in c:\anaconda3\lib\site-packages (from websocket) (0.4.15)
Requirement already satisfied: cffi>=1.11.5 in c:\anaconda3\lib\site-packages (from gevent->websocket) (1.12.3)
Requirement already satisfied: pycparser in c:\anaconda3\lib\site-packages (from cffi>=1.11.5->gevent->websocket) (2.19)

Building wheels for collected packages: websocket
  Building wheel for websocket (setup.py) ... done
  Created wheel for websocket: filename=websocket-0.2.1-cp37-none-any.whl size=192139 sha256=b3e433a00de63bf7068c26158aebfb06b6ef4340ac0938762b694e3dcccdfed3f
  Stored in directory: C:\Users\calvi\AppData\Local\pip\Cache\wheels\35\7f\5c\9e8243838269ea93f05295708519a6e183fa6b515d9ce3b636
Successfully built websocket
Installing collected packages: websocket
Successfully installed websocket-0.2.1

(base) C:\Users\calvi>pip install websockets
Requirement already satisfied: websockets in c:\anaconda3\lib\site-packages (9.1)

(base) C:\Users\calvi>
```

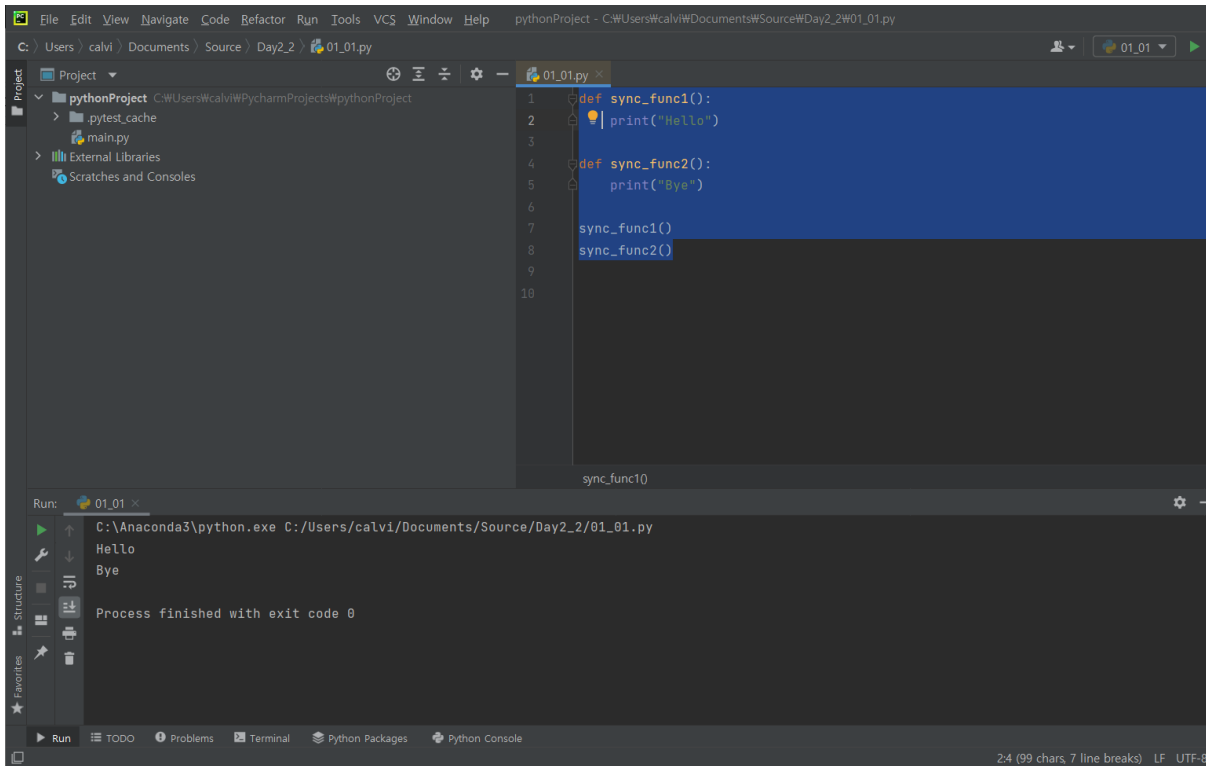
2. 동기방식과 비동기 방식의 차이를 알아보도록 합니다.

```
def sync_func1():
    print("Hello")

def sync_func2():
    print("Bye")

sync_func1()
sync_func2()
```

동기방식으로 처리되는 로직으로, 정의된 코드 순서에 따라 처리되는 것을 확인할 수 있습니다.



비동기방식으로 호출되는 방식으로, Async 키워드가 있는 함수를 코루틴(coroutine)이라고 합니다.

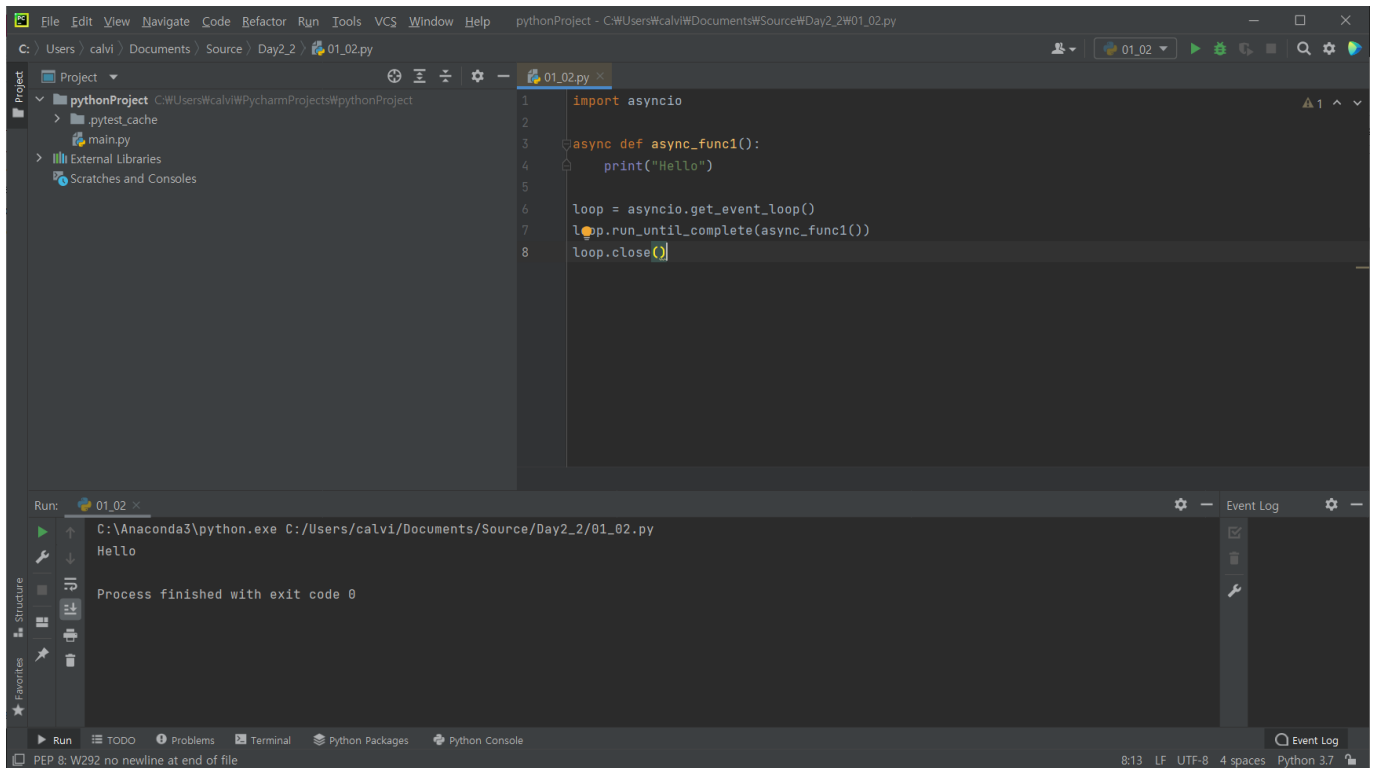
```
import asyncio

async def async_func1():
    print("Hello")

loop = asyncio.get_event_loop()
loop.run_until_complete(async_func1())
loop.close()
```

코루틴이 정상적으로 동작하기 위해서는 스케줄러인 이벤트 루프가 필요합니다.

Asyncio 를 사용할 경우, 기본적으로 `asyncio.run` 함수를 사용하여 이벤트 루프를 처리해야 합니다.



\_\_3. 비동기 처리를 위한 **asyncio** 를 기반으로 코루틴을 만들어 봅니다.

```
import asyncio

async def make_car1():
    print("Car1 Start")
    await asyncio.sleep(3)
    print("Car1 End")

async def make_car2():
    print("Car2 Start")
    await asyncio.sleep(5)
    print("Car2 End")

async def main():
    coro1 = make_car1()
    coro2 = make_car2()
    await asyncio.gather(
        coro1,
        coro2
    )

print("Main Start")
asyncio.run(main())
print("Main End")
```

호출한

코루틴은 **Car1**과 **Car2**의 코루틴을 각각 호출하여, 각각의 코루틴이 실행된 후에 **Main** 코루틴이 종료되는 구조입니다.

4. 비동기 처리 영역에서, 코루틴이 처리되면서 **Return** 값을 전달하고, 배열로 저장되는 구조로 동작 여부를 확인합니다.

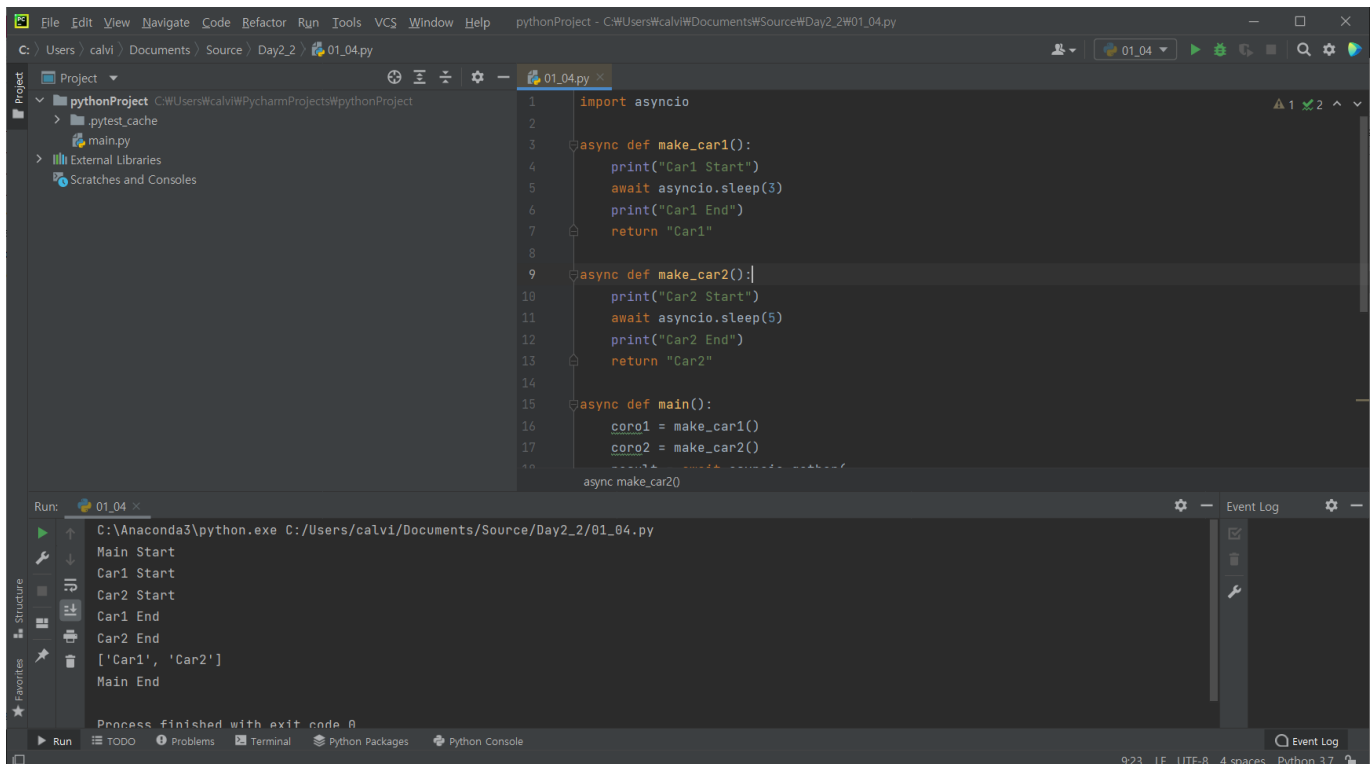
```
import asyncio

async def make_car1():
    print("Car1 Start")
    await asyncio.sleep(3)
    print("Car1 End")
    return "Car1"

async def make_car2():
    print("Car2 Start")
    await asyncio.sleep(5)
    print("Car2 End")
    return "Car2"

async def main():
    coro1 = make_car1()
    coro2 = make_car2()
    result = await asyncio.gather(
        coro1,
        coro2
    )
    print(result)

print("Main Start")
asyncio.run(main())
print("Main End")
```



The screenshot displays the PyCharm IDE interface. The main editor window shows the Python code from the previous block. The Run console at the bottom shows the output of the program execution:

```
Run: 01_04.py
C:\Anaconda3\python.exe C:/Users/calvi/Documents/Source/Day2_2/01_04.py
Main Start
Car1 Start
Car2 Start
Car1 End
Car2 End
['Car1', 'Car2']
Main End
Process finished with exit code 0
```

The status bar at the bottom indicates the Python version is 3.7.



\_\_5. 웹소켓 기반으로 실시간 데이터를 처리하기 위해서는 프로세스와 스레드를 적용해야 합니다.

기본적으로 프로세스의 실행 단위를 스레드라고 하며, 프로세스는 최소 하나의 스레드 구성되는 형태를 가집니다.

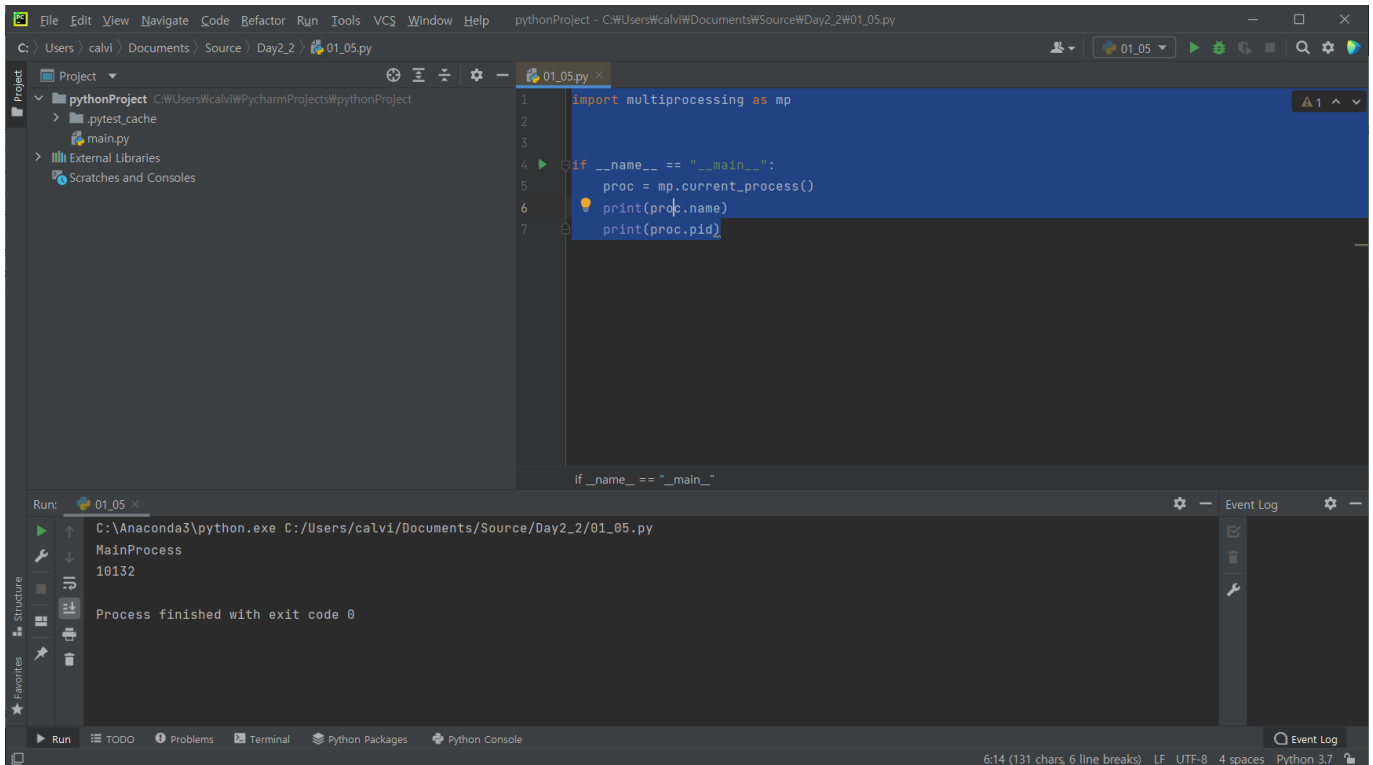
프로세스간에 데이터를 공유하기 위해서는 별도의 통신 방식을 적용해야 하며, 한 프로세스내에 구성된 스레드 간에는 자원을 공유할 수 있습니다.

프로세스를 사용하기 위해서는 **multiprocessing** 모듈을 사용해야 합니다.

```
import multiprocessing as mp

if __name__ == "__main__":
    proc = mp.current_process()
    print(proc.name)
    print(proc.pid)
```

Multiprocessing 모듈의 **current\_process** 함수를 통해서 현재 실행되는 프로세스에 대한 정보를 가지고 오게 됩니다. (PID 는 운영체제가 가 프로세스에 부여한 고유 번호)



6. 프로세스는 부모프로세스와 자식 프로세스로 나누어 지게 되며, 부모 프로세스가 자식 프로세스를 새로 생성하기 위해 운영체제에 요청하는 것을 프로세스 스폰닝(spawning) 이라고 하며, 스폰닝 처리 방식을 실행해 봅니다.

```
import multiprocessing as mp
import time

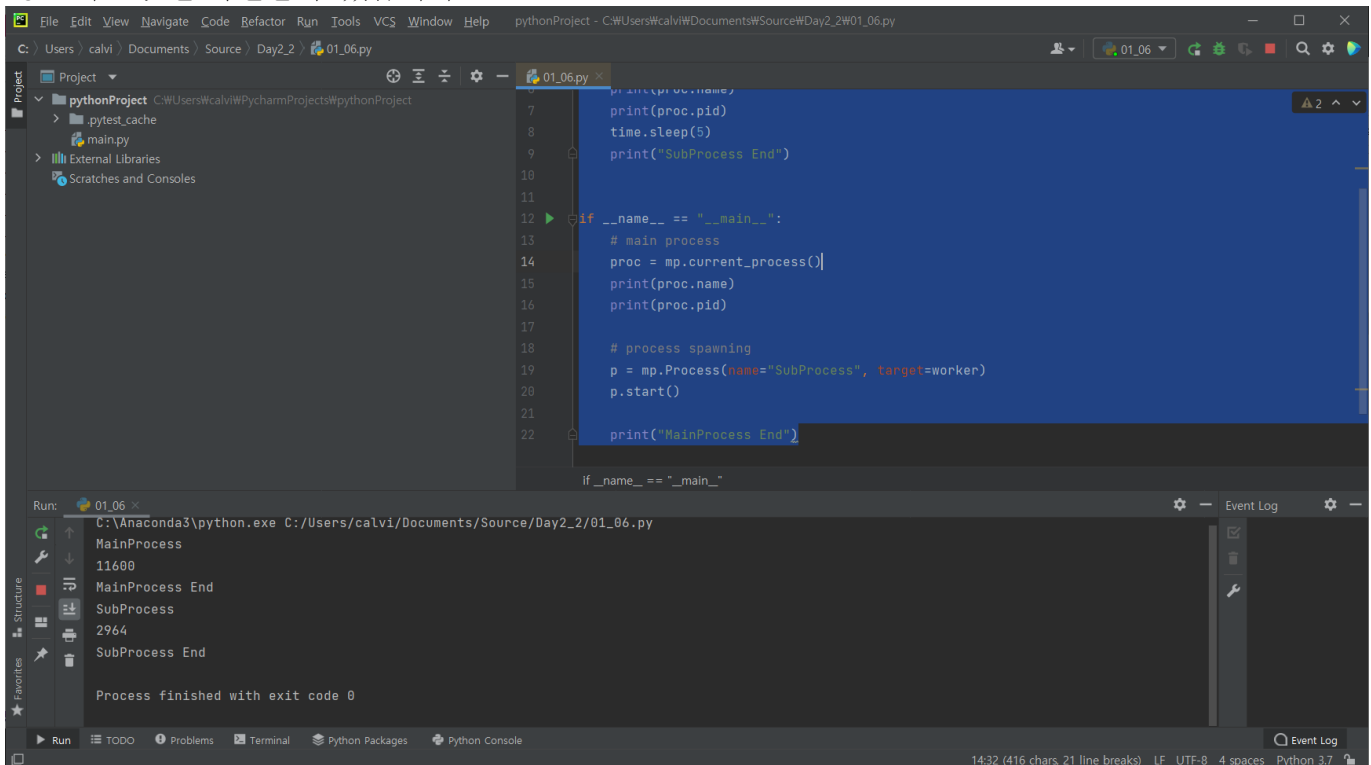
def worker():
    proc = mp.current_process()
    print(proc.name)
    print(proc.pid)
    time.sleep(5)
    print("SubProcess End")

if __name__ == "__main__":
    # main process
    proc = mp.current_process()
    print(proc.name)
    print(proc.pid)

    # process spawning
    p = mp.Process(name="SubProcess", target=worker)
    p.start()

    print("MainProcess End")
```

부모 프로세스인 Main process 가 실행되어, 종료되고 나면 자식 프로세스인 sub process 가 실행되고 종료되는 것을 확인할 수 있습니다.



7. 웹소켓 모듈 기반으로 가상화폐 거래소의 웹소켓 서버로부터 실시간 데이터를 전송하는 클라이언트를 실행해 봅니다.

```
import websockets
import asyncio

async def bithumb_ws_client():
    uri = "wss://pubwss.bithumb.com/pub/ws"

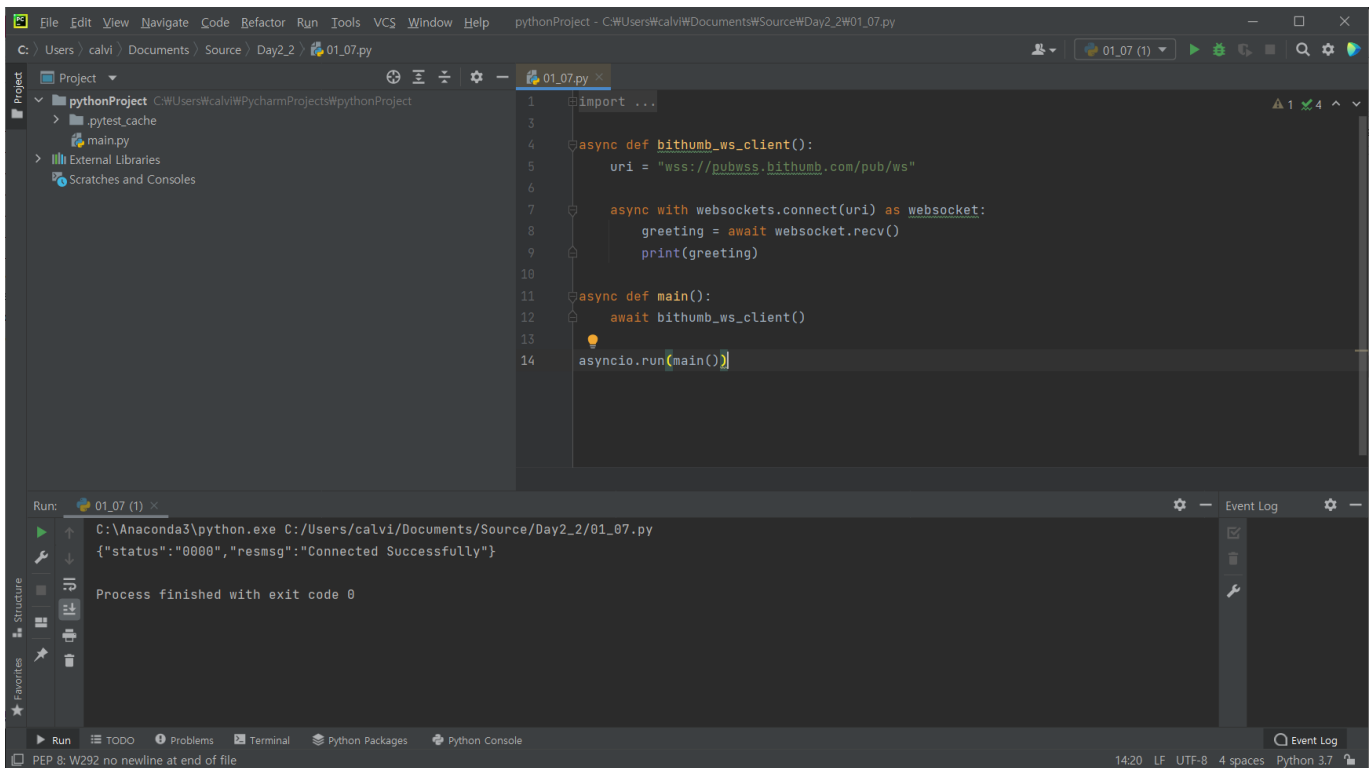
    async with websockets.connect(uri) as websocket:
        greeting = await websocket.recv()
        print(greeting)

async def main():
    await bithumb_ws_client()

asyncio.run(main())
```

가상화폐 거래소의 웹소켓 서버에서 접속하여 메시지를 하나 수신해서 출력하는 결과 값을 확인합니다.

`{"status": "0000", "resmsg": "Connected Successfully"}`



```
File Edit View Navigate Code Refactor Run Tools VCS Window Help pythonProject - C:\Users\calvi\Documents\Source\Day2_2\01_07.py
C:\Users\calvi\Documents\Source\Day2_2\01_07.py
Project
  pythonProject C:\Users\calvi\PycharmProjects\pythonProject
    .pytest_cache
    main.py
    External Libraries
    Scratches and Consoles
  01_07.py x
1 import ...
2
3
4 async def bithumb_ws_client():
5     uri = "wss://pubwss.bithumb.com/pub/ws"
6
7     async with websockets.connect(uri) as websocket:
8         greeting = await websocket.recv()
9         print(greeting)
10
11 async def main():
12     await bithumb_ws_client()
13
14 asyncio.run(main())
Run: 01_07 (1) x
C:\Anaconda3\python.exe C:/Users/calvi/Documents/Source/Day2_2/01_07.py
{"status": "0000", "resmsg": "Connected Successfully"}
Process finished with exit code 0
PEP 8: W292 no newline at end of file
14:20 LF UTF-8 4 spaces Python 3.7
```

8. 가상화폐 거래소의 웹소켓 서버 구독 요청을 통해, 실시간 데이터를 수신하는 방법으로 비트코인의 현재가를 받는 로직을 실행해 봅니다.

```
import websockets
import asyncio
import json

async def bithumb_ws_client():
    uri = "wss://pubwss.bithumb.com/pub/ws"

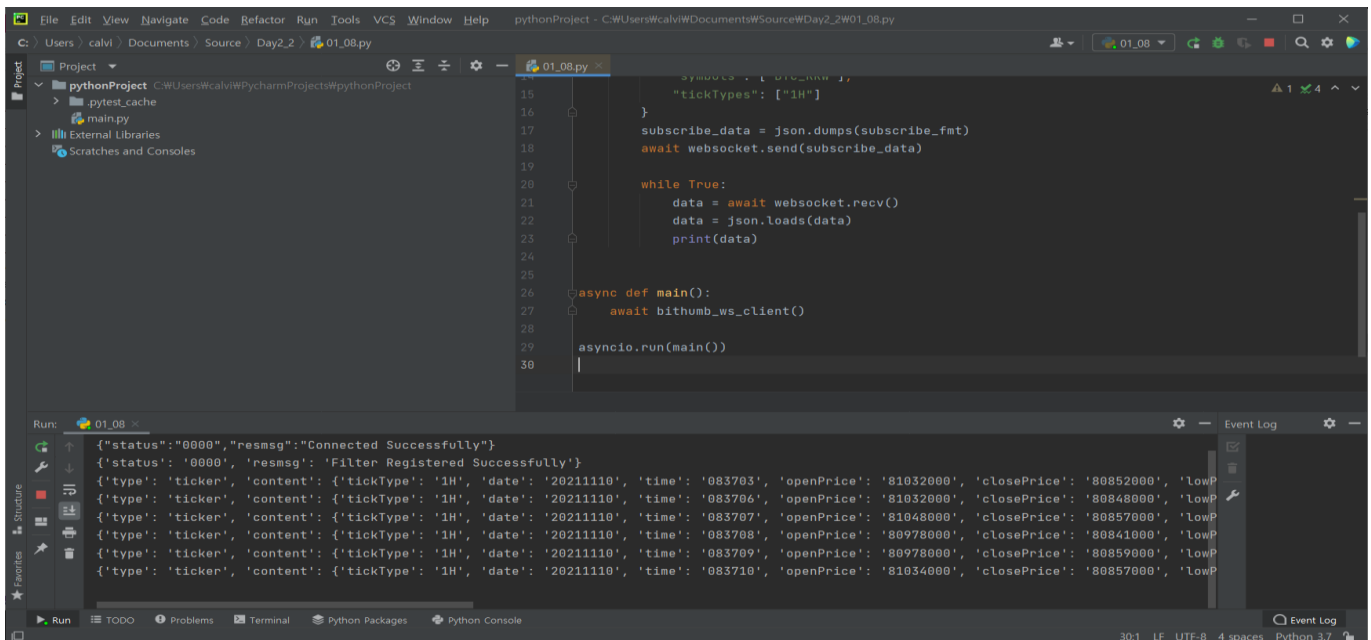
    async with websockets.connect(uri, ping_interval=None) as websocket:
        greeting = await websocket.recv()
        print(greeting)

        subscribe_fmt = {
            "type": "ticker",
            "symbols": ["BTC_KRW"],
            "tickTypes": ["1H"]
        }
        subscribe_data = json.dumps(subscribe_fmt)
        await websocket.send(subscribe_data)

        while True:
            data = await websocket.recv()
            data = json.loads(data)
            print(data)

async def main():
    await bithumb_ws_client()

asyncio.run(main())
```



The screenshot shows the PyCharm IDE with the file `01_08.py` open. The code is the same as shown in the previous block. The Run console at the bottom displays the output of the script, showing a successful connection to the Bithumb WebSocket server and a series of ticker data for BTC\_KRW. The data includes status, resmsg, type, content, and various price and volume metrics.

```
{
  "status": "0000",
  "resmsg": "Connected Successfully"
}
{
  "status": "0000",
  "resmsg": "Filter Registered Successfully"
}
{
  "type": "ticker",
  "content": {
    "tickType": "1H",
    "date": "20211110",
    "time": "083703",
    "openPrice": "81032000",
    "closePrice": "80852000",
    "lowP": "80852000",
    "highP": "81032000",
    "open": "81032000",
    "close": "80852000",
    "low": "80852000",
    "high": "81032000",
    "volume": "1000000000",
    "volumeUnit": "KRW"
  }
}
{
  "type": "ticker",
  "content": {
    "tickType": "1H",
    "date": "20211110",
    "time": "083706",
    "openPrice": "81032000",
    "closePrice": "80848000",
    "lowP": "80848000",
    "highP": "81032000",
    "open": "81032000",
    "close": "80848000",
    "low": "80848000",
    "high": "81032000",
    "volume": "1000000000",
    "volumeUnit": "KRW"
  }
}
{
  "type": "ticker",
  "content": {
    "tickType": "1H",
    "date": "20211110",
    "time": "083707",
    "openPrice": "81048000",
    "closePrice": "80857000",
    "lowP": "80857000",
    "highP": "81048000",
    "open": "81048000",
    "close": "80857000",
    "low": "80857000",
    "high": "81048000",
    "volume": "1000000000",
    "volumeUnit": "KRW"
  }
}
{
  "type": "ticker",
  "content": {
    "tickType": "1H",
    "date": "20211110",
    "time": "083708",
    "openPrice": "80978000",
    "closePrice": "80841000",
    "lowP": "80841000",
    "highP": "80978000",
    "open": "80978000",
    "close": "80841000",
    "low": "80841000",
    "high": "80978000",
    "volume": "1000000000",
    "volumeUnit": "KRW"
  }
}
{
  "type": "ticker",
  "content": {
    "tickType": "1H",
    "date": "20211110",
    "time": "083709",
    "openPrice": "80978000",
    "closePrice": "80859000",
    "lowP": "80859000",
    "highP": "80978000",
    "open": "80978000",
    "close": "80859000",
    "low": "80859000",
    "high": "80978000",
    "volume": "1000000000",
    "volumeUnit": "KRW"
  }
}
{
  "type": "ticker",
  "content": {
    "tickType": "1H",
    "date": "20211110",
    "time": "083710",
    "openPrice": "81034000",
    "closePrice": "80857000",
    "lowP": "80857000",
    "highP": "81034000",
    "open": "81034000",
    "close": "80857000",
    "low": "80857000",
    "high": "81034000",
    "volume": "1000000000",
    "volumeUnit": "KRW"
  }
}
```

\_\_9. 가상화폐 거래소에서 받은 실시간 데이터를 GUI를 통해서 실행해 봅니다.

```
import multiprocessing as mp
import websockets
import asyncio
import json
import sys
import datetime
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

async def bithumb_ws_client(q):
    uri = "wss://pubwss.bithumb.com/pub/ws"

    async with websockets.connect(uri, ping_interval=None) as websocket:
        subscribe_fmt = {
            "type": "ticker",
            "symbols": ["BTC_KRW"],
            "tickTypes": ["1H"]
        }
        subscribe_data = json.dumps(subscribe_fmt)
        await websocket.send(subscribe_data)

        while True:
            data = await websocket.recv()
            data = json.loads(data)
            q.put(data)

async def main(q):
    await bithumb_ws_client(q)

def producer(q):
    asyncio.run(main(q))

class Consumer(QThread):
    popped = pyqtSignal(dict)

    def __init__(self, q):
        super().__init__()
        self.q = q

    def run(self):
        while True:
            if not self.q.empty():
                data = self.q.get()
                self.popped.emit(data)
```

```

class MyWindow(QMainWindow):
    def __init__(self, q):
        super().__init__()
        self.setGeometry(200, 200, 400, 200)
        self.setWindowTitle("Bithumb Websocket with PyQt")

        # thread for data consumer
        self.consumer = Consumer(q)
        self.consumer.popped.connect(self.print_data)
        self.consumer.start()

        # widget
        self.label = QLabel("Bitcoin: ", self)
        self.label.move(10, 10)

        # QLineEdit
        self.line_edit = QLineEdit(" ", self)
        self.line_edit.resize(150, 30)
        self.line_edit.move(100, 10)

    @pyqtSlot(dict)
    def print_data(self, data):
        content = data.get('content')
        if content is not None:
            current_price = int(content.get('closePrice'))
            self.line_edit.setText(format(current_price, ",d"))

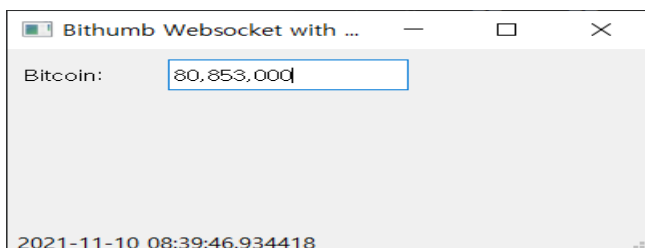
        now = datetime.datetime.now()
        self.statusBar().showMessage(str(now))

if __name__ == "__main__":
    q = mp.Queue()
    p = mp.Process(name="Producer", target=producer, args=(q,),
daemon=True)
    p.start()

    # Main process
    app = QApplication(sys.argv)
    mywindow = MyWindow(q)
    mywindow.show()
    app.exec_()

```

PyQt5 GUI 기반으로 비트코인 현재 가격을 가상화폐 거래소의 웹소켓 서버를 통해 실시간으로 데이터를 전달받아 GUI에 반영되게 됩니다.



- \_\_10. 가상화폐 거래소의 파이썬 모듈을 통해서, 서버로부터 데이터를 받은 후, 내부에 구성한 큐에 데이터를 저장하고, 큐에 저장된 데이터를 스레드를 통해서 가져가서 GUI에 보여주는 로직을 실행해 봅니다.

```
from pyupbit import WebSocketManager
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtCore import *

class Worker(QThread):
    recv = pyqtSignal(dict)

    def run(self):
        # create websocket for Bithumb
        wm = WebSocketManager("ticker", ["KRW-BTC"])
        while True:
            data = wm.get()
            print(data)
            self.recv.emit(data)

class MyWindow(QMainWindow):
    def __init__(self):
        super().__init__()

        label = QLabel("BTC", self)
        label.move(20, 20)

        self.price = QLabel("", self)
        self.price.move(80, 20)
        self.price.resize(100, 20)

        button = QPushButton("Start", self)
        button.move(20, 50)
        button.clicked.connect(self.click_btn)

        self.th = Worker()
        self.th.recv.connect(self.receive_msg)

    @pyqtSlot(dict)
    def receive_msg(self, data):
        print(data)
        close_price = data.get("trade_price")
        self.price.setText(str(close_price))

    def click_btn(self):
        self.th.start()

if __name__ == '__main__':
    app = QApplication(sys.argv)
    mywindow = MyWindow()
    mywindow.show()
    app.exec_()
```

실행을 하게 되면, UI가 보이게 되고, 여기서 **Start** 버튼을 클릭하면 데이터 요청을 해서, 수신한 데이터를 **Console** 창에 보여지게 됩니다.

