TCP CHAT

- 다중채팅
- 누가 말하는지 알도록 할 것
- 상속, super, @classmethod, @staticmethod

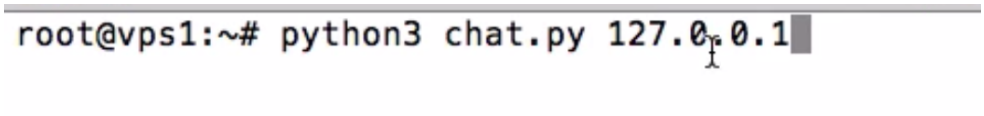
Command

kill progress

- lsof -i :<port number> <==========띄어쓰기 정확히
- 해당 <port number>에서 돌아가고 있는 progress들이 모두 나옴
- pid를 참고
- kill <해당 pid> 하면
- ok

sys.argv

(sys.argv) > 1): => import 해온 sys에서 argv는 무엇인가?



=> 위의 예를 보면 => “python3 chat.py” 뒤(127.0.0.1)가 바로 “두번째(==[1]) argv”

except keyboardInterrupt:

- 여기서 KeyboardInterrupt 는 Terminal에서 ctrl + c 하는 것을 말한다.

문법 for socket.send, socket.recv

if c = socket(AF_INET, SOCK_STREAM),

c.send => server가 보낸다 client에게
c.recv => server가 받는다 client로 부터

=> 주어 (server) , 동사(after dot), 목적어(before dot)

Socket

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # connection type, protocol

print("s: "s)

server = 'google.com'
port = 80

server_ip = socket.gethostbyname(server)

print("server_ip: ", server_ip)

request = "GET / HTTP/1.1\nHOST: " + server + "\n\n"

s.connect((server,port))
s.send(request.encode())                    # from python3 differentiate between string and byte that's why using encode()
result = s.recv(4096)                       # how much data are we gonna download in one moment

print(result)

while (len(result) > 0):
    print(result)
    result = s.recv(1024)
```

Output :

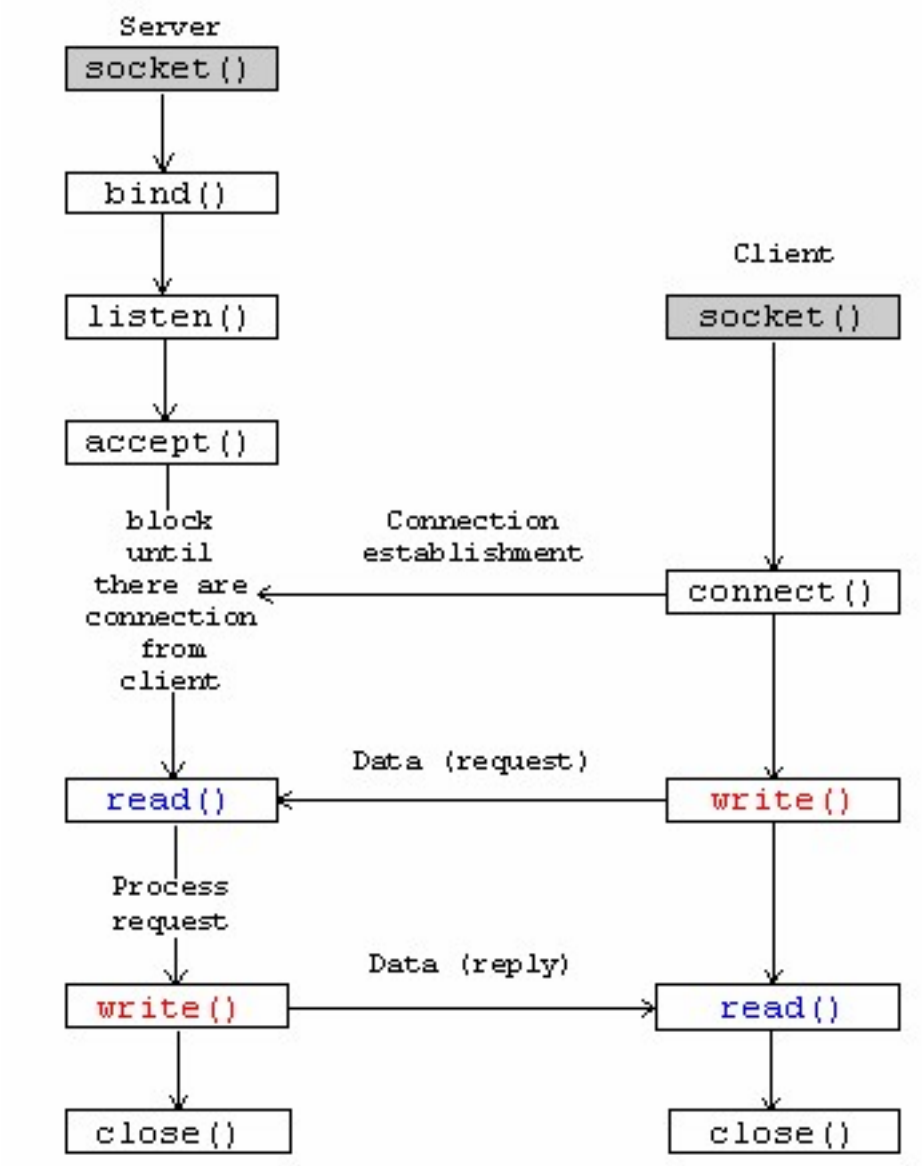
```
/Users/imac/Desktop/tcp_chat/venv/bin/python /Users/imac/Desktop/tcp_chat/tcp_chat/intergrated_classes_copy.py
s:  <socket.socket fd=3, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM, proto=0, laddr=('0.0.0.0', 0)>
server_ip:  216.58.199.14
result:  b'HTTP/1.1 301 Moved Permanently\r\nLocation: http://www.google.com/\r\nContent-Type: text/html; charset=UTF-8\r\nDate: Wed, 02 Jan 2019 08:54:05 GMT\
b'HTTP/1.1 301 Moved Permanently\r\nLocation: http://www.google.com/\r\nContent-Type: text/html; charset=UTF-8\r\nDate: Wed, 02 Jan 2019 08:54:05 GMT\r\nExpires
```

위의 코드에서

- socket 연결 => 여기선 IPV4(우리가 사용하길 원하는 "연결 type"), 통신protocol(TCP)
- socket이 찍힌다.
- 도메인 네임
- 도메인 네임으로 ip address를 얻는다.
- s.connect((server,port)) => 인자들의 조건으로 socket을 연결
- encode(), decode()의 이유는 python3부터 str과 byte를 엄격하게 구분하기 때문

What is difference between socket.bind() and socket.connect()

To make understanding better , lets find out where exactly bind and connect comes into picture,

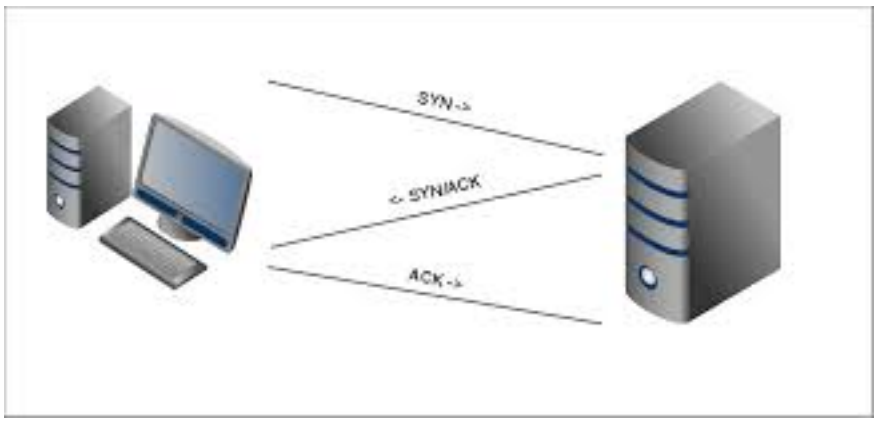


Further to positioning of two calls , as clarified by Sourav,

bind() associates the socket with its local address [that's why server side binds, so that clients can use that address to connect to server.] connect() is used to connect to a remote [server] address, that's why is client side, connect [read as: connect to server] is used.

We cannot use them interchangeably (even when we have client/server on same machine) because of specific roles and corresponding implementation.

I will further recommend to correlate these calls TCP/IP handshake .



So , who will send SYN here , it will be connect() . While bind() is used for defining the communication end point.

Hope this helps!!

=>bind() => Server side
=>connect() => Client side

Threading

`threading` — Thread-based parallelism(평행)

Source code: Lib/threading.py

This module constructs higher-level threading interfaces on top of the lower level `_thread` module. See also the `queue` module.

```
import threading
from queue import Queue
import time

print_lock = threading.Lock()

def exampleJob(worker):
    time.sleep(0.5)

    with print_lock:
        print(threading.current_thread().name, worker)

def threader():
    while True:
        worker = q.get()
        exampleJob(worker)
        q.task_done()

q = Queue()

for x in range(10):
    t = threading.Thread(target = threader)
    t.daemon = True
    t.start()

start = time.time()

for worker in range(20):
    q.put(worker)

q.join()

print('Entire job took: ', time.time()-start)
```

```
/Users/imac/Desktop/tcp_chat/venv/bin/pyth
Thread-3 1
Thread-1 0
Thread-2 4
Thread-7 3
Thread-6 5
Thread-9 6
Thread-4 7
Thread-10 8
Thread-8 9
Thread-5 2
Thread-1 11
Thread-3 10
Thread-5 19
Thread-7 13
Thread-6 14
Thread-4 16
Thread-10 17
Thread-8 18
Thread-2 12
Thread-9 15
Entire job took:  1.0052688121795654

Process finished with exit code 0
```

Threading의 동작 방식과 효용가치를 볼수 있는 예제이다.
=> 10개의 thread를 만든다.
=> 그 thread들은 threader 함수를 이용한다.
=> daemon => main threader 끝나면, 관련 thread들이 모두 끝나게 하는 세팅 => 기본 True로 해줘야 한다.
=> thread들은 만듦과 동시에 start!
=> threader가 돌아가지만 현재는 queue에 아무것도 없기때문에 가져오는게 없다.
=> q에 worker 20개를 넣는다.
=> threader가 실행되고
=> exampleJob은 0.5초의 시간을 할애하지만 => 그래서 20개를 순서대로 처리하면 20초 이상이 될 것 같지만
=> 그렇지 않다
=> thread는 여러개가 동시에 진행되게 할 수있는 power가 있기 때문!
=> q.join()으로 queue안의 작업이 모두 끝날때까지 기다리고 끝난 뒤 시간을 계산해보면 약 1초 정도 밖에 안나온다!

+Synchronizing Threads

```
threadLock = threading.Lock()
```

daemon => main threader 끝나면, 관련 thread들이 모두 끝나게 하는 세팅 => 기본 True로 해줘야 한다.

A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before `start()` is called, otherwise `RuntimeError` is raised. Its initial value is inherited from the creating thread; the main thread is not a daemon thread and therefore all threads created in the main thread default to `daemon` = `False`.

The entire Python program exits when no alive non-daemon threads are left.

thread.join(timeout=None) => thread가 다 끝날때까지 기다린다.

Wait until the thread terminates. This blocks the calling thread until the thread whose `join()`method is called terminates – either normally or through an unhandled exception – or until the optional timeout occurs.

queue =>여러개를 생성하고 사용할 수 있는 thread programming에 적절히 사용되어질 수 있다, locking 에도 쓰이고

The `queue` module implements multi-producer, multi-consumer queues. It is especially useful in threaded programming when information must be exchanged safely between multiple threads. The `Queue` class in this module implements all the required locking semantics. It depends on the availability of thread support in Python; see the `threading` module.

+ Queue.join()
q = Queue()
q.join() => When you call queue.join() in the main thread, all it does is block the main threads until the workers have processed everything that is in the queue. It does not stop the worker threads, which continue executing their infinite loops.
If the worker threads are non-deamon, their continuing execution prevents the program from stopping irrespective of whether the main
thread has finished.

AF_INET

AF_INET is an address family that is used to designate the type of addresses that your socket can communicate with (in this case, Internet Protocol v4 addresses). When you create a socket, you have to specify its address family, and then you can only use addresses of that type with the socket. The Linux kernel, for example, supports 29 other address families such as UNIX (AF_UNIX) sockets and IPX (AF_IPX), and also communications with IRDA and Bluetooth (AF_IRDA and AF_BLUETOOTH, but it is doubtful you'll use these at such a low level).

For the most part, sticking with AF_INET for socket programming over a network is the safest option. There is also AF_INET6 for Internet Protocol v6 addresses.

BUFSIZ

The bufsize argument of 1024 used above is the maximum amount of data to be received at once. It doesn't mean that recv() will return 1024 bytes.

1024

Special use in computers[edit]

In binary notation, 1024 is represented as 10000000000, making it a simple round number occurring frequently in computer applications.

1024 is the maximum number of computer memory addresses that can be referenced with ten binary switches. This is the origin of the organization of computer memory into 1024-byte chunks or kibibytes.

In the Rich Text Format (RTF), language code 1024 indicates the text is not in any language and should be skipped over when proofing. Most used languages codes in RTF are integers slightly over 1024.

1024×768 pixels and 1280×1024 pixels are common standards of display resolution.

Python dict

- Python은 dict에서 값을 가져올때 ['] bracket으로만 찾는다.
- Dot notation은 class에서 method나 member 변수를 찾을때 사용
- somedict.get() 은 dict 라는 class에 대한 소속 magic method를 호출한 개념의 것이므로 somedict에 있는 key로 value를 가져오는 것과는 아예 다른 개념이다. 헷갈리지 말 것.

-

그럼 somedict.get()의 쓰임새를 살펴보자

```
print(3, config.get('hi', 'say'))
print(4, config['hi'])
```

위의 첫번째 라인은: Dict config에 ‘hi’라는 key에 대한 vlaue값을 가져오고 싶은데 만약 ‘hi’라는 key가 없을 경우 ’say’라는 value값이 있는 것으로 치고 가져오겠다는 말. =>

```
3 hi
```

=> 근데 결론적으로 그때만(일시적으로) 그렇게 대용품으로서 쓰는 것이고
=> 실제 config에 key = ‘hi’, value = ‘say’의 pair가 추가 되는 것은 아니다.
=> 그러므로 위의 두번째 라인을 찍어보면
=> 아래와 같이 keyError 가 뜨게 된다.

```
    print(4, config['hi'])
KeyError: 'hi'
```