- Python math has functions math.ceil(args) math.floor(args) etc etc

- For complex numbers, use d.real d.imag d.conjugate()

- +/*- have left to right associativity

- If a string is prepended with r, it is a raw string which means things like / , ‘ are treated as string. Example - r’He said ‘Let Us Python’ “ . ANother way of using these is by using escape sequences such as backslash \” \’

- A substring can be extracted by s[start:end] which extracts elements from start to end-1

- Strings are immutable, can be printed multiple times using print(’-’,50) #prints 50 dashes, character existence can be found by print(’e’ in Hello) #returns true or false.

- A string can be reversed by [::-1]

- Form of print expanded is print(objects,sep=’’,end=’\n’,file=sys.stdout,flush=False)

- lst2=lst1 is shallow copy.

- lst2=[], lst2=lst2+lst1 is deep copy.

- List can be sorted by sorted(lst,reverse=True/False)

-

- Tuples are immutable, therefore append, remove and insert cannot be used in them.

- Concatenation,merging etc can be done(list1.extend(list2), list.append(args),list.insert(index,data)). Also remove, clear and pop

- def func(args,*kwargs) means that number of positional and keyword arguments are not defined , therefore a tiuple args and a dictionary kwargs is passed.

- 2 types recursion-head and tail. In head,recursive call made before other processing, vice versa for tail.

- Recursive error is real. Limit=10**4.

- Lambda functions are nameless functions built at execution time. Example= lambda n:n*n*n returns n cube. lambda x,y,z:(x+y+z)/3 returns average of three values, lambda s:s.trim().upper() returns s trimmed and in uppercase.

- Lambda functions can be generated inline. Example - print((lambda l:sum(l)/len(l))(lst1)) prints sum of lst1 divided by length of lst1.

- *Map* applies a function to each element in the sequence and returns a new sequence containing the result. Example- ls[1,2,3], m1=map(math.radians,lst)

- *Filter* operation applies a function to all the elements of a sequence and returns sequence of those elements for which result returns True. Example- def fun(n):n%5==0?True:False ,lst1=[45,23,42,65], lst2=filter(fun,lst1) returns [45,65]

- *Reduce* operation performs a rolling computation to sequential values in a sequence and prints the result. Example-def getsum(x,y):return x+y , lst=[1,2,3,4,5], s=reduce(getsum,lst) returns 15.

- *Lambda* functions can be used with map,reduce, filter. Example- lst2=map(lambda n:n*n, lst1)

- *Module* is a .py file containing all definitions and statements. During execution of program, module name is _main_ and is stored in the variable name _main_ .

- Modules can be imported as - import module, import function/* from module or import function from module as function_name.

- *Package* is a directory containing modules and a _init.py_ file.

- To use a variable in the global namespace, define it with the keyword global.

- Using double underscore before variable name in class,sets it to private.

- *self* is like *this* in java.

- _init_ is used as default constructor.

- *yield* is used in a function to return the value without destrying the local variable value

- When yield is used, we can iterate through the items by using *next*

Multithreading

- Python is an interepreted language, isiliye niche line wala function call can execute before upar wala function call finishes execution.

- time.sleep(t) —> t is in seconds

- To initialize— variable=threading.Thread(target=function,args=tuple of arguments)

- To start- t1.start()

- t1.join() basically fucks up the multithreading by converting the threads into a single one process

- Multiprocessing syntax- replace threading bhy multiprocessing and Thread by Process

- A *process* is what we call a program that has been loaded into memory along with all the resources it needs to operate. It has its own memory space.

- A *thread* is the unit of execution within a process. A process can have multiple threads running as a part of it, where each thread uses the process's memory space and shares it with other threads.

- • *Multiprocessing* is a technique where parallelism in its truest form is achieved. Multiple processes are run across multiple CPU cores, which do not share the resources among them. Each process can have many threads running in its own memory space. In Python, each process has its own instance of Python interpreter doing the job of executing the instructions.

- *Multithreading* is a technique where multiple threads are spawned by a process to do different tasks, at about the same time, just one after the other. This gives you the illusion that the threads are running in parallel, but they are actually run in a concurrent manner. In Python, the Global Interpreter Lock (GIL) prevents the threads from running simultaneously.

- If 2 processes are called simultaneously, one may execute before other finishes executing and mess up the output. For this we use the join method.

- Producer consumer problem- one producer must inform consumer that it is done with production and only then consumer method will run. THis is done by communication b/w threads.

- Python communication- Lock (for synchronized access) ,RLock(for nested access),Semaphore(for limited number of access to a resource). *acquire* is like chaabi ghumao to open and *release* is like lock it again.

lck=threading.Lock()

lck=threading.acquire() #lck free for use

#use lck

lck=threading.release() #release the resource

- Problem with just lock() is that it doesn't support recursion i.e. if acquire is used on a lock twice, it is implemented only once. Therefore we use RLock (release should be called as much times acquire is called, this is like locks of my house where one is opened after that other whereas Lock is like lock of classroom where locking a lock again leads to it being in the same state)

rlck=Threading.Lock()

rlck.acquire()

rlck.acquire()

- The acquire method forces the thread to wait if the thread is already in locked state and then locks it unlocks by the release of the previous method which locked it first.This works like if lck.acquire is called by fun1 and before lck.release of fun1 is called if lck.acquire of fun2 is called, then lck.acquire of fun2 will wait till the lck.release of fun1 is executed. *WE cannot lock a lock again if it is locked*

- Semaphore- aukat ke bahr atm

- 2 ways for Inter Threaded Communication- using Event (wait,set and clear) and using Condition(acquire release wait notify notifyall)

- In event, wait method waits for the flag to be set before exection of codes below it. set and clear are used to change the flags.

Example-

```
def fun1():

while True:

ev.wait();#wait for flag to be clear

#do corresponding work in the thread

ev.clear() #clear the flag again

def fun2():

while True:

#perform work

#set the flag

#ev.set()
```

ev=Event()

th1=threading.Thread(target=fun1)

th2=threading.Thread(target=fun2) #fun2 executes its work first and then fun 1 work is executed once fun2 work finishes

- Condition uses lock release acquire wait notify and notifyall. The wait method release the lockuntil a notify wakes it up in another thread. Acquire and release have previous functionalities(block and unblock like your ex), acquire waits if the current thread is blocked.

Producer/Consumer problem-

#Producer thread

cond.acquire()

#produce one item

cond.notify()

cond.release()

#Consumer thread

cond.acquire()

while(item unavailable):

cond.wait

cond.release()

- TO sum it up, for multithreading, we have lock,rlock which can be utilised by acquire and release. For interthreaded communication we use condition, which has got acquire,release,wait,notify and notifyAll, we also have Event() which is used by set wait and clear. Rest notify and notifyall concepts are same as in JAva.
- To achieve producer consumer by using Events-

ev=Event()

#we have set wait and clear

#Producer code

ev.set()

#Create object

ev.clear()

#Consumer code

ev.set()

if(Item unavailable):

ev.wait()

ev.clear()

.

How to do TCP and UDP and derive UDP from TCP

Basically create a server for TCP by creating a socket first and then binding the socket to a specific port. Now the difference between a socket and a port for our perspective is that if we bind a socket to a port, we can use that socket to listen to all the incoming and send messages via that port. Therefore we first bind the server to a port and then we start listening for connection requests from it. Then we

run a infinite loop in which when a accept a request and extract the socket details and address from it. Then we send further messages to that address and receive messages in bytecode format and utf 8 encoding. For the client side, we first create socket and then connect to the same port as the server. Then we send and receive messages in bytecode and utf8.

How to get UDP- For UDP for the server everything is same till the point we receive details from the specific client socket. We send a message via the client socket to the server socket and then we extract the socket data from it if we want to send a message to the client from the server afterwards. We do not connect directly to the socket from the client and therefore we do not receive a direct connection request in the server side so that the operation socketdetails,addr=s.accept() can be executed, instead we do socketdetails,addr=s.recvfrom(portnumber)