**Exception handling-:**

**Q1 difference between error and exception ?**

**Error-: error is a problem which is not allow to execution the program.**

**Error is two type-:**

1. **Basic/fundamental error**
2. **Runtime error**

**Basic/fundamental error-:** basic error is also three type……

1. **Syntax error-**mistake in python syntax-

Ex

a=10 ----------->valid

10=a------------->not valid

If a in range(1)

Print(‘hello’)

1. **Lexical error-** mistake in lexemes or token

Ex.

If i==10:---------------> valid

fi ==10: --------------------> invalid

1. **Semantic error-performing** operation with incompatible type-

Ex. i=10

Str=’abc’

Result=i-str

Invalid ----------------- > this type error is semantic error

**Runtime error-:**

These errors occurred at runtime which are not having any solution programmatically.

EX: Insufficient Main memory

PVM Internal Problems

Unavailability of IO Components.

**Exception-:**

An unwanted unexpected event that disturb the normal flow of the program that called exception.

Ex. - ZeroDivisionError, Value Error,Type Error

Or

Exception is an unexpected event occurred at Runtime, it may be provided by the users while entering dynamic input in Python programs, it may be provided by Database Engine while executing sql queries in PDBC applications, It may be provided by the Network when we establish connection between client and Server in distributed applications,.... causes abnormal termination to the applications.

In Python, there are two types of Terminations are existed.

1. Smooth Termination

2. Abnormal Termination

1. Smooth Termination:

---> If any program is terminated at the end of code then that termination is called as Smooth Termination.

2. Abnormal Termination:

--> If any Program is terminated in middle of the program then that termination is called as Abnormal Termination

In general, in applications execution, abnormal terminations may provide the following problems

1. It may crash local OS.

2. It may provide Hanged out situation for Network based applications.

3. It may collapse the database which we are using in Python applications

**Solution-**

To overcome the above problems we have to handle excerptions properly, for this, we have to use a set of mechanisms explicitly called as Exception Handling mechanisms.

In Python, there are two types of Exceptions are existed.

1. Predefined Exceptions

2. User Defined Exceptions

**1. Predefined Exceptions:**

These Exceptions are defined by Python programming language.

In Python, all the Errors and Exceptions are sub types to Base Exception either directly or indirectly.

In general, almost all the Exceptions are sub classes to Exception class.

To handle Exceptions in python applications, we have to use try-except-else-finally block

Syntax:

--------

try:

---instructions---

except ExceptionName:

---instructions----

else:l

---instructions----

finally:

---instructions----

try block:

--> It include as set of instructions which may raise exceptions

--> It will include doubtful code to get an exception, where doubtful code may or may not generate exception.

--> If any exception is generated in try block then PVM will bypass flow of execution to except block then finally block by skipping remaining instructions in try block.

--> if no exception is identified in try block then PVM will execute the complete try block, at the end of try block PVM will bypass flow of execution to else block.

except block:

--> Its main intention is to catch exception from try block and to provide exception details on console.

--> Along with except keyword we have to provide Exception name and we have to defined an alias name [Reference variable ] to Exception name.

--> In Python applications, if any exception is raised in try block there PVM will execute except block, if no exception is raised in try block then PVM will not execute except block.

**else block:**

--> It will be executed when no exceptions are generated in try block, it is an alternative to except block.

--> If any exception is identified in try block then PVM will execute except block only, PVM will not execute else block.

try:

print("Inside try block")

except:

print("in side except block")

else:

print("Inside else block")

finally block:

--> finally block is able to include a set of instructions which must be executed irrespective of executing try block, except block and else block.

--> In general, in Python applications, we will use resources , we will create resources inside try block and we will close these resources in side finally, because, finally block is giving guarantee for execution irrespective getting exceptions in try block. EX:

try:

print("Inside try block")

except:

print("In side except block")

else:

print("Inside else block")

finally:

print("Inside finally block")

Q)Is it possible to write try block with out except block?

-------------------------------------------------------------

Ans:

Yes, it is possible to write try block with out except block , but, we must provide finally block. ex…

**try**:  
 print(**"Inside try"**)  
 a = 100 /0  
**finally**:  
 print(**"Inside finally"**)

Q)Is it possible to provide try block with out finally block?

Yes, it is possible to provide try block with out finally block, but, we must provide except block. EX:

try:

print("Inside try")

a = 100 /0

except :

print("Inside except")

Q)Is it possible to provide try-except-finally in side try block, inside except block and inside finally block?

Yes, It is possible to provide try-excepti-finally block inside try block, inside except block and inside finally block.

try:

print("Inside try")

try:

print("Inside nested try")

except:

print("Inside nested except")

finally:

print("Inside nested finally")

except :

print("Inside except")

finally:

print("Inside finally")

note-:

In Python applications, we are able to provide more than one except block for a single try block, where if we provide any default except block then it must be provided as last statement .

try:

print("Inside try")

a = 100 / 0

except ZeroDivisionError as e:

print(e)

except AttributeError as e:

print(e)

except NameError as e:

print(e)

except:

print("Unknown Error")

**2. User Defined Exceptions**

These exceptions are defined by the developers as per their application requirements.

To prepare User defined exceptions we have to use the following steps.

1. Define User defined Exception class.

2. Raise and handle User defined Exception.

Ex.

class InsufficientFundsException(Exception):

def \_\_init\_\_(self, exceptionDescription):

self.exceptionDescription = exceptionDescription

class Transaction:

def \_\_init\_\_(self, accNo, accHolderName, accType, balance):

self.accNo = accNo

self.accHolderName = accHolderName

self.accType = accType

self.balance = balance

def withdraw(self, wdAmt):

print("Transaction Details")

print("------------------------")

print("Account Number :", self.accNo)

print("Account Holder Name :", self.accHolderName)

print("Account Type :",self.accType)

print("Transaction Type : WITHDRAW")

print("Withdraw Amount :",wdAmt)

try:

if wdAmt > self.balance:

print("Total Balance :", self.balance)

print("Transaction Status : FAILURE")

raise InsufficientFundsException("Reasone : Funds are not Sufficient in Your Account")

else:

self.balance = self.balance - wdAmt

print("Total Balance :", self.balance)

print("Transaction Status : SUCCESS")

except InsufficientFundsException as ex:

print(ex)

finally:

print("\*\*\*\*\*\*ThanQ, Visit Again\*\*\*\*\*\*\*\*")

tx1 = Transaction("abc123", "Durga", "Savings", 10000)

tx1.withdraw(5000)

print()

tx1 = Transaction("xyz123", "Anil", "Savings", 10000)

tx1.withdraw(15000)

**Multi Threadding**

Q)What is the difference between process , procedure and processor?

Procedure is a set of instructions to perform/represent a particular task.

Process is a flow of execution to execute set of instruction to perform a particular task.

Processor is a hardware component to generate process.

Multitasking-: is the process of executing multi task simultaneously.

There are two type of multitasking…..

1. Process base multi tasking
2. Thread base multi tasking

**Process base multitasking-:** Executing several tasks simultaneously where each task is a separate independent process…..

Ex. we are making ppt on , download movies from internet, listen audio song

Note: In general, Processes are managed by Operating System.

**Thread base multi tasking-:** Executing several task simultaneously where each task having separate independent part of same program… and each independent part is called as thread.

**Process base multitasking-:** type-

**Single tasking/single process tasking**- :

-- it allow to perform only one task at a time.

--it will allows only one process to execute the complete task

---it follows sequential execution.

---it will increase application execution time.

--it will reduce application performance.

**Multi tasking/multi process tasking(mechanism)-:**-

It will allows more than one task at at a time.

It will allows more than one process at a time to execute task.

It follows parallel execution.

It will reduce application execution time.

It will improve application performance.

Note-: In multitasking, controlling is navigated for one process to another process, it is called context switching. there are two type of context switching…….

**Heavy weight context switching**-: if context switching is going on in between two heavy weight components, then this type switching called as Heavy weight context switching.

**light weight context switching**-: if context switching is going on in between two light weight components, then this type switching called as light weight context switching. Ex. context switching between two threads.

Q)What is the difference between process and Thread?

**Process** is a heavy weight components, to handle process, System has to consume more no of resources, that is, more memory and execution time, It will reduce application performance.

Note: In general, Processes are managed by Operating System.

**Thread** is Light Weight components, to handle threads, System has to consume less no of resources, that is , less memory and less execution time, it will increase application performance.

Note: In general, Threads are managed by Python Software or PVM.

There are two thread models to execute applications.

1. Single Thread Model

2. Multi Thread Model

Single Thread Model:

--> It able to allow only one thread to execute application

--> It able to allow sequential execution in our application

--> It will increase application execution time

--> It will reduce application performance.

Multi Thread Model-:

--> It able to allow more than one thread to execute applications.

--> It allows parallel execution in our applications.

--> It will reduce application execution time.

--> It will increase application performance.

Note-: Python is able to follow Multi Thread Model, because, Python is able to provide very good environment to create and execute more than one thread.

To create threads and to execute threads, Python has provided separate predefined module in the form of "threading" module.

To create Threads, "threading" module has provided a predefined class in the form of "Thread".

**Q)What is Thread and in how many ways we are able to create Threads in Python?**

Thread is flow of execution to perform a particular task.

As per the predefined library, there are two ways to create threads.

1. without using any class
2. Extending Thread class.
3. With out Extending Thread class.

Without using any class-:

**from** threading **import**\*  
**def** display():  
 print(**'this code executed by thred :'**,current\_thread().getName())  
t=Thread(target=display())  
print(**'this code executed by thred :'**,current\_thread().getName())

**Extending Thread class-:**

Declare an user define class as child class to thread class.

Overriding thread class run() method in user define class with application logic. Which we want to execute by creating new thread.

Create object for user define class.

Access start() method on user define class object reference variable.

from threading import Thread

import time

class WelcomeThread(Thread):

def run(self):

for x in range(0,10):

time.sleep(1)

print("Welcome Thread :",x)

wt = WelcomeThread()

wt.start()

for x in range(0,10):

time.sleep(1)

print("Main Thread ",x)

In thread.

Run()-: it will include an application logic to execute by creating thread.

Start()-: it will create new thread( flow of execution) and it will send that thread to run method().

Internal Flow:

1. When we execute the above application, PVM will create a thread to execute python application called as "Main Thread".

2. When Main Thread encounter start() method then start() method will create a new thread and start() method will bypass new thread to User defined thread class provided run() method.

3. In the above context, User Thread executes run() method implementation and Main Thread executes remaining part of the Python file parllely, so the above python application is able to generate mixed output.

Drawbacks: In the above approach, we must extend Thread class and we must override run() method, it is not possible to replace run() method with any other method.

**With out Extending Thread class-:**

Declare a function with application logic which we want to execute by creating new thread.

Create Thread class object with "target" attribute in Thread class constructor.

Note: To the target attribute we must provide method name which we want to execute by creating new thread.

Access start() method by using Thread class reference variable.

**import** time  
**from** threading **import** Thread  
  
**def** welcome():  
 **for** x **in** range(0,10):  
 time.sleep(1)  
 print(**'user thread welcome'**,x)  
  
t=Thread(target=welcome)  
t.start()  
**for** x **in** range(0, 10):  
 time.sleep(1)

print(**'Main thread '**, x)

Internal Flow:

---------------

1. When we execute the above application, PVM will create a thread called as Main Thread to execute the above python file.

2. When Main thread encounter start() method, start() method will create new thread and start() method will bypass new thread to a method which we specify along with "target" attribute.

3. In the above context, User thread executes user defined function and main thread executes remaining part of Python file parallely, it will provide mixed output.

In Python applications, we are able to create thread and we are able to submit that thread to a method which is available in a particular class…

**import** time  
**from** threading **import** Thread  
**class** Task:  
 **def** dotask(self):  
 **for** x **in** range(0,6):  
 time.sleep(2)  
 print(**"user thread"**,x)  
  
t=Task()  
t.dotask()  
**for** x **in** range(0, 6):  
 time.sleep(2)  
 print(**"main thread"**,x)

In Python applications, if the target method contains explicit parameters then we are able to pass that parameter values by using "args" attribute in Thread class constructer.

**import** time  
**from** threading **import** Thread  
**class** Bank:  
 **def** displayCustmerNames(self,customerNames):  
 **for** customerName **in** customerNames:  
 time.sleep(1)  
 print(customerName)  
bank = Bank()  
t = Thread(target=bank.displayCustmerNames,args=([**"AAA"**, **"BBB"**, **"CCC"**, **"DDD"**,**"EEE"**, **"FFF"**],))  
t.start()

Ex………….

**import** time **as** tm  
**from** threading **import** Thread  
**def** sayWelcome():  
 **for** x **in** range(0,10):  
 tm.sleep(1)  
 print(**"Welcome User!"**)  
**def** sayHello():  
 **for** x **in** range(0,10):  
 tm.sleep(1)  
 print(**"Hello User!"**)  
**def** sayHai():  
 **for** x **in** range(0,10):  
 tm.sleep(1)  
 print(**"Hi User!"**)  
t1 = Thread(target=sayWelcome)  
t2 = Thread(target=sayHello)  
t3 = Thread(target=sayHai)  
t1.start()  
t2.start()  
t3.start()

In Python applications, when we create a thread automatically an identity number will be create that is called as Thread Identity Number, to access Thread Identity number we have to use "ident" attribute oin Thread reference.

**from** threading **import** Thread  
t1 = Thread()  
t2 = Thread()  
t3 = Thread()  
  
t1.start()  
t2.start()  
t3.start()  
  
print(**"t1 Identity :"**,t1.ident)  
print(**"t2 Identity :"**,t2.ident)  
print(**"t3 Identity :"**,t3.ident)

IN Python applications, each and every thread has its own internal name in the form of Thread-1, Thread2,..... , If we want to get Name of the thread we have to use getName() method and if we want to set a particular name to the thread we have to use setName().

EX:

**from** threading **import** Thread  
t=Thread()  
print(t.getName())  
t.setName(**"aAA"**)  
print(t.getName())

IN Python applications, it is possible to get currently executed thread by using current\_thread() function.

**from** threading **import** Thread, current\_thread  
**import** time  
**def** fn():  
 **for** x **in** range(0,10):  
 time.sleep(2)  
 print(current\_thread().getName())  
t1 = Thread(target=fn)  
t2 = Thread(target=fn)  
t3 = Thread(target=fn)  
t1.start()  
t2.start()  
t3.start()

In Python applications, we are able to get the no of threads which are active in present python applications by using active\_count().

**from** threading **import** Thread, current\_thread,active\_count  
**import** time  
**def** fn():  
 **for** x **in** range(0,10):  
 time.sleep(2)  
  
t1 = Thread(target=fn)  
t2 = Thread(target=fn)  
t3 = Thread(target=fn)  
t1.start()  
t2.start()  
t3.start()  
print(active\_count())

In Python applications, we are able to check whether thread is in live or not by using is\_Alive() function.

**from** threading **import**\*  
**import** time  
**def** fn():  
 time.sleep(2)  
t1 = Thread(target=fn)  
print(t1.is\_alive())  
t1.start()  
print(t1.is\_alive())

Note: In general, Thread is in live when we access start() method, before acessing start() method thread is not in live.

IN Python applications, we are able to get all the active threads references in a list by using enumerate() function.

**from** threading **import**\*  
**import** time  
**def** fn():  
 **for** i **in** range(0,3):  
 time.sleep(1)  
t1 = Thread(target=fn)  
t2 = Thread(target=fn)  
t3 = Thread(target=fn)  
t1.start()  
t2.start()  
t3.start()  
list = enumerate()  
**for** thread **in** list:  
 print(thread.getName())

In Python applications, if we want to pause one thread to complete other thread and if we want to conitnue the paused thread after completion of the other thread we have to use join() method.

Note-join() can be define inside the class and outside the class. If it declare inside the class , called as method

If we declare outside the class , we called function

**from** threading **import**\*  
**def** fn(name):  
 **for** x **in** range(0,5):  
 print(**'hello'**,name,**'!'**)  
t1 = Thread(target=fn,args=(**"mukesh"**,))  
t1.start()  
t1.join()  
**for** x **in** range(0,5):  
 print(**'hi durga'**)

note-: the adv of threading is that it take less execution time and improve application performance.

Without threading program-

**import** time  
**def** doubles(numbers):  
 **for** n **in** numbers:  
 time.sleep(1)  
 print(**"doubles :"**,2\*n)  
**def** squares(numbers):  
 **for** n **in** numbers:  
 time.sleep(1)  
 print(**"squares :"**,n\*n)  
numbers=[1,2,3,4,5,6]  
begintime=time.time()  
doubles(numbers)  
squares(numbers)  
endtime=time.time()  
print(**'Total time taken :'**,endtime-begintime)

op-12.008sec

with threading module-:

**from** threading **import**\*  
**import** time  
**def** doubles(numbers):  
 **for** n **in** numbers:  
 time.sleep(1)  
 print(**"doubles :"**,2\*n)  
**def** squares(numbers):  
 **for** n **in** numbers:  
 time.sleep(1)  
 print(**"squares :"**,n\*n)  
numbers=[1,2,3,4,5,6]  
begintime=time.time()  
t1=Thread(target=doubles,args=(numbers,))  
t2=Thread(target=squares,args=(numbers,))  
t1.start()  
t2.start()  
t1.join()  
t2.join()  
endtime=time.time()  
print(**'Total time taken :'**,endtime-begintime)

op-6.008sec

**Daemon Thread:-**

Daemon thread is a thread, it will be executed internally and gives some services to some other thread and demean threads are terminated automatically along with the threads termination which are taking services from threads.

Ex. Garbage collector in PVM is Daemon thread. it will be execute and provide Garbage collection service to PVM and it will be terminated automatically along with PVM termination. And provide mixed output.

In Python applications, to make a thread as daemon thread we have to access

setDaemon(True) method on thread reference variable.

Note: If we want to access setDaemon(True) method on a thread, then we have to access that method before accessing start() method, not after accessing start() method. If we access setDaemon(True) method after calling start() method then PVM is able to provide an error like "RuntimeError: cannot set daemon status of active thread".

To check whether a thread is daemon thread or not we have to use isDaemon() method.

Note-

**from** threading **import** \*  
**class** GarbageCollectorThread(Thread):  
 **def** run(self):  
 **while True**:  
 print(**"Garbage Collector Thread"**)  
  
gct = GarbageCollectorThread()  
gct.setDaemon(**True**)  
gct.start()  
*#gct.setDaemon(True)***for** x **in** range(0,10):  
 print(**"PVM Thread"**,x)  
print(gct.isDaemon())

**Synchronization-:**

In multithreading, if we execute more than one thread at a time on single data or on single program, then that thread as called Concurrent thread and this process is called as “Thread Concurrency”..

IN Threads concurrentcy, if more than one thread is executed on single data object ,then, there may be a chance to get Data Inconsistency and In this context, to get data consistency we have to use "Synchronization".

Synchronization is a mechanism, it able to allow only one thread at a time to access data , it will not allow more than one thread at a time, after completion of the present thread only other threads are allowed.

IN Python, Synchronization is running on the top of Locking Mechanisms only. it will provide sequential execution. it will reduce application performance, so it’s not suggested to application development.

**There are three ways to achieve synchronization in python**

1. Lock [Simple Lock]
2. RLock[Re-Entrent Lock]
3. Semaphores.

**By using Lock [Simple Lock]-:**

If any thread acquire lock, then it will eligible to execute program. And the program is executed by thread, which is already lock, then thread must release the lock. After releasing lock from thread, then that avaible lock assign to another thread.

To represent a Simple Lock in Python Applications, Python has provided a predefined class that is Lock

l = Lock()

To acquire a lock we have to use a function like acquire().

Perform synchronization operation……………….

To release lock we have to use a function like release().

**from** threading **import** current\_thread, Thread, Lock  
  
l = Lock()  
**def** wish():  
 l.acquire()  
 **for** x **in** range(0,10):  
 print(current\_thread().getName())  
 l.release()  
t1 = Thread(target=wish)  
t1.setName(**"AAA Thread"**)  
t2 = Thread(target=wish)  
t2.setName(**"BBB Thread"**)  
t3 = Thread(target=wish)  
t3.setName(**"CCC Thread"**)  
  
t1.start()  
t2.start()  
t3.start()

Drawback:-

1. It is not supporting for acquiring lock by same thread again and again.

2. It is not suitable for Recursive functions.

Synchronization through RLock-:

RLock is called as Re-Entrent Lock, it is very much suitable for acquiring lock by the same thread again and again and it is very much suitable in Recursive functions.

To reprepsent Re-Entrent Lock , Pyth*on has provided a predefined class in the form of RLock.*

*Q) difference between Lock and RLock.*

|  |  |
| --- | --- |
| *Lock-* | RLock |
| *1. only one thread at a time .but owner thread can’t acquire multiple times* | *only one thread at a time but owner thread acquire multiple times* |
| 2. it will not suitable to execute recursive function and nested access call | 2. it will more suitable to execute recursive function and nested access call |
| 3. It will not suitable for acquiring lock by the same thread again and again | 3. It will more suitable for acquiring lock by the same thread again and again. |

**Synchronization by Semaphores-:**

Lock and RLock are able to allow only one thread at a time. but our requirement is to execute particular (n) no of threads at a time. Where If we want to allow a fixed no of threads [Not all the threads] to execute, then we have to use Semaphore. Semaphore is advance synchronization mechanism.

To represent Semaphore, Python has provided a predefined class in the form of Semaphore.Syntaxes:

s = Semaphore() --> It able to allow 1 thread at a time in Synchronized area

s = Semaphore(count) --> It able to allow the specified no of threads in Synchronized Area.

**from** threading **import**\*  
**import** time  
s=Semaphore(2)  
**def** wish(name):  
 s.acquire()  
 **for** x **in** range(2):  
 print(**'good evening :'**,end=**' '**)  
 time.sleep(3)  
 print(name)  
 s.release()  
t1=Thread(target=wish,args=(**'ram'**,))  
t2=Thread(target=wish,args=(**'mukesh'**,))  
t3=Thread(target=wish,args=(**'neelam'**,))  
t4=Thread(target=wish,args=(**'Ayush'**,))  
t1.start()  
t2.start()  
t3.start()  
t4.start()

IN Normal Semaphores, it is not at all mandatory condition to match no of acquire()

**from** threading **import** \*  
s = Semaphore(3)  
**def** wish(name):  
 s.acquire()  
 s.acquire()  
 **for** i **in** range(3):  
 print(**'good morning :'**,end=**' '**)  
 print(name)  
 s.release()  
   
t=Thread(target=wish,args=(**'mukesh'**,))  
t.start()

print(**"End of Program"**)

In Python applications, if we want to make no of acquire() functions calls and no of release() function as equal then we have to use "BoundedSemaphore".

**from** threading **import** \*  
s = BoundedSemaphore()  
s.acquire()  
  
s.release()  
s.release()  
print(**"End of Program"**)  
*#OP:  
#ValueError: Semaphore released too many times*

or

**from** threading **import** \*  
s = BoundedSemaphore()  
s.acquire()  
  
s.release()  
print(**"End of Program"**)

Note: In case of Semaphores, if any thread access acquire() method then the provided threads count value will be decremented by 1, if any thread access release() function then threads count value is incremented by 1. in this process, if count value is 0 then Semaphore will make the next threads in waiting state until the completion of threads.

Note: In Python applications, Synchronization is able to allow only one thread at a time, it followis sequential execution of the threads, it will increase application execution time, it will reduce application performance, due to this reason, in python applications , it is not suggestible to use "Synchronization", if we are not thinking about the data consistency then it is suggestible to remove Synchronization in python applications.

**Inter Thread Communication-:**

The process of providing communication between more than one threads then it is called as Inter Thread Communication.

To provide inter thread communication between threads Python has provided a set of predefined classes.

1. Event

2. Condition

3. Queue

In general, Inter Thread Communication is able to provide solutions for the problems like "Producer-Consumer" problems.

In Producer-Consumer problem, both Producer and Consumer are two threads, where Producer Thread has to produce an item and Consumer Thread has to consume that item, the same sequence has to manage upto infinite no of times.

**Inter thread Communication through Event object-:**

Event is an object, it will provide the methods inorder to make a thread to wait and to send notification to some other threads to active which are existed in waiting state.

To create an Event object we have to use the following instruction.

event = threadding.Event()

Note: Event has an internal flag that we can set and clear depending on our requirement.

Methods in Event:

1. set(): It can be used to set True value to internal flag and it will give notification to other threads which are available in waiting state.

2. clear(): It can be used to set False value to internal flag and it will give notification to other threads like not to come out from waiting state.

3. isSet(): It can be used to check whether event is set or not.

4. wait() wait(seconds): It will used to make current thread to wait state.

**from** threading **import** \*  
**import** time  
e=Event()  
**def** consumer():  
 print(**'consumer thread waiting for updation...'**) #line1  
 e.wait()  
 print(**'consumer thread got notification and item cunsumed'**) #line4  
**def** producer():  
 time.sleep(5)  
 print(**'producer thread producing items..'**) #line2  
 print(**'producer thread giving notification by event setting'**) #line3  
 e.set()  
t1=Thread(target=producer)  
t2=Thread(target=consumer)  
t1.start()  
t2.start()

and other example

**from** threading **import** \*  
**import** time  
e=Event()  
**def** Trafficpolice():  
 **while True**:  
 time.sleep(15)  
 print(**'traffic police Given Green signal...'**)  
 e.set()  
 time.sleep(20)  
 print(**'traffic police given RED signal..'**)  
 e.clear()  
  
**def** driver():  
 num=0  
 **while True**:  
 print(**'driver waiting for Green signal..'**)  
 e.wait()  
 print(**'traffic signal is green...vicle can move..'**)  
 **while** e.isSet():  
 num=num+1  
 print(**'vehical number'**,num,**'crossing the signal'**)  
 time.sleep(2)  
 print(**'traffic signal is red .. drver is wait'**)  
  
t1=Thread(target=Trafficpolice)  
t2=Thread(target=driver)  
t1.start()  
t2.start()

**Inter Thread Communication By Using Condition:**

Condition is more advanced than Event, It able to manage a Condition internally and it will change the state on the basis of the condition.

In Producer-Consumer problem, we will use Condition to make threads wait and to send notification when condition happened.

Condition has the following methods.

1. acquire() --> it is used for acquire lock before producing a item in produce thread and before consuming an item in consume thread.

2. release() --> it is used for releasing a lock after producing a item in produce thread and after consuming an item in consume thread.

3. wait() -----> To make a thread to wait.

4. notify() ---> To give notification to other thread which is in waiting state.

5. notifyAll()-> To give notification to all threads which are in waiting state.

Note-Condition is always associated with a lock (ReentrantLock).l

**from** threading **import** \*  
**import** time  
count = 0  
c = Condition()  
**def** produce():  
 **global** count  
 **while True**:  
 c.acquire()  
 time.sleep(2)  
 count = count + 1  
 print(**"Producer Produced Item :"**,count)  
 c.notify()  
 c.wait()  
 c.release()  
**def** consume():  
 **global** count  
 **while True**:  
 c.acquire()  
 print(**"Consumer Consumed Item :"**,count)  
 c.notify()  
 c.wait()  
 c.release()  
  
  
producer = Thread(target=produce)  
consumer = Thread(target=consume)  
producer.start()  
consumer.start()

**Inter Thread Communication By Using Queue-:**

Queue is most advance Inter Thread Communication mechanism to establish communication between threads and to share data between threads.

Queue internally has condition and a lock , it will set automatically when we put elements and retrieving elements from Queue.

Queue is an element provided by queue module, if we want to use Queue in Python application then we have to import queue module.

Methods of Queue:

1. put(): It will add an element in Queue.

2. get(): It will remove and return an element from Queue.

Producer Thread will use put() method to insert Item in Queue , put() method has logic to acquire lock before inserting item in queue and it will release lock after inserting item.

Note-put() method will check whether Queue is full or not, if Queue is full then Producer Thread will come to waiting state.

Consumer thread will use get() method to remove and return element from Queue, it has internal logic to acquire lock before removing element and to release lock after removing element.

get() method will check first whether item is existed or not, if no item is existed then get() method will keep consumer thread in waiting state.

**import** queue **as** q  
**from** threading **import** \*  
**import** time  
queue=q.Queue()  
count=0  
**def** producer():  
 **global** count  
 **while True** :  
 count=count+1  
 print(**'producer is producing items'**,count)  
 queue.put(count)  
 time.sleep(2)  
**def** consumer():  
 **global** count  
 **while True**:  
 print(**'consumer is consuming items'**, count)  
 queue.get(count)  
 time.sleep(2)  
t1=Thread(target=producer)  
t2=Thread(target=consumer)  
t1.start()  
t2.start()

There are three types of Queues in python.

1. FIFO
2. LIFO
3. PriorityQueue

**FIFO Queue:**

It is default behaviour of Queue, it will retrieve all the elements in the same order in which we entered.

**import** queue **as** q *#inter thred communication by FIFO Queue*q1=q.Queue()  
q1.put(**"AAA"**)  
q1.put(**'BBB'**)  
q1.put(123)  
q1.put(**'nnn'**)  
**for** x **in** range(0,q1.qsize()):  
 print(q1.get())

**LIFO Queue:**

It able to retrieve all elements in Last In First Out Manner.

**import** queue **as** q

*#inter thred communication by LIFO Queue*queue=q.LifoQueue()  
queue.put(40)  
queue.put(60)  
queue.put(80)  
queue.put(100)  
**for** x **in** range(0,queue.qsize()):  
 print(queue.get())

**PriorityQueue:**

It retrieve all elements on the basis of Priorities(But window not support any priority system, so it will provide in sorting order.).

*#inter thred communication by PriorityQueue*

**import** queue **as** qqueue=q.PriorityQueue()  
queue.put(70)  
queue.put(60)  
queue.put(90)  
queue.put(40)  
**for** x **in** range(0,queue.qsize()):  
 print(queue.get())

----------------------------------------------------MULTI THREADING END-------------------------------------------------------