# Python Concurrency for Senior Engineering Interviews

# Global Interpreter Lock:

The program that interprets user code is called the Interpreter. An interpreter is a program that executes other programs. At a higher level when we run a Python program (.py file), the Python interpreter compiles the source code into byte code. The generated byte code is a lower-level platform-independent representation that can be understood by the Python Virtual Machine (PVM). In the next step, the byte code is routed to the PVM for execution. Note that PVM isn't a separate component. Rather, it is just a loop in the Python interpreter that is responsible for executing byte code line by line. The PVM is really a part of the interpreter.

The Python interpreter as explained is responsible for executing a program, but it can only execute a single thread at a time. This is the falling of the reference implementation of Python - CPython, called so because it is written in the C language. So if your machine has one, ten, or a hundred processors, the Python interpreter is only able to run a single thread at a time using a single processor. Two threads on a machine with two available processors can't be executed in parallel each running on a single CPU.

One may wonder what was the design decision behind restricting the interpreter to run a single thread. The answer lies in how memory management works in Python - reference counter.

import sys

# declare a variable

some\_var = "Educative"

# check reference count

print sys.getrefcount(some\_var)

# create another refrence to someVar

another\_var = some\_var

# verify the incremented reference count

print sys.getrefcount(some\_var)

If you run the above snippet, you'll see the reference count of the variable some\_var increase. When references to an object are removed, the reference count for an object is decremented. When the reference count becomes zero, the object is deallocated. The interpreter executes a single thread in order to ensure that the reference count for objects is safe from race conditions.

A reference count is associated with each object in a program. One possible solution could have been to associate one lock per object so that multiple threads could work on the object in a thread-safe manner. However, this approach would have resulted in too many locks being managed with the possibility of deadlocks. Thus, a compromise was made to have a single lock that provides exclusive access to the Python interpreter. This lock is known as the Global Interpreter Lock.

Execution of Python bytecode requires acquiring the GIL. This approach prevents deadlocks as there's a single global lock to manage and introduces little overhead. However, the cost is paid by making CPU-bound tasks essentially single-threaded.

**Removing GIL:**

One may wonder why the GIL can't be removed from Python because of the limitations it imposes on CPU-bound programs. Attempts at removing GIL resulted in breaking C extensions and degrading the performance of single and multithreaded I/O bound programs. Therefore, so far GIL hasn't been removed from Python.

# Thread Safety

The primary motivation behind using multiple threads is improving program performance that may be measured with metrics such as throughput, responsiveness, latency, etc. Whenever threads are introduced in a program, the shared state amongst the threads becomes vulnerable to corruption. If a class or a program has immutable state then the class is necessarily thread-safe.

It increments an object of class Counter using 5 threads. Each thread increments the object a hundred thousand times. The final value of the counter should be half a million (500,000).

from threading import Thread

import sys

class Counter:

    def \_\_init\_\_(self):

        self.count = 0

    def increment(self):

        for \_ in range(100000):

            self.count += 1

if \_\_name\_\_ == "\_\_main\_\_":

    # Sets the thread switch interval

    sys.setswitchinterval(0.005)

    numThreads = 5

    threads = [0] \* numThreads

    counter = Counter()

    for i in range(0, numThreads):

        threads[i] = Thread(target=counter.increment)

    for i in range(0, numThreads):

        threads[i].start()

    for i in range(0, numThreads):

        threads[i].join()

    if counter.count != 500000:

        print(" count = {0}".format(counter.count), flush=True)

    else:

        print(" count = 50,000 - Try re-running the program.")

### ***Fixing Thread Unsafe Class***

We fix the above example using the equivalent of a mutex in Python called a **Lock**

from threading import Thread

from threading import Lock

import sys

class Counter:

    def \_\_init\_\_(self):

        self.count = 0

        self.lock = Lock()

    def increment(self):

        for \_ in range(100000):

            self.lock.acquire()

            self.count += 1

            self.lock.release()

if \_\_name\_\_ == "\_\_main\_\_":

    # Sets the thread switch interval

    sys.setswitchinterval(0.005)

    numThreads = 5

    threads = [0] \* numThreads

    counter = Counter()

    for i in range(0, numThreads):

        threads[i] = Thread(target=counter.increment)

    for i in range(0, numThreads):

        threads[i].start()

    for i in range(0, numThreads):

        threads[i].join()

    if counter.count != 500000:

        print(" If this line ever gets printed, " + \

        "the author is a complete idiot and " + \

        "you should return the course for a full refund!")

    else:

        print(" count = {0}".format(counter.count))

# Barrier

A barrier is a synchronization construct to wait for a certain number of threads to reach a common synchronization point in code. The involved threads each invoke the barrier object's wait() method and get blocked till all of threads have called wait(). When the last thread invokes wait() all of the waiting threads are released simultaneously. The below snippet shows example usage of barrier:

**from threading import Barrier  
from threading import Thread  
import random  
import time  
  
  
def thread\_task():  
    time.sleep(random.randint(0, 7))  
    print("\nCurrently {0} threads blocked on barrier".format(barrier.n\_waiting))  
    barrier.wait()  
  
  
num\_threads = 5  
barrier = Barrier(num\_threads)  
threads = [0] \* num\_threads  
  
for i in range(num\_threads):  
    threads[i - 1] = Thread(target=thread\_task)  
  
for i in range(num\_threads):  
    threads[i].start()**

The barrier constructor also accepts a callable argument as an action to be performed when threads are released. Only one of the threads released will invoke the action. An example is given below:

**from threading import Barrier  
from threading import Thread  
from threading import current\_thread  
import random  
import time  
  
  
def thread\_task():  
    time.sleep(random.randint(0, 5))  
    print("\nCurrently {0} threads blocked on barrier".format(barrier.n\_waiting))  
    barrier.wait()  
  
  
def when\_all\_threads\_released():  
    print("All threads released, reported by {0}".format(current\_thread().getName()))  
  
  
num\_threads = 5  
barrier = Barrier(num\_threads, action=when\_all\_threads\_released)  
threads = [0] \* num\_threads  
  
for i in range(num\_threads):  
    threads[i - 1] = Thread(target=thread\_task)  
  
for i in range(num\_threads):  
    threads[i].start()**

# Lock

**As we have read GIL , It allows one thread at a time , then do we still needs a lock?**

GIL protects the Python interals. That means:

1. you don't have to worry about something in the interpreter going wrong because of multithreading
2. most things do not really run in parallel, because python code is executed sequentially due to GIL

But GIL does not protect your own code. For example, if you have this code:

self.some\_number += 1

That is going to read value of self.some\_number, calculate some\_number+1 and then write it back to self.some\_number.

If you do that in two threads, the operations (read, add, write) of one thread and the other may be mixed, so that the result is wrong.

This could be the order of execution:

1. thread1 reads self.some\_number (0)
2. thread2 reads self.some\_number (0)
3. thread1 calculates some\_number+1 (1)
4. thread2 calculates some\_number+1 (1)
5. thread1 writes 1 to self.some\_number
6. thread2 writes 1 to self.some\_number

You use locks to enforce this order of execution:

1. thread1 reads self.some\_number (0)
2. thread1 calculates some\_number+1 (1)
3. thread1 writes 1 to self.some\_number
4. thread2 reads self.some\_number (1)
5. thread2 calculates some\_number+1 (2)
6. thread2 writes 2 to self.some\_number

Python's Lock is the equivalent of a Mutex.

Lock is the most basic or primitive synchronization construct available in Python. It offers two methods: acquire() and release(). A Lock object can only be in two states: **locked** or **unlocked**. A Lock object can only be unlocked by a thread that locked it in the first place.

## **Acquire**

Whenever a Lock object is created it is initialized with the unlocked state. Any thread can invoke acquire() on the lock object to lock it. Advanced readers should note that acquire() can only be invoked by a single thread at any point because the GIL ensures that only one thread is being executed by the interpreter. If a Lock object is already acquired/locked and a thread attempts to acquire() it, the thread will be blocked till the Lock object is released.

## **Release**

The release() method will change the state of the Lock object to unlocked and give a chance to other waiting threads to acquire the lock. If multiple threads are already blocked on the acquire call then only one arbitrarily chosen (varies across implementations) thread is allowed to acquire the Lock object and proceed.

## **Deadlock**

Consider the example below, where two threads are instantiated and each tries to invoke release() on the lock acquired by the other thread, resulting in a deadlock.

from threading import \*

import time

def thread\_one(lock1, lock2):

    lock1.acquire()

    time.sleep(1)

    lock2.release()

def thread\_two(lock1, lock2):

    lock2.acquire()

    time.sleep(1)

    lock1.release()

if \_\_name\_\_ == "\_\_main\_\_":

    lock1 = Lock()

    lock2 = Lock()

    t1 = Thread(target=thread\_one, args=(lock1, lock2))

    t2 = Thread(target=thread\_one, args=(lock1, lock2))

    t1.start()

    t2.start()

    t1.join()

    t2.join()

The above example demonstrates that a thread can't release a lock it has not locked. Furthermore, trying to release an unacquired lock will result in an exception.

# RLock

A reentrant lock is defined as a lock which can be reacquired by the same thread. A RLock object carries the notion of ownership. If a thread acquires a RLock object, it can chose to reacquire it as many times as possible. Consider the following snippet:

## **Reentrant lock**

**# create a reentrant lock  
rlock = RLock()  
  
# acquire the lock twice  
rlock.acquire()  
rlock.acquire()  
  
# release the lock twice  
rlock.release()  
rlock.release()**

As explained, each reentrant lock is owned by some thread when in the locked state. Only the owner thread is allowed to exercise a release() on the lock. If a thread different than the owner invokes release() a RuntimeError is thrown as shown in the example below:

# Condition Variables

Synchronization mechanisms need more than just mutual exclusion; a general need is to be able to wait for another thread to do something. Condition variables provide mutual exclusion and the ability for threads to wait for a predicate to become true.

Imagine a scenario where we have two threads working together to find prime numbers and print them. Say the first thread finds the prime number and the second thread is responsible for printing the found prime. The first thread (finder) sets a boolean flag whenever it determines an integer is a prime number. The second (printer) thread *needs to know when the finder thread has hit upon a prime number.* The naive approach is to have the printer thread do a busy wait and keep polling for the boolean value. Let's see what this approach looks like:

**def printer\_thread\_func():  
    global prime\_holder  
    global found\_prime  
  
    while not exit\_prog:  
        while not found\_prime and not exit\_prog:  
            time.sleep(0.1)  
  
        if not exit\_prog:  
            print(prime\_holder)  
  
            prime\_holder = None  
            found\_prime = False  
  
  
def is\_prime(num):  
    if num == 2 or num == 3:  
        return True  
  
    div = 2  
  
    while div <= num / 2:  
        if num % div == 0:  
            return False  
        div += 1  
  
    return True  
  
  
def finder\_thread\_func():  
    global prime\_holder  
    global found\_prime  
  
    i = 1  
  
    while not exit\_prog:  
  
        while not is\_prime(i):  
            i += 1  
  
        prime\_holder = i  
        found\_prime = True  
  
        while found\_prime and not exit\_prog:  
            time.sleep(0.1)  
  
        i += 1  
  
  
found\_prime = False  
prime\_holder = None  
exit\_prog = False  
  
printer\_thread = Thread(target=printer\_thread\_func)  
printer\_thread.start()  
  
finder\_thread = Thread(target=finder\_thread\_func)  
finder\_thread.start()  
  
# Let the threads run for 5 seconds  
time.sleep(3)  
  
# Let the threads exit  
exit\_prog = True  
  
printer\_thread.join()  
finder\_thread.join()**

The above program is essentially a producer-consumer problem. The printer thread is a consumer and the finder thread is a producer. The printer thread needs to be signaled somehow that a prime number has been discovered for it to print. Do you see a condition here? The condition in our program is the discovery of the prime number represented by the boolean variable found\_prime. Realize that locks don't help us signal other threads when a condition becomes true.

One shortcoming of the above code is we have the printer thread constantly polling in a while loop for the found\_prime variable to become true. This is called busy waiting and is highly discouraged as it unnecessarily wastes CPU cycles.

Creating a condition variable

cond\_var = Condition()

The two important methods of a condition variable are:

* wait() - invoked to make a thread sleep and give up resources
* notify() - invoked by a thread when a condition becomes true and the invoking threads want to inform the waiting thread or threads to proceed

A condition variable is always associated with a lock. The lock can be either reentrant or a plain vanilla lock. The associated lock must be acquired before a thread can invoke wait()or notify() on the condition variable.

**Creating a condition variable by passing a custom lock**

**lock = Lock()  
cond\_var = Condition(lock) # pass custom lock to condition variable  
cond\_var.acquire()  
cond\_var.wait()**

We can also create a condition variable without passing in a custom lock.

**Creating a condition variable without passing a lock**

**cond\_var = Condition()  
cond\_var.acquire()  
cond\_var.wait()**

Rewriting the above prime number finding and printing code using the condition variable as below:

from threading import Thread

from threading import Condition

import time

def printer\_thread\_func():

    global prime\_holder

    global found\_prime

    while not exit\_prog:

        cond\_var.acquire()

        while not found\_prime and not exit\_prog:

            cond\_var.wait()

        cond\_var.release()

        if not exit\_prog:

            print(prime\_holder)

            prime\_holder = None

            cond\_var.acquire()

            found\_prime = False

            cond\_var.notify()

            cond\_var.release()

def is\_prime(num):

    if num == 2 or num == 3:

        return True

    div = 2

    while div <= num / 2:

        if num % div == 0:

            return False

        div += 1

    return True

def finder\_thread\_func():

    global prime\_holder

    global found\_prime

    i = 1

    while not exit\_prog:

        while not is\_prime(i):

            i += 1

            # Add a timer to slow down the thread

            # so that we can see the output

            time.sleep(.01)

        prime\_holder = i

        cond\_var.acquire()

        found\_prime = True

        cond\_var.notify()

        cond\_var.release()

        cond\_var.acquire()

        while found\_prime and not exit\_prog:

            cond\_var.wait()

        cond\_var.release()

        i += 1

cond\_var = Condition()

found\_prime = False

prime\_holder = None

exit\_prog = False

printerThread = Thread(target=printer\_thread\_func)

printerThread.start()

finderThread = Thread(target=finder\_thread\_func)

finderThread.start()

# Let the threads run for 3 seconds

time.sleep(3)

# Let the threads exit

exit\_prog = True

cond\_var.acquire()

cond\_var.notifyAll()

cond\_var.release()

printerThread.join()

finderThread.join()

There are however two questions we need to answer:

* If the printer thread acquires the lock on the condition variable cond\_var then how can the finder thread acquire() the lock when it needs to invoke the notify() method?
* Can the condition, which is the variable found\_prime, change once the printer thread is woken up?

The answer to the first question is that when a thread invokes wait() it simultaneously gives up the lock associated with the condition variable. Only when the sleeping thread wakes up again on a nofity(), will it reacquire the lock.

def printer\_thread\_func():

    global prime\_holder

    global found\_prime

    while not exit\_prog:

        cond\_var.acquire()

        while not found\_prime and not exit\_prog:

            cond\_var.wait()

        cond\_var.release()

In the highlighted statement, why to use “while”, why “if” cannot be used?

 The reason is that if a thread invokes notifyAll() on a condition variable, then all the threads waiting on the condition variable will be woken up but only one thread will be allowed to make progress. Once the first thread exits the critical section and releases the lock associated with the condition variable, another thread, from the set of threads that were waiting when the original notifyAll() call was made, is allowed to make progress. This may not be appropriate for every use case and certainly not for ours if we had multiple printer threads. We would want a printer thread to make progress only when the condition found\_prime is set to true. This can only be possible with a while loop where we check if the condition found\_prime is true before allowing a printer thread to move ahead.

A peculiarity of condition variables is the possibility of spurious wakeups. It means that a thread might wakeup as if it has been signaled even though nobody called notify() on the condition variable in question. This is specifically allowed by the POSIX standard because it allows more efficient implementations of condition variables under some circumstances. Such wakeups are called spurious wakeups.

A thread that has been woken up does not imply that the conditions for it to move forward hold. The thread must test the conditions again for validity before moving forward. In conclusion, we must always check for conditions in a loop and wait() inside it. The correct idiomatic usage of a condition variable appears below:

*Idiomatic use of wait()*

**acquire lock  
while(condition\_to\_test is not satisfied):  
    wait  
  
# condition is now true, perform necessary tasks  
  
release lock**

## **Quiz1:**

Consider an abridged version of the code we discussed in this lesson. The child\_task method exits without releasing the lock. What would be the outcome of running the program? The changed program is shown below:

**flag = False  
  
lock = Lock()  
cond\_var = Condition(lock)  
  
  
def child\_task():  
    global flag  
    name = current\_thread().getName()  
  
    cond\_var.acquire()  
    while not flag:  
        cond\_var.wait()  
        print("\n{0} woken up \n".format(name))  
  
    print("\n{0} exiting\n".format(name))  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
    thread1 = Thread(target=child\_task, name="thread1")  
    thread1.start()  
      
    # give the child task to wait on the condition variable  
    time.sleep(1)  
  
    cond\_var.acquire()  
    flag = True  
    cond\_var.notify\_all()  
    cond\_var.release()  
  
    thread1.join()  
    print("main thread exits")**

*Does the program hang?*

*Answer is : NO*

This is an interesting case, the single waiting thread exits without releasing the lock but since no other thread including the main thread attempts to acquire the lock the program sucessfully completes with the lock in locked state

# Semaphore:

Rewriting the prime find and print program using Semaphore

from threading import Thread

from threading import Semaphore

import time

def printer\_thread():

    global primeHolder

    while not exitProg:

        # wait for a prime number to become available

        sem\_find.acquire()

        # print the prime number

        print(primeHolder)

        primeHolder = None

        # let the finder thread find the next prime

        sem\_print.release()

def is\_prime(num):

    if num == 2 or num == 3:

        return True

    div = 2

    while div <= num / 2:

        if num % div == 0:

            return False

        div += 1

    return True

def finder\_thread():

    global primeHolder

    i = 1

    while not exitProg:

        while not is\_prime(i):

            i += 1

            # Add a timer to slow down the thread

            # so that we can see the output

            time.sleep(.01)

        primeHolder = i

        # let the printer thread know we have

        # a prime available for printing

        sem\_find.release()

        # wait for printer thread to complete

        # printing the prime number

        sem\_print.acquire()

        i += 1

sem\_find = Semaphore(0)

sem\_print = Semaphore(0)

primeHolder = None

exitProg = False

printerThread = Thread(target=printer\_thread)

printerThread.start()

finderThread = Thread(target=finder\_thread)

finderThread.start()

# Let the threads run for 3 seconds

time.sleep(3)

exitProg = True

printerThread.join()

finderThread.join()

# Using with Statement in Multithreading:

Some classes in the threading module such as Lock, support the context management protocol and can be used with the with statement. In the example below, we reproduce an example from an earlier section and use the with statement with the Lock object my\_lock. Note, we don't need to explicitly acquire() and release() the lock object. The context manager automatically takes care of managing the lock for us.

# Concurrent Package

managing these threads and process entities can be taxing on the developer so Python alleviates this burden by providing an interface which abstracts away the subtleties of starting and tearing down threads or processes. The **concurrent.futures** package provides the Executor interface which can be used to submit tasks to either threads or processes. The two subclasses are:

* ThreadPoolExecutor
* ProcessPoolExecutor

## ThreadPoolExecutor

The ThreadPoolExecutor uses threads for executing submitted tasks. Let's look at a very simple example.

**from concurrent.futures import ThreadPoolExecutor  
from threading import current\_thread  
  
  
def say\_hi(item):  
  
    print("\nhi " + str(item) + " executed in thread id " + current\_thread().name, flush=True)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
    executor = ThreadPoolExecutor(max\_workers=10)  
    lst = list()  
    for i in range(1, 10):  
        lst.append(executor.submit(say\_hi, "guest" + str(i)))  
  
    for future in lst:  
        future.result()  
  
    executor.shutdown()**

We create a thread pool with a maximum of ten threads. Next, we run in a loop and submit tasks to be executed. The first argument to the submit() is a callable which gets invoked with the arguments that follow. If you examine the output you'll see that tasks are executed by threads with different names. The submit calls return what we call a future. The Future class represents the execution of the callable. Note that the invocation future.result() is blocking. Interestingly, if we change the code within the first for loop as follows, the execution becomes serial.

**for i in range(1, 10):  
        future = executor.submit(say\_hi, "guest" + str(i))  
        future.result()**

## threading.Future

You can think of Future as an entity that represents a deferred computation that may or may not have been completed. It is an object that represents the outcome of a computation to be completed in future. We can query about the status of the deferred computation using the methods exposed by the Future class. Some of the useful ones are:

* done: Returns true if the execution of the callable was successfully completed or cancelled.
* cancel: Attempts to cancel execution of a callable. Note if the task is already finished or executing the method returns False.
* cancelled: Returns True if the task was successfully cancelled.
* running: Returns True if the task is currently running and can't be cancelled.
* **from concurrent.futures import ThreadPoolExecutor  
  import time  
    
    
  def square(item):  
      # simulate a computation by sleeping  
      time.sleep(5)  
      return item \* item  
    
    
  if \_\_name\_\_ == '\_\_main\_\_':  
      executor = ThreadPoolExecutor(max\_workers=10)  
    
      future = executor.submit(square, 7)  
    
      print("is running : " + str(future.running()))  
      print("is done : " + str(future.done()))  
      print("Attempt to cancel : " + str(future.cancel()))  
      print("is cancelled : " + str(future.cancelled()))  
    
      executor.shutdown()**

### Exception in Future

If an exception occurs in the callable, it can be retrieved using the exception() method. Examine line#14 in the runnable code below, where we retrieve the exception occurred in the callable and print it. Note that if you asked for result() the exception from the callable would be thrown and the program would exit.

from concurrent.futures import ThreadPoolExecutor

def square(item):

    item = None

    return item \* item

if \_\_name\_\_ == '\_\_main\_\_':

    executor = ThreadPoolExecutor(max\_workers=1)

    lst = list()

    future = executor.submit(square, 7)

    ex = future.exception()

    print(ex)

    executor.shutdown()

### Adding Callbacks

So far we have retrieved results from futures using the result() method. However, this call is blocking and there may be situations where we don't want our program to block. The solution to this dilemma is to add a callback to the future which is invoked when the future has completed or is canceled. The add\_done\_callback() takes a callable as the only argument. In the example below, we attach two callbacks to the future we submit. The callbacks are invoked in the order in which they are added.

#### Adding callbacks to futures

**future = executor.submit(square, 7)  
    future.add\_done\_callback(my\_special\_callback)  
    future.add\_done\_callback(my\_other\_special\_callback)**

from concurrent.futures import ThreadPoolExecutor

def square(item):

    return item \* item

def my\_special\_callback(ftr):

    res = ftr.result()

    print("my\_special\_callback invoked " + str(res))

def my\_other\_special\_callback(ftr):

    res = ftr.result()

    print("my\_other\_special\_callback invoked " + str(res \* res))

if \_\_name\_\_ == '\_\_main\_\_':

    executor = ThreadPoolExecutor(max\_workers=10)

    future = executor.submit(square, 7)

    future.add\_done\_callback(my\_special\_callback)

    future.add\_done\_callback(my\_other\_special\_callback)

    executor.shutdown(wait=False)

# Async.io

## Sending and Receiving in a Generator:

Consider the snippet below:

**def generate\_numbers():  
    i = 0  
    while True:  
        i += 1  
        yield i  
        k = yield  
        print(k)**

You may be surprised how this snippet behaves when we send and receive data.

1. First we create the generator object as follows:

**generator = generate\_numbers()**

Remember creating the generator object doesn't run the generator function.

1. Next, we start the generator by invoking next(). We'll receive a value from the generator function since the first yield statement returns a value. We can do that as follows:

**item = next(generator)  
   print(item)**

1. It is very important to understand that at this point, the generator's execution is suspended at the first yield statement. If we try to send() data, it'll not be received since the generator isn't suspended at a yield assignment statement. Let's run this scenario so that we understand the concept clearly.

def generate\_numbers():

    i = 0

    while True:

        i += 1

        yield i

        k = yield

        print(k)

if \_\_name\_\_ == "\_\_main\_\_":

    generator = generate\_numbers()

    item = next(generator)

    print(item)

    # Nothing is received by the generator function

    generator.send(5)





1. Note that in the above code the generator doesn't receive 5 when we send() it. The value 5 is lost as the generator isn't suspended at a yield assignment statement. In fact, the generator resumes execution from the first yield statement and immediately blocks at the second yield statement. In between, the two yield statements no other line of code is executed. The main method which invokes send() on the generator object receives None because by definition send() returns the next yielded value in a generator function which is None.
2. We can insert a next or a send to move the generator execution from the first yield to the second yield statement. You can consider this a noop.
3. Once the generator object suspends at the second yield statement, we can invoke send() to pass data into the generator function. The generator function would successfully receive the data and at the same time, it'll loop back to the first yield statement and return the value of i as the return value of the send() method. This is demonstrated by the runnable script below:

def generate\_numbers():

    i = 0

    while True:

        i += 1

        yield i

        k = yield

        print("Received in generator function: " + str(k))

if \_\_name\_\_ == "\_\_main\_\_":

    generator = generate\_numbers()

    item = next(generator)

    print("Received in main script: " + str(item))

    # Nothing is received by the generator function

    item = generator.send(5)

    print("Received in main script: " + str(item))

    # The second send is successful

    item = generator.send(5)

    print("Received in main script: " + str(item))





1. Note that the generator again suspends itself at the first yield statement and will require another noop send or next to move to the second yield statement. The code below adds more statements to send and receive data from the generator function alongwith noop operations.

We can instead use a single one to do both and without the need to do noop operations. The generator method to do this appears below:

**def generate\_numbers():  
    i = 0  
  
    while True:  
        i += 1  
        k = (yield i)  
        print(k)**

## Coroutine

*Coroutine isn't a concept specific to Python. In fact, it is a general programming concept also found in other programming languages. A coroutine can be defined as a special function that can give up control to its caller without losing its state.*

### Difference with Generators:

The distinction between generators and coroutines, in general, is that:

* Generators yield back a value to the invoker whereas a coroutine yields control to another coroutine and can resume execution from the point it gives up control.
* A generator can't accept arguments once started whereas a coroutine can.
* Generators are primarily used to simplify writing iterators. They are a type of coroutine and sometimes also called as semicoroutines.

In case, of Python, generators are used as producers of data and coroutines as consumers data. Before support for native coroutines was introduced in Python 3.5, coroutines were implemented using generators. Objects of both, however, are of type generator.

The two types of Python coroutines are:

* Generator based coroutines
* Native coroutines

### Difference with threads

Following are the differences between thread and coroutines:

* One of the major benefits of coroutines over threads is that coroutines don’t use as much memory as threads do.
* Coroutines don't require operating system support or invoke system calls.
* Coroutines don't need to worry about synchronizing access to shared data-structures or guarding critical sections. Mutexes, semaphore and other synchronization constructs aren't required.
* Coroutines are concurrent but not parallel.

## Yield From

The yield from syntax is as follows:

Yield from <expr>

The expression must be an iterable from which an iterator is extracted. Let's understand the problem that **yield from** solves.

Consider the following snippet of code:

**def nested\_generator():  
    i = 0  
    while i < 5:  
        i += 1  
        yield i  
  
  
def outer\_generator():  
    nested\_gen = nested\_generator()  
  
    for item in nested\_gen:  
        yield item  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
    gen = outer\_generator()  
  
    for item in gen:  
        print(item)**

The yield from expects an iterable on its right and runs it to exhaustion. Remember a generator is after all an iterator! In fact the following test for instance-of will return true.

**isinstance(nested\_gen, collections.abc.Iterable)**

In this example the outer\_generator() is called the ***delegating generator*** and the nested generator is called the ***subgenerator.***

Rewriting the code using ***yield from:***

**def nested\_generator():  
    i = 0  
    while i < 5:  
        i += 1  
        yield i  
  
def outer\_generator\_with\_yield\_from():  
    nested\_gen = nested\_generator()  
    yield from nested\_gen  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
    
    gen\_using\_yield\_from = outer\_generator\_with\_yield\_from()  
  
    for item in gen\_using\_yield\_from:  
        print(item)**

### Yield from Using Send

Consider the snippet below:

**def nested\_generator():  
    for \_ in range(5):  
        k = yield  
        print("inner generator received = " + str(k))  
  
  
def outer\_generator():  
    nested\_gen = nested\_generator()  
    next(nested\_gen)  
  
    for \_ in range(5):  
        # receive the value from the caller  
        k = yield  
        try:  
            # send the value to the inner generator  
            nested\_gen.send(k)  
        except StopIteration:  
            pass  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
    gen = outer\_generator()  
    next(gen)  
  
    for i in range(5):  
        try:  
            gen.send(i)  
        except StopIteration:  
            pass**

The outer generator is acting as an intermediary to pass the values it receives from the caller to the inner generator. The above monstrosity can be simplified as follows:

**def nested\_generator():  
    for \_ in range(5):  
        k = yield  
        print("inner generator received = " + str(k))  
  
  
def outer\_generator():  
    nested\_gen = nested\_generator()  
    yield from nested\_gen  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
    gen = outer\_generator()  
    next(gen)  
  
    for i in range(5):  
        try:  
            gen.send(i)  
        except StopIteration:  
            pass**

using yield from:

def nested\_generator():

    for \_ in range(5):

        try:

            k = yield

            print("inner generator received = " + str(k))

        except Exception:

            print("caught an exception")

def outer\_generator():

    nested\_gen = nested\_generator()

    next(nested\_gen)

    for \_ in range(5):

        try:

            k = yield

        except Exception as e:

            nested\_gen.throw(e)

        try:

            nested\_gen.send(k)

        except StopIteration:

            pass

if \_\_name\_\_ == "\_\_main\_\_":

    gen = outer\_generator()

    next(gen)

    for i in range(5):

        try:

            if i == 1:

                gen.throw(Exception("delibrate exception"))

            else:

                gen.send(i)

        except StopIteration:

            pass

***So***

**yield from** can be best thought of as **creating transparent bidirectional communication between the caller and the subgenerator.**

### Yield from with close:

Without using **yield from** if we execute **close()** on the outer generator the inner generator will be left suspended.

import inspect

var = None

def nested\_generator():

    for \_ in range(5):

        k = yield

        print("inner generator received = " + str(k))

def outer\_generator():

    global var

    nested\_gen = nested\_generator()

    var = nested\_gen

    next(nested\_gen)

    for \_ in range(5):

        k = yield

        try:

            nested\_gen.send(k)

        except StopIteration:

            pass

if \_\_name\_\_ == "\_\_main\_\_":

    gen = outer\_generator()

    next(gen)

    try:

        gen.close()

        print("Outer generator state: " + inspect.getgeneratorstate(gen))

        print("Inner generator state: " + inspect.getgeneratorstate(var))

    except StopIteration:

        pass

Output

Outer generator state: GEN\_CLOSED

Inner generator state: GEN\_SUSPENDED

Contrast the above output with the output we get when we use **yield from**. Both the generators are closed.

import inspect

var = None

def nested\_generator():

    for \_ in range(5):

        k = yield

        print("inner generator received = " + str(k))

def outer\_generator():

    global var

    nested\_gen = nested\_generator()

    var = nested\_gen

    yield from nested\_gen

if \_\_name\_\_ == "\_\_main\_\_":

    gen = outer\_generator()

    next(gen)

    try:

        gen.close()

        print("Outer generator state: " + inspect.getgeneratorstate(gen))

        print("Inner generator state: " + inspect.getgeneratorstate(var))

    except StopIteration:

        pass

Output

Outer generator state: GEN\_CLOSED

Inner generator state: GEN\_CLOSED

## Generator Based Coroutines

Generator based coroutines use ***yield from*** syntax instead of yield. A coroutine can:

* yield from another coroutine
* yield from a future
* return an expression
* raise exception
* To sum up, a function that uses ***yield from*** becomes a coroutine and requires the ***@asyncio.coroutine*** decorator. If a function doesn't use yield from adding the decorator will make it a coroutine. Consider the following method
* **@asyncio.coroutine  
  def hello\_world():  
      print("hello world")**
* The above method becomes a coroutine with the addition of the decorator and can be run using the event loop as follows:
* **coro\_obj = hello\_world()  
  asyncio.get\_event\_loop().run\_until\_complete(coro\_obj)**
* In the runnable script below remove the decorator and observe the event loop throw an exception.

## Native Coroutines

Native coroutines can be defined using the async/await syntax. Before getting into further details, here is an example of a very simple native coroutine:

The above coroutine can be run with an event loop as follows:

import asyncio

async def coro():

    await asyncio.sleep(1)

if \_\_name\_\_ == "\_\_main\_\_":

    # run the coroutine

    loop = asyncio.get\_event\_loop()

    loop.run\_until\_complete(coro())

#### Async

We can create a native coroutine by using **async** def. A method prefixed with async def automatically becomes a native coroutine.

The **inspect.iscoroutine()** method would return True for a coroutine object returned from the above coroutine function. Note that yield or yield from can't appear in the body of an async-defined method, else the occurrence would be flagged as a syntax error.

#### Await

**await** can be used to obtain the result of a coroutine object's execution. We use await as:

**await <expr>**

The following objects are awaitable:

* A native coroutine object returned from calling a native coroutine function.
* A generator based coroutine object returned from a generator decorated with @types.coroutine or @asyncio.coroutine. Decorated generator-based coroutines are awaitables, even though they do not have an \_\_await\_\_() method.
* Future objects are awaitable.
* Task objects are awaitable and Task is a subclass of Future.
* Objects defined with CPython C API with a tp\_as\_async.am\_await() function, returning an iterator (similar to \_\_await\_\_() method).

Additionally, await must appear inside an async-defined method, else it's a syntax error.

import asyncio

import types

import inspect

from collections.abc import Iterable, Awaitable

# Ordinary Function

def ordinary\_function():

    pass

# Ordinary Function with @asyncio.coroutine decorator

@asyncio.coroutine

def ordinary\_function\_with\_asyncio\_coroutine\_dec():

    pass

# Ordinary Function with @types.coroutine decorator

@types.coroutine

def ordinary\_function\_with\_types\_coroutine\_dec():

    pass

# Simple Generator

def simple\_generator():

    assign\_me = yield 0

# Simple Generator with @asyncio.coroutine decorator

@asyncio.coroutine

def simple\_generator\_with\_asyncio\_coroutine\_dec():

    assign\_me = yield 0

# Simple Generator with @types.coroutine decorator

@types.coroutine

def simple\_generator\_with\_types\_coroutine\_dec():

    assign\_me = yield 0

# Generator-based coroutine

def generator\_based\_coroutine():

    yield from asyncio.sleep(1)

# Generator-based coroutine with @asyncio.coroutine decorator

@asyncio.coroutine

def generator\_based\_coroutine\_with\_asyncio\_coroutine\_dec():

    yield from asyncio.sleep(1)

# Generator-based coroutine with @types.coroutine decorator

@types.coroutine

def generator\_based\_coroutine\_with\_types\_coroutine\_dec():

    yield from asyncio.sleep(1)

# Native coroutine

async def native\_coroutine():

    pass

if \_\_name\_\_ == "\_\_main\_\_":

    of\_aio\_dec = ordinary\_function\_with\_asyncio\_coroutine\_dec()

    print(of\_aio\_dec)

    print("simple generator instance of collections.abc.Iterable : " + str(isinstance(of\_aio\_dec, Iterable)))

    print("simple generator instance of collections.abc.Awaitable : " + str(isinstance(of\_aio\_dec, Awaitable)))

    print("simple generator instance of types.Generator : " + str(isinstance(of\_aio\_dec, types.GeneratorType)))

    print("simple generator instance of types.CoroutineType : " + str(isinstance(of\_aio\_dec, types.CoroutineType)))

    print("simple generator instance of asyncio.iscoroutine : " + str(asyncio.iscoroutine(of\_aio\_dec)))

    print("simple generator instance of asyncio.iscoroutinefunction : " + str(

        asyncio.iscoroutinefunction(ordinary\_function\_with\_asyncio\_coroutine\_dec)))

    print("simple generator instance of inspect.iscoroutine : " + str(inspect.iscoroutine(of\_aio\_dec)))

    print("generator instance of inspect.iscoroutinefunction : " + str(

        inspect.iscoroutinefunction(ordinary\_function\_with\_asyncio\_coroutine\_dec)))

    print("simple generator instance of inspect.isawaitable : " + str(inspect.isawaitable(of\_aio\_dec)))

    print("\n\n")

    of\_types\_dec = ordinary\_function\_with\_asyncio\_coroutine\_dec()

    print(of\_types\_dec)

    print("simple generator instance of collections.abc.Iterable : " + str(isinstance(of\_types\_dec, Iterable)))

    print("simple generator instance of collections.abc.Awaitable : " + str(isinstance(of\_types\_dec, Awaitable)))

    print("simple generator instance of types.Generator : " + str(isinstance(of\_types\_dec, types.GeneratorType)))

    print("simple generator instance of types.CoroutineType : " + str(isinstance(of\_types\_dec, types.CoroutineType)))

    print("simple generator instance of asyncio.iscoroutine : " + str(asyncio.iscoroutine(of\_types\_dec)))

    print("simple generator instance of asyncio.iscoroutinefunction : " + str(

        asyncio.iscoroutinefunction(ordinary\_function\_with\_types\_coroutine\_dec)))

    print("simple generator instance of inspect.iscoroutine : " + str(inspect.iscoroutine(of\_types\_dec)))

    print("generator instance of inspect.iscoroutinefunction : " + str(

        inspect.iscoroutinefunction(ordinary\_function\_with\_types\_coroutine\_dec)))

    print("simple generator instance of inspect.isawaitable : " + str(inspect.isawaitable(of\_aio\_dec)))

    print("\n\n")

    sg = simple\_generator()

    print(sg)

    print("simple generator instance of collections.abc.Iterable : " + str(isinstance(sg, Iterable)))

    print("simple generator instance of collections.abc.Awaitable : " + str(isinstance(sg, Awaitable)))

    print("simple generator instance of types.Generator : " + str(isinstance(sg, types.GeneratorType)))

    print("simple generator instance of types.CoroutineType : " + str(isinstance(sg, types.CoroutineType)))

    print("simple generator instance of asyncio.iscoroutine : " + str(asyncio.iscoroutine(sg)))

    print("simple generator instance of asyncio.iscoroutinefunction : " + str(

        asyncio.iscoroutinefunction(simple\_generator)))

    print("simple generator instance of inspect.iscoroutine : " + str(inspect.iscoroutine(sg)))

    print("generator instance of inspect.iscoroutinefunction : " + str(

        inspect.iscoroutinefunction(simple\_generator)))

    print("simple generator instance of inspect.isawaitable : " + str(inspect.isawaitable(sg)))

    print("\n\n")

    sg\_aio\_dec = simple\_generator\_with\_asyncio\_coroutine\_dec()

    print(sg\_aio\_dec)

    print("simple generator instance of collections.abc.Iterable : " + str(isinstance(sg\_aio\_dec, Iterable)))

    print("simple generator instance of collections.abc.Awaitable : " + str(isinstance(sg\_aio\_dec, Awaitable)))

    print("simple generator instance of types.Generator : " + str(isinstance(sg\_aio\_dec, types.GeneratorType)))

    print("simple generator instance of types.CoroutineType : " + str(isinstance(sg\_aio\_dec, types.CoroutineType)))

    print("simple generator instance of asyncio.iscoroutine : " + str(asyncio.iscoroutine(sg\_aio\_dec)))

    print("simple generator instance of asyncio.iscoroutinefunction : " + str(

        asyncio.iscoroutinefunction(simple\_generator\_with\_asyncio\_coroutine\_dec)))

    print("simple generator instance of inspect.iscoroutine : " + str(inspect.iscoroutine(sg\_aio\_dec)))

    print("generator instance of inspect.iscoroutinefunction : " + str(

        inspect.iscoroutinefunction(simple\_generator\_with\_asyncio\_coroutine\_dec)))

    print("simple generator instance of inspect.isawaitable : " + str(inspect.isawaitable(sg\_aio\_dec)))

    print("\n\n")

    sg\_types\_dec = simple\_generator\_with\_types\_coroutine\_dec()

    print(sg\_types\_dec)

    print("simple generator instance of collections.abc.Iterable : " + str(isinstance(sg\_types\_dec, Iterable)))

    print("simple generator instance of collections.abc.Awaitable : " + str(isinstance(sg\_types\_dec, Awaitable)))

    print("simple generator instance of types.Generator : " + str(isinstance(sg\_types\_dec, types.GeneratorType)))

    print("simple generator instance of types.CoroutineType : " + str(isinstance(sg\_types\_dec, types.CoroutineType)))

    print("simple generator instance of asyncio.iscoroutine : " + str(asyncio.iscoroutine(sg\_types\_dec)))

    print("simple generator instance of asyncio.iscoroutinefunction : " + str(

        asyncio.iscoroutinefunction(simple\_generator\_with\_types\_coroutine\_dec)))

    print("simple generator instance of inspect.iscoroutine : " + str(inspect.iscoroutine(sg\_types\_dec)))

    print("generator instance of inspect.iscoroutinefunction : " + str(

        inspect.iscoroutinefunction(simple\_generator\_with\_types\_coroutine\_dec)))

    print("simple generator instance of inspect.isawaitable : " + str(inspect.isawaitable(sg\_types\_dec)))

    print("\n\n")

    gbc = generator\_based\_coroutine()

    print(gbc)

    print("generator instance of collections.abc.Iterable : " + str(isinstance(gbc, Iterable)))

    print("generator instance of collections.abc.Awaitable : " + str(isinstance(gbc, Awaitable)))

    print("generator instance of types.Generator : " + str(isinstance(gbc, types.GeneratorType)))

    print("generator instance of types.CoroutineType : " + str(isinstance(gbc, types.CoroutineType)))

    print("generator instance of asyncio.iscoroutine : " + str(asyncio.iscoroutine(gbc)))

    print("generator instance of asyncio.iscoroutinefunction : " + str(

        asyncio.iscoroutinefunction(generator\_based\_coroutine)))

    print("generator instance of inspect.iscoroutine : " + str(inspect.iscoroutine(gbc)))

    print("generator instance of inspect.iscoroutinefunction : " + str(

        inspect.iscoroutinefunction(generator\_based\_coroutine)))

    print("generator instance of inspect.isawaitable : " + str(inspect.isawaitable(gbc)))

    print("\n\n")

    gbc\_aio\_dec = generator\_based\_coroutine\_with\_asyncio\_coroutine\_dec()

    print(gbc\_aio\_dec)

    print("generator instance of collections.abc.Iterable : " + str(isinstance(gbc\_aio\_dec, Iterable)))

    print("generator instance of collections.abc.Awaitable : " + str(isinstance(gbc\_aio\_dec, Awaitable)))

    print("generator instance of types.Generator : " + str(isinstance(gbc\_aio\_dec, types.GeneratorType)))

    print("generator instance of types.CoroutineType : " + str(isinstance(gbc\_aio\_dec, types.CoroutineType)))

    print("generator instance of asyncio.iscoroutine : " + str(asyncio.iscoroutine(gbc\_aio\_dec)))

    print("generator instance of asyncio.iscoroutinefunction : " + str(

        asyncio.iscoroutinefunction(generator\_based\_coroutine\_with\_asyncio\_coroutine\_dec)))

    print("generator instance of inspect.iscoroutine : " + str(inspect.iscoroutine(gbc\_aio\_dec)))

    print("generator instance of inspect.iscoroutinefunction : " + str(

        inspect.iscoroutinefunction(generator\_based\_coroutine\_with\_asyncio\_coroutine\_dec)))

    print("generator instance of inspect.isawaitable : " + str(inspect.isawaitable(gbc\_aio\_dec)))

    print("\n\n")

    gbc\_types\_dec = generator\_based\_coroutine\_with\_types\_coroutine\_dec()

    print(gbc\_types\_dec)

    print("generator instance of collections.abc.Iterable : " + str(isinstance(gbc\_types\_dec, Iterable)))

    print("generator instance of collections.abc.Awaitable : " + str(isinstance(gbc\_types\_dec, Awaitable)))

    print("generator instance of types.Generator : " + str(isinstance(gbc\_types\_dec, types.GeneratorType)))

    print("generator instance of types.CoroutineType : " + str(isinstance(gbc\_types\_dec, types.CoroutineType)))

    print("generator instance of asyncio.iscoroutine : " + str(asyncio.iscoroutine(gbc\_types\_dec)))

    print("generator instance of asyncio.iscoroutinefunction : " + str(

        asyncio.iscoroutinefunction(generator\_based\_coroutine\_with\_types\_coroutine\_dec)))

    print("generator instance of inspect.iscoroutine : " + str(inspect.iscoroutine(gbc\_types\_dec)))

    print("generator instance of inspect.iscoroutinefunction : " + str(

        inspect.iscoroutinefunction(generator\_based\_coroutine\_with\_types\_coroutine\_dec)))

    print("generator instance of inspect.isawaitable : " + str(inspect.isawaitable(gbc\_types\_dec)))

    print("\n\n")

    nc = native\_coroutine()

    print("native coro instance of collections.abc.Iterable : " + str(isinstance(nc, Iterable)))

    print("native coro instance of collections.abc.Awaitable : " + str(isinstance(nc, Awaitable)))

    print("native coro instance of types.Generator : " + str(isinstance(nc, types.GeneratorType)))

    print("native coro instance of types.CoroutineType : " + str(isinstance(nc, types.CoroutineType)))

    print("native coro instance of asyncio.iscoroutine : " + str(asyncio.iscoroutine(nc)))

    print("native coro instance of asyncio.iscoroutinefunction : " + str(asyncio.iscoroutinefunction(native\_coroutine)))

    print("native coro instance of inspect.iscoroutine : " + str(inspect.iscoroutine(nc)))

    print("generator instance of inspect.iscoroutinefunction : " + str(

        inspect.iscoroutinefunction(native\_coroutine)))

    print("native coro instance of inspect.isawaitable : " + str(inspect.isawaitable(nc)))

    print(nc)

    print("\n\n")

#### Native Vs Generator Based Coroutines

Generator based coroutines and native coroutines have differences between themselves which are listed below:

* Native coroutines don't implement the **\_\_iter\_\_()** and **\_\_next\_\_()** methods and therefore can't be iterated upon.
* Generator based coroutines can't **yield from** a native coroutine. The following will result in a syntax error:

**def gen\_based\_coro():  
    yield from asyncio.sleep(10)**

However, if we decorate the **gen\_based\_coro()** with the decorator @asyncio.coroutine then it is allowed to **yield from** a native coroutine. The following is thus legal:

**@asyncio.coroutine  
def gen\_based\_coro():  
    yield from asyncio.sleep(10)**

* Methods **inspect.isgenerator()** and **inspect.isgeneratorfunction()** return false for native coroutine objects while true for generator-based coroutine objects and functions.

#### @asyncio.coroutine

Adding the @asyncio.coroutine decorator makes generator based coroutines compatible with native coroutines. Without the decorator it would not be possible to **yield from** a native coroutine inside of a generator based coroutine. Consider the example below:

**import asyncio  
  
  
@asyncio.coroutine  
def gen\_based\_coro():  
    yield from asyncio.sleep(1)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
    gen = gen\_based\_coro()  
    next(gen)**

The decorator also allows a generator based coroutine to be awaited in a native coroutine. Consider the below example:

**import asyncio  
  
@asyncio.coroutine  
def gen\_based\_coro():  
    return 10  
  
async def main():  
    rcvd = await gen\_based\_coro()  
    print("native coroutine received: " + str(rcvd))  
  
if \_\_name\_\_ == "\_\_main\_\_":  
    loop = asyncio.get\_event\_loop()  
    loop.run\_until\_complete(main())**

## Chaining Coroutines

**def coro3(k):  
    yield (k + 3)  
  
  
def coro2(j):  
    j = j \* j  
    yield from coro3(j)  
  
  
def coro1():  
    i = 0  
    while True:  
        yield from coro2(i)  
        i += 1  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
    # The first 100 natural numbers evaluated for the following expression  
    # x^2 + 3  
  
    cr = coro1()  
    for v in range(100):  
        print("f({0}) = {1}".format(v, next(cr)))**

The setup is as follows:

* The first coroutine produces natural numbers starting from 1.
* The second coroutine computes the square of each passed in input.
* The last function is a generator and adds 3 to the value passed into it and yields the result.
* In the example above, the end of the chain consists of a generator, however, this chain wouldn't run with the asyncio's event loop since it doesn't work with generators. One way to fix is to change the last generator into an ordinary function that returns a future with the result computed. The method **coro3()** would change to:
* **def coro3(k):  
      f = Future()  
      f.set\_result(k + 3)  
      f.done()  
      return f**

Note that in the previous examples we didn't decorate coro1() and coro2() with @asyncio.coroutine. Both the functions are generator-based coroutine functions because of the presence of yield from in their function bodies.

### Chaining Native Coroutines

**import asyncio  
  
  
async def coro3(k):  
    return k + 3  
  
  
async def coro2(j):  
    j = j \* j  
    res = await coro3(j)  
    return res  
  
  
async def coro1():  
    i = 0  
    while i < 100:  
        res = await coro2(i)  
        print("f({0}) = {1}".format(i, res))  
        i += 1  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
    # The first 100 natural numbers evaluated for the following expression  
    # x^2 + 3  
    cr = coro1()  
    loop = asyncio.get\_event\_loop()  
    loop.run\_until\_complete(cr)**

# Event Loop

# The event loop is a programming construct that waits for events to happen and then dispatches them to an event handler. An event can be a user clicking on a UI button or a process initiating a file download. At the core of asynchronous programming, sits the event loop. The concept isn't novel to Python. In fact, many programming languages enable asynchronous programming with event loops. In Python, event loops run asynchronous tasks and callbacks, perform network IO operations, run subprocesses and delegate costly function calls to pool of threads.

# One of the most common use cases you'll find in the wild is of webservers implemented using asynchronous design. A webserver waits for an HTTP request to arrive and returns the matching resource. Folks familiar with JavaScript would recall NodeJS works on the same principle. It is a webserver that runs an event loop to receive web requests in a single thread. Contrast that to webservers which create a new thread or worse fork a new process, to handle each web request. In some benchmarks, the asynchronous event loop based webservers outperformed multithreaded ones, which may seem counterintuitive.

## Running the Event Loop

With Python 3.7+ the preferred way to run the event loop is to use the **asyncio.run()** method. The method is a blocking call till the passed-in coroutine finishes. A sample program appears below:

**async def do\_something\_important():  
    await asyncio.sleep(10)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
  
  asyncio.run(do\_something\_important())**

If you are working with Python 3.5, then the **asyncio.run()** API isn't available. In that case, we explicitly retrieve the event loop using **asyncio.new\_event\_loop()** and run our desired coroutine using **run\_until\_complete()** defined on the loop object.

# Future & Tasks

## Future

**Future** represents a computation that is either in progress or will get scheduled in the future. It is a special low-level awaitable object that represents an eventual result of an asynchronous operation. Don't confuse threading.Future and asyncio.Future. The former is part of the threading module and doesn't have an \_\_iter\_\_() method defined on it. asyncio.Future is an awaitable and can be used with the yield from statement.

# Interview Practice Problems

## Implementing a Barrier

A barrier can be thought of as a point in the program code, which all or some of the threads need to reach at before any one of them is allowed to proceed further.

A barrier allows multiple threads to congregate at a point in code before any one of the thread is allowed to move forward. Python and most other languages provide libraries which make barrier construct available for developer use. Even though we are re-inventing the wheel but this makes for a good interview question.

We can immediately realize that our solution will need a count variable to track the number of threads that have arrived at the barrier. If we have n threads, then n-1 threads must wait for the nth thread to arrive. This suggests we have the n-1 threads execute the wait method and the nth thread wakes up all the asleep n-1 threads.

from threading import Condition

from threading import Thread

from threading import current\_thread

import time

class Barrier(object):

    def \_\_init\_\_(self, size):

        self.barrier\_size = size

        self.reached\_count = 0

        self.cond = Condition()

    def arrived(self):

        self.cond.acquire()

        self.reached\_count += 1

        if self.reached\_count == self.barrier\_size:

            self.cond.notifyAll()

            self.reached\_count = 0

        else:

            self.cond.wait()

        self.cond.release()

def thread\_process(sleep\_for):

    time.sleep(sleep\_for)

    print("Thread {0} reached the barrier".format(current\_thread().getName()))

    barrier.arrived()

    time.sleep(sleep\_for)

    print("Thread {0} reached the barrier".format(current\_thread().getName()))

    barrier.arrived()

    time.sleep(sleep\_for)

    print("Thread {0} reached the barrier".format(current\_thread().getName()))

    barrier.arrived()

if \_\_name\_\_ == "\_\_main\_\_":

    barrier = Barrier(3)

    t1 = Thread(target=thread\_process, args=(0,))

    t2 = Thread(target=thread\_process, args=(0.5,))

    t3 = Thread(target=thread\_process, args=(1.5,))

    t1.start()

    t2.start()

    t3.start()

    t1.join()

    t2.join()

    t3.join()

**The above code has a subtle but very crucial bug!** Can you spot the bug and try to fix it before reading on?

We discussed in previous sections that wait() should always be used with a while loop that checks for a condition, and if found false, should make the thread wait again.

The condition the while loop can check for is simply how many threads have incremented the reached\_count variable so far. A thread that wakes up spuriously should go back to waiting if the reached\_count is less than the size of the barrier. We can check for this condition as follows:

**while self.reached\_count < self.barrier\_size  
       wait();**

Below is the improved version:

**class Barrier(object):  
1.     def \_\_init\_\_(self, size):  
2.         self.barrier\_size = size  
3.         self.reached\_count = 0  
4.         self.released\_count = self.barrier\_size  
5.         self.cond = Condition()  
6.   
7.     def arrived(self):  
8.   
9.         self.cond.acquire()  
10.  
11.        self.reached\_count += 1  
12.  
13.        if self.reached\_count == self.barrier\_size:  
14.            self.released\_count = self.barrier\_size  
15.              
16.        else:  
17.             while self.reached\_count < self.barrier\_size:  
18.                self.cond.wait()  
19.  
20.        self.released\_count -= 1  
21.  
22.        if self.released\_count == 0:  
23.            self.reached\_count = 0  
24.  
25.        self.cond.notifyAll()  
26.        self.cond.release()**

**There is still a bug in the above code!** Can you guess what it is?

### Final Version

To understand why the above code is broken, consider three threads t1, t2, and t3 trying to await on a barrier object in an infinite loop. Note the following sequence of events:

1. Threads t1 and t2 invoke arrived() and end up waiting at line#18. The reached\_count variable is set to 2 and any spurious wakeups will cause t1 and t2 to go back to waiting. So far so good.
2. Threads t3 comes along, executes the if block on line#13 and finds reached\_count == barrier\_size condition to be true. Thread t3 doesn't wait, notifies threads t1 and t2 to wake up, and exits.
3. If thread t3 attempts to invoke arrived() immediately after exiting it and is successful before threads t1 or t2 get a chance to acquire the condition variable back, then the reached\_count variable will be incremented to 4.
4. With reached\_count equal to 4, t3 will not block at the barrier and exit which breaks the contract for the barrier.
5. The invocation order of the arrived() method was t1, t2, t3, and then t3 again. The right behaviour would have been to release t1, t2, or t3 in any order and then block t3 on its second invocation of the arrived() method.
6. Another flaw with the above code is that it can cause a deadlock. Suppose we wanted the three threads t1, t2, and t3 to congregate at a barrier twice. The first invocation was in the order [t1, t2, t3] and the second was in the order [t3, t2, t1]. If t3 immediately invoked arrived() after the first barrier, it would go past the second barrier without stopping while t2 and t1 would become stranded at the second barrier, since reached\_count would never equal barrier\_size.

The fix requires us to block any new threads from proceeding until all the threads that have reached the previous barrier are released. The code with the fix appears below:

from threading import Condition

from threading import Thread

from threading import current\_thread

import time

class Barrier(object):

    def \_\_init\_\_(self, size):

        self.barrier\_size = size

        self.reached\_count = 0

        self.released\_count = self.barrier\_size

        self.cond = Condition()

    def arrived(self):

        self.cond.acquire()

        while self.reached\_count == self.barrier\_size:

            self.cond.wait()

        self.reached\_count += 1

        if self.reached\_count == self.barrier\_size:

            self.released\_count = self.barrier\_size

        else:

            while self.reached\_count < self.barrier\_size:

                self.cond.wait()

        self.released\_count -= 1

        if self.released\_count == 0:

            self.reached\_count = 0

        print("{0} released".format(current\_thread().getName()), flush=True)

        self.cond.notifyAll()

        self.cond.release()

def thread\_process(sleep\_for):

    time.sleep(sleep\_for)

    print("Thread {0} reached the barrier".format(current\_thread().getName()), flush=True)

    barrier.arrived()

    time.sleep(sleep\_for)

    print("Thread {0} reached the barrier".format(current\_thread().getName()))

    barrier.arrived()

    time.sleep(sleep\_for)

    print("Thread {0} reached the barrier".format(current\_thread().getName()))

    barrier.arrived()

if \_\_name\_\_ == "\_\_main\_\_":

    barrier = Barrier(3)

    t1 = Thread(target=thread\_process, args=(0,))

    t2 = Thread(target=thread\_process, args=(0.5,))

    t3 = Thread(target=thread\_process, args=(1.5,))

    t1.start()

    t2.start()

    t3.start()

    t1.join()

    t2.join()

    t3.join()

## Rate Limiting Using Token Bucket Filter

Problem Statement:

*This is an actual interview question asked at Uber and Oracle.*

Imagine you have a bucket that gets filled with tokens at the rate of 1 token per second. The bucket can hold a maximum of N tokens. Implement a thread-safe class that lets threads get a token when one is available. If no token is available, then the token-requesting threads should block.

The class should expose an API called **get\_token()** that various threads can call to get a token.

## Thread Safe Deferred Callback

Problem Statement:

Design and implement a thread-safe class that allows registration of callback methods that are executed after a user specified time interval in seconds has elapsed.

Solution

Let us try to understand the problem without thinking about concurrency. Let's say our class exposes an API called add\_action() that'll take a parameter action, which will get executed after user specified seconds. Anyone calling this API should be able to specify after how many seconds should our class invoke the passed-in action.

One naive way to solve this problem is to have a busy thread that continuously loops over the list of actions and executes them as they become due. However, the challenge here is to design a solution which doesn't involve a busy thread.

One possible solution is to have an execution thread that maintains a priority queue (min-heap) of actions ordered by the time remaining to execute each of the actions. The execution thread can sleep for the duration equal to the time duration before the earliest action in the min-heap becomes due for execution.

Consumer threads can come and add their desired actions in the min-heap within the critical section. The caveat here is that the execution thread will need to be woken up to recalculate the minimum duration it would sleep for before an action is due for execution. An action with an earlier due timestamp might have been added while the executor thread was sleeping on a duration calculated for an action due later than the one just added.

Consider this example: initially, the execution thread is sleeping for 30 mins before any action in the min-heap is due. A consumer thread comes along and adds an action to be executed after 5 minutes. The execution thread would need to wake up and reset itself to sleep for only 5 minutes instead of 30 minutes. Once we find an elegant way of achieving this our problem is pretty much solved.

from threading import Condition

from threading import Thread

import heapq

import time

import math

class DeferredCallbackExecutor():

    def \_\_init\_\_(self):

        self.actions = list()

        self.cond = Condition()

        self.sleep = 0

    def add\_action(self, action):

        # add exec\_at time for the action

        action.execute\_at = time.time() + action.exec\_secs\_after

        self.cond.acquire()

        heapq.heappush(self.actions, action)

        self.cond.notify()

        self.cond.release()

    def start(self):

        while True:

            self.cond.acquire()

            while len(self.actions) is 0:

                self.cond.wait()

            while len(self.actions) is not 0:

                # calculate sleep duration

                next\_action = self.actions[0]

                sleep\_for = next\_action.execute\_at - math.floor(time.time())

                if sleep\_for <= 0:

                    # time to execute action

                    break

                self.cond.wait(timeout=sleep\_for)

            action\_to\_execute\_now = heapq.heappop(self.actions)

            action\_to\_execute\_now.action(\*(action\_to\_execute\_now,))

            self.cond.release()

class DeferredAction(object):

    def \_\_init\_\_(self, exec\_secs\_after, name, action):

        self.exec\_secs\_after = exec\_secs\_after

        self.action = action

        self.name = name

    def \_\_lt\_\_(self, other):

        return self.execute\_at < other.execute\_at

def say\_hi(action):

        print("hi, I am {0} executed at {1} and required at {2}".format(action.name, math.floor(time.time()),

                                                                    math.floor(action.execute\_at)))

if \_\_name\_\_ == "\_\_main\_\_":

    action1 = DeferredAction(3, ("A",), say\_hi)

    action2 = DeferredAction(2, ("B",), say\_hi)

    action3 = DeferredAction(1, ("C",), say\_hi)

    action4 = DeferredAction(7, ("D",), say\_hi)

    executor = DeferredCallbackExecutor()

    t = Thread(target=executor.start, daemon=True)

    t.start()

    executor.add\_action(action1)

    executor.add\_action(action2)

    executor.add\_action(action3)

    executor.add\_action(action4)

    # wait for all actions to execute

    time.sleep(15)

## Blocking Queue | Bounded Buffer | Consumer Producer

Classical synchronization problem involving a limited size buffer which can have items added to it or removed from it by different producer and consumer threads. This problem is known by different names: consumer producer problem, bounded buffer problem or blocking queue problem.

A blocking queue is defined as a queue which blocks the caller of the enqueue method if there's no more capacity to add the new item being enqueued. Similarly, the queue blocks the dequeue caller if there are no items in the queue. Also, the queue notifies a blocked enqueuing thread when space becomes available and a blocked dequeuing thread when an item becomes available in the queue.

We'll need two variables: one to keep track of the maximum size of the queue and another to track the current size of the queue. Moreover, we'll also need a condition variable that either the producer or the consumer can wait on if the queue is full or empty respectively.

Initially, our class will look like as follows:

**class BlockingQueue:  
  
    def \_\_init\_\_(self, max\_size):  
        self.max\_size = max\_size  
        self.curr\_size = 0  
        self.cond = Condition()  
        self.q = []  
  
    def enqueue(self):  
        pass  
  
    def dequeue(self):  
        pass**

The complete code appears in the code widget below along with an example.

from threading import Thread

from threading import Condition

from threading import current\_thread

import time

import random

class BlockingQueue:

    def \_\_init\_\_(self, max\_size):

        self.max\_size = max\_size

        self.curr\_size = 0

        self.cond = Condition()

        self.q = []

    def dequeue(self):

        self.cond.acquire()

        while self.curr\_size == 0:

            self.cond.wait()

        item = self.q.pop(0)

        self.curr\_size -= 1

        self.cond.notifyAll()

        self.cond.release()

        return item

    def enqueue(self, item):

        self.cond.acquire()

        while self.curr\_size == self.max\_size:

            self.cond.wait()

        self.q.append(item)

        self.curr\_size += 1

        self.cond.notifyAll()

        print("\ncurrent size of queue {0}".format(self.curr\_size), flush=True)

        self.cond.release()

def consumer\_thread(q):

    while 1:

        item = q.dequeue()

        print("\n{0} consumed item {1}".format(current\_thread().getName(), item), flush=True)

        time.sleep(random.randint(1, 3))

def producer\_thread(q, val):

    item = val

    while 1:

        q.enqueue(item)

        item += 1

        time.sleep(0.1)

if \_\_name\_\_ == "\_\_main\_\_":

    blocking\_q = BlockingQueue(5)

    consumerThread1 = Thread(target=consumer\_thread, name="consumer-1", args=(blocking\_q,), daemon=True)

    consumerThread2 = Thread(target=consumer\_thread, name="consumer-2", args=(blocking\_q,), daemon=True)

    producerThread1 = Thread(target=producer\_thread, name="producer-1", args=(blocking\_q, 1), daemon=True)

    producerThread2 = Thread(target=producer\_thread, name="producer-2", args=(blocking\_q, 100), daemon=True)

    consumerThread1.start()

    consumerThread2.start()

    producerThread1.start()

    producerThread2.start()

    time.sleep(15)

    print("Main thread exiting")

**Follow Up Question**

Does it matter if we use **notify()** or **notifyAll()** method in our implementation?

In both the **enqueue()** and **dequeue()** methods we use the **notifyAll()** method instead of the **notify()** method. The reason behind the choice is very crucial to understand. Consider a situation with two producer threads and one consumer thread all working with a queue of size one. It's possible that when an item is added to the queue by one of the producer threads, the other two threads are blocked waiting on the condition variable. If the producer thread after adding an item invokes **notify()** it is possible that the other producer thread is chosen by the system to resume execution. The woken-up producer thread would find the queue full and go back to waiting on the condition variable, causing a deadlock. Invoking **notifyAll()** assures that the consumer thread also gets a chance to wake up and resume execution.

## Non-Blocking Queue

We have seen the blocking version of a queue in the previous question that blocks a producer or a consumer when the queue is full or empty respectively. In this problem, you are asked to implement a queue that is non-blocking.

Let's first define the notion of **non-blocking**. If a consumer or a producer can successfully enqueue or dequeue an item, it is considered non-blocking. However, if the queue is full or empty then a producer or a consumer (respectively) need not wait until the queue can be added to or taken from.

### First Cut

The trivial solution is to return a boolean value indicating the success of an operation. If the invoker of either **enqueue()** or **dequeue()** receives False, then it is the responsibility of the invoker to retry the operation at a later time. This trivial solution appears in the code widget below.

from threading import Thread

from threading import Lock

from threading import current\_thread

from concurrent.futures import Future

import time

import random

class NonBlockingQueue:

    def \_\_init\_\_(self, max\_size):

        self.max\_size = max\_size

        self.q = []

        self.lock = Lock()

    def dequeue(self):

        with self.lock:

            curr\_size = len(self.q)

            if curr\_size != 0:

                return self.q.pop(0)

            else:

                return False

    def enqueue(self, item):

        with self.lock:

            curr\_size = len(self.q)

            if curr\_size == self.max\_size:

                return False

            else:

                self.q.append(item)

                return True

def consumer\_thread(q):

    while 1:

        item = q.dequeue()

        if item == False:

            print("Consumer couldn't dequeue an item")

        else:

            print("\n{0} consumed item {1}".format(current\_thread().getName(), item), flush=True)

        time.sleep(random.randint(1, 3))

def producer\_thread(q):

    item = 1

    while 1:

        result = q.enqueue(item)

        if result is True:

            print("\n {0} produced item".format(current\_thread().getName()), flush=True)

            item += 1

if \_\_name\_\_ == "\_\_main\_\_":

    no\_block\_q = NonBlockingQueue(5)

    consumerThread1 = Thread(target=consumer\_thread, name="consumer", args=(no\_block\_q,), daemon=True)

    producerThread1 = Thread(target=producer\_thread, name="producer", args=(no\_block\_q,), daemon=True)

    consumerThread1.start()

    producerThread1.start()

    time.sleep(15)

    print("Main thread exiting")

### Second Cut

If we want to get more sophisticated in our approach we can return an object of **concurrent.futures.Future** class to the invoker of the queue APIs incase the requested operation can't be completed at the time of invocation.

from threading import Thread

from threading import Lock

from threading import current\_thread

from concurrent.futures import Future

import time

import random

class NonBlockingQueue:

    def \_\_init\_\_(self, max\_size):

        self.max\_size = max\_size

        self.q = []

        self.q\_waiting\_puts = []

        self.q\_waiting\_gets = []

        self.lock = Lock()

    def dequeue(self):

        result = None

        future = None

        with self.lock:

            curr\_size = len(self.q)

            if curr\_size != 0:

                result = self.q.pop()

                if len(self.q\_waiting\_puts) > 0:

                    self.q\_waiting\_puts.pop().set\_result(True)

            else:

                future = Future()

                self.q\_waiting\_gets.append(future)

        return result, future

    def enqueue(self, item):

        future = None

        with self.lock:

            curr\_size = len(self.q)

            if curr\_size == self.max\_size:

                future = Future()

                self.q\_waiting\_puts.append(future)

            else:

                self.q.append(item)

                if len(self.q\_waiting\_gets) != 0:

                    future\_get = self.q\_waiting\_gets.pop()

                    future\_get.set\_result(True)

        return future

def consumer\_thread(q):

    while 1:

        item, future = q.dequeue()

        if item is None:

            print("Consumer received a future but we are ignoring it")

        else:

            print("\n{0} consumed item {1}".format(current\_thread().getName(), item), flush=True)

        # slow down consumer thread

        time.sleep(random.randint(1, 3))

def producer\_thread(q):

    item = 1

    while 1:

        future = q.enqueue(item)

        if future is not None:

            while future.done() == False:

                print("waiting for future to resolve")

                time.sleep(0.1)

        else:

            item += 1

if \_\_name\_\_ == "\_\_main\_\_":

    no\_block\_q = NonBlockingQueue(5)

    consumerThread1 = Thread(target=consumer\_thread, name="consumer", args=(no\_block\_q,), daemon=True)

    producerThread1 = Thread(target=producer\_thread, name="producer", args=(no\_block\_q,), daemon=True)

    consumerThread1.start()

    producerThread1.start()

    time.sleep(15)

    print("Main thread exiting")