PYTHON FEATURES

Contents

Python Zip Function—Real Python

Zip function is a bulitin. It aggregates elements from each of the iterables passed to it. It returns a tuple of the elements of the iterables. Stops bundling when the shortest length iterables is exhausted. For lists of length 5,7,8 the tuple will be of length 5.

Python's zip() function is defined as zip(*iterables). The function takes in iterables as arguments and returns an iterator. This iterator generates a series of tuples containing elements from each iterable. zip() can accept any type of iterable, such as files, lists, tuples, dictionaries, sets, and so on.

If you use zip() with n arguments, then the function will return an iterator that generates tuples of length n.

Zip itself returns an iterator, which must be consumed before being used.

```
>>> numbers = [1, 2, 3]
>>> letters = ['a', 'b', 'c']
>>> zipped = zip(numbers, letters)
>>> zipped  # Holds an iterator object
<zip object at 0x7fa4831153c8>
>>> type(zipped)
<class 'zip'>
>>> list(zipped)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

Related links: zip(*iterables),

(python.org),

If you're working with sequences like lists, tuples, or strings, then your iterables are guaranteed to be evaluated from left to right. However, for other types of iterables (like sets), you might see some weird results. Set objects don't keep their elements in any particular order. This means that the tuples returned by zip() will have elements that are paired up randomly. If you're going to use the Python zip() function with unordered iterables like sets, then this is something to keep in mind.

Zips are generator objects, you can call next() on them to retrieve items. They raise StopIteration at the end.

```
>>> a = [1,2,3]
>>> b = [5,6,7]
>>> c = zip(a,b)
>>> c
<zip object at 0x000002434111B180>
>>> next(c)
(1, 5)
```

Passing Arguments of Unequal Length

When you're working with the Python zip() function, it's important to pay attention to the length of your iterables. It's possible that the iterables you pass in as arguments aren't the same length. In these cases, the number of elements that zip() puts out will be equal to the length of the shortest iterable. The remaining elements in any longer iterables will be totally ignored by zip()

```
>>> list(zip(range(5), range(100)))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

Since 5 is the length of the first (and shortest) range() object, zip() outputs a list of five tuples. There are still 95 unmatched elements from the second range() object. These are all ignored by zip() since there are no more elements from the first range() object to complete the pairs.

If trailing or unmatched values are important to you, then you can use itertools.zip_longest() instead of zip(). With this function, the missing values will be replaced with whatever you pass to the fillvalue argument (defaults to None). The iteration will continue until the longest iterable is exhausted: Here, you use itertools.zip_longest() to yield five tuples with elements from letters, numbers, and longest. The iteration only stops when longest is exhausted. The missing elements from numbers and letters are filled with a question mark ?, which is what you specified with fillvalue.

```
>>> from itertools import zip_longest
>>> numbers = [1, 2, 3]
>>> letters = ['a', 'b', 'c']
>>> longest = range(5)
>>> zipped = zip_longest(numbers, letters, longest, fillvalue='?')
>>> list(zipped)
[(1, 'a', 0), (2, 'b', 1), (3, 'c', 2), ('?', '?', 3), ('?', '?', 4)]
```

Comparing zip() in Python 3 and 2

In Python 2 zip returns a list and in 3 it returns a zip object, which is an iterator.

Looping Over Multiple Iterables

Looping over multiple iterables is one of the most common use cases for Python's zip() function. If you need to iterate through multiple lists, tuples, or any other sequence, then it's likely that you'll fall back on zip().Python's zip() function allows you to iterate in parallel over two or more iterables. Since zip() generates tuples, you can unpack these in the header of a for loop.

```
>>> letters = ['a', 'b', 'c']
>>> numbers = [0, 1, 2]
>>> operators = ['*', '/', '+']
>>> for l, n, o in zip(letters, numbers, operators):
...     print(f'Letter: {l}')
...     print(f'Number: {n}')
...     print(f'Operator: {o}')
...
Letter: a
Number: 0
Operator: *
Letter: b
Number: 1
Operator: /
Letter: c
Number: 2
Operator: +
```

Here, you iterate through the series of tuples returned by zip() and unpack the elements into l, o, and n. When you combine zip(), for loops, and tuple unpacking, you can get a useful and Pythonic idiom for traversing two or more iterables at once.

In Python 3.6 and beyond, dictionaries are ordered collections, meaning they keep their elements in the same order in which they were introduced.

```
>>> dict_one = {'name': 'John', 'last_name': 'Doe', 'job': 'Python Consultant'}
>>> dict_two = {'name': 'Jane', 'last_name': 'Doe', 'job': 'Community Manager'}
>>> for (k1, v1), (k2, v2) in zip(dict_one.items(), dict_two.items()):
...     print(k1, '->', v1)
...     print(k2, '->', v2)
...
name -> John
name -> Jane
last_name -> Doe
last_name -> Doe
job -> Python Consultant
job -> Community Manager
```

Here, you iterate through dict_one and dict_two in parallel. In this case, zip() generates tuples with the items from both dictionaries. Then, you can unpack each tuple and gain access to the items of both dictionaries at the same time.

Unzipping a Sequence

The opposite of zip() is zip(). Zip can be used to unzip with * because it zips things into separate iterables by aggregating them in the same order they were packed in. Zip is a bijective function, and much like a packed up box, things get out the way they were packed in.

```
>>> pairs = [(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
>>> numbers, letters = zip(*pairs)
>>> numbers
(1, 2, 3, 4)
>>> letters
('a', 'b', 'c', 'd')
```

Sorting in Parallel

Sorting is a common operation in programming. Suppose you want to combine two lists and sort them at the same time. To do this, you can use zip() along with .sort().

```
>>> letters = ['b', 'a', 'd', 'c']
>>> numbers = [2, 4, 3, 1]
>>> data1 = list(zip(letters, numbers))
```

```
>>> data1
[('b', 2), ('a', 4), ('d', 3), ('c', 1)]
>>> data1.sort()  # Sort by letters
>>> data1
[('a', 4), ('b', 2), ('c', 1), ('d', 3)]
>>> data2 = list(zip(numbers, letters))
>>> data2
[(2, 'b'), (4, 'a'), (3, 'd'), (1, 'c')]
>>> data2.sort()  # Sort by numbers
>>> data2
[(1, 'c'), (2, 'b'), (3, 'd'), (4, 'a')]
```

In this example, you first combine two lists with zip() and sort them. Notice how data1 is sorted by letters and data2 is sorted by numbers. List of tuples get sorted by the first item in the tuple.

```
>>> letters = ['b', 'a', 'd', 'c']
>>> numbers = [2, 4, 3, 1]
>>> data = sorted(zip(letters, numbers))  # Sort by letters
>>> data
[('a', 4), ('b', 2), ('c', 1), ('d', 3)]
```

You can also use sorted() and zip() together to achieve a similar result. In this case, sorted() runs through the iterator generated by zip() and sorts the items by letters, all in one go. This approach can be a little bit faster since you'll need only two function calls: zip() and sorted(). With sorted(), you're also writing a more general piece of code.

Calculating in Pairs

You can use the Python `zip()` function to make some quick calculations. Suppose you have the following data in a spreadsheet:

Element/Month	January	February	March
Total Sales	52,000.00	51,000.00	48,000.00
Production Cost	46,800.00	45,900.00	43,200.00

```
>>> total_sales = [52000.00, 51000.00, 48000.00]
>>> prod_cost = [46800.00, 45900.00, 43200.00]
>>> for sales, costs in zip(total_sales, prod_cost):
...     profit = sales - costs
...     print(f'Total profit: {profit}')
...
Total profit: 5200.0
Total profit: 5100.0
Total profit: 4800.0
```

Here, you calculate the profit for each month by subtracting costs from sales. Python's zip() function combines the right pairs of data to make the calculations. You can generalize this logic to make any kind of complex calculation with the pairs returned by zip().

Building Dictionaries

To build a dictionary from two different but closely related sequences, a convenient way is to use dict() and zip() together.

```
>>> fields = ['name', 'last_name', 'age', 'job']
>>> values = ['John', 'Doe', '45', 'Python Developer']
>>> a_dict = dict(zip(fields, values))
>>> a_dict
{'name': 'John', 'last_name': 'Doe', 'age': '45', 'job': 'Python Developer'}
>>> new_job = ['Python Consultant']
>>> field = ['job']
>>> a_dict.update(zip(field, new_job))
>>> a_dict
{'name': 'John', 'last_name': 'Doe', 'age': '45', 'job': 'Python Consultant'}
```

Here, you create a dictionary that combines the two lists. zip(fields, values) returns an iterator that generates 2-items tuples. If you call dict() on that iterator, then you'll be building the dictionary you need. The elements of fields become the dictionary's keys, and the elements of values represent the values in the dictionary. You can also update an existing dictionary by combining zip() with dict.update().

Threading in Python—Real Python

Threads are lightweight processes. They can be thought of as separate flow of execution from a common parent program. But for most Python 3 implementations the different threads do not actually execute at the same time: they merely appear to. Getting multiple tasks running simultaneously requires a non-standard implementation of Python, writing some of your code in a different language, or using multiprocessing which comes with some extra overhead.

Because of the way CPython implementation of Python works, threading may not speed up all tasks. This is due to interactions with the GIL that essentially limit one Python thread to run at a time. Tasks that spend much of their time waiting for external events are generally good candidates for threading. Problems that require heavy CPU computation and spend little time waiting for external events might not run faster at all. This is true for code written in Python and running on the standard CPython implementation. If your threads are written in C they have the ability to release the GIL and run concurrently.

Starting a Thread

To start a separate thread, you create a **Thread** instance and then tell it to **.start()**

```
import logging
 2 import threading
 3 import time
 4
 5 def thread_function(name):
 6     logging.info("Thread %s: starting", name)
 7     time.sleep(2)
 8     logging.info("Thread %s: finishing", name)
 9
10 if __name__ == "__main__":
11     format = "%(asctime)s: %(message)s"
12     logging.basicConfig(format=format, level=logging.INFO,
13                         datefmt="%H:%M:%S")
14
15     logging.info("Main    : before creating thread")
16     x = threading.Thread(target=thread_function, args=(1,))
17     logging.info("Main    : before running thread")
18     x.start()
19     logging.info("Main    : wait for the thread to finish")
20     # x.join()
21     logging.info("Main    : all done")
```

When you create a Thread, you pass it a function and a list containing the arguments to that function. In this case, you're telling the Thread to run thread_function() and to pass it 1 as an argument. Here x is a **Thread** object that is passed a function, and it's arguments are passed in the **args=** parameter. The thread then must be started using **.start()** .

threading.get_ident() returns a unique name for each thread, but these are usually neither short nor easily readable.

Output:

```
$ ./single_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
Thread 1: finishing
```

The Thread finished after the Main section of code did.

Daemon Threads

In computer science, a daemon is a process that runs in the background. Python threading has a more specific meaning for daemon. A daemon thread will shut down immediately when the program exits. One way to think about these definitions is to consider the daemon thread a thread that runs in the background without worrying about shutting it down. If a program is running Threads that are not daemons, then the program will wait for those threads to complete before it terminates. Threads that are daemons, however, are just killed wherever they are when the program is exiting.

When you run the program, you'll notice that there is a pause (of about 2 seconds) after __main__ has printed its all done message and before the thread is finished. This pause is Python waiting for the non-daemonic thread to complete. When your Python program ends, part of the shutdown process is to clean up the threading routine. If you look at the source for Python threading, you'll see that threading._shutdown() walks through all of the running threads and calls .join() on every one that does not have the daemon flag set. So your program waits to exit because the thread itself is waiting in a sleep. As soon as it has completed and printed the message, .join() will return and the program can exit.

The program refactored with a daemon thread by changing how you construct the Thread, adding the daemon=True flag:

```
x = threading.Thread(target=thread_function, args=(1,), daemon=True)
```

output:

```
$ ./daemon_thread.py
Main    : before creating thread
Main    : before running thread
Thread 1: starting
Main    : wait for the thread to finish
Main    : all done
```

The difference here is that the final line of the output is missing. thread_function() did not get a chance to complete. It was a daemon thread, so when __main__ reached the end of its code and the program wanted to finish, the daemon was killed.

join() a Thread

This is for when you want to wait for a thread to stop, when you want to do that and not exit your program.

To tell one thread to wait for another thread to finish, you call .join(). If you uncomment that line in the example, the main thread will pause and wait for the thread x to complete running. If you .join() a thread, that statement will wait until either kind of thread is finished.

Working With Many Threads

Frequently, you'll want to start a number of threads and have them do interesting work. Let's start by looking at the harder way of doing that

```
import logging, threading, time

def thread_example(index):
    logging.info(f"Thread {index} starting")
    time.sleep(index+10)
    logging.info(f"Thread {index} finishing")

if __name__ == "__main__":
    format="%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")

    threads = []
    for index in range(5):
        logging.info(f"Main  : create and start thread {index}")
        new_thread = threading.Thread(target=thread_example, args=(index,))
        threads.append(new_thread)
        new_thread.start()

    for index, thread in enumerate(threads):
        logging.info(f"Main  : before joining thread {index}")
        thread.join()
        logging.info(f"Main  : finishing thread {index}")
```

This code uses the same mechanism you saw above to start a thread, create a Thread object, and then call .start(). The program keeps a list of Thread objects so that it can then wait for them later using .join(). Multiple runs will produce different orderings. Look for the Thread x: finishing message to tell you when each thread is done. The order in which threads are run is determined by the operating system and can be quite hard to predict. It may (and likely will) vary from run to run, so you need to be aware of that when you design algorithms that use threading.

```
'(Anaconda3)'J:\Education\Code\Python\Python-Features\scripts>python threads.py
07:32:56: Main  : create and start thread 0
07:32:56: Thread 0 starting
07:32:56: Main  : create and start thread 1
07:32:56: Thread 1 starting
07:32:56: Main  : create and start thread 2
```

```
07:32:56: Thread 2 starting
07:32:56: Main  : create and start thread 3
07:32:56: Thread 3 starting
07:32:56: Main  : create and start thread 4
07:32:56: Thread 4 starting
07:32:56: Main  : before joining thread 0
07:33:06: Thread 0 finishing
07:33:06: Main  : finishing thread 0
07:33:06: Main  : before joining thread 1
07:33:07: Thread 1 finishing
07:33:07: Main  : finishing thread 1
07:33:07: Main  : before joining thread 2
07:33:08: Thread 2 finishing
07:33:08: Main  : finishing thread 2
07:33:08: Main  : before joining thread 3
07:33:09: Thread 3 finishing
07:33:09: Main  : finishing thread 3
07:33:09: Main  : before joining thread 4
07:33:10: Thread 4 finishing
07:33:10: Main  : finishing thread 4
```

This example might have started and ended in the correct order of execution, but it wont' always be like that.

Using a ThreadPoolExecutor

ThreadPoolExecutor, is a part of the standard library in concurrent.futures (as of Python 3.2). The easiest way to create it is as a context manager, using the with statement to manage the creation and destruction of the pool.

```python
import logging, threading, time
import concurrent.futures

def thread_example(index):
    logging.info(f"Thread {index} starting")
    time.sleep(index+10)
    logging.info(f"Thread {index} finishing")

if __name__ == "__main__":
    format="%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")

    with concurrent.futures.ThreadPoolExecutor(max_workers=3) as executor:
        executor.map(thread_example, range(3))
```

The code creates a ThreadPoolExecutor as a context manager, telling it how many worker threads it wants in the pool. It then uses .map() to step through an iterable of things, in your case range(3), passing each one to a thread in the pool. The end of the with block causes the ThreadPoolExecutor to do a .join() on each of the threads in the pool. It is strongly recommended that you use ThreadPoolExecutor as a context manager when you can so that you never forget to .join() the threads.

Using a ThreadPoolExecutor can cause some confusing errors. For example, if you call a function that takes no parameters, but you pass it parameters in .map(), the thread will throw an exception. Unfortunately, ThreadPoolExecutor will hide that exception, and (in the case above) the program terminates with no output. This can be quite confusing to debug at first.

```
'(Anaconda3)'J:\Education\Code\Python\Python-Features\scripts>python threads2.py
07:54:16: Thread 0 starting
07:54:16: Thread 1 starting
07:54:16: Thread 2 starting
07:54:26: Thread 0 finishing
07:54:27: Thread 1 finishing
07:54:28: Thread 2 finishing
```

Race Conditions

FakeDatabase is keeping track of a single number: .value. This is going to be the shared data on which you'll see the race condition. .__init__() simply initializes .value to zero. So far, so good. .update() looks a little strange. It's simulating reading a value from a database, doing some computation on it, and then writing a new value back to the database. In this case, reading from the database just means copying .value to a local variable. The computation is just to add one to the value and then .sleep() for a little bit. Finally, it writes the value back by copying the local value back to .value.

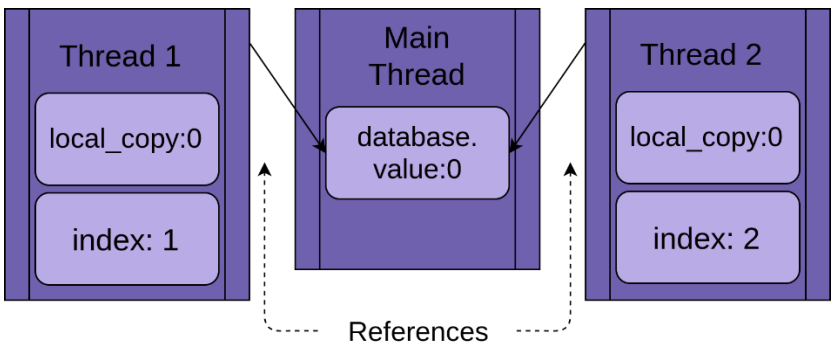
```python
import logging, threading, time
import concurrent.futures
class FakeDatabase:
    def __init__(self):
        self.value = 0

    def update(self, name):
        logging.info(f"Thread {name} starting update")
        local_copy = self.value
```

```
        local_copy += 1
        time.sleep(0.1)
        self.value = local_copy
        logging.info(f"Thread {name} finishing update")
if __name__ == "__main__":
    format="%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")
    database = FakeDatabase()
    logging.info(f"Start = {database.value}")
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        for index in range(2):
            executor.submit(database.update, index)
    logging.info(f"End = {database.value}")
```

The program creates a ThreadPoolExecutor with two threads and then calls .submit() on each of them, telling them to run database.update(). .submit() has a signature that allows both positional and named arguments to be passed to the function running in the thread

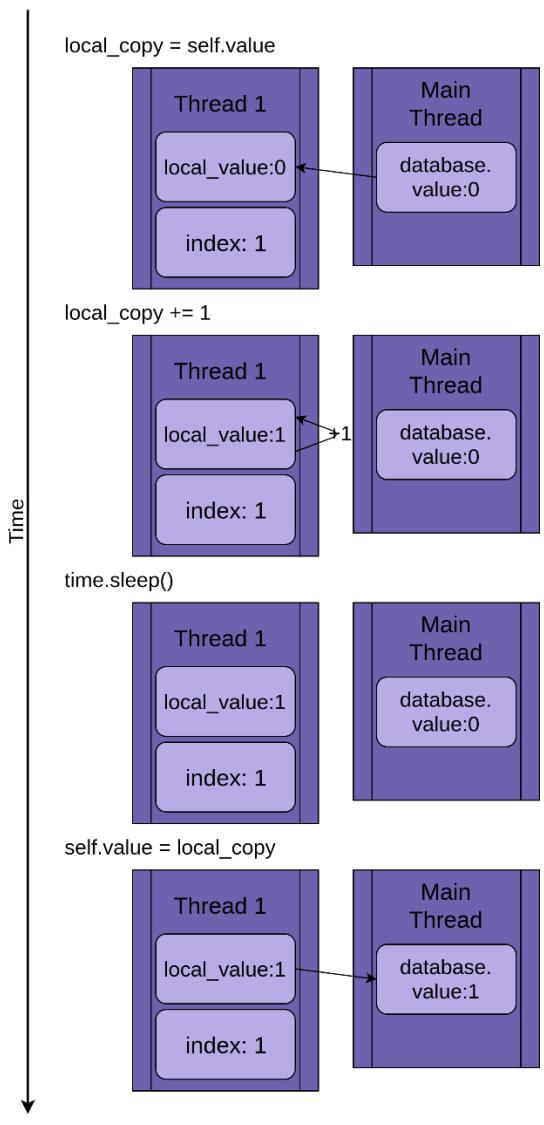
In the usage above, index is passed as the first and only positional argument to database.update(). Since each thread runs .update(), and .update() adds one to .value, you might expect database.value to be 2 when it's printed out at the end.

When you tell your ThreadPoolExecutor to run each thread, you tell it which function to run and what parameters to pass to it: executor.submit(database.update, index). The result of this is that each of the threads in the pool will call database.update(index). Note that database is a reference to the one FakeDatabase object created in __main__. Calling .update() on that object calls an instance method on that object. Each thread is going to have a reference to the same FakeDatabase object, database. Each thread will also have a unique value, index, to make the logging statements a bit easier to read.



When the thread starts running .update(), it has its own version of all of the data local to the function. In the case of .update(), this is local_copy. This is definitely a good thing. Otherwise, two threads running the same function would always confuse each other. It means that all variables that are scoped (or local) to a function are thread-safe.

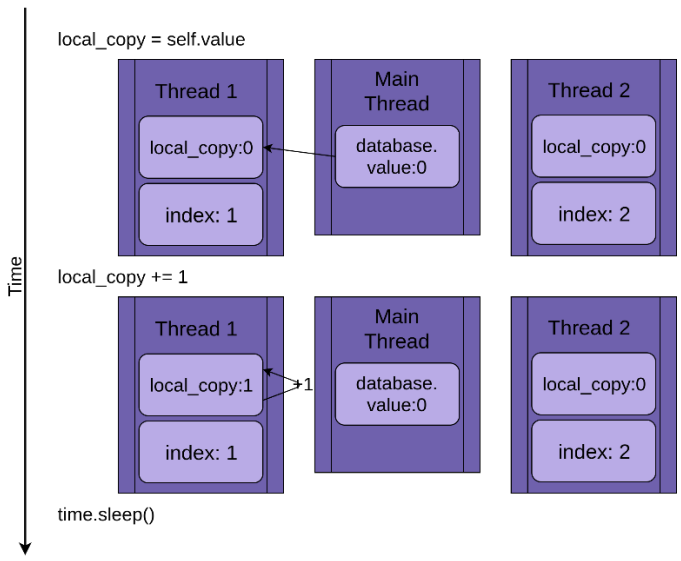
The image below steps through the execution of .update() if only a single thread is run. The statement is shown on the left followed by a diagram showing the values in the thread's local_value and the shared database.value.



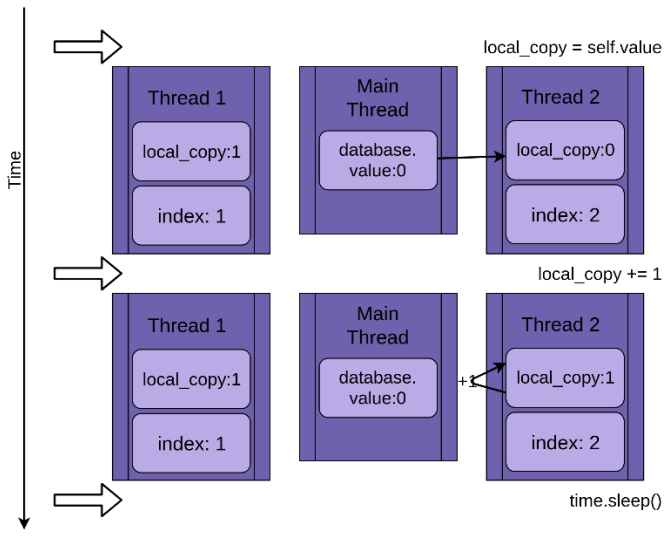
The diagram is laid out so that time increases as you move from top to bottom. It begins when Thread 1 is created and ends when it is terminated. When Thread 1 starts, FakeDatabase.value is zero. The first line of code in the method, local_copy = self.value, copies the value zero to the local variable. Next it

increments the value of local_copy with the local_copy += 1 statement. You can see .value in Thread 1 getting set to one. Next time.sleep() is called, which makes the current thread pause and allows other threads to run. Since there is only one thread in this example, this has no effect. When Thread 1 wakes up and continues, it copies the new value from local_copy to FakeDatabase.value, and then the thread is complete. You can see that database.value is set to one. So far, so good. You ran .update() once and FakeDatabase.value was incremented to one.

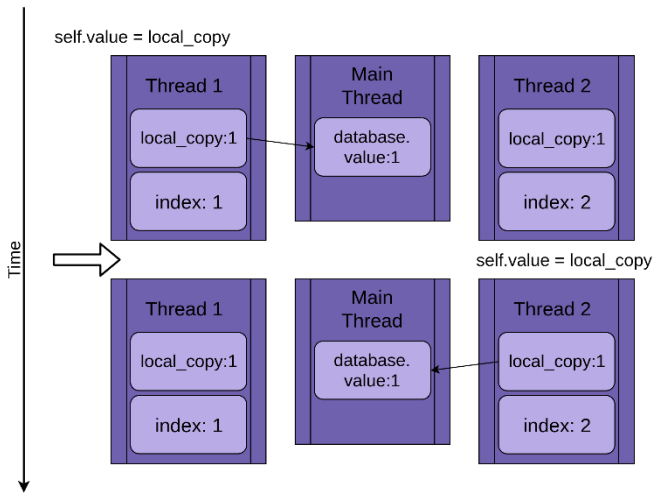
Getting back to the race condition, the two threads will be running concurrently but not at the same time. They will each have their own version of local_copy and will each point to the same database. It is this shared database object that is going to cause the problems. The program starts with Thread 1 running .update()



When Thread 1 calls time.sleep(), it allows the other thread to start running. Thread 2 starts up and does the same operations. It's also copying database.value into its private local_copy, and this shared database.value has not yet been updated.



When Thread 2 finally goes to sleep, the shared database.value is still unmodified at zero, and both private versions of local_copy have the value one.Thread 1 now wakes up and saves its version of local_copy and then terminates, giving Thread 2 a final chance to run. Thread 2 has no idea that Thread 1 ran and updated database.value while it was sleeping. It stores its version of local_copy into database.value, also setting it to one.



The two threads have interleaving access to a single shared object, overwriting each other's results. Similar race conditions can arise when one thread frees memory or closes a file handle before the other thread is finished accessing it.

There are two things to keep in mind when thinking about race conditions:

1. Even an operation like x += 1 takes the processor many steps. Each of these steps is a separate instruction to the processor.
2. The operating system can swap which thread is running at any time. A thread can be swapped out after any of these small instructions. This means that a thread can be put to sleep to let another thread run in the middle of a Python statement.

It's rare to get a race condition like this to occur, but remember that an infrequent event taken over millions of iterations becomes likely to happen. The rarity of these race conditions makes them much, much harder to debug than regular bugs.

Basic Synchronization Using Lock

To solve your race condition above, you need to find a way to allow only one thread at a time into the read-modify-write section of your code. The most common way to do this is called Lock in Python. In some other languages this same idea is called a mutex. Mutex comes from MUTual EXclusion, which is

exactly what a Lock does. A Lock is an object that acts like a hall pass. Only one thread at a time can have the Lock. Any other thread that wants the Lock must wait until the owner of the Lock gives it up.

The basic functions to do this are .acquire() and .release(). A thread will call my_lock.acquire() to get the lock. If the lock is already held, the calling thread will wait until it is released. There's an important point here. If one thread gets the lock but never gives it back, your program will be stuck.

Python's Lock will also operate as a context manager, so you can use it in a with statement, and it gets released automatically when the with block exits for any reason.

FakeDatabase with a Lock added to it.

```
import logging, threading, time
import concurrent.futures
class FakeDatabase:
    def __init__(self):
        self.value = 0
        self._lock = threading.Lock()

    def update_locked(self, name):
        logging.info(f"Thread {name} starting update")
        logging.debug(f"Thread {name} about to lock")
        with self._lock:
            logging.debug(f"Thread {name} has lock")
            local_copy = self.value
            local_copy += 1
            time.sleep(0.1)
            self.value = local_copy
            logging.debug(f"Thread {name} releasing lock")
        logging.debug(f"Thread {name} after lock")
        logging.info(f"Thread {name} finishing update")
if __name__ == "__main__":
    format="%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")
    logging.getLogger().setLevel(logging.DEBUG)
    database = FakeDatabase()
    logging.info(f"Start = {database.value}")
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        for index in range(2):
            executor.submit(database.update_locked, index)
    logging.info(f"End = {database.value}")
```

Other than adding a bunch of debug logging so you can see the locking more clearly, the big change here is to add a member called ._lock, which is a threading.Lock() object. This ._lock is initialized in the unlocked state and locked and released by the with statement. It's worth noting here that the thread running this function will hold on to that Lock until it is completely finished updating the database. In this case, that means it will hold the Lock while it copies, updates, sleeps, and then writes the value back to the database. In this output you can see Thread 0 acquires the lock and is still holding it when it goes to sleep. Thread 1 then starts and attempts to acquire the same lock. Because Thread 0 is still holding it, Thread 1 has to wait. This is the mutual exclusion that a Lock provides.

```
'(Anaconda3)'J:\Education\Code\Python\Python-Features\scripts>python threads4.py
08:46:20: Start = 0
08:46:20: Thread 0 starting update
08:46:20: Thread 0 about to lock
08:46:20: Thread 0 has lock
08:46:20: Thread 1 starting update
08:46:20: Thread 1 about to lock
08:46:21: Thread 0 releasing lock
08:46:21: Thread 0 after lock
08:46:21: Thread 1 has lock
08:46:21: Thread 0 finishing update
08:46:21: Thread 1 releasing lock
08:46:21: Thread 1 after lock
08:46:21: Thread 1 finishing update
08:46:21: End = 2
```

Deadlock

As you saw, if the Lock has already been acquired, a second call to .acquire() will wait until the thread that is holding the Lock calls .release().

```
import threading


l = threading.Lock()
print("before first acquire")
l.acquire()
```

```
print("before second acquire")
l.acquire()
print("acquired lock twice")
```

When the program calls l.acquire() the second time, it hangs waiting for the Lock to be released. In this example, you can fix the deadlock by removing the second call, but deadlocks usually happen from one of two subtle things:

1. An implementation bug where a Lock is not released properly
2. A design issue where a utility function needs to be called by functions that might or might not already have the Lock

The first situation happens sometimes, but using a Lock as a context manager greatly reduces how often. It is recommended to write code whenever possible to make use of context managers, as they help to avoid situations where an exception skips you over the .release() call.

The design issue can be a bit trickier in some languages. Thankfully, Python threading has a second object, called RLock, that is designed for just this situation. It allows a thread to .acquire() an RLock multiple times before it calls .release(). That thread is still required to call .release() the same number of times it called .acquire(), but it should be doing that anyway. Lock and RLock are two of the basic tools used in threaded programming to prevent race conditions.

Producer-Consumer Threading

The Producer-Consumer Problem is a standard computer science problem used to look at threading or process synchronization issues.

For this example, you're going to imagine a program that needs to read messages from a network and write them to disk. The program does not request a message when it wants. It must be listening and accept messages as they come in. The messages will not come in at a regular pace, but will be coming in bursts. This part of the program is called the producer.

On the other side, once you have a message, you need to write it to a database. The database access is slow, but fast enough to keep up to the average pace of messages. It is not fast enough to keep up when a burst of messages comes in. This part is the consumer.

In between the producer and the consumer, you will create a Pipeline that will be the part that changes as you learn about different synchronization objects.

Producer-Consumer Using Lock

The general design is that there is a producer thread that reads from the fake network and puts the message into a Pipeline:

```
import random

SENTINEL = object()

def producer(pipeline):
    """Pretend we're getting a message from the network."""
    for index in range(10):
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        pipeline.set_message(message, "Producer")

    # Send a sentinel message to tell consumer we're done
    pipeline.set_message(SENTINEL, "Producer")
```

The producer also uses a SENTINEL value to signal the consumer to stop after it has sent ten values. This is a little awkward, but don't worry, you'll see ways to get rid of this SENTINEL value after you work through this example.

On the other side of the pipeline is the consumer:

```
def consumer(pipeline):
    """Pretend we're saving a number in the database."""
    message = 0
    while message is not SENTINEL:
        message = pipeline.get_message("Consumer")
        if message is not SENTINEL:
            logging.info("Consumer storing message: %s", message)
```

The consumer reads a message from the pipeline and writes it to a fake database, which in this case is just printing it to the display. If it gets the SENTINEL value, it returns from the function, which will terminate the thread.

```
if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")
    # logging.getLogger().setLevel(logging.DEBUG)
```

```
    pipeline = Pipeline()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(producer, pipeline)
        executor.submit(consumer, pipeline)
```

Pipeline that passes messages from the producer to the consumer:

```
class Pipeline:
    """
    Class to allow a single element pipeline between producer and consumer.
    """
    def __init__(self):
        self.message = 0
        self.producer_lock = threading.Lock()
        self.consumer_lock = threading.Lock()
        self.consumer_lock.acquire()

    def get_message(self, name):
        logging.debug("%s:about to acquire getlock", name)
        self.consumer_lock.acquire()
        logging.debug("%s:have getlock", name)
        message = self.message
        logging.debug("%s:about to release setlock", name)
        self.producer_lock.release()
        logging.debug("%s:setlock released", name)
        return message

   def set_message(self, message, name):
        logging.debug("%s:about to acquire setlock", name)
        self.producer_lock.acquire()
        logging.debug("%s:have setlock", name)
        self.message = message
        logging.debug("%s:about to release getlock", name)
        self.consumer_lock.release()
        logging.debug("%s:getlock released", name)
```

The Pipeline has three members:

1. .message stores the message to pass.
2. .producer_lock is a threading.Lock object that restricts access to the message by the producer thread.
3. .consumer_lock is also a threading.Lock that restricts access to the message by the consumer thread.

__init__() initializes these three members and then calls .acquire() on the .consumer_lock. This is the state you want to start in. The producer is allowed to add a new message, but the consumer needs to wait until a message is present.

.get_message() and .set_messages() are nearly opposites. .get_message() calls .acquire() on the consumer_lock. This is the call that will make the consumer wait until a message is ready. Once the consumer has acquired the .consumer_lock, because the producer has released this lock, it copies out the value in .message and then calls .release() on the .producer_lock. Releasing this lock is what allows the producer to insert the next message into the pipeline. Before you go on to .set_message(), there's something subtle going on in .get_message() that's pretty easy to miss. It might seem tempting to get rid of message and just have the function end with return self.message. However this opens up a race condition.

The producer and consumer act only when the release() is used by the other.

As soon as the consumer calls .producer_lock.release(), it can be swapped out, and the producer can start running. That could happen before .release() returns! This means that there is a slight possibility that when the function returns self.message, that could actually be the next message generated, so you would lose the first message. This is another example of a race condition.

Moving on to .set_message(), you can see the opposite side of the transaction. The producer will call this with a message. It will acquire the .producer_lock, then the consumer will release the lock, set the .message, and the call .release() on then consumer_lock, which will allow the consumer to read that value.

The consumer lock is acquired. The consumer tries to acquire this same lock but has to wait for the previous one to be released. The producer then acquires lock, since it is free to do so. It then sets a message. It releases the consumer's lock, and consumer can acquire a new lock to get the message. The producer tries to acquire a new lock, but since the previous one wasn't release, it has to wait for the consumer to finish and then release it's lock and it can acquire a new one.

```python
import random
import logging, threading, time
import concurrent.futures

SENTINEL = object()

def producer(pipeline):
    """
    Pretend we are getting a msg
    """
    for index in range(10):
        message = random.randint(1,101)
        logging.info(f"Producer got message: {message}")
        pipeline.set_message(message, "Producer")

    pipeline.set_message(SENTINEL, "Producer")

def consumer(pipeline):
    """
    Pretend we are saving a msg
    """
    message = 0
    while message is not SENTINEL:
        message=pipeline.get_message("Consumer")
        if message is not SENTINEL:
            logging.info(f"Consumer storing message: {message}")

class Pipeline:
    """
    Class to allow a single element pipeline between producer and consumer
    """
    def __init__(self):
        self.message = 0
        self.producer_lock = threading.Lock()
        self.consumer_lock = threading.Lock()
        self.consumer_lock.acquire()

    def get_message(self,name):
        logging.debug(f"{name}: about to acquire getlock")
        self.consumer_lock.acquire()
        logging.debug(f"{name}: have getlock")
        message = self.message
        logging.debug(f"{name}: getting message {message}")
        logging.debug(f"{name}: about to release setlock")
        self.producer_lock.release()
        logging.debug(f"{name}: released setlock")
        return message

    def set_message(self, message, name):
        logging.debug(f"{name}: about to acquire setlock")
        self.producer_lock.acquire()
        logging.debug(f"{name}: have setlock")
        logging.debug(f"{name}: setting message {message}")
        self.message = message
        logging.debug(f"{name}: about to release getlock")
        self.consumer_lock.release()
        logging.debug(f"{name}: released getlock")

if __name__ == "__main__":
    format="%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")
    logging.getLogger().setLevel(logging.INFO)
    pipeline = Pipeline()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(producer, pipeline)
        executor.submit(consumer, pipeline)
```

Output:

```
'(Anaconda3)'J:\Education\Code\Python\Python-Features\scripts>python threads5.py
12:56:57: Producer got message: 25
12:56:57: Producer got message: 32
12:56:57: Consumer storing message: 25
```

```
12:56:57: Producer got message: 66
12:56:57: Consumer storing message: 32
12:56:57: Producer got message: 10
12:56:57: Consumer storing message: 66
12:56:57: Producer got message: 91
12:56:57: Consumer storing message: 10
12:56:57: Producer got message: 78
12:56:57: Consumer storing message: 91
12:56:57: Producer got message: 15
12:56:57: Consumer storing message: 78
12:56:57: Producer got message: 39
12:56:57: Consumer storing message: 15
12:56:57: Consumer storing message: 39
12:56:57: Producer got message: 87
12:56:57: Producer got message: 81
12:56:57: Consumer storing message: 87
12:56:57: Consumer storing message: 81
```

If you look back at the producer and .set_message(), you will notice that the only place it will wait for a Lock is when it attempts to put the message into the pipeline. This is done after the producer gets the message and logs that it has it. When the producer attempts to send this second message, it will call .set_message() the second time and it will block. The operating system can swap threads at any time, but it generally lets each thread have a reasonable amount of time to run before swapping it out. That's why the producer usually runs until it blocks in the second call to .set_message().

Once a thread is blocked, however, the operating system will always swap it out and find a different thread to run. In this case, the only other thread with anything to do is the consumer. The consumer calls .get_message(), which reads the message and calls .release() on the .producer_lock, thus allowing the producer to run again the next time threads are swapped. Notice that the first message was 43, and that is exactly what the consumer read, even though the producer had already generated the 45 message.

While it works for this limited test, it is not a great solution to the producer-consumer problem in general because it only allows a single value in the pipeline at a time. When the producer gets a burst of messages, it will have nowhere to put them.

Producer-Consumer Using Queue

To handle more than one value in the pipeline at a time, you'll need a data structure for the pipeline that allows the number to grow and shrink as data backs up from the producer. Python's standard library has a queue module which, in turn, has a Queue class. Let's change the Pipeline to use a Queue instead of just a variable protected by a Lock. A different way to stop the worker threads by using a different primitive from Python threading is an Event.

The threading.Event object allows one thread to signal an event while many other threads can be waiting for that event to happen. The key usage in this code is that the threads that are waiting for the event do not necessarily need to stop what they are doing, they can just check the status of the Event every once in a while. The triggering of the event can be many things.

In this example, the main thread will simply sleep for a while and then .set() it:

```
if __name__ == "__main__":
 2      format = "%(asctime)s: %(message)s"
 3      logging.basicConfig(format=format, level=logging.INFO,
 4                          datefmt="%H:%M:%S")
 5      # logging.getLogger().setLevel(logging.DEBUG)
 6
 7      pipeline = Pipeline()
 8      event = threading.Event()
 9      with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
10          executor.submit(producer, pipeline, event)
11          executor.submit(consumer, pipeline, event)
12
13          time.sleep(0.1)
14          logging.info("Main: about to set event")
15          event.set()
```

The only changes here are the creation of the event object on line 6, passing the event as a parameter on lines 8 and 9, and the final section on lines 11 to 13, which sleep for a second, log a message, and then call .set() on the event.

```
def producer(pipeline, event):
 2      """Pretend we're getting a number from the network."""
 3      while not event.is_set():
 4          message = random.randint(1, 101)
 5          logging.info("Producer got message: %s", message)
 6          pipeline.set_message(message, "Producer")
 7
```

```
    logging.info("Producer received EXIT event. Exiting")
```

It now will loop until it sees that the event was set on line 3. It also no longer puts the SENTINEL value into the pipeline.

consumer changes:

```
def consumer(pipeline, event):
    """Pretend we're saving a number in the database."""
    while not event.is_set() or not pipeline.empty():
        message = pipeline.get_message("Consumer")
        logging.info(
            "Consumer storing message: %s  (queue size=%s)",
            message,
            pipeline.qsize(),
        )

    logging.info("Consumer received EXIT event. Exiting")
```

While you got to take out the code related to the SENTINEL value, you did have to do a slightly more complicated while condition. Not only does it loop until the event is set, but it also needs to keep looping until the pipeline has been emptied.

Making sure the queue is empty before the consumer finishes prevents another fun issue. If the consumer does exit while the pipeline has messages in it, there are two bad things that can happen. The first is that you lose those final messages, but the more serious one is that the producer can get caught attempting to add a message to a full queue and never return.

This happens if the event gets triggered after the producer has checked the .is_set() condition but before it calls pipeline.set_message(). If that happens, it's possible for the consumer to wake up and exit with the queue still completely full, since in this hypothetical the consumer doesn't check for emptiness of the queue.. The producer will then call .set_message() which will wait until there is space on the queue for the new message. The consumer has already exited, so this will not happen and the producer will not exit.

Because the producer cannot add to a full queue, and must wait till there is space, the producer will wait indefinitely.

```
class Pipeline(queue.Queue):
    def __init__(self):
        super().__init__(maxsize=10)

    def get_message(self, name):
        logging.debug("%s:about to get from queue", name)
        value = self.get()
        logging.debug("%s:got %d from queue", name, value)
        return value

    def set_message(self, value, name):
        logging.debug("%s:about to add %d to queue", name, value)
        self.put(value)
        logging.debug("%s:added %d to queue", name, value)
```

You can see that Pipeline is a subclass of queue.Queue. Queue has an optional parameter when initializing to specify a maximum size of the queue. If you give a positive number for maxsize, it will limit the queue to that number of elements, causing .put() to block until there are fewer than maxsize elements. This means that .put( ) will wait for a part of the quee to empty so that it can add stuff. If you don't specify maxsize, then the queue will grow to the limits of your computer's memory.

.get_message() and .set_message() got much smaller. They basically wrap .get() and .put() on the Queue. You might be wondering where all of the locking code that prevents the threads from causing race conditions went. The core devs who wrote the standard library knew that a Queue is frequently used in multi-threading environments and incorporated all of that locking code inside the Queue itself. Queue is thread-safe. This means that there is no need to impose the locking mechanism manually on the pipeline, since all of that has already been implemented.

```
import random
import logging, threading, time
import concurrent.futures
import queue

SENTINEL = object()

def producer(pipeline, event):
    """
    Pretend we are getting a msg
    """
    while not event.is_set():
```

```
        message = random.randint(1,101)
        logging.info(f"Producer got message: {message}")
        pipeline.set_message(message, "Producer")
    logging.info("Producer received EXIT event. Exiting....")

def consumer(pipeline,event):
    """
    Pretend we are saving a msg
    """
    while not event.is_set() or not pipeline.empty():
        message=pipeline.get_message("Consumer")
        if message is not SENTINEL:
            logging.info(f"Consumer storing message: {message} (queue size = {pipeline.qsize()}")
    logging.info("Consumer received EXIT event. Exiting....")

class Pipeline(queue.Queue):
    """
    Class to allow a single element pipeline between producer and consumer
    """
    def __init__(self):
        super().__init__(maxsize=10)

    def get_message(self,name):
        logging.debug(f"{name}: about to get from queue")
        value = self.get()
        logging.debug(f"{name}: got value {value}")
        return value

    def set_message(self, value, name):
        logging.debug(f"{name}: about to add {value} to queue")
        self.put(value)
        logging.debug(f"{name}: {value} added to queue")

if __name__ == "__main__":
    format="%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO, datefmt="%H:%M:%S")
    logging.getLogger().setLevel(logging.INFO)
    pipeline = Pipeline()
    event = threading.Event()
    with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
        executor.submit(producer, pipeline, event)
        executor.submit(consumer, pipeline, event)

        time.sleep(0.1)
        logging.info("Main: about to set event")
        event.set()
```

```
'(Anaconda3)'J:\Education\Code\Python\Python-Features\scripts>python threads6.py
13:33:09: Producer got message: 1
13:33:09: Producer got message: 61
13:33:09: Producer got message: 18
13:33:09: Consumer storing message: 1 (queue size = 1
13:33:09: Consumer storing message: 61 (queue size = 1
13:33:09: Producer got message: 59
13:33:09: Consumer storing message: 18 (queue size = 0
13:33:09: Consumer storing message: 59 (queue size = 0
13:33:09: Producer got message: 96
13:33:09: Producer got message: 21
13:33:09: Consumer storing message: 96 (queue size = 0
13:33:09: Producer got message: 23
13:33:09: Consumer storing message: 21 (queue size = 0
13:33:09: Consumer storing message: 23 (queue size = 0
13:33:09: Producer got message: 69
13:33:09: Producer got message: 50
13:33:09: Consumer storing message: 28 (queue size = 0
13:33:09: Producer got message: 41
13:33:09: Consumer storing message: 40 (queue size = 0
13:33:09: Producer got message: 35
13:33:09: Producer got message: 51
13:33:09: Consumer storing message: 41 (queue size = 0
13:33:09: Producer got message: 48
```

```
13:33:09: Producer got message: 71
13:33:09: Consumer storing message: 35 (queue size = 1
13:33:09: Producer got message: 81
13:33:09: Consumer storing message: 51 (queue size = 2
13:33:09: Producer got message: 38
13:33:09: Consumer storing message: 48 (queue size = 2
13:33:09: Producer got message: 47
13:33:09: Producer got message: 5
13:33:09: Consumer storing message: 71 (queue size = 2
13:33:09: Producer got message: 17
13:33:09: Consumer storing message: 81 (queue size = 3
13:33:09: Main: about to set event
13:33:09: Producer got message: 51
13:33:09: Consumer storing message: 38 (queue size = 3
13:33:09: Producer received EXIT event. Exiting....
13:33:09: Consumer storing message: 47 (queue size = 3
13:33:09: Consumer storing message: 5 (queue size = 2
13:33:09: Consumer storing message: 17 (queue size = 1
13:33:09: Consumer storing message: 51 (queue size = 0
13:33:09: Consumer received EXIT event. Exiting....
```

Right at the top, you can see the producer got to create 3 messages and place four of them on the queue. It got swapped out by the operating system before it could place the fifth one. The consumer then ran and pulled off the first message. It printed out that message as well as how deep the queue was at that point.

This is how you know that the second message hasn't made it into the pipeline yet.

The queue can hold ten messages, so the producer thread didn't get blocked by the queue. It was swapped out by the OS.

As the program starts to wrap up, can you see the main thread generating the event which causes the producer to exit immediately. The consumer still has a bunch of work do to, so it keeps running until it has cleaned out the pipeline.

Even slight changes to different queue sizes and calls to time.sleep() in the producer or the consumer to simulate longer network or disk access times will make large differences in your results.

This is a much better solution to the producer-consumer problem, but you can simplify it even more. The Pipeline really isn't needed for this problem. Once you take away the logging, it just becomes a queue.Queue.

```
import concurrent.futures
import logging
import queue
import random
import threading
import time

def producer(queue, event):
    """Pretend we're getting a number from the network."""
    while not event.is_set():
        message = random.randint(1, 101)
        logging.info("Producer got message: %s", message)
        queue.put(message)

    logging.info("Producer received event. Exiting")

def consumer(queue, event):
    """Pretend we're saving a number in the database."""
    while not event.is_set() or not queue.empty():
        message = queue.get()
        logging.info(
            "Consumer storing message: %s (size=%d)", message, queue.qsize()
        )

    logging.info("Consumer received event. Exiting")

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
```

```
                    datefmt="%H:%M:%S")

pipeline = queue.Queue(maxsize=10)
event = threading.Event()
with concurrent.futures.ThreadPoolExecutor(max_workers=2) as executor:
    executor.submit(producer, pipeline, event)
    executor.submit(consumer, pipeline, event)

    time.sleep(0.1)
    logging.info("Main: about to set event")
    event.set()
```

Threading Objects

Other primitives offered by threading

Semaphore

A Semaphore is a counter with a few special properties. The first one is that the counting is atomic. This means that there is a guarantee that the operating system will not swap out the thread in the middle of incrementing or decrementing the counter. The internal counter is incremented when you call .release() and decremented when you call .acquire(). The next special property is that if a thread calls .acquire() when the counter is zero, that thread will block until a different thread calls .release() and increments the counter to one. Semaphores are frequently used to protect a resource that has a limited capacity. An example would be if you have a pool of connections and want to limit the size of that pool to a specific number.

This means that the semaphore counts the global max num of resources available. Each thread acquiring a lock on a resource means that the count decreases, since a resource is being used up. Each release means that the count increases since a resource is freed up. When the count reaches o, this means that there are no more resources to acquire and the threads that want to acuire access must wait for another thread to release the resource.

Timer

A threading.Timer is a way to schedule a function to be called after a certain amount of time has passed. You create a Timer by passing in a number of seconds to wait and a function to call.

```
t = threading.Timer(30.0, my_function)
```

You start the Timer by calling .start(). The function will be called on a new thread at some point after the specified time, but be aware that there is no promise that it will be called exactly at the time you want. If you want to stop a Timer that you've already started, you can cancel it by calling .cancel(). Calling .cancel() after the Timer has triggered does nothing and does not produce an exception. A Timer can be used to prompt a user for action after a specific amount of time. If the user does the action before the Timer expires, .cancel() can be called.

Barrier

A threading.Barrier can be used to keep a fixed number of threads in sync. When creating a Barrier, the caller must specify how many threads will be synchronizing on it. Each thread calls .wait() on the Barrier. They all will remain blocked until the specified number of threads are waiting, and then the are all released at the same time. Remember that threads are scheduled by the operating system so, even though all of the threads are released simultaneously, they will be scheduled to run one at a time. One use for a Barrier is to allow a pool of threads to initialize themselves. Having the threads wait on a Barrier after they are initialized will ensure that none of the threads start running before all of the threads are finished with their initialization.

This means that a barrier is a way to block or bar program execution until something has already happened for all threads.

Python Decorators—Real Python

Heading

Some text

References

There are no sources in the current document.