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, zip(*iterables)),

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.

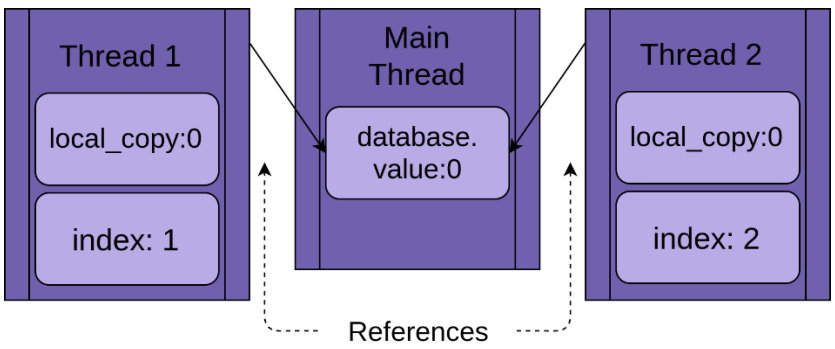
```python
import logging, threading, time
import concurrent.futures
class FakeDatabase:
    def __init__(self):
        self.value = 0

    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

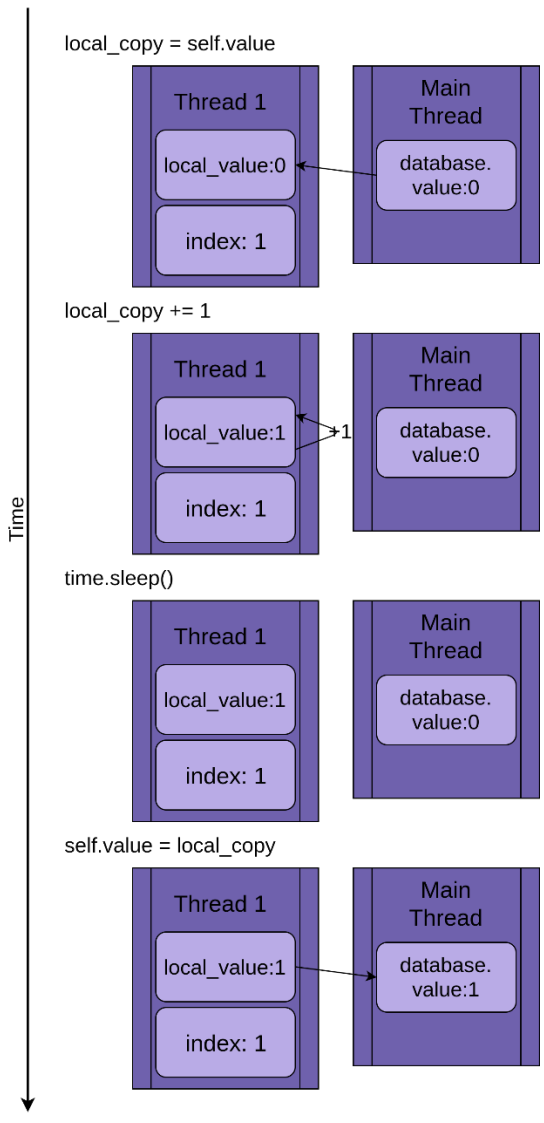
In the usage above, index is passed as the first and only positional argument to database.update(). Since each thread runs .update(), and .update() adds one to .value, you might expect database.value to be 2 when it's printed out at the end.

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.

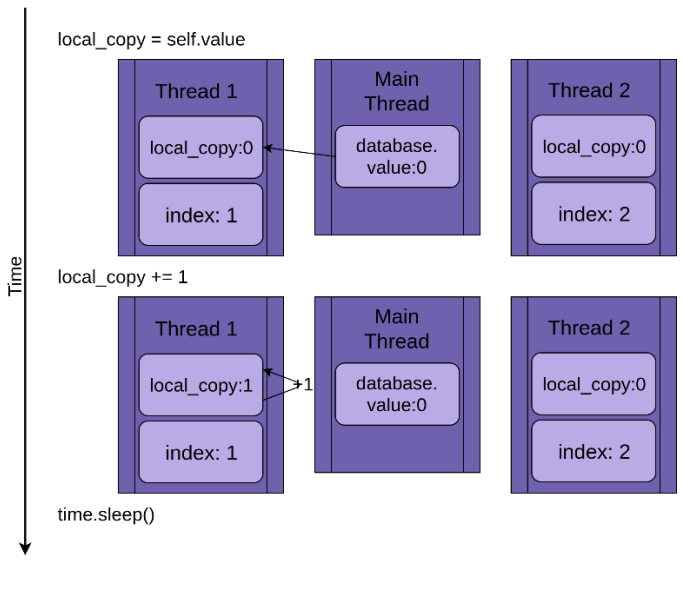
The image below steps through the execution of .update() if only a single thread is run. The statement is shown on the left followed by a diagram showing the values in the thread's local_value and the shared database.value.



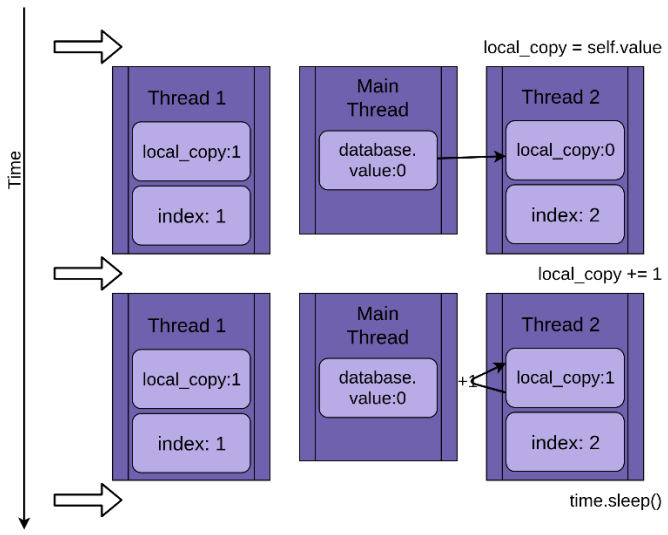
The diagram is laid out so that time increases as you move from top to bottom. It begins when Thread 1 is created and ends when it is terminated. When Thread 1 starts, FakeDatabase.value is zero. The first line of code in the method, local_copy = self.value, copies the value zero to the local variable.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

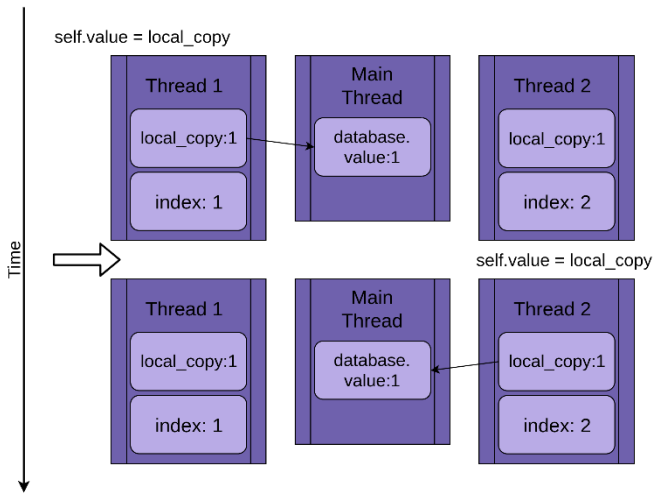
Getting back to the race condition, the two threads will be running concurrently but not at the same time. They will each have their own version of local_copy and will each point to the same database. It is this shared database object that is going to cause the problems. The program starts with Thread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

```
8     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):
2     """Pretend we're saving a number in the database."""
3     while not event.is_set() or not pipeline.empty():
4         message = pipeline.get_message("Consumer")
5         logging.info(
6             "Consumer storing message: %s  (queue size=%s)",
7             message,
8             pipeline.qsize(),
9         )
10
11    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):
2     def __init__(self):
3         super().__init__(maxsize=10)
4
5     def get_message(self, name):
6         logging.debug("%s:about to get from queue", name)
7         value = self.get()
8         logging.debug("%s:got %d from queue", name, value)
9         return value
10
11    def set_message(self, value, name):
12        logging.debug("%s:about to add %d to queue", name, value)
13        self.put(value)
14        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

Decorators provide a simple syntax for calling higher-order functions. By definition, a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.

Linked resources: "pie" syntax, Type introspection, Data Classes, Use of class decorators, Python Meta classes, functools.partial, emulating callable objects, python-is-identity-vs-equality, Memoization, Cache computing, function annotations, decorator module docs, decorator module, PythonDecoratorLibrary, PythonDecorators,

(python.org, "pie" syntax), (Wikipedia, Introspection), (Python, Dataclasses), (python.org, PEP-3129), (Python, Python Meta Classes), (python.org, functools.partial), (python.org, emulating-callable-objects), (Python, python-is-identity-vs-equality), (Wikipedia, Memoization), (python.org, PEP-3107),

(python.org, PythonDecorators), (python.org, PythonDecoratorLibrary), (Decorator module docs)

First-Class Objects

A function returns a value based on the given arguments. In Python, functions are first-class objects. This means that functions can be passed around and used as arguments, just like any other object (string, int, float, list, and so on).

```
def say_hello(name):
    return f"Hello {name}"


def be_awesome(name):
    return f"Yo {name}, together we are the awesomest!"


def greet_bob(greeter_func):
    return greeter_func("Bob")
>>> greet_bob(say_hello)
'Hello Bob'


>>> greet_bob(be_awesome)
'Yo Bob, together we are the awesomest!'
```

Here, say_hello() and be_awesome() are regular functions that expect a name given as a string. The greet_bob() function however, expects a function as its argument. Note that greet_bob(say_hello) refers to two functions, but in different ways: greet_bob() and say_hello. The say_hello function is named without parentheses. This means that only a reference to the function is passed. The function is not executed. The greet_bob() function, on the other hand, is written with parentheses, so it will be called as usual.

Inner Functions

It's possible to define functions inside other functions. Such functions are called inner functions. Functions are just another block of code, and all other objects can be defined and executed inside them.

```
def parent():
    print("Printing from the parent() function")


    def first_child():
        print("Printing from the first_child() function")


    def second_child():
        print("Printing from the second_child() function")


    second_child()
    first_child()
>>> parent()
Printing from the parent() function
Printing from the second_child() function
Printing from the first_child() function
```

Whenever you call parent(), the inner functions first_child() and second_child() are also called. But because of their local scope, they aren't available outside of the parent() function.

Note that the order in which the inner functions are defined does not matter. Like with any other functions, the printing only happens when the inner functions are executed. Furthermore, the inner functions are not defined until the parent function is called. They are locally scoped to parent(): they only exist inside the parent() function as local variables.

Returning Functions From Functions

Python also allows you to use functions as return values.

```
def parent(num):
    def first_child():
```

```
        return "Hi, I am Emma"

    def second_child():
        return "Call me Liam"

    if num == 1:
        return first_child
    else:
        return second_child
>>> first = parent(1)
>>> second = parent(2)

>>> first
<function parent.<locals>.first_child at 0x7f599f1e2e18>

>>> second
<function parent.<locals>.second_child at 0x7f599dad5268>
>>> first()
'Hi, I am Emma'

>>> second()
'Call me Liam'
```

Note that you are returning first_child without the parentheses. Recall that this means that you are returning a reference to the function first_child. In contrast first_child() with parentheses refers to the result of evaluating the function.

The somewhat cryptic output simply means that the first variable refers to the local first_child() function inside of parent(), while second points to second_child(). You can now use first and second as if they are regular functions, even though the functions they point to can't be accessed directly

Simple Decorators

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

def say_whee():
    print("Whee!")

say_whee = my_decorator(say_whee)
>>> say_whee()
Something is happening before the function is called.
Whee!
Something is happening after the function is called.
>>> say_whee
<function my_decorator.<locals>.wrapper at 0x7f3c5dfd42f0>
```

In effect, the name say_whee now points to the wrapper() inner function. Remember that you return wrapper as a function when you call my_decorator(say_whee). However, wrapper() has a reference to the original say_whee() as func, and calls that function between the two calls to print(). Put simply: decorators wrap a function, modifying its behavior.

Because wrapper() is a regular Python function, the way a decorator modifies a function can change dynamically.

```
from datetime import datetime

def not_during_the_night(func):
    def wrapper():
        if 7 <= datetime.now().hour < 22:
            func()
        else:
```

```
            pass  # Hush, the neighbors are asleep
    return wrapper

def say_whee():
    print("Whee!")

say_whee = not_during_the_night(say_whee)
>>> say_whee()
>>>
```

If you try to call say_whee() after bedtime, nothing will happen.

Syntactic Sugar!

say_whee() above is a little clunky. First of all, you end up typing the name say_whee three times. In addition, the decoration gets a bit hidden away below the definition of the function.

Instead, Python allows you to use decorators in a simpler way with the @ symbol, sometimes called the "pie" syntax.

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@my_decorator
def say_whee():
    print("Whee!")
```

So, @my_decorator is just an easier way of saying say_whee = my_decorator(say_whee). It's how you apply a decorator to a function.

Reusing Decorators

A decorator is just a regular Python function. All the usual tools for easy reusability are available. They can be moved to their own module, so that they can be used with any other function.

```
def do_twice(func):
    def wrapper_do_twice():
        func()
        func()
    return wrapper_do_twice
```

Create a file called decorators.py and save the above. You can now use this new decorator in other files by doing a regular import. When you run this example, you should see that the original say_whee() is executed twice.

```
from decorators import do_twice

@do_twice
def say_whee():
    print("Whee!")
>>> say_whee()
Whee!
Whee!
```

You can name your inner function whatever you want, and a generic name like wrapper() is usually okay.

Decorating Functions With Arguments

To decorate a function that accepts arguments:

```
from decorators import do_twice

@do_twice
def greet(name):
    print(f"Hello {name}")
```

This might give an error without changes to the decorator function wrapper.

```
>>> greet("World")
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
TypeError: wrapper_do_twice() takes 0 positional arguments but 1 was given
```

The problem is that the inner function wrapper_do_twice() does not take any arguments, but name="World" was passed to it. You could fix this by letting wrapper_do_twice() accept one argument, but then it would not work for the say_whee() function you created earlier. The solution is to use *args and **kwargs in the inner wrapper function. Then it will accept an arbitrary number of positional and keyword arguments. The wrapper_do_twice() inner function now accepts any number of arguments and passes them on to the function it decorates.

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        func(*args, **kwargs)
    return wrapper_do_twice
>>> say_whee()
Whee!
Whee!

>>> greet("World")
Hello World
Hello World
```

Returning Values From Decorated Functions

To return a value from a decorated function, you need to make sure the wrapper function returns the return value of the decorated function.

```
def do_twice(func):
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
from decorators import do_twice

@do_twice
def return_greeting(name):
    print("Creating greeting")
    return f"Hi {name}"

>>> return_greeting("Adam")
Creating greeting
Creating greeting
'Hi Adam'
```

Inside the wrapper function, the second function call's value is returned. To return both, return both function calls.

Who Are You, Really?

A great convenience when working with Python, especially in the interactive shell, is its powerful introspection ability. Introspection is the ability of an object to know about its own attributes at runtime. For instance, a function knows its own name and documentation

(Wikipedia, Introspection)

The introspection works for functions you define yourself as well.

```
>>> say_whee
<function do_twice.<locals>.wrapper_do_twice at 0x7f43700e52f0>

>>> say_whee.__name__
'wrapper_do_twice'

>>> help(say_whee)
Help on function wrapper_do_twice in module decorators:

wrapper_do_twice()
```

It now reports being the wrapper_do_twice() inner function inside the do_twice() decorator. Although technically true, this is not very useful information.

To fix this, decorators should use the @functools.wraps decorator, which will preserve information about the original function.

In decorators.py:

```
import functools

def do_twice(func):
    @functools.wraps(func)
    def wrapper_do_twice(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return wrapper_do_twice
>>> say_whee
<function say_whee at 0x7ff79a60f2f0>

>>> say_whee.__name__
'say_whee'

>>> help(say_whee)
Help on function say_whee in module whee:

say_whee()
```

decorators.py:

```
from functools import wraps

def do_twice(func):
    @wraps(func)
    def wrapper_do_twice(*args, **kwargs):
        return func(*args, **kwargs), func(*args, **kwargs)
    return wrapper_do_twice
```

```
from decorators import do_twice

@do_twice
def return_greting(name):
    print("Creaing greeting")
    return f"Hi {name}"

print(return_greting("Kitty"))
```

A Few Real World Examples

```
import functools

def decorator(func):
    @functools.wraps(func)
    def wrapper_decorator(*args, **kwargs):
        # Do something before
        value = func(*args, **kwargs)
        # Do something after
        return value
    return wrapper_decorator
```

This formula is a good boilerplate template for building more complex decorators.

Timing Functions

Let's create a @timer decorator. It will measure the time a function takes to execute and print the duration to the console.

```
import functools
import time


def timer(func):
    """Print the runtime of the decorated function"""
```

```
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1
        value = func(*args, **kwargs)
        end_time = time.perf_counter()      # 2
        run_time = end_time - start_time    # 3
        print(f"Finished {func.__name__!r} in {run_time:.4f} secs")
        return value
    return wrapper_timer

@timer
def waste_some_time(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])
>>> waste_some_time(1)
Finished 'waste_some_time' in 0.0010 secs

>>> waste_some_time(999)
Finished 'waste_some_time' in 0.3260 secs
```

This decorator works by storing the time just before the function starts running (at the line marked # 1) and just after the function finishes (at # 2). The time the function takes is then the difference between the two (at # 3). We use the time.perf_counter() function, which does a good job of measuring time intervals.

The @timer decorator is great if you just want to get an idea about the runtime of your functions. If you want to do more precise measurements of code, you should instead consider the timeit module in the standard library. It temporarily disables garbage collection and runs multiple trials to strip out noise from quick function calls.

Debugging Code

The @debug decorator will print the arguments a function is called with as well as its return value every time the function is called.

```
import functools

def debug(func):
    """Print the function signature and return value"""
    @functools.wraps(func)
    def wrapper_debug(*args, **kwargs):
        args_repr = [repr(a) for a in args]                      # 1
        kwargs_repr = [f"{k}={v!r}" for k, v in kwargs.items()]  # 2
        signature = ", ".join(args_repr + kwargs_repr)           # 3
        print(f"Calling {func.__name__}({signature})")
        value = func(*args, **kwargs)
        print(f"{func.__name__!r} returned {value!r}")           # 4
        return value
    return wrapper_debug
@debug
def make_greeting(name, age=None):
    if age is None:
        return f"Howdy {name}!"
    else:
        return f"Whoa {name}! {age} already, you are growing up!"
>>> make_greeting("Benjamin")
Calling make_greeting('Benjamin')
'make_greeting' returned 'Howdy Benjamin!'
'Howdy Benjamin!'

>>> make_greeting("Richard", age=112)
Calling make_greeting('Richard', age=112)
'make_greeting' returned 'Whoa Richard! 112 already, you are growing up!'
'Whoa Richard! 112 already, you are growing up!'
```

```
>>> make_greeting(name="Dorrisile", age=116)
Calling make_greeting(name='Dorrisile', age=116)
'make_greeting' returned 'Whoa Dorrisile! 116 already, you are growing up!'
'Whoa Dorrisile! 116 already, you are growing up!'
```

The signature is created by joining the string representations of all the arguments. The numbers in the following list correspond to the numbered comments in the code.

1. Create a list of the positional arguments. Use repr() to get a nice string representing each argument.
2. Create a list of the keyword arguments. The f-string formats each argument as key=value where the !r specifier means that repr() is used to represent the value.
3. The lists of positional and keyword arguments is joined together to one signature string with each argument separated by a comma.
4. The return value is printed after the function is executed.

The @debug decorator just repeats what you just wrote. It's more powerful when applied to small convenience functions that you don't call directly yourself.

The following example calculates an approximation to the mathematical constant e:

```
import math
from decorators import debug

# Apply a decorator to a standard library function
math.factorial = debug(math.factorial)

def approximate_e(terms=18):
    return sum(1 / math.factorial(n) for n in range(terms))
>>> approximate_e(5)
Calling factorial(0)
'factorial' returned 1
Calling factorial(1)
'factorial' returned 1
Calling factorial(2)
'factorial' returned 2
Calling factorial(3)
'factorial' returned 6
Calling factorial(4)
'factorial' returned 24
2.708333333333333
```

This example might not seem immediately useful since the @debug decorator just repeats what you just wrote. It's more powerful when applied to small convenience functions that you don't call directly yourself.

The following example calculates an approximation to the mathematical constant e

```
import math
from decorators import debug

# Apply a decorator to a standard library function
math.factorial = debug(math.factorial)

def approximate_e(terms=18):
    return sum(1 / math.factorial(n) for n in range(terms))
>>> approximate_e(5)
Calling factorial(0)
'factorial' returned 1
Calling factorial(1)
'factorial' returned 1
Calling factorial(2)
'factorial' returned 2
Calling factorial(3)
'factorial' returned 6
```

```
Calling factorial(4)
'factorial' returned 24
2.708333333333333
```

Slowing Down Code

Probably the most common use case is that you want to rate-limit a function that continuously checks whether a resource—like a web page—has changed. The @slow_down decorator will sleep one second before it calls the decorated function

```
import functools
import time

def slow_down(func):
    """Sleep 1 second before calling the function"""
    @functools.wraps(func)
    def wrapper_slow_down(*args, **kwargs):
        time.sleep(1)
        return func(*args, **kwargs)
    return wrapper_slow_down

@slow_down
def countdown(from_number):
    if from_number < 1:
        print("Liftoff!")
    else:
        print(from_number)
        countdown(from_number - 1)
>>> countdown(3)
3
2
1
Liftoff!
```

The countdown() function is a recursive function. In other words, it's a function calling itself. The @slow_down decorator always sleeps for one second.

Registering Plugins

Decorators don't have to wrap the function they're decorating. They can also simply register that a function exists and return it unwrapped. This can be used, for instance, to create a light-weight plug-in architecture.

```
import random
PLUGINS = dict()

def register(func):
    """Register a function as a plug-in"""
    PLUGINS[func.__name__] = func
    return func

@register
def say_hello(name):
    return f"Hello {name}"

@register
def be_awesome(name):
    return f"Yo {name}, together we are the awesomest!"

def randomly_greet(name):
    greeter, greeter_func = random.choice(list(PLUGINS.items()))
    print(f"Using {greeter!r}")
    return greeter_func(name)
>>> PLUGINS
{'say_hello': <function say_hello at 0x7f768eae6730>,
```

```
 'be_awesome': <function be_awesome at 0x7f768eae67b8>}

>>> randomly_greet("Alice")
Using 'say_hello'
'Hello Alice'
>>> globals()
{..., # Lots of variables not shown here.
 'say_hello': <function say_hello at 0x7f768eae6730>,
 'be_awesome': <function be_awesome at 0x7f768eae67b8>,
 'randomly_greet': <function randomly_greet at 0x7f768eae6840>}
```

The @register decorator simply stores a reference to the decorated function in the global PLUGINS dict. Note that you do not have to write an inner function or use @functools.wraps in this example because you are returning the original function unmodified. The randomly_greet() function randomly chooses one of the registered functions to use. Note that the PLUGINS dictionary already contains references to each function object that is registered as a plugin.

The main benefit of this simple plugin architecture is that you do not need to maintain a list of which plugins exist. That list is created when the plugins register themselves. This makes it trivial to add a new plugin: just define the function and decorate it with @register.

If you are familiar with globals() in Python, you might see some similarities to how the plugin architecture works. globals() gives access to all global variables in the current scope, including your plugins. Using the @register decorator, you can create your own curated list of interesting variables, effectively hand-picking some functions from globals().

Is the User Logged In?

Commonly used when working with a web framework. In this example, we are using Flask to set up a /secret web page that should only be visible to users that are logged in or otherwise authenticated.

```
from flask import Flask, g, request, redirect, url_for
import functools
app = Flask(__name__)

def login_required(func):
    """Make sure user is logged in before proceeding"""
    @functools.wraps(func)
    def wrapper_login_required(*args, **kwargs):
        if g.user is None:
            return redirect(url_for("login", next=request.url))
        return func(*args, **kwargs)
    return wrapper_login_required

@app.route("/secret")
@login_required
def secret():
    ...
```

While this gives an idea about how to add authentication to your web framework, you should usually not write these types of decorators yourself. For Flask, you can use the Flask-Login extension instead, which adds more security and functionality.

```
from functools import wraps
import time
from flask import g, request, redirect, url_for

def do_twice(func):
    @wraps(func)
    def wrapper_do_twice(*args, **kwargs):
        return func(*args, **kwargs), func(*args, **kwargs)
    return wrapper_do_twice

def timer(func):
    """
    Print the runtimeof the decorated function
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        start_time = time.perf_counter()
        value  = func(*args, **kwargs)
```

```
        end_time = time.perf_counter()
        taken = end_time - start_time
        run_time = f"Finished {func.__name__!r} in {taken: .4f} secs"
        print(run_time)
        return value
    return wrapper

def debug(func):
    """
    Print the function signature and return value
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        args_repr = [repr(a) for a in args]
        kwargs_repr = [f"{k} = {v!r}" for k,v in kwargs.items()]
        signature = ", ".join(args_repr+kwargs_repr)
        print(f"Calling {func.__name__}({signature})")
        value  = func(*args, **kwargs)
        print(f"{func.__name__}returned {value!r}")
        return value
    return wrapper

def slow_down(func):
    """
    Slows down a function, by 1 second of sleep
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        time.sleep(1)
        return func(*args, **kwargs)
    return wrapper

def login_required(func):
    """
    Make sure user is logged in before proceeding
    """
    @wraps(func)
    def wrapper(*args, **kwargs):
        if g.user is None:
            return redirect(url_for("login", next=request.url))
        return func(*args, **kwargs)
    return wrapper
```

Fancy Decorators

Decorating Classes

There are two different ways you can use decorators on classes. The first one is very close to what you have already done with functions: you can decorate the methods of a class. This was one of the motivations for introducing decorators back in the day.

Some commonly used decorators that are even built-ins in Python are @classmethod, @staticmethod, and @property. The @classmethod and @staticmethod decorators are used to define methods inside a class namespace that are not connected to a particular instance of that class. The @property decorator is used to customize getters and setters for class attributes.

The following definition of a Circle class uses the @classmethod, @staticmethod, and @property decorators:

```
class Circle:
    def __init__(self, radius):
        self._radius = radius

    @property
    def radius(self):
        """Get value of radius"""
        return self._radius

    @radius.setter
    def radius(self, value):
        """Set radius, raise error if negative"""
        if value >= 0:
            self._radius = value
        else:
            raise ValueError("Radius must be positive")

    @property
    def area(self):
        """Calculate area inside circle"""
        return self.pi() * self.radius**2

    def cylinder_volume(self, height):
        """Calculate volume of cylinder with circle as base"""
        return self.area * height

    @classmethod
    def unit_circle(cls):
        """Factory method creating a circle with radius 1"""
        return cls(1)

    @staticmethod
    def pi():
        """Value of π, could use math.pi instead though"""
        return 3.1415926535
>>> c = Circle(5)
>>> c.radius
5

>>> c.area
78.5398163375

>>> c.radius = 2
>>> c.area
12.566370614

>>> c.area = 100
AttributeError: can't set attribute
```

```
>>> c.cylinder_volume(height=4)
50.265482456

>>> c.radius = -1
ValueError: Radius must be positive

>>> c = Circle.unit_circle()
>>> c.radius
1

>>> c.pi()
3.1415926535

>>> Circle.pi()
3.1415926535
```

In this class:

- .cylinder_volume() is a regular method.
- .radius is a mutable property: it can be set to a different value. However, by defining a setter method, we can do some error testing to make sure it's not set to a nonsensical negative number. Properties are accessed as attributes without parentheses.
- .area is an immutable property: properties without .setter() methods can't be changed. Even though it is defined as a method, it can be retrieved as an attribute without parentheses.
- .unit_circle() is a class method. It's not bound to one particular instance of Circle. Class methods are often used as factory methods that can create specific instances of the class.
- .pi() is a static method. It's not really dependent on the Circle class, except that it is part of its namespace. Static methods can be called on either an instance or the class.

Here is a class where we decorate some of its methods using the @debug and @timer decorators from earlier:

```
from decorators import debug, timer


class TimeWaster:
    @debug
    def __init__(self, max_num):
        self.max_num = max_num

    @timer
    def waste_time(self, num_times):
        for _ in range(num_times):
            sum([i**2 for i in range(self.max_num)])
>>> tw = TimeWaster(1000)
Calling __init__(<time_waster.TimeWaster object at 0x7efccce03908>, 1000)
'__init__' returned None

>>> tw.waste_time(999)
Finished 'waste_time' in 0.3376 secs
```

The other way to use decorators on classes is to decorate the whole class. This is, for example, done in the new dataclasses module in Python 3.7:

```
from dataclasses import dataclass


@dataclass
class PlayingCard:
    rank: str
    suit: str
```

The meaning of the syntax is similar to the function decorators. In the example above, you could have done the decoration by writing PlayingCard = dataclass(PlayingCard).

A common use of class decorators is to be a simpler alternative to some use-cases of metaclasses. In both cases, you are changing the definition of a class dynamically. Writing a class decorator is very similar to writing a function decorator. The only difference is that the decorator will receive a class and not a function as an argument. In fact, all the decorators you saw above will work as class decorators.

Nesting Decorators

You can apply several decorators to a function by stacking them on top of each other. Think about this as the decorators being executed in the order they are listed.

```
from decorators import debug, do_twice


@debug
@do_twice
def greet(name):
    print(f"Hello {name}")
>>> greet("Eva")
Calling greet('Eva')
Hello Eva
Hello Eva
'greet' returned None
```

In other words, @debug calls @do_twice, which calls greet(), or debug(do_twice(greet())):

If we change the order of @debug and @do_twice, @do_twice will be applied to @debug as well:

```
from decorators import debug, do_twice


@do_twice
@debug
def greet(name):
    print(f"Hello {name}")
>>> greet("Eva")
Calling greet('Eva')
Hello Eva
'greet' returned None
Calling greet('Eva')
Hello Eva
'greet' returned None
```

Decorators With Arguments

Sometimes, it's useful to pass arguments to your decorators. For instance, @do_twice could be extended to a @repeat(num_times) decorator. The number of times to execute the decorated function could then be given as an argument. So far, the name written after the @ has referred to a function object that can be called with another function. To be consistent, you then need repeat(num_times=4) to return a function object that can act as a decorator.

Typically, the decorator creates and returns an inner wrapper function, so writing the example out in full will give you an inner function within an inner function.

```
def repeat(num_times):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper_repeat(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper_repeat
    return decorator_repeat
@repeat(num_times=4)
def greet(name):
    print(f"Hello {name}")
>>> greet("World")
Hello World
Hello World
Hello World
Hello World
```

This wrapper_repeat() function takes arbitrary arguments and returns the value of the decorated function, func(). This wrapper function also contains the loop that calls the decorated function num_times times. This is no different from the earlier wrapper functions you have seen, except that it is using the

num_times parameter that must be supplied from the outside. Again, decorator_repeat() looks exactly like the decorator functions you have written earlier, except that it's named differently. That's because we reserve the base name—repeat()—for the outermost function, which is the one the user will call.

There are a few subtle things happening in the repeat() function:

- Defining decorator_repeat() as an inner function means that repeat() will refer to a function object—decorator_repeat. Earlier, we used repeat without parentheses to refer to the function object. The added parentheses are necessary when defining decorators that take arguments.
- The num_times argument is seemingly not used in repeat() itself. But by passing num_times a closure is created where the value of num_times is stored until it will be used later by wrapper_repeat().

The inner function is passed the func object automatically

Both Please, But Never Mind the Bread

You can also define decorators that can be used both with and without arguments.

Since the function to decorate is only passed in directly if the decorator is called without arguments, the function must be an optional argument. This means that the decorator arguments must all be specified by keyword. You can enforce this with the special * syntax, which means that all following parameters are keyword-only.

```
def name(_func=None, *, kw1=val1, kw2=val2, ...):  # 1
    def decorator_name(func):
        ...  # Create and return a wrapper function.

    if _func is None:
        return decorator_name                    # 2
    else:
        return decorator_name(_func)             # 3
```

Here, the _func argument acts as a marker, noting whether the decorator has been called with arguments or not/

- If name has been called without arguments, the decorated function will be passed in as _func. If it has been called with arguments, then _func will be None, and some of the keyword arguments may have been changed from their default values. The * in the argument list means that the remaining arguments can't be called as positional arguments.
- In this case, the decorator was called with arguments. Return a decorator function that can read and return a function.
- In this case, the decorator was called without arguments. Apply the decorator to the function immediately.

Using this boilerplate on the @repeat decorator:

```
def repeat(_func=None, *, num_times=2):
    def decorator_repeat(func):
        @functools.wraps(func)
        def wrapper_repeat(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper_repeat

    if _func is None:
        return decorator_repeat
    else:
        return decorator_repeat(_func)
@repeat
def say_whee():
    print("Whee!")

@repeat(num_times=3)
def greet(name):
    print(f"Hello {name}")
>>> say_whee()
Whee!
Whee!

>>> greet("Penny")
Hello Penny
Hello Penny
Hello Penny
```

The only changes are the added _func parameter and the if-else at the end.

Stateful Decorators

It's useful to have a decorator that can keep track of state.

```
import functools

def count_calls(func):
    @functools.wraps(func)
    def wrapper_count_calls(*args, **kwargs):
        wrapper_count_calls.num_calls += 1
        print(f"Call {wrapper_count_calls.num_calls} of {func.__name__!r}")
        return func(*args, **kwargs)
    wrapper_count_calls.num_calls = 0
    return wrapper_count_calls

@count_calls
def say_whee():
    print("Whee!")
Call 1 of 'say_whee'
Whee!

>>> say_whee()
Call 2 of 'say_whee'
Whee!

>>> say_whee.num_calls
2
```

A decorator that counts the number of times a function is called. The state—the number of calls to the function—is stored in the function attribute .num_calls on the wrapper function.

Here, we need to remember that this is what happens when we decorate a function with @decorator

@decorator

func()

Is func = decorator(func)

This means that a .num_calls = 0 is returned, along with the logic to increment it. Then when it is called, the attribute is incremented by 1. Decorating it again will reset .num_calls to 0.

Classes as Decorators

The typical way to maintain state is by using classes.

Recall that the decorator syntax @my_decorator is just an easier way of saying func = my_decorator(func). Therefore, if my_decorator is a class, it needs to take func as an argument in its .__init__() method. Furthermore, the class needs to be callable so that it can stand in for the decorated function. For a class to be callable, you implement the special .__call__() method:

```
class Counter:
    def __init__(self, start=0):
        self.count = start

    def __call__(self):
        self.count += 1
        print(f"Current count is {self.count}")
>>> counter = Counter()
>>> counter()
Current count is 1

>>> counter()
Current count is 2

>>> counter.count
2
import functools
```

```
class CountCalls:
    def __init__(self, func):
        functools.update_wrapper(self, func)
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
        print(f"Call {self.num_calls} of {self.func.__name__!r}")
        return self.func(*args, **kwargs)

@CountCalls
def say_whee():
    print("Whee!")
>>> say_whee()
Call 1 of 'say_whee'
Whee!


>>> say_whee()
Call 2 of 'say_whee'
Whee!


>>> say_whee.num_calls
2
```

The .__init__() method must store a reference to the function and can do any other necessary initialization. The .__call__() method will be called instead of the decorated function. It does essentially the same thing as the wrapper() function in our earlier examples. Note that you need to use the functools.update_wrapper() function instead of @functools.wraps. This @CountCalls decorator works the same as the one in the previous section. Therefore, a typical implementation of a decorator class needs to implement .__init__() and .__call__()

```
def repeat(_func=None, *, num_times=2):
    def decorator_repeat(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            for _ in range(num_times):
                value = func(*args, **kwargs)
            return value
        return wrapper
    if _func is None:
        return decorator_repeat
    else:
        return decorator_repeat(_func)

def count_calls(func):
    @wraps(func)
    def wrapper(*args, **kwargs):
        wrapper.count_calls += 1
        print(f"Call {wrapper.count_calls} of {func.__name__!r}")
        return func(*args, **kwargs)
    wrapper.count_calls = 0
    return wrapper

class Counter:
    def __init__(self, start=0):
        self.count = start

    def __call__(self):
        self.count += 1
        print(f"Current calls count is {self.count}")

class CountCalls:
    def __init__(self, func):
        self.func = func
        self.num_calls = 0

    def __call__(self, *args, **kwargs):
        self.num_calls += 1
```

```
        print(f"Call {self.num_calls} of {self.func.__name__!r}")
        return self.func(*args, **kwargs)
```

More Real World Examples

Slowing Down Code, Revisited

Our previous implementation of @slow_down always sleeps for one second. Let's rewrite @slow_down using an optional rate argument that controls how long it sleeps:

```
import functools
import time

def slow_down(_func=None, *, rate=1):
    """Sleep given amount of seconds before calling the function"""
    def decorator_slow_down(func):
        @functools.wraps(func)
        def wrapper_slow_down(*args, **kwargs):
            time.sleep(rate)
            return func(*args, **kwargs)
        return wrapper_slow_down

    if _func is None:
        return decorator_slow_down
    else:
        return decorator_slow_down(_func)
@slow_down(rate=2)
def countdown(from_number):
    if from_number < 1:
        print("Liftoff!")
    else:
        print(from_number)
        countdown(from_number - 1)
>>> countdown(3)
3
2
1
Liftoff!
```

The same recursive countdown() function as earlier now sleeps two seconds between each count.

Creating Singletons

A singleton is a class with only one instance. There are several singletons in Python that you use frequently, including None, True, and False. It is the fact that None is a singleton that allows you to compare for None using the is keyword. Using is returns True only for objects that are the exact same instance. The @singleton decorator turns a class into a singleton by storing the first instance of the class as an attribute. Later attempts at creating an instance simply return the stored instance.

```
import functools

def singleton(cls):
    """Make a class a Singleton class (only one instance)"""
    @functools.wraps(cls)
    def wrapper_singleton(*args, **kwargs):
        if not wrapper_singleton.instance:
            wrapper_singleton.instance = cls(*args, **kwargs)
        return wrapper_singleton.instance
    wrapper_singleton.instance = None
    return wrapper_singleton

@singleton
class TheOne:
    pass
```

```
>>> first_one = TheOne()
>>> another_one = TheOne()

>>> id(first_one)
140094218762280

>>> id(another_one)
140094218762280

>>> first_one is another_one
True
```

This class decorator follows the same template as our function decorators. The only difference is that we are using cls instead of func as the parameter name to indicate that it is meant to be a class decorator. It seems clear that first_one is indeed the exact same instance as another_one.

Singleton classes are not really used as often in Python as in other languages. The effect of a singleton is usually better implemented as a global variable in a module.

Caching Return Values

Decorators can provide a nice mechanism for caching and memoization.

The usual solution is to implement Fibonacci numbers using a for loop and a lookup table. However, simple caching of the calculations will also do the trick.

```
import functools
from decorators import count_calls

def cache(func):
    """Keep a cache of previous function calls"""
    @functools.wraps(func)
    def wrapper_cache(*args, **kwargs):
        cache_key = args + tuple(kwargs.items())
        if cache_key not in wrapper_cache.cache:
            wrapper_cache.cache[cache_key] = func(*args, **kwargs)
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = dict()
    return wrapper_cache

@cache
@count_calls
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
>>> fibonacci(10)
Call 1 of 'fibonacci'
...
Call 11 of 'fibonacci'
55

>>> fibonacci(8)
21
import functools

@functools.lru_cache(maxsize=4)
def fibonacci(num):
    print(f"Calculating fibonacci({num})")
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

In the standard library, a Least Recently Used (LRU) cache is available as @functools.lru_cache.

This decorator has more features than the one you saw above. You should use @functools.lru_cache instead of writing your own cache decoratorThe maxsize parameter specifies how many recent calls are cached. The default value is 128, but you can specify maxsize=None to cache all function calls. However, be aware that this can cause memory problems if you are caching many large objects.

You can use the .cache_info() method to see how the cache performs, and you can tune it if needed. In our example, we used an artificially small maxsize to see the effect of elements being removed from the cache The cache works as a lookup table, so now fibonacci() only does the necessary calculations once.

Adding Information About Units

The following does not really change the behavior of the decorated function. Instead, it simply adds unit as a function attribute. The following example calculates the volume of a cylinder based on its radius and height in centimeters. This .unit function attribute can later be accessed when needed.

```
def set_unit(unit):
    """Register a unit on a function"""
    def decorator_set_unit(func):
        func.unit = unit
        return func
    return decorator_set_unit
import math

@set_unit("cm^3")
def volume(radius, height):
    return math.pi * radius**2 * height
>>> volume(3, 5)
141.3716694115407


>>> volume.unit
'cm^3'
```

Note that you could have achieved something similar using function annotations. However, since annotations are used for type hints, it would be hard to combine such units as annotations with static type checking.

Units become even more powerful and fun when connected with a library that can convert between units. One such library is pint. With pint installed (pip install Pint), you can for instance convert the volume to cubic inches or gallons. You could also modify the decorator to return a pint Quantity directly. Such a Quantity is made by multiplying a value with the unit. In pint, units must be looked up in a UnitRegistry. The registry is stored as a function attribute to avoid cluttering the namespace

```
>>> import pint
>>> ureg = pint.UnitRegistry()
>>> vol = volume(3, 5) * ureg(volume.unit)


>>> vol
<Quantity(141.3716694115407, 'centimeter ** 3')>


>>> vol.to("cubic inches")
<Quantity(8.627028576414954, 'inch ** 3')>

>>> vol.to("gallons").m  # Magnitude
0.0373464440537444
def use_unit(unit):
    """Have a function return a Quantity with given unit"""
    use_unit.ureg = pint.UnitRegistry()
    def decorator_use_unit(func):
        @functools.wraps(func)
        def wrapper_use_unit(*args, **kwargs):
            value = func(*args, **kwargs)
            return value * use_unit.ureg(unit)
        return wrapper_use_unit
    return decorator_use_unit

@use_unit("meters per second")
def average_speed(distance, duration):
    return distance / duration
```

```
>>> bolt = average_speed(100, 9.58)
>>> bolt
<Quantity(10.438413361169102, 'meter / second')>

>>> bolt.to("km per hour")
<Quantity(37.578288100208766, 'kilometer / hour')>

>>> bolt.to("mph").m  # Magnitude
23.350065679064745
```

Validating JSON

Take a quick look at the following Flask route handler:

```
@app.route("/grade", methods=["POST"])
def update_grade():
    json_data = request.get_json()
    if "student_id" not in json_data:
        abort(400)
    # Update database
    return "success!"
```

Here we ensure that the key student_id is part of the request. Although this validation works, it really does not belong in the function itself. Plus, perhaps there are other routes that use the exact same validation. So, let's keep it DRY and abstract out any unnecessary logic with a decorator.

```
from flask import Flask, request, abort
import functools
app = Flask(__name__)

def validate_json(*expected_args):                  # 1
    def decorator_validate_json(func):
        @functools.wraps(func)
        def wrapper_validate_json(*args, **kwargs):
            json_object = request.get_json()
            for expected_arg in expected_args:      # 2
                if expected_arg not in json_object:
                    abort(400)
            return func(*args, **kwargs)
        return wrapper_validate_json
    return decorator_validate_json
@app.route("/grade", methods=["POST"])
@validate_json("student_id")
def update_grade():
    json_data = request.get_json()
    # Update database.
    return "success!"
```

In the above code, the decorator takes a variable length list as an argument so that we can pass in as many string arguments as necessary, each representing a key used to validate the JSON data:

- The list of keys that must be present in the JSON is given as arguments to the decorator.
- The wrapper function validates that each expected key is present in the JSON data.

The route handler can then focus on its real job—updating grades—as it can safely assume that JSON data are valid .

```
def slowdown(_func=None, *, rate=1):
    """
    Sleep given amount of seconds before calling the function
    """
    def decorator(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            time.sleep(rate)
            return func(*args, **kwargs)
        return wrapper

    if _func is None:
```

```
        return decorator
    else:
        return decorator(_func)

def set_unit(unit):
    def wrapper(func):
        func.unit = unit
        return func
    return wrapper

def use_unit(unit):
    """
    Have a function return a Quantity with given unit
    """
    use_unit.ureg = pint.UnitRegistry()
    def decorator(func):
        wraps(func)
        def wrapper(*args, **kwargs):
            value = func(*args, **kwargs)
            return value *use_unit.ureg(unit)
        return wrapper
    return decorator

def validate_json(*expected_args):
    def decorator(func):
        wraps(func)
        def wrapper(*args, **kwargs):
            json_object = request.get_json()
            for expected_arg in expected_args:
                if expected_arg not in json_object:
                    abort(400)
                return func(*args, **kwargs)
        return wrapper
    return decorator
```

Heading

Some text

Heading

Some text

References

Decorator module docs. (n.d.). Retrieved from https://decorator.readthedocs.io/

Python, R. (n.d.). *Dataclasses.* Retrieved from Real Python: https://realpython.com/python-data-classes/

Python, R. (n.d.). *Python Meta Classes.* Retrieved from Real Python: https://realpython.com/python-metaclasses/

Python, R. (n.d.). *python-is-identity-vs-equality.* Retrieved from Real Python: https://realpython.com/python-is-identity-vs-equality/

python.org. (n.d.). *"pie" syntax.* Retrieved from python.org: https://www.python.org/dev/peps/pep-0318/#background

python.org. (n.d.). *emulating-callable-objects.* Retrieved from python.org: https://docs.python.org/3/reference/datamodel.html#emulating-callable-objects

python.org. (n.d.). *functools.partial.* Retrieved from python.org: https://docs.python.org/library/functools.html#functools.partial

python.org. (n.d.). *PEP-3107.* Retrieved from python.org: https://www.python.org/dev/peps/pep-3107/

python.org. (n.d.). *PEP-3129.* Retrieved from python.org: https://www.python.org/dev/peps/pep-3129/#rationale

python.org. (n.d.). *PythonDecoratorLibrary.* Retrieved from python.org: https://wiki.python.org/moin/PythonDecoratorLibrary

python.org. (n.d.). *PythonDecorators.* Retrieved from python.org: https://wiki.python.org/moin/PythonDecorators

python.org. (n.d.). *zip(*iterables).* Retrieved from docs.python.org: https://docs.python.org/3/library/functions.html#zip

Wikipedia. (n.d.). *Introspection.* Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Type_introspection

Wikipedia. (n.d.). *Memoization.* Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Memoization