Getting Started with Python Concurrency

by Tim Ojo

Start Course

Bookmark Add to Channel Download Course

Table of contents Description **Transcript** Exercise files Discussion Learnin

Course Overview

Course Overview

Hi everyone, my name is Tim Ojo. I'm a software engineer at trivago, and I'd like to welcome you to my course, Python Concurrency Getting Started. Despite Python's popularity and its use in multiple areas from desktop and web development to scientific computing, many still regard it as a single-threaded language. In this course, we're going to learn how to execute tasks concurrently using pure Python. Some of the topics we'll cover include a Python-threading API, using multiprocessing to get around the GIL, the new asyncio module, and how to choose the appropriate concurrency mechanism for the task. The only prior knowledge needed for this course is a basic understanding of Python. And by the end of this course, you'll be able to develop faster and more efficient concurrent apps in Python. I hope you'll join me on this journey to learn Python concurrency with the Python Concurrency Getting Started course at Pluralsight.

Concurrency Concepts

Introduction to the Course

Welcome to the Python Concurrency Getting Started course. My name is Tim Ojo. And in this course, we will explain the forms and use cases of concurrency, take a look at the thread-based concurrency mechanisms available in Python, use how to use process-based concurrency to overcome limitations CPython has concerning threads, examine the concurrent. futures high-level concurrency API, which was provided to simplify the execution of parallel tasks, and finally we'll examine the asynchronous programming capabilities available from Python 3. 5 on. But before we dive in, let's take a quick detour to talk about Moore's Law as motivation for this course. In 1965, Gordon E. Moore, the director of research and development for Fairchild Semiconductor, predicted in an article in Electronics Magazine that the number of transistors in a dense integrated circuit would continue to double approximately every two years. This has been paraphrased to mean that processor speeds or overall processing power will double every two years. Gordon Moore went on to become the cofounder of Intel in 1968, and his prediction went on to be popularly known as Moore's Law. Moore's Law held true for several decades and has been used by the semiconductor industry as a guide for longterm planning and as a target for research and development. Many of the advancements in electronics we have today are strongly linked to Moore's Law. The development of chips that are increasingly cheaper, smaller, more powerful, and more energy efficient has led to all of the modern technology we know today. From the internet itself, to better electronic devices, modern healthcare, transportation, social media, data analytics, and the list goes on. But now Moore's Law is slowly dying. In 2015, Gordon Moore stated that he believed that the rate of progress would soon reach saturation. And Intel announced that same year that the pace of advancement had slowed. To keep up with Moore's Law today, chip makers are turning to multi-core CPUs. That is CPUs with two or more independent processing units that can read and execute instructions. So what does this mean for us developers? It means that today, more so in the future, we cannot make our programs perform better and solve harder problems by simply running them a newer, faster hardware. In order to make our applications run faster and to evolve the way hardware is evolving, we must employ concurrency techniques to allow them to take full advantage of multi-core hardware by executing instructions on multiple cores simultaneously.

Concurrency Concepts

Now that we've seen why concurrency is important, let's talk about what concurrency is. Concurrency is the execution of multiple instruction sequences at the same time. This is possible when the instruction sequences to be executed simultaneously have a very important characteristic, which is that they are largely independent of each other. This characteristic is important both in terms of the order of execution and in the use of shared resources. In terms of the order of execution, this means that the order of execution of these instruction sequences should have no effect on the eventual outcome. If task 1 finishes after tasks 2 and 3, or if task 2 is initiated first, but finishes last, the eventual outcome should still be the same. In terms of shared resources, the different instruction sequences should share as few resources between each other as possible. The more shared resources that exist between concurrently executing instructions, the more coordination is necessary between those instructions in order to make sure that the shared resource stays in a consistent state. This coordination is typically what makes concurrent programming complicated. However, we can avoid many of these complications by choosing the right concurrency patterns and mechanisms depending on what we are trying to achieve. This course will help show you many of the concurrency patterns and techniques offered in Python, and guide you on how to choose and use them to achieve your goals. But before we dive into Python, we should get a general understanding of the different types of concurrency in use today.

Types of Concurrency

The two main forms of concurrency that we will focus on in this course are parallel programming and asynchronous programming. It should be noted that distributed computing also offers to us another form of concurrency. However in this course, we will be focusing our efforts on currency that operates on the level of a single machine. Parallel programming involves taking a computational task and splitting it into smaller subtasks that are then assigned to multiple threads or processes to be executed on multiple processor cores simultaneously. With single-threaded code, if you have multiple processor cores on your system, only one core will be charged with executing your task, while the other cores sit idle or execute instructions for other programs. With parallel programming, all your processor cores can be engaged, theoretically cutting your processing time by a factor of the number of cores that you have available. Because it's designed to take advantage of multi-core systems, parallel programming is best suited for tasks that are CPU intensive. That is, tasks in which most of the time is spent solving the problem rather than reading to or writing from a device. Another term for those types of tasks is that they are CPU-bound tasks, which means that they will perform better if you could get better performance out of the CPU. Examples of such workloads are string operations, search algorithms, graphics processing, any number crunching algorithm, and more. If you have a task in which most of the time is spent reading from or writing to a device, which is more commonly known as performing input or output, in other words, IO, then that task is more suited for asynchronous programming. Asynchronous programming's concurrency model works by delegating a subtask to another actor such as another thread or device, and then continuing to do other work instead of waiting for a response. When the subtask is completed, the actor then notifies the main thread, which calls a function to handle the results. This is usually known as a callback function. In some languages, instead of executing a callback, the main thread is given an object that represents the results from a not-yet-completed operation. This object is typically called a future, promise, or a task, and the application can either wait on its completion or check back at a later time. Examples of IO-bound tasks are database reads and writes, web service calls, or any form of copying, downloading, or uploading data either to disk or to a network.

Concurrency in Python

Python supports both parallel and asynchronous programming natively. The threading module was introduced way back in Python 1. 5, and allows you to create thread objects that are mapped to native operating system threads, and can be used for the concurrent execution of code. However, it should be noted that in CPython, which is the most common implementation of the Python specification, threads are limited to executing Python code serially by a mechanism known as the global interpreter lock. Therefore with threading in Python, concurrency is limited to what you get when the operating system switches between threads. We'll discuss this in more detail in the next module. The multiprocessing package is an implementation of parallelism that uses sub-processes instead of threads. This technique avoids the global interpreter lock and allows Python to take advantage of multiple processor cores. The multiprocessing package was introduced in Python 2. 6 and has an API that is similar to the threading module, but also introduces some new APIs. The concurrent. futures module was introduced in Python 3. 2, and it provides a common high-level interface for using thread pools or process pools. And finally in Python 3. 4, the asyncio module was introduced as a provisional package to enable asynchronous programming. Starting with 3. 6, the provisional status was removed, and the API was declared to be stable. However, if you're still running Python 3. 5 and cannot upgrade versions, you shouldn't have any problems working with the examples we will be using in this course.

Introduction of Demo Application

Our main example throughout this course would be the development of a thumbnail maker library. The function of this library is that it takes multiple images of a certain size and produces smaller copies of the original image, while keeping the dimensions intact. How is this useful, you may ask? Let's say you've built a service or website that has a notion of user profiles and allows the user to upload a profile picture of themselves. A profile picture taken on a smart phone camera can easily have a resolution of 3264 x 1836, and a size of 4 MB per image. This image is way too big for your site to serve, especially as a thumbnail, which is usually a small image on the top-right corner of the web page, or a small image that accompanies comments. Attempting to serve such a large image will lead to very slow load times and high data usage, two things which will quickly drive away users from your site, especially mobile users. For reference, Facebook uses a 200 x 200 image as the max profile image size, a 50 x 50 image throughout the website, and a 32 x 32 image for comments on threads. Skype uses a 96 x 96 image for the chat window profile picture and a 48 x 48 image throughout the rest of the app. Similarly, Pinterest displays a profile page picture at 165 x 165 and uses a 32 x 32 thumbnail everywhere else. Therefore for our example application, we will follow a similar model, and resize the images offered to us by our users into smaller sizes for use in different situations. A typical workflow would be to download the images from some form of file store like Amazon S3 to the image or application server, perform the resizing, and then re-upload the resized image back to the file store or some other location where they can be served. However, to keep this example simple, we will only download the images from the source location, which in our case will be DropBox, and then perform the resize operation. We won't worry about the re-upload for this example, because these two operations are sufficient for us to illustrate the different forms and use cases of concurrency. The download of the images is an IO-bound task, and the resizing is a CPU-bound task. For our example application, I have arbitrarily picked target image sizes of 200 x 200 for the large version, 64 x 64 for the medium size, and 32 x 32 for the small size. Let's take a look at the initial code for the application. The initial code is single threaded, and fairly simple, and straightforward. We will use it as a starting point in this course, and then throughout the course, we will refactor the code to use different concurrency mechanisms to speed up parts of the application. This model of starting with a single-threaded program and then refactoring it is good practice in general, because it allows you to ensure that your code works properly, debug it, handle exceptions, etc., before introducing the complications that can be inherent in running code in multiple threads. The starter code for this course is hosted on GitHub at the URL showed on your screen. If you're not currently a Git user, you can simply download the code from the site. If you are a Git user, then you have the option to clone the repository.

Demo Code Walkthrough

I tend to rely on the commandline for working with Git, but GUIs work just as well. The clone command will be gitclone, and then the URL with. git appended to it. I'm going to paste in the

URL here, and hit Enter. Done. Before running the code, the first thing to do will be to check what version of Python 3 you're running by typing in the command python3--version. If you are running on Windows, the command may be python--version instead. As of the release of this course, the most recent version of Python is 3. 6, but for the purposes of this course, any version of Python from 3. 5 onwards is acceptable. If your Python version is 3. 4 or less, you will need to upgrade your Python version. A quick internet search will show you how to do this on whatever platform you're running on. If you do not have the luxury of upgrading your Python version, with the exception of some of the asyncio code, most of the examples in this course will work for you. Once you have your Python version squared away, the next step will be to install the Python libraries required for this project. We have only two for this project, Pillow and pytest. Pillow is a well-maintained fork of the Python imaging library and provides us with tools to manipulate images. And pytest is a unit test runner that we'll be using to run our ThumbnailMakerService. Both of these libraries are included in the requirements. txt file. So you can go to your shell, and install them using the pip command, pip3 or pip if you're on Windows, install -r requirements. txt. If you don't have these libraries installed already, it may take a few moments depending on how fast your computer and network connection is. Since I already have these libraries installed, it lets me know that. Now let's explore the code. In thumbnail_maker. py, we define the ThumbnailMakerService class. Its initializer takes in the home directory of the service, defaulting to the current directory if nothing is passed in. In the body of the initializer, we set the input directory and the output directory. The input directory is where we will download images to be resized, and the output directory is where we will place the resized images on completion. Our next method is the download images method, which takes in a list of image URLs to be downloaded. Our first step is to verify that the list is not empty and the target folder exists. We then simply iterate through the img_url_list, and download each image one at a time, saving them to the input folder using the filename from the URL. I added timers to see how long this process takes. Our next method is the perform resizing method. As usual, we first start with validating the inputs, checking that the input directory is not empty and the output directory exists. We then specify our target sizes. I'm also getting a count of the number of files in the input directory, so that I can log that later on. Then in a loop, we load each image into memory, and for each target size, we calculate the target height of the image that allows the image to maintain the aspect ratio. This is achieved by figuring out the ratio of the desired width to the current width, and then multiplying the current height by that ratio. Once we have our target height, we can then call resize, passing in our target height and width at a sampling filter. When the resize completes, we save the image to the output directory using the same name, but with the base size appended to the name. The last method we will take a look at is the make_thumbnails method, which is essentially the public interface for our ThumbnailMakerService. It takes in the image URL list and calls the methods needed to complete the resizing process. You may have noticed by now that the code is structured into these two methods on purpose. The download_images method is primary IO bound, while the perform_resize method is primarily CPU bound. We could've easily written our application to download each image and then resize them within the same loop, but this will run a file of a best practice in Python concurrency, which is not to mix different types of workloads. Because Python's concurrency mechanisms are optimized for either IO-bound or CPU-bound tasks, mixing those two in one operation often reduces the efficiency of Python concurrency mechanisms. To run our ThumbnailMakerService, we will use a test. In test_thumbnail_maker. py, we have a list of URLs to some example images that were uploaded to DropBox. In a test_thumbnail_maker method, we initialized our ThumbnailMakerService class, and then called the make_thumbnails method, passing in that list. To run the code, we will open our terminal application, ensuring we're in the project folder, and then type in the command pytest to execute the test. Pytest will find your test files based on naming conventions and execute your test methods. The execution time of this code will depend on the speed of your internet connection for the downloading of the images and your processor speed for the resizing of them. However, by in large, the download takes significantly more time than the resizing. Once the test is completed, we can take a look at the log file that logged to verify this. So in my run, the download took 27 seconds, and the resize took 3 seconds. If we go take a look at the outgoing folder, we can see the resized images with their width appended to the image name. Now that we have the simple synchronous version of the code, we will learn how to use Python concurrency to make this code run faster, starting with the use of threading in the next module. But first, let's look back at what we've covered so far.

Module Summary

We started out using Moore's law as motivation for why the industry is moving towards concurrency, and why every developer needs to be familiar with concurrent programming. We then defined concurrency and the conditions necessary for concurrency to be possible. We briefly introduced the different types of concurrency, and the workloads that they are suited for. We noted the different concurrency mechanisms in Python. And finally, we set up the example that we'll be using for the rest of this course. That's it for module 1. I hope you learned something and are excited for the next module in this course, Threading.

Threading

Introduction to Threads

Welcome to the Threading module in the Python Concurrency Getting Started course. I'm Tim Ojo. In the last module, we got a general introduction to the field of concurrency and discussed the different forms of concurrency, which includes both parallel and asynchronous programming. In this module, we'll be diving deeper into parallelism using threads. We will learn how to use the threading package in Python to create and execute threads. We'll discuss the behavior and structural characteristics of threads. We'll learn how to do thread synchronization and inter-thread communication in Python. And since we're in Python, we'll talk about the global interpreter lock and its effect on the usage of threads. So let's get started. The predominant method of concurrent or parallel programming for the past couple of decades has been the use of multiple threads, multithreading. To introduce threads, let's start with defining a process. A process can be defined as the execution context of a running program. An alternative and maybe more approachable definition is that a process is a running instance of a computer program. While a computer program is a passive collection of instructions in the form of code, a process is the construct we use to define the execution of those instructions. Every executing process has system resources and a section of a memory that is assigned to it. It also has security attributes in a process state. A process is composed of one or more threads of execution. A thread is the smallest sequence of instructions that can be managed, that is scheduled and executed by the operating system. A program can be composed of a single thread of execution or multiple threads of execution. When multiple CPU cores are available, each thread's instructions can be executed at the same time in parallel of multiple cores. If only a single core is available, the threads share time on that core. In either scenario, the result is that the use of multiple threads allows a process to perform multiple tasks at once. For example in a media player, one thread can be playing the current song, while another is figuring out the next song to play and downloading it, while again another thread is responding to user clicks and navigation. Another example is a web or application server that uses a pool of threads to respond to multiple requests simultaneously. Each request is handled by a thread from the pool. The thread executes whatever task is assigned, and then when it is completed, it returns to the pool to wait for the next request. Multithreading is supported by virtually all operating systems and almost all programming languages, and Python is no exception.

Creating Threads in Python

Python has had support for threading since version 1. 5. 2 via the threading package in the standard library. This package allows you to create thread objects that map directly to operating system threads. The simplest way to create a thread is to instantiate an object of the thread class, passing in the thread function, as well as any function arguments, and then calling the start method on the thread object you just created. In this example, we start by importing the threading package, and then we define the function that we want our thread to run. All we're doing in this function is printing out some text, including the passed-in parameter called val. The interesting part comes in line 7. This is where we construct a thread using the threading. Thread constructor. Here we define our target function and any optional arguments that the target function requires. In line 8, we schedule the new thread to start by calling the start method, and then in line 9, we suspend our execution until the new thread completes, by calling the join method on the thread object that we want to wait for its completion. The full threading. Thread constructor is showed on your screen. The group parameters are reserved for future use, so it's always set to None. The target is the function to be invoked. It can specify a name for a thread using the name parameter. If you choose not to, a default name will be used, which will be the word Thread and a counter appended to the thread. The args parameter is the argument tuple for the target function invocation, while the kwargs parameter is a dictionary of keyword arguments for the function, if you choose to use that instead. And lastly, there is a daemon parameter, which is added in Python 3. 3. that specifies whether the thread will be terminated, if it's parent thread terminates or not. This way of executing tasks with threads is the most common usage pattern for threading. It considers the thread to be a worker executing the instructions in a target function. There's a second usage pattern of threading where we use threads known as the worker function performing some tasks, but as a unit of work instead. To do this, we create a class that inherits from the thread class in the threading package and overrides the run method. The _init method can also be overwritten to provide additional state in the object via the constructor, but we should keep in mind that whenever you override a _init method in the sub-class of thread, the super class' _init method must always be called first before performing any other operations. Here in our overridden run method, we are calculating the Fibonacci number for the number passed into the constructor and printing out the value. We see an example of the usage here. We can create objects of our class, and because they are thread objects, we can call start to schedule the threads to start, and we can call join to wait for the threads to complete. These threads we run concurrently on the machine, and then the program would exit. Let's get our hands dirty and try out some of this code ourselves using the demo. The first thing we'll do is import the threading package. Then we'll go ahead and create a new method. I'll call it download_image. This method will be the target function for the threads. So I'll cut and paste the critical code from the download_images method to the new download_image method. I can then come in and create a new thread, setting download_image as the target and passing in the URL as the argument, then start the tread. I could also go ahead and call thread. join here, but that'll be a mistake, because what would happen is that in a loop, the main thread will create a new thread, start it, and then wait for it to complete before continuing the loop. This isn't what we want. What we want is to create the worker threads, and then wait for all of them to complete. So to accomplish that, I'll create a list that will hold references to the thread objects, and then every time I create a thread in the loop, I'll add it to the list. Then after all the threads are started, I'm going to loop through my thread list and call join on each of the threads, so the main thread is forced to wait for all the threads to complete before continuing. I'm ready to run this code, but before I do, I'd like to add a couple of log messages to my download_images method in order to get a better picture of what actually happens. And then I'm going to specify a format for the log messages that will allow me to see the threadName, the current time, the logging level, and of course the message. The last thing I want to do is scroll down to the make_thumbnails method, and comment out the perform_resizing method, just because I don't really need this part of the code to run for this illustration. Alright, so I'll open my terminal, and run pytest to run my code. The log file will have my results as they come in. So we can see that the threads get kicked off within milliseconds of each other. On the left, there is the threadNames and the times. And because they run concurrently, the entire download completes in 6. 5 seconds, instead of the 27 seconds that it previously took, which is almost a 4 times improvement in performance. Let's talk a bit more about how threads work, as this would give us the foundations needed to understand thread synchronization and inter-thread communication.

How Threads Work

Using our simple example, we can trace the lifecycle of threads that are created and executed when this code is run. When the program starts, there's only one thread in existence, the mainThread. The mainThread executes the instructions for importing the threading library, defining the do_some_work method, and creating the val variable. The mainThread then creates the new thread. At this point in time, the new thread is in a new state. When the mainThread calls start on the newThread object, it goes into the ready state, which means that the thread is now available for the OS to schedule to run on a CPU. After this, the mainThread calls join. Prior to this, the mainThread had been in the running state, but now it goes into the blocked state, which means that it is suspended and can't execute until something happens. In this case, the something that needs to happen is a completion of thread 1. Thread 1 goes into the running stage, executes the 2 print instructions, and terminates once the method is completed. This signals to the blocked main thread that it can now go back into the ready state and finish execution. Here's a diagram depicting the full threads lifecycle. A thread starts out in a new state, and from there it goes into the ready state when start is called. From there it moves back and forth between the running state and the ready state. Whenever it's actively executing on a CPU, it's in the running state, and if its execution gets paused, it goes back into the ready state. If a situation arises where a thread has to wait for a particular condition to occur before it can continue executing, it goes into the blocked state. For example, it may block while waiting for an IO operation to complete or while waiting for another thread to complete. A thread cannot move from the blocked state back into the ready state until the condition that it's waiting on occurs. The final state is of course the terminated state, which occurs when the thread completes or if an unhandled exception occurs. In a program with multiple threads, each thread has its own counter, which maintains the instructions that has been executed at the current time, its own registers to hold the data being used in computation, and its own stack, which is memory set aside as scratch space for a thread. Local variables within a function are kept on the stack and are therefore thread safe as each thread has its own stack space. However, threads within a process share code, a common memory space, and other OS resources such as open files or network sockets. Therefore any data or area of memory owned by a process can be read and modified by any of the threads that are part of the process. To make things more interesting, once a thread is started, we have little control over how it runs. The operating system using an algorithm determines what threads run, for how long they get to run before they get suspended, and on what processor core they run on. A thread may be running, and then after a certain amount of processor time as determined by the OS scheduler, the thread gets suspended mid-execution, so that other threads that are waiting for processor time get to run. The scheduler then uses an algorithm to select which threads should run next and performs a context switch to cycle the running thread out of the CPU and the selected thread into the CPU. If the newly-selected thread is from a different process, a full process switch occurs, which is a fairly expensive process. However, if it's a thread from the safe process, the thread switch is executed, which is a less expensive process. This is one of the reasons why in most cases parallel programming using threads is preferred over parallel processing using processors. But thread-based parallelism is not without its perils. Memory sharing combined with indeterminate scheduling can lead to a situation known as thread interference. The canonical example used to describe thread interference is the BankAccount example. If two threads are allowed to run this code, and one of the threads is running the deposit method, while the other is running the withdrawal method, we could run into a situation where thread 1 reads the balance as 0, it gets interrupted and suspended, then thread 2 comes in and reads the balance as 0 as well, it subtracts 50 from 0, resulting in -50, thread 2 stores the resulting balance as -50 and then exits. Thread 1 gets switched onto the processor, and its stack gets restored. It adds 100 to 0, resulting in 100, and stores the resulting balance as 100 and exits. Here the 50 dollar withdrawal is completely lost as a result of thread interference. And the balance that should have been 50 ends up incorrectly being 100. Thread interference is also commonly referred to as a race condition, as in both threads are in a race to read or update the same variable, and you may get different results depending on what threads run the code at what time. And since this is out of programmer control, sometimes your program results in a correct outcome, and other times it doesn't. This can be very hard to debug, as the data corruption may occur only intermittently in your production environment under heavy load, but the code runs fine in your local development environment. To solve this problem, we use a technique called thread synchronization.

Thread Synchronization

Before we talk about thread synchronization, I must mention that it's best practice in concurrent programming to keep accessing shared memory for multiple threads to a minimum. The more shared memory access that is done by threads, the more complicated your code gets, and the less concurrent it runs, because of the need to combat thread interference. But sometimes the function of your program unavoidably requires threads to read and write to shared memory. Python has a number of mechanisms that can be used to synchronize access to shared resources. The most fundamental of these is the lock. A lock is in one of two states, locked and unlocked. When put in the locked state by one thread, it could only be unlocked by the same thread. It cannot be locked or unlocked by another thread. For this reason, we use the word acquire to signify a thread's putting of a lock into the lock state, because once the lock is acquired by a thread, it can't be acquired by another thread until it has been released by the thread that owns it. So what happens if a thread tires to acquire a lock that's being held by another thread? Well, that thread goes into the block state, which means that its execution is paused and can't continue until the lock that it is trying to acquire is released by the owning thread. We can use this locking and unlocking mechanism to synchronize access to shared resources. To do so, we'll need to create a lock for the shared resource. Before accessing the shared resource, the thread must acquire the lock for that particular resource. This ensures that no other thread can also acquire the lock and access the resource at the same time. When the thread is done with the resource, it releases the lock, so that other threads are now allowed to access the resource. But there's a small flaw in that example code. If something goes wrong during the resource access, and an exception is thrown, we may never release the lock, which would cause all the other threads waiting on the lock to block forever. We can solve that simply by using a try finally block. By putting the releasing of the lock in a finally block, we ensure that it will be run even if an exception is thrown. But we can take it a step further. In Python 2. 5, the with statement was introduced to make the acquisition and releasing of certain types of objects, such as locks, files, and network connections easier. This example shows how to use the with statement to manage the acquisition and releasing of a lock. The lock is automatically acquired when entering the with block and automatically released when leaving. As with the try finally block using the with statement, the lock will always be released, even if an exception occurs during resource access. Let's try this out using our thumbnail_maker example. Let's say we wanted to get the total number of bytes downloaded for accounting purposes or to see how much space we would save after we resize the image. The initial value for our downloaded_bytes variable will be 0. In the download_image method, I'm going to do a bit of cleanup by creating this dest_path variable, so I don't have to keep computing it over and over again. I'll get the size of the download image into a variable called img_size. Next I'm going to update the downloaded_bytes counter by adding the img_size for this particular image to whatever the current value is. While this may seem like one operation, it's really three. First the Python interrupter reads the current value of downloaded_bytes, then it adds that value to the value of img_size, and lastly it stores the summed value back into downloaded_bytes. If one of the threads running this code gets interrupted after it's read the value of downloaded_bytes, summed data could be lost, as some other thread could've also updated the same value without its knowledge. The solution here is to use a lock. We'll call our lock dl_lock, and then we'll move our critical section of code into a with block for that lock. Now we have the guarantee that even if the current thread gets interrupted, no other thread can modify the downloaded_bytes variable while it has the lock. The last thing we want to do here is to print out the downloaded_bytes value. Now I'm going to run this and expect to see the size and bytes for each image, and the total number of bytes downloaded. One quick thing to note is when to use locks and when they aren't needed. If we're reading the value of a shared variable that isn't modified, like we're doing in line 27, then we don't need to synchronize access, since we're just reading the value and never modifying it. Also, if instead of updating the value of downloaded_bytes, we simply replaced it, then we won't need a lock, since these are both atomic operations, which means they happen in a single step, therefore they can't be interfered with. Other examples of atomic operations are getting an item from a list or dictionary and adding in an item to a list or dictionary. Now before we move on to other synchronization methods, there are a couple of things to note about the lock API. The lock. acquire method takes in an optional Boolean argument that defines whether the thread trying to acquire the lock should block or not, if the lock it's trying to acquire has already been acquired by another thread. By default, this argument is set to true, but if you set it to false, the call to the lock doesn't block the calling thread. Instead it returns false if the lock cannot be acquired and true if it's successfully acquired the lock. There is also the locked method. The locked method allows the caller to determine whether the particular lock has been acquired or not. This may come in handy if there is some code that you want to run in case where a lock has already been acquired. In the category of locks, there is a type of lock called reentry lock or RLock for short. With a regular lock, the lock can only be acquired once, even by the same thread. If a thread tried to acquire a lock that it already owns, it will block. The RLock fixes this by allowing a thread to acquire a lock is already holds. If a thread tried to do so, it would just continue.

Inter-thread Communication Using Queues

Another widely-used thread synchronization mechanism is the semaphore. A semaphore is a synchronization primitive that manages an internal counter instead of a single locked or unlocked switch. Each time a thread call is acquired, the internal counter is decremented. And each time release is called, the counter is incremented. The internal counter can never go below 0, so if a thread makes a call to acquire, and the value of the internal counter is 0, the thread gets blocked and must wait for another thread to call release to increment the counter before it can continue. A common analogy is that the semaphore maintains a set of permits. Every call to acquire attempts to acquire one of the permits. When the thread is done, it must then release the permit. If there are no permits currently available, then the thread must wait for another thread to release a permit back into the pool before it can acquire the permit, whereas a lock can be used to permit only one thread to run a critical section of code at the same time. A semaphore can allow one or more threads to run at the same time, depending on the number used when initializing the semaphore. If no parameter is passed to the constructor of the semaphore like the example on the screen, then the internal counters defaults to 1. In the next example, we're passing in the number of permits that can be acquired without blocking. We are also using a bounding semaphore instead of the standard semaphore implementation. This is because the standard semaphore implementation allows you to call release an unlimited number of times, more times than you called acquire. With the bounded semaphore, if you inadvertently attempt to release more times than you have calls to acquire, an error will be thrown. Let's use our thumbnail_maker to illustrate how we can use semaphores for thread synchronization. Let's say we have a limit of the number of concurrent image downloads we are allowed to perform at once. Now I don't believe DropBox has such a limit, but image we had to cap our downloads at a maximum of 4 concurrent downloads. One way to do this would be to use a semaphore. I'll start by setting the value of max_concurrent_dl to 4, and instantiate a semaphore object with that value. Then in my download method, I'll call acquire at the start of the method body, and release at the end of it. To ensure that release is always called, I'll use a try finally block. Now when I run this code, what I should see is that no more than 4 threads can execute the download code at the same time. Looking at the logs, I can verify this. Initially only 4 threads can download at the same time, because we have only 4 permits. Once thread 4 is done and releases its permit, then thread 5 can acquire the permit and begin running. One thing to note is that instead of following acquire and release in the finally block, we can also use the with statement just like we did with the lock. In the context of thread synchronization, the with statement can be used with locks, semaphores, and condition objects. We'll talk about conditions shortly, but first let's briefly discuss events. Events are pretty simple. One thread signals an event, and the other threads wait for it. An event object has an internal flag. The thread or threads that need to wait for the event to occur do so by calling the wait method on the event. As long as the event's internal flag is false, the threads that call wait will block. A manager or server thread can then call the set method on the event, which sets the internal flag to true and releases the block threads or threads. The server or manager method can also call the clear method on the event, which resets the internal flag to false and causes subsequent threads calling wait to block until set is called to set the internal flag to true again. A condition combines the properties of a lock at an event. Like a lock, it has an acquire and a release method to synchronize access to some shared state, and like an event, it has a wait, a notify, and a notify_all method, so that threads interesting in knowing when the state changes can call the wait method and can be notified when the condition changes. The typical use case for a condition is in implementing a producer/consumer pattern. I have a simplified example of this on the screen. In this example, the consumer thread gets the lock, and checks whether an item is available, and calls wait if one isn't. Calling wait releases the lock, allowing the producer thread to acquire it, produce the item, and then notify the waiting consumer thread. It must also release the lock in order for the waiting consumer thread to be able to actually return from its wait call. The consumer thread wakes up when notify is called, reacquires the lock, and then gets and processes the produced item. It releases the lock when it's done. This example is as simple as it gets with conditions, yet I still have a problem with it, because it's not intuitive to the reader of the code, and as a writer, I have to constantly keep the state of the threads and condition in mind, which can be complex and error prone. What if I don't have to deal with that? What if there was an easier way of communicating between threads that didn't involve locks, conditions, and thread states? Well, there is, and it's called a queue.

Demo: Multiple Reader Threads

The Python standard library queue is a construct that makes it easier to exchange messages between multiple threads. The queue handles all the locking semantics for us, the programmer, and allows us to focus on just four methods, put, which puts an item into the queue, get, which removes an item from the queue and returns it, task_done, which marks that an item has been gotten from the queue and processed completely, and join, which blocks until all the items in the queue have been processed. There are more than four methods in the queue API, but these four are the most common and important ones. Using a queue to communicate between threads brings us closer to a message-passing architecture, resulting in code that's cleaner, safer, and more readable most of the time, as opposed to the messier world of synchronized access to shared state. Therefore if the situation fits, I'd recommend using queues over conditions or events. Here's our producer/consumer example again, but using a queue instead. We have a queue object and our two threads, t1 and t2 that act in a capacity of the producer and consumer. In the body of the producer method, we can see that the thread produces 10 items and puts each one in the queue. Then our consumer thread simply loops and gets the items from the queue one at a time, and calls task_done each time it completes the processing of an item. It's important to know that both the get call and the put call are blocking calls. When the queue object is instantiated, you can specify the maximum number of items that can be in the queue. If you don't specify a value, like we have in our example, then you have an unbounded queue; however, if we had specified a value for max_items, then the put can block if the queue is full. It will block until a consumer thread removes an item from the queue, or until a user-specified timeout threshold is reached. The get call can also block, and it actually blocks more frequently in practice. The get call blocks when there are no items in the queue to get. In this situation, the thread making the get call is blocked until the new item is put in the queue for it to read, or until a user-specified time out is reached. We can pass a flag into the get call, instructing it not to block if the queue is empty, and we'll see an example of that later. Now that you've seen how easy it is to pass messages amongst threads, let's put it into practice in our thumbnail_maker example. In the last couple of sections, we've spent time building up locks and semaphores, but I'd like to temporarily back away from that right now, and go back to the state of the code when we first cloned or downloaded the code, but I don't want to lose the code we already have, so I'm going to create and check out a new branch. I'll call it threading-intro. And then commit the code that we have so far with the commit message of threads, locks, and semaphores. And then I'll switch back to the master branch, which should be in the original state it was when I first cloned the repo, but all the changes I made earlier are not lost, they're just in a different branch. If you're following along and aren't using Git, then the way to do it will be to rename your Python file with the stuff you've done so far, that is maybe thumbnail_maker_threading-into, and then grab an original copy of the thumbnail_maker. py, and place it in your project folder. So the first thing we want to do is from queue import Queue, and from threading import Thread. I'm also going to set up the format string for the logger, just like we had before. Specify the threadName, the time, the log level, and the message, and pass it to the config method. Then I'm going to instantiate a queue instance in the constructor. I'll call it img_queue. In my download method, all I need to do is after a download is complete, I'll pop the file name for the file I just downloaded into the queue. For the resize method, now instead of reading the folder, I want to go into the loop, and in the loop I want to read the files from the queue. Then I mustn't forget to mark a task as done after I'm done resizing. Oh yeah, I don't need that piece of code anymore since I'm reading off the queue and not the filesystem, and that will actually check the filesystem before it runs. Now let's go to our make_thumbnails method, and create our two threads. T1, which will target the download_images method, and t2, which will target the perform_resizing method. As usual, we need to start and join both threads. That's the majority of the changes that need to be made. For informational purposes, I'm going to add a login statement to the beginning and end of the resize operation. But I see a small problem here though. Up here we have an infinite loop, and so our thread will never end, which means that our program will also never end. So how do we fix this? We could change the while condition to test the flag that gets set by the producer thread, but then we go back into the world of shared state and race conditions, and the chances are very high that the resizing thread misses some items that are still in the queue. Since what we need is a way for the download thread to tell the resize thread that there are no more items, we could have the download thread do so by putting a special message into the queue, so when the resize thread receives the message, it can exit the loop and terminate. This technique is known as the poison pill technique. Here's how that would look. After downloading all the files in the download_images method, we could put a non-value or empty string into the queue. This message marks that there are no more items to be enqueued. Now in the resize method, we need to check the items that we get from the queue. If it's not a none or an empty string, we process it; otherwise, we break out of the loop. Of course we must not forget to mark the task as done. I think we're ready to run this, we'll run it using pytest as usual, and we can switch over to the log file and view the progress. So what we have now is we have one thread downloading and putting items into the queue, this is our producer thread, and we have another thread reading items from the queue and making thumbnails, this is our consumer thread. We only expect a minimal amount of performance gains, because the downloads are still happening sequentially. However, we're not waiting for all the downloads to complete before we start resizing, so that results in some performance improvements as we can see here. But we can take this a step further, and get much more significant performance gains by parallelizing our downloads. Because we're now using a message-passing style, and queues handle all the locking for us behind the scenes, it's just as easy to have multiple threads that are reading and writing from the queue, as it is to have one thread on each side. So we can come up with a design where we have our main thread throw all the URLs from the list into a queue, and then have multiple download threads pulling items off of that queue, performing the download, and then pushing the message into another queue for the resize thread to process. We'll start by adding another queue from the downloads. The main thread will load all the URLs for the images to download into this queue, and then spin up some threads to drain the queue and perform the download. Let's create the target method for the download threads. We'll need to go into a loop and read messages off the queue, but instead of going into an infinite loop, let's loop until the download queue is empty, because we know that once the queue is empty, no other URLs will be added. Then as usual, we'll get an item from the queue, but this time we don't want to block and wait if the queue is empty, we just want to exit the loop. If we pass the block=False flag on a get call, and the queue happens to be empty, then the call throws a queue. empty exception. So let's catch that exception. We don't have to do anything with it though, we'll just write a log statement. My ID doesn't recognize the queue. empty object, so it's giving me a red squiggly line, but that can be ignored. So you may be wondering, if at 9:25 we only go into the loop if there are items in the queue, then why do we have to guard against the queue being empty in line 27? This is because since we plan on having multiple threads reading off the queue, we could have a situation where this is one last item in the queue, and the thread comes along and checks if the queue is empty, and gets back a result of false. Then it goes to read the item of the queue, but by that time another thread has come in and taken that item off the queue. So that's the situation we're guarding against with the block=False flag in the tri catch. Okay, so after this we do our download, and put the item into the image queue for the resize thread to pick up. To save on typing, I'm just copying and pasting the code. Lastly, we mustn't forget to mark the task as done for every URL we get off the queue. There are no changes to be done in the perform_resizing method, so I'll just move down to the make_thumbnails method. Here we'll start by loading all the URLs into the dl_queue. Then in the loop, we'll create four threads for pulling URLs off the queue and downloading them. And of course start the threads. We don't need to call join on the download threads, just on the resize thread. This is because, as you may remember, a join blocks the calling thread until the join thread is complete. And her we know that the resize thread is more important to join on, because once the resize is complete, then we are done, not when any other downloads threads are complete. But we still have a small problem left to solve, the resize thread only terminates when it receives the poison pill, and we're not doing that in the download_image method. This is intentional, because if we try doing it in the download_image method, then we would run into a situation where one of the four threads completes, and sends the poison pill, killing the resize thread before it can process messages from the other three. So we need a way to wait for all the download threads to complete before sending the poison pill. This is where the join API on the queue comes in. What we can do is after starting the resize thread, we can block the main thread until all the downloads are complete, by calling dl_queue. join. And then we put the poison pill into the image queue, so that the resize thread can terminate when it reaches the end of the queue. I'll go ahead and rerun our test. This time we expect a much more significant performance increase now that we have four threads downloading, and that's what we get. Our code runs in roughly a quarter of the time. So we see now we can use queues to communicate between threads with minimal or no need for logs, events, conditions, etc. We were even able to implement a pool of four threads to read from the queue and do the download in parallel. This approach to parallelizing our downloads is better than our previous approach, because we can control the number of threads that do the download, and we don't run the risk of creating too many threads. In the previous approach, we were looping through the list and creating and kicking off a thread to download to each item from the list. This is fine if we have a small list of items to download, say less than 100 or in the low hundreds. In our case, our list is 26, so we end up spinning up 26 threads. If our list size happened to be a 1000, 100 thousand, or more, it would be really bad for the system we're working on to spin up that many threads. In most systems, you'll either run out of stack memory space sometime before you hit the 100 thousand threads, or you'll hit an operating system limit, and your program will fail. So instead of creating a thread for each item in the list, and you can't anticipate its size beforehand, the better option is to have a fixed pool of threads for the download. That concludes our discussion on the threading module, but before we move on from parallelism using threads, we need to discuss the most talked about lock in Python, the global interpreter lock.

The Global Interpreter Lock

If you take a look at best practices surrounding concurrency in Python, you'll notice that it says to use Python threading only in cases where you're doing IO-bound operations. The reason for this is the global interpreter lock or GIL for short. The global interpreter lock is a lock within the

Python interpreter code that prevents multiple native threads from simultaneously executing Python code. And Python interpreters that aren't thread safe, which is the majority of Python interpreters, the GIL exists to to protect internal data structures from thread interference and race conditions. This allows Python interpreters to be easier to write and to integrate more easily with C libraries that typically aren't thread safe either. It also allows interpreters to run much faster for single-threaded programs, because with the single lock for the interpreter, there is no need to acquire and release locks on all of the internal data structures individually. There's just one global lock to acquire, and the interpreter is off to the races. The downside of the GIL though is that only one Python thread can execute at a time. If this is true, then how are we seemingly able to perform operations in parallel using Python threads? This occurs because after a certain interval, Python threads are forced to give up the lock in order to allow another thread to run. So instead of having true concurrency, which is multiple threads running at the exact same time on different processor cores, what we actually have it corporative multithreading, in which threads give way to one another to allow each other to take turns to run on the same core. This actually degrades the performance if you're using multiple threads to try and speed up CPU-bound operations, but IO-bound operations benefit from multiple threads, because during IO-bound operations, the thread releases the GIL and blocks until the IO operation completes. As a result, it's not in contention for the CPU, and other threads can run during that time. That's the reason best practices state that the use of Python threads should be limited to IO-bound operations, because the global interpreter lock denies the interpreter the ability to operate concurrently on multiple threads, and instead forces only one thread to operate at a time in order to ensure thread safety. There is an ongoing attempt to remove the GIL from the CPython interpreter, which has been coined the Gilectomy, but until data is successfully completed, the Python community has two workarounds for parallelizing CPU-bound tasks with Python. One of the workarounds is to use a Python implementation that doesn't have a GIL. Currently, the only two Python interpreters without the GIL are the Python implementation on top of Java known as Jython and the Python implementation on the. NET Framework known as IronPython. Because neither of these interpreters has a GIL, executing multiple threads over multiple processor cores is possible, but the most popular workaround for the GIL is the use of multiprocessing. Here we use multiple processes instead of multiple threads for parallel execution of code. We'll dive into this in the next module. But first, let's look back at what we've covered in this module. We started by introducing the concepts of threads and processes, and we talked about to create and manage threads in Python. We then dove deeper into thread components, thread states, and how the operating system schedules threads. We discussed synchronizing multithreaded access to shared state using mechanisms such as locks, semaphores, events, and conditions, and then we looked at using queues to simplify inter-thread communication. Our final topic in this module was the global interpreter lock, which gives us the motivation for our next module, Multiprocessing.

Multiprocessing

Processes vs. Threads

In the last module, we introduced the concept of processes. We defined a process as a running instance of a computer program, and we mentioned that every running instance, meaning every process, has a processor state, security attributes, a section of memory assigned by the operating system, and possibly some system resources. We also mentioned that a process can be composed of one or more threads. It is important to know that while threads within a process automatically share the memory assigned to the process, processes do not share their memory space with other processes. There is an exception to this rule, however. A process can explicitly ask the operating system for a segment of memory that can be shared with other processes, but again, that's the exception and requires special system calls. In running Python programs, there is one interpreter per process, and since the GIL is a lock on the interpreter, there is one GIL for every Python process. With this in mind, it's an easy logical step to say that if we want to get around the GIL blocking multiple instructions from running simultaneously on different cores, then we can simply create copies of the process. Each separate process will have its own GIL, and can therefore run concurrently with other processes on different processor cores. In addition to the benefit of being able to side step the GIL and concurrently run on multiple cores, there are a couple of other advantages to using multiprocessing in Python. The separate memory space means that there is less need for synchronization primitives when using multiprocessing. Like I mentioned earlier, there's a way for processes to share memory with other processes, and when that happens, then it becomes necessary to synchronize access to shared memory. But in the absence of that, code making use of multiprocessing can look simpler and more straightforward once you take away the concerns of synchronization. Another benefit is that processes are interruptible and killable using OS-provided APIs, where as threads aren't. And lastly, since processes are insulated from each other by the OS, an error in one process cannot bring down another process. This is not to say that multiprocessing is always the better solution. Processes have a higher memory footprint, and context switches are more expensive. However in certain cases, like in parallelizing CPU-intensive workloads, multiprocessing is invaluable.

The Multiprocessing API

The Python multiprocessing API was designed to be similar to the threading API, so that in some cases, switching between the two is as simple as changing one or two lines of code. As a result, creating a new process is very similar to creating a new thread. Here's our previous example where we created a new thread. To use a process to execute the do_some_work method concurrently instead of a thread, all we need to do is important multiprocessing instead of threading, and construct a multiprocessing that processes objects instead. You'll also notice that in line 6, we have this extra check for _main. We'll get back to why this is here in a few minutes, but first let's take a look at the steps needed to spawn the process. We start in line 8 by instantiating an object of multiprocessing. Process. Here we pass in our target function and any optional arguments needed. Arguments passed to the process constructor must be picklable. Pickling is process whereby a Python object is converted into a byte stream, and unpickling is the reverse operation, where by a byte stream is converted back into an object hierarchy. In some languages, this is known as object serialization and deserialization or object martialing and unmartialing. Pickable objects include types such as none, Booleans, numbers, strings, byte arrays, collections to pickable objects, top-level functions, and classes whose attributes are pickable. After instantiating our process object, we call start on it to kick it off, and then we can call join on the object to block until the process completes its work and exits. The end results of this code is that our main Python process creates a worker Python process which executes to do_some_work method and prints out the two strings. The child process then exits, which allows the main process to exit. Now let's talk about the necessity of the if_name block in our example. When the child process gets started, it needs to import the script containing the target function. If the code for creating the new process in the top level of the script, it gets executed during the import, which means that a new process is created, which in turn tries to import the script, causing new processes to keep getting recursively created until a runtime error occurs. Putting the code for creating the new process in the _main block ensures that it only gets run when the script is executed, and not when it's imported. Here's the full signature of the multiprocessing. Process constrictor. As you can see, it's fully identical to the threading. Thread constructor, so much so that the group parameter exists solely for compatibility with the threading. Thread constructor. We have the target parameter, which is the function to be invoked, The name parameter, which sets the process, args is the argument tuple for the target invocation, and kwargs is a dictionary of keyword arguments for the target invocation. The daemon argument sets the process daemon flag to true or false. If none the default, this flag will be inherited from the creating process. The difference between a daemon process and a non-daemon process is that when a process exits, it terminates all its daemon child processes. If a process has child processes that are not daemon processes, the process will by default not exit until all its non-daemon child processes have exited. Therefore, in situations where we want a background process that runs without blocking the main program from exiting, setting the child process as a daemon will come in handy. It's important to note that a daemon child process is not allowed to create its own child processes, so keep that in mind when determining whether to flag a child process as a daemon or not. As we mentioned before, one of the advantages of using processes for concurrency is the fact that unlike threads, processes are killable via an operating system-provided API. In Python, two API calls are used to manage the aliveness of a process, the is_alive method and the terminate method. The code below shows the methods in action. In line 10, we define the process, but we have not started it yet, so the print statement on line 11 will print out an is_alive value of false. After starting the process in line 12, the next print statement prints an is_alive value of true. In line 14, we're sending the kill signal by calling the terminate method, and then waiting for the process to terminate by calling join on the process object, which means that the is_alive value in line 16 will be false. From line 17, you'll notice that we can find out the exit code of our process by reading the exit code attribute. A process with an exit code of 0 means that no error occurred. A code of greater than 0 means that the process had an error and exited with that code, and a code of less than 0 means that the process was killed with a signal of -1 multiplied by the signal code. While the option of forcibly killing a child process by calling terminate is available in Python, the better alternative will be to use the poison pill method. However, if you do decide to go down a terminate route, there are a couple of things that are important to note. One is that if the child process uses any shared resources such as locks, semaphores, pipes, and queues, and gets forcibly killed, those resources may be put in an inconsistent state, and therefore made unusable to other processes. Therefore it's advisable to only consider using process. terminate on processes which don't use any shared resources. The other thing to note is that on a forcibly killed process, finally clauses and exit handlers will not get run. So if you have critical code in a finally clause or exit handler, then it's best not to use a terminate method.

Process Pools

An easy way to set up a fixed pool of worker processes that can accept and execute tasks in parallel is to use a process pool. This comes built into the Python standard library in the form of the multiprocessing. pool class. When creating a multiprocessing. pool object, you can specify the number of worker processes. If you leave the parameter set to none, then by default the number of worker processes will be set to the number of CPU cores available on the machine at the time. You can also optionally specify an initializer function and initialization args. If set, each worker process executes the initialization function once at startup. An interesting side note concerning the initalizer function in args is that they don't have to be pickable. This makes it useful as a workaround for when you need to pass a non-pickable object to the worker processes. The final constructor parameter is maxtasksperchild. By default, this is set to none, meaning that worker processes live as long as the pool is alive. However, if the value is set, then after executing the specified number of tasks, a worker process is killed and replaced with a new worker process. This ensures that a long-running worker process periodically has to release any system resources it holds. The most common usage pattern for using a process pool is to define a function to be executed, an iterable of items that serve as the function argument, and then using the pool. map method to apply the function to each value in parallel. Here's an example. Here we start by inputting multiprocessing, and then we define our target function do_work. I've also created a function start_process, which will serve as our initializer function. Our initializer function takes in no parameters, since it simply prints the current processes name. We can access the process object for the currently-running process by calling the multiprocessing. current_process function, and once we have a reference to the process object, the name attribute, it gives us the process name. In line 10, we get the number of CPU cores by calling the cpu_count method, and use it to define the number of processes that we want to have in the pool. We then instantiate our pool object, creating a list of integers, and call the pool. map method. The map method applies a target function to every item in the interable in parallel by chopping up the iterable into chunks and submitting the chunks, as well as the target function to the process pool. Therefore, both the target function and the iterable must be pickable. The map method is a blocking call, and upon completion, it returns a list of the results in the original order of the iterable. There is a non-blocking version of map called map_async, which also allows you to specify an optional callback function, which can act on the results of each function call as they become available. For now, we'll continue to focus on the synchronous map method. In line 17, we can call close on the process pool, which prevents any more tasks from being submitted to the pool and causes the worker threads to begin exiting. And then we call pool. join, which blocks the main process until all the worker processes in the pool have exited. You should note that the join method can only be called after calling pool. close or pool. terminate. Let's put this into practice in our thumbnail_maker example. We'll use a process pool to execute the more CPU-heavy perform_resizing method in parallel, but I'd like to do this in a branch, so that I can come back to our code in its current state. So in order to do this, I'll commit the work I currently have, give it a commit message, and then I'll create and check out my new branch. Now I can start coding in this branch. The goal I want to achieve is that instead of using a thread to perform the resize operation, I want to use a process pool. So I'm going to start by importing multiprocessing. And then I'll go ahead and remove t2, which is the thread responsible for reading the filenames from the img_queue and performing the resize, and I'll also delete the code where I put the poison pill into the img_queue. Then I'll go ahead and create my process pool object. I'm not processing any arguments to the pool constructor, because I'm fine with the default value for the number of processes. I also don't have any special initialization function that needs to be run. To use pool. map to run the resizing operation in parallel, I need the function to apply and the iterable of items that the function will work on. So first, I'll create the target function, I'll call it resize_image, and I'll just copy the interesting parts of the perform_resizing method. I'll also need to copy over the target_size list for the function to be complete. Next I need the iterable of function arguments. This will be the sequence of filenames that we will need to apply the resize_image function to. Currently, when a thread downloads an image, it puts the filename into the img_queue, what we want to do is to add it to a list instead, so that it can be used as an input for the pool. map method. So we'll delete self. img_queue, and in its place create img_list, and when the download is completed, we should append to img_list instead of putting to img_queue. We can now write our pool. map statement. There is no return value in this case, so we won't be storing any results into a list. We can close and join here, and we are done, or are we? If I run this code, I will encounter an error, and the reason for this error is that both the target function and the iterable arg must be picklable. Here our iterable is picklable, but our function isn't, because in this case, Python needs to serialize what is the function definition, but also all the attributes of the thumbnail_maker service object, one of which is the dl_queue. Unfortunately, dl_queue is an object of threading. queue, and threading. queue objects are not picklable. To work around this, we will need to remove the dl_queue from the thumbnail_maker service object, and instead pass it into the download_image method. This means that I have to also remove the reference to self for the dl_queue since it's now a local variable. In my make_thumbnails method, I need to create the dl_queue, and then I need to pass the dl_queue as an argument in the threading constructor. And also here as well, we'll delete the self keyword in these two places, since dl_queue is a local variable. Now our pickling problem should be solved, and we shouldn't have any issues pickling the target function for our pool. map. But before I run this, I want to do a few housekeeping things. I don't want process spinup time and shutdown time to be included in our performance measurement, so I'm going to move them out of the perf_counter context. I also want to find out how quickly the perform_resize method runs, so I'll create timers for that, start_resize and stop_resize. And then I'm going to log how long it takes. Now I'm ready to run. And we can see that the entire operation completes in 9. 5 seconds, while the resize operation completes in 1. 5

seconds. Our initial performance measurements for the resize operation was 3. 19 seconds. So we achieved a 2x increase in performance by parallelizing the resize using the processor pool. In addition to the pool. map method, there's the map_async method like I mentioned earlier. The map_async method allows you to specify a callback, which gets called for every completed function execution with the results. The map_async method is also non-blocking, therefore instead of pausing execution until the map operation is complete, it immediately returns an async result object, allowing the call to store a reference to the async result object, and then continue doing other things. When the caller needs the result of the map operation, they call the get method on the result object, which would then fetch the results and block only if the results are not immediately available. The pool class also has the apply and apply_async method, which allows you to pass a function and its args to be executed on one of the workers in the process pool, whereas pool. map limits you to applying only a single function on a iterable of arguments. With apply and apply_async, you can send any number of different functions and arguments to the process pool. In this example, we send the multiply function and a tuple containing the args to the process pool to run on one of the processes in the pool. Since we're using the async variant of the apply method, the results are returned immediately and can only be fetched when we call result. get.

Inter-process Communication

As we mentioned earlier, by default, processes do not share memory space with each other. This means unlike threads, sharing of data between processes is more involved than one process, writing the data to memory, and the other process reading the data from memory in a synchronized manner. Instead, processes must use OS-supported communication channels if they want to exchange data with each other. The two communication channels implemented in Python are pipes and queues. The multiprocessing pipe involves two connection objects, which represent two ends of a pipe. A process can write to one of the pipe, while another process reads from the other end of the pipe, and vice versa. By default, the pipe is duplex, which means that the pipe is bidirectional, but you can specify that you only want a unidirectional pipe. In our example, we have two methods that will be executed by two different processes. In line 15, we create our pipe and specify that it should be bidirectional, which it is by default. We get back a tuple of connection objects, which represents both ends of the pipe, as we pass one end of the pipe to one of the processes, and the other end to the other process. In the make_tuple method, we are creating a tuple which has a string and a randomly-generated integer between 1 and 9. We then send that tuple into the pipe using the send method, and then listen for a response using the receive method. The receive method blocks until a message is received, and once the message arrives, it prints it. The make_string process is on the other end of the pipe listening for the tuple. Once it receives it, it creates a string, which is composed of the string in the tuple replicated by the integer in the tuple. When it's done with its computation, it sends a response back to the process of the other side of the pipe. The pipe is fairly simple construct with no built-in locking or consistency guarantees. Two processes trying to write to the same end of the pipe can cause data in the pipe to become corrupted. For this and other reasons, the multiprocessing queue is the more common method of interprocess communication. The multiprocessing queue uses a pipe, as well as a few locks and semaphores behind the scenes for process safety. It also implements all the methods of standard library's queue class, with the exception of the task_done and join methods. Even the queue that. empty and queue. full exceptions are used to raise timeout exceptions in reading and writing to the queue. There is a type of multiprocessing queue called a JoinableQueue that also implements the task_done and join methods for 100% API compatibility with the standard library queue. Therefore the JoinableQueue can be used as a drop and replacement for the standard queue. Both the queue and the JoinableQueue are multi-producer, multi-consumer queues, which means that multiple processes can read from and write to the queue at the same time without data corruption occurring. To illustrate the use of a queue, let's go back to our earlier example. We can replace the pipe with a queue. Instead of the send and receive methods, we have the standard get and put methods for getting a message from the queue and putting a message into the queue. As with the standard library queue, when the get is called, the multiprocessing queue by default blocks until an item is available to get from the queue. Here's a full get method signature. If the block flag is set to false, then instead of blocking, the get call returns immediately with the retrieved item or throws the queue that empty exception if there is no available item in the queue. For blocking calls, a timeout can also be specified. The put method also has the optional block and timeout parameter. When put is called on a queue that has a max size set, if there is no free slot in the queue, the put call blocks until one becomes available or until the timeout expires if one is was specified. When the timeout expires, a queue full exception gets thrown. If the blocked flag is set to false instead, if a full queue is encountered, the put call immediately throws the queue that full exception. Let's take a look at how we can use inter-processing communication in our thumbnail_maker application. In the current state of our code, we are collecting all the downloads into a list, and then using a process pool to execute resize operations on multiple processes. The process pool uses a type of multiprocessing queue under the covers, but to illustrate interprocess communication using queues, we'll back off using the process pool and implement the queuing ourselves. So let's start by communicating the current branch, and then go back to the previous state of our code. Now what I'd like to do is after the download threads downloads each image, they should put the filename into a multiprocessing queue instead of a standard library queue. Then on the other side, we'll have a number of processes reading items off the queue and performing the resize. So let's get started. First I need to import multiprocessing and change image_queue from being a queue. Queue object to being an object of multiprocessing. JoinableQueue. I'm going to use a joinableQueue instead of the regular multiprocessing queue, because the joinableQueue has 100% compatibility with the standard queue and can be used as a drop and replacement. So when using the joinableQueue, I'm able to made as few code changes as possible. Now all I have to do is in the make_thumbnails method, change the thread here to an instance of multiprocessing. process. I want more than one process performing the resize though, so I'll specify a number of processes which will be equal to the number of CPU cores, and then loop that number of times, so that we're now creating multiple processes that will read from the queue and do the resizing. Now here we're only putting one poison pill into the image_queue, which means that only one of our worker processes will terminate. So to get all of our worker processes to terminate, we need to put as many poison pills as there are workers. We're almost ready to run this code. But first, since we're going to be passing the resize function to separate processes to execute it, it must be picklable, so I need to put back in the code that we had earlier to ensure that the function can be pickled, which entailed removing the download_queue from the thumbnail_maker object, and instead passing it into the download_image method. And now we can run the code. So what we now have is our two operations, download_image, which is IO bound, and perform_resize, which is CPU bound. Our IO-bound operation is being parallelized using multithreading, and the CPU-bound operation is being parallelized using multiprocessing with queues for communicating between the different actors. Doing so, we've managed to improve performance by a factor of almost 5x, and still kept our code fairly clean and readable. Nicely done. We've accomplished one of the major milestones of this course, but there is still a number of exciting Python features to examine like the asyncio library and the ability to share state between processes, even potentially across machines.

Sharing State Between Processes

Before talking about sharing state between processes, I should reiterate that as much as possible, it is best to keep shared state to a minimum, because manipulating shared state can be error prone and require a lot of complicated synchronization steps. Queues and other message-passing mechanisms are great ways to design systems that minimize or eliminate state sharing. However, in cases where program design works best by sharing state, Python makes it possible to have shared state even between processes. The two options for sharing state between processes are shared memory and the manager process. If you only need to share a single variable or array between processes, then shared memory is the way to go. The multiprocessing module provides the value and array-shape objects, which wrap an area of shared memory and can be inherited by the child processes. To create a value object, we call the multiprocessing. Value constructor, passing in the ctypes typecode, more on that later, and the initial value of the object, and optionally a lock argument that specifies whether or not access to the value should be synchronized by a lock. By default, the lock argument is set to true, which means that an RLock is created for synchronizing access to the value. If set to false, then no lock is created, and no synchronization is done. You can also use this argument to pass a lock or RLock object to be used for the synchronization. So what's a ctype? Ctypes are c-compatible data types that are necessary for interacting with C libraries from within the Python code. Here's a table showing the ctypes and their corresponding Python type. Because the multiprocessing value and array objects are ctype objects that are allocated from shared memory, the ctype or typecode needs to specified in the constructor. Here's an example of how to create a shared value object. For the counter variable, we're initializing it to a shared object of type int. She char I is a typecode for the int ctype. We also haven't specified an initial value, so it defaults to 0, and also since we don't specify the lock variable, it defaults to synchronized. The next example is a shared Boolean. Here we specify the ctype in full and set the initial value to false. We also specify that access to this object should not be synchronized. Keep in mind that the lock argument is a keywords-only argument. In our final example, instead of using the built-in lock, we specify the lock object that we want to use for synchronizing access to our shared object. This comes in handy if we want to use one lock for several objects. We'll take a look at an example of shared memory objects in action a little later. But first, let's take a look at using manager objects for shared state. A manager controls a server process, which maintains objects that can be shared amongst other processes. An example of shared objects are values, arrays, lists, dictionaries, namespaces, locks, semaphores, and more. Processes that access those shared objects do so using a proxy. To the process, it looks like they're interacting with the local object; however, any interactions they have with the proxy object is serialized to and from the server process. As a result of this, interactions with objects that are shared via a manager are slower than interactions with objects that are shared through a shared memory, because every action involves making a remote call. However, the benefit to using a manager for shared objects is that you can share much more complex types than just a single value or array. You can share lists, dictionaries, arbitrary objects, etc. Another benefit is that with manager processes, we can share objects across a network. The manager process may be on the same local machine as the process using its shared object or on a remote machine. As long as the proxy object knows how to connect to the manager process, it doesn't matter. The only difference would be the performance of your calls when the separate processes local versus remote. We won't discuss remote managers any further in this course, but it's good to know that such functionality is available to us when needed. Creating a manager object is simply a matter of calling the constructor. This spawns the server process that will maintain the shared objects. This server process is a child process of the process that spawns it. Therefore, when the parent process exits, or if the manager object goes out of scope and is garbage collected, the server process will be terminated. It is also important to remember that calling the manager constructor spawns a process, because if the line where the manager object is instantiated is in the top level of the module, it will get called recursively when the new process is spawned. So care must be taken to ensure that the object is instantiated in the _main block or in the method. Here's a complete list of the objects that can be shared via a manager as of the publishing of this course. On the data structure side we have Value, Array, Namespace, List, Dict, and Queue, and on the synchronization side we have Lock, RLOCK, BoundedSemaphore, Event, Condition, and Barrier. An example of how to use a manager is shown on the screen. In line 7, we create a manager object which starts the server process. We then call the dict method, which creates a dict object on the server process and returns a proxy for it. We then pass that dict proxy to our processes as one of the arguments of the target function. I'm using list comprehension here for the looping, but it's tangential to the example. A regular for loop function would function the exact same way. Now that we've learned about how to share state between processes, let's put it into practice. Let's imagine that we're pretty happy with the performance of our thumbnail_maker app, but now we're curious to know how much memory do we save by resizing the original image into three smaller images. To find this out, we know the total size of the files we downloaded and the total size of the resized files. Let's start by creating a variable for the downloaded file size, and we'll need a lock for this variable, since it will be updated in a download_image method by multiple threads simultaneously. So let's add lock to the import statement, and pass a lock object to said method. In here, I want to do a little bit of cleanup by creating an img_filepath variable so that I don't have to repeatedly compute it. Next, I'm going to update the dl_size variable within the context of the dl_size lock. I'll use os. path. getsize to get the size on disk of the image I just downloaded. So now we've computed the total size of the downloaded images. I'm going to go down to the perform_resize method and instantiate the dl_size lock, and pass it to the downloaded_image method. Next, we need to know the total size of the resized image. Since the resize operation takes place in different processes, we could either have each work process send the resize_img sizes back to the parent process using the queue, and then let the parent process aggregate the sizes, or we could use shared state and have each process update the shared_value object whenever it completes a resize operation. We're going to go with the later approach, the shared state approach. And now again we have two options to choose from, using shared memory or using a manager process. Since all we need is a single value and not a list or a dictionary, it's best to go with the shared memory object in this case. This is because the shared memory approach is more lightweight in that we don't need to spin up a whole new process to manage the memory, and shared memory access is also much faster. So let's create a shared value object that will be inherited by all the worker processes. It's of type int, it's initialized to 0, and it's synchronized by default. Then in my perform_resizing method that gets run by multiple processes simultaneously, I'll do the same clean up by creating an out_filepath variable. Now I'm going to update the shared value with the on disk size of the file I just resized. What about locking, you may ask. If multiple processes are updating the same shared variable at the same time, isn't there the potential for race conditions? Well, yes there is. By default, access to shared value objects are synchronized using an internal lock. In this case though where we have the plus equals, we have two operations, reading the value and then updating it. Both of those operations will be synchronized internally separately, so to ensure that they run as an atomic unit, we need to lock both operations, preferably using the same internal RLock that the shared value uses. We can get this lock by calling get_lock on the value object, so we can move this code into a with block. And now we're locking the update using the resize_size. value objects internal lock. The final thing we want to do is log the initial size of the downloads and the final size of the resized images, and we can go ahead and run this. And a few seconds later, we'll see that we downloaded 5. 6 MB of images, and after resizing, we now have 199 KB of images. Nicely done. We get to save disk space, and now we can transmit perfectly sized images to our users, wins all around. Now is a good as time as any to wrap up this module.

Process Synchronization

Before we wrap up the module, I'd like to bring up something that snuck in to our last example, and that's the use of a lock across multiple processes. In the following code snippet, we are using a lock returned by the get_lock call to synchronize access to the shared memory value across multiple processes. The reason this is possible is because get_lock does not return an object of type threading. lock. Instead, it returns an object of type multiprocessing. lock. While threading. lock can be used to synchronize access to a critical section of code by multiple threads, multiprocessing. lock is used to synchronize access to a critical section of code by multiple processes. And it's not just locks, the multiprocessing module contains process-enabled equivalents of all of the synchronization primitives in the threading module. Therefore, when synchronizing between processes, you should use the primitives for the multiprocessing module in place of the threading version. However, since processes have separate memory spaces from one another, ideally there should be very few situations other than synchronizing access to shared resources where significant amounts of interprocess synchronization is needed. Instead, queues or pipes should be used to pass messages between processes. So to recap what we've done in this module, we started out by looking at when and why you should use multiprocessing over multithreading, including a discussion on the benefits and drawbacks of multiprocessing. We then took a look at the multiprocessing API, it's similarities to the threading API, and how to manage processes. We took a look at the multiprocessing pool API, which makes it easier to distribute work over a pool of processes, and then we explored interprocess communication with queues and pipes. This is the preferred means of communication between processes as processes don't share memory by default, but sometimes this is necessary. So we looked at ways to share state between processes, which can be done either through the use of shard memory or a manager process, depending on the complexity of data that needs to be shared. Lastly, we briefly touched on synchronization between processes. We now two ways of executing code concurrently using threads or processes. As a result of the similarities between the two, we can begin to develop some abstractions that'll help us standardize the way we interact with threads or processes and deal with the results. This is what we'll look at in the next module.

Abstracting Concurrency

Abstracting Concurrency Mechanisms

Before we jump into asyncio, we need to take a quick look at the executing API and the future object that gets returned from it. We'll start by doing a recap of the concurrency mechanisms we've looked at so far. To create a thread to execute a task, we need to define the function that we want to run, and then create an instance of the thread class, passing in a target function and any arguments. Then we need to start the thread to begin execution. To wait for a function execution to complete, we can call join on the thread. If we want multiple threads to execute that task in parallel, then we need to create and start multiple threads in a loop. We'll also need to join on those objects in a separate loop in order to wait for execution to complete. The API for processes is similar, with the difference being that the task to be executed should be a CPU-bound task instead of an IO-bound task. Like the thread, we instantiate the process object, start it and join it to wait for the completion of the task. We can also have multiple processes concurrently executing the function, where we create and kick off multiple process objects and join on them in a separate loop. In both cases at a high level, what we are doing is defining a task, passing it to some sort of executor, and then getting the results back, or waiting for it to complete. For IO-bound tasks, the executor is a thread, and for CPU-bound tasks, the executor is a process. We can turn this high-level description into an interface. With this interface, all we need to do is instantiate an executor, submit the function to execute, and get back the results. For executing multiple tasks concurrently, we can have a map method that applies a target function to a iterable or iterable of arguments using multiple threads or processes in the background. What this buys is that instead of directly dealing with the underlying threads or processes, we simply submit tasks to the interface, and let the implementation manage the execution of the task. We don't need to worry anymore about instantiating the threads or processes, starting them and joining them, all this gets taken care of for us. Now if you have a program design where you need a finer grain control over the threads or processes that will be running your tasks, then the executor interface is not for you, and you should stick to the process and threading API. But if your needs are straightforward, then the executor interface makes it easier to parallelize operations. Another thing it buys us is that we could easily switch between threads and processes. Switching is simply a matter of re-specifying what type of executor you want to instantiate, whether going from a thread pool executor to process pool executor, or vice versa, but the rest of the API stays the same. Now in reality, this benefit may not be very practical if you've appropriately divided up your tasks into IO bound and CPU bound. Your CPU-bound tasks should run on processes, and there should rarely be a need to switch, and your IO-bound tasks should run on threads, and there should barely be a need to switch either. But there might be cases where your tasks mix IO and CPU, therefore you'd like to maintain the flexibility of running on either, depending on your platform, etc., or you might foresee a migration to a GIL-less interpreter, so you want to keep your options open. Using the executor interface will provide the benefit of making that switch easier. So now that we've introduced the why of the executor interface, let's dive into it and how to use it.

The Executor API

The executor API is provided by the concurrent. futures module, and it exposes only three methods, submit, map, and shutdown. The submit method schedules the passing function to run on one of the executor's workers and returns a future object that represents the execution state of the function. We'll talk more about the return future object a little later, but it's important to know that the call to submit is non-blocking. The submit method immediately returns the future object to the caller, so that the caller can continue executing and get the results of the computation at a later time by calling the results method on the returned future object. The map method uses the executor worker pool to apply the passed-in function to every member of the iterable or iterables concurrently. Each worker concurrently operates on an item from the iterable or a tuple of items from the iterables until all the items are processed. Therefore the degree of concurrency depends on the number of workers in the worker pool. The map function returns an iterator that can raise a timeout exception if a timeout is set and the value is not available by the timeout period. If the timeout is set to none, then there is no limit to the wait time. The chunksize parameter is only used by the process pool executor. It defines how to chop up the iterable into chunks when submitting the tasks to the workers in the pool. By default, the value is set to 1, which means that each item in the iterables get send individually to the worker on the task. But for larger iterables, there might be some efficiency gains that may be had by setting this value to a number larger than 1. The third method is the shutdown method. The shutdown method is used to signal to the executor that no more tasks will be submitted to it, and that it should free up any resources that it's currently using once the currently-running tasks, if any, are done executing. Once an execute is shut down, any subsequent attempts to call submit or map on it will result in a runtime error. The wait parameter specifies whether the shutdown method should block or not. If set to true, then the method will not return until any pending tasks have completed, and the executor pool has been shut down. If set to false, the shutdown method returns immediately, but it should be noted that your application will not exit until all the pending tasks are done executing. Like several other resources in Python, executor instances can be called within a with context, in which case it would be unnecessary to call the shutdown method, as it would automatically get called when leaving the with block. The executor class is an abstract class, so to use the executor API, we need to instantiate one of its concrete subclasses, ThreadPoolExecutor or ProcessPoolExecutor. For IO-bound tasks, we should instantiate a ThreadPoolExecutor. When we instantiate the ThreadPoolExecutor, we can specify the maximum number of works and the thread name prefix. the maximum number of workers determines, like the name implies, the maximum number of worker threads in the thread pool. If the value is set to none, then it will default to the number of cores in the machine, multiplied by 5. The thread name prefix allows you to specify the prefix for the thread names. This can make debugging easier when you print out the thread name in the application logs. In this example, we use a ThreadPoolExecutor to attempt to download two HTML pages concurrently. We start by importing the necessary modules, and then defining the load_url task, which opens a connection to the specified URL and downloads the HTML content in that URL. Then in a width block, we instantiate an object of the

ThreadPoolExecutor with a max_workers parameter of 2, and refer to it as executor. We then submit two tasks to the executor, one to download a legitimate web page, cnn. com, and another to download a page we know doesn't exist. The submitting of the tasks is non-blocking and returns a future object. So both tasks get submitted immediately one after the other. To get the results of the computation, we call the result method on the future object. This is where we actually get the downloaded bytes array returned by the load_url method, and refer to it using the data variable. If the download is not complete, then the call to result will block. We expect request 1 to succeed and request 2 to fail, because it's being made to a nonexistent URL. But notice that the try catch isn't around the submit, but rather it's around the result method. This is because exceptions that happen during the execution of submitted functions don't get raised to the caller during the submit or map operation. Instead, they get raised when the result method is called in the case of a submit method or when the resulting iterator is being iterated over in the case of the map method. The last thing I want to point out here is that because we make use of the width statement, we don't need to explicitly shut down the executor. This happens automatically for us when we exit the with block. For CPU-bound tasks, we should instantiate a ProcessPoolExecutor. The only parameter in the ProcessPoolExecutor constructor is the max_workers parameter, which determines the maximum number of worker processes in the pool. If this value is set to none, then by default it will be set o the number of processor cores on the machine. The ProcessPoolExecutor uses multiprocessing under the hood, which means that just like we discussed in the last module, the target function and the iterables must be picklabe. In this example, we're using a ProcessPoolExecutor to concurrently generate a hash of each input text. We start by importing the necessary modules. Next, we define a list of byte strings and our callable function generate_hash. The generate_hash function returns a sha2 hash of the passed in byte string. Then we create a ProcessPoolExecutor instance in a with block, and use the map method to run the generate_hash function on all the items in a text iterable concurrently using the ExecutorPoolWorkers. The resulting integrator is zipped together with a text iterable into a tuple, and then we print out the byte string and the hash. It's important to note that the instantiating of the ProcessPoolExecutor was done inside of a _main block, as opposed to the ThreadPoolExecutor, which didn't need to be. As we discussed in the last module, this is done to protect your application from infinitely creating new ProcessPoolExecutor instances in the process of creating new worker processes until our runtime error occurs. When a newly forked process importing the script from the parent process, it won't execute the code inside of the _main block. Therefore we avoid the infinite recurrent problem.

The Future Object

Let's talk in more detail about the future object, which is returned by the submit method. As an FYI, the map method, even though it doesn't return a future object, also uses an iterable of future objects internally in its implementation, so it's certainly a worthwhile exercise to take a few moments to understand what a future object is and how to use it. A future object is an object that acts as a proxy for a result that is initially unknown, typically because a computation of its value is not yet complete. In Python, the future object takes on the extra responsibility of encapsulating the execution state of the computation, allowing the developer to manage that state to some extent and be notified or perform some action when the computation completes by making use of callbacks. By being strictly non-blocking and returning a future object, the executor API drives us to think about programming asynchronously. In asynchronous program, we typically focus on the main thread. The actor that actually executes the task is concealed from us. The main thread simply submits the task to the actor, which may be another thread, process, or OS function, and then continues executing until it needs the result from the execution or until the execution is completed. In Python, the future object enables asynchronous programming. The executor represents the actor and immediately returns the future object so that the main thread is not blocked and can go on doing other things. When the main threads needs the result, then it calls the future. result method to get back the result of the function. If the function is not yet completed, then future. result will block until it completes or until a timeout occurs if one is specified. Also important to remember is that as I've mentioned earlier, if the execution of the task resulted in an exception, then when you call the future. result method, the exception will get raised at that time. The future object has a number of methods that allow you to manage the execution of a task. The cancel method allows you to attempt to cancel the execution of the task. If the function call is currently being executed and cannot be cancelled, then the method will return false; otherwise, the call will be cancelled and the method will return true. The done method is usually checked when the function call has completed execution or was successfully cancelled. In both cases, it returns true, otherwise it returns false. The exception method returns the exception raised by the call if any. If the execution is not yet completed and a time out is specified, then the method will wait until the time out expires. On expiry of the timeout, a timeout error is raised. If there are no exceptions raised by the task execution, then the method returns none. The add_done_callback method attaches a callback function to the future that will be executed on completion or cancellation. The callback function accepts only one argument, which is the future object. Multiple callback functions can be attached to a future object and will be called in the order of which they were added. If the future has already been completed or cancelled, the attached callback functions are called immediately. And if an exception is raised within a callback function, the exception is logged and ignored. When you have a collection of future objects, you may want to wait for all of them to complete. The concurrent. futures module has a function for this. The concurrent. futures wait function takes in an iterable of future objects and blocks until the futures are completed. The timeout parameter can be used to control how long to wait before returning. If it's set to none, then the wait time is unlimited. By default, the wait method waits until all futures are completed as specified by the return_when parameter, but the return_when can also be set to first completed, which then means that the method returns when the first future from the collection completes or is cancelled. If you don't want to wait for all the futures to complete but would rather process the results as they complete, then there is the as_completed function. The as_completed function takes a group of future objects and returns an iterator that yields futures as they complete. Any futures that are completed before the as_completed function was called will be yielded first. If set, the timeout parameter determines how long to wait for the completion of the futures. By default, it's set to none. So that was a quick look at the executor API and the future object. We started off talking about how the executor API abstracts away the concurrency mechanism and what that buys us. We then discussed how to use the executor API with the concrete ThreadPoolExecutor and ProcessPoolExecutor classes. And lastly we looked at the future object and how it enables asynchrony in Python. In the next module, we'll dive deeper into asynchronous programming as we'll be introduced to the latest member of the Python concurrency family, the asyncio module.

Asynchronous Programming

Introduction to Single Threaded Asynchrony

This single-threaded asynchronous model is a concurrency model in which a single thread achieves concurrency by interleaving the execution of multiple tasks. Typically, a single thread executes tasks sequentially. If you want to execute multiple tasks simultaneously, you need an array of threads or processes executing those tasks in parallel. With single-threaded asynchrony, we can achieve concurrency on a single thread by interleaving the execution of the tasks. While executing a task, let's say t1, a thread can pause the execution of that task and then work on another task, say t4, and then come back and resume t1 at a later time. Single-threaded asynchrony is most often applied to IO-bound tasks, because IO-bound tasks typically have a lot of idle time while waiting for an IO operation to complete. During that idle time, the thread can be working on another task and then come back to the original task after the IO operation is completed. Implementations of single-threaded asynchrony are typically based on event-driven architectures. An event-driven architecture is a software design that orchestrates behavior around the production, detection, and consumption of events, as well as reaction to those events. While event-driven architectures have existed for a long time, particularly in GUI development, two products are pushed for public understanding of the technique as a solution for general IO multiplexing, particularly on the server side, NGINX and NodeJS. NGINX ditched the traditional model for serving web requests. In the traditional model, there's a thread or process that's assigned to each connection. The issue with that is that any time the thread or process sends a message to the client, it would have to block for awhile waiting for the response. This results in a lot of waste, as threads and processes are heavy-weight system objects. NGINX chose to go with an event-driven model where there's only a single worker process for many connections. An event on the listen socket means that a new client is available, and the worker process responds by creating a new non-blocking connection socket. An event on the connection socket means that there is new data to read or write. Here the worker performs the appropriate response and moves on. The worker process never blocks, instead it immediately goes on to process other events, as a result, NGINX is able to scale to support hundreds of thousands of connections per worker process, and this caught people's attention. NodeJS expanded the popularity of server-side event-driven architecture by bringing it to the popular JavaScript language. With NodeJS, terms like asynchronous IO and event loop became a common part of web developer lingo. As NodeJS provides the tools for developers to build highly-scalable web servers using callbacks that signal the occurrence of events and completion of tasks. So what is the benefit that the single-threaded asynchronous model provides? In the traditional model of concurrency, IO-bound tasks are typically handled by multiple threads, with each thread handling a task. When those threads have nothing to do because they are blocked and waiting on IO, the operating system suspends the thread and picks up a thread that is ready for execution. This model works well up to a point, but doesn't scale well to handling thousands of IO operations because of the memory overhead and scheduling and switching costs associated with threads. A more efficient solution is to have one thread handle multiple IO-bound tasks. When the task needs to wait for some IO operation to complete, instead of blocking, the thread suspends the task and moves on to a task that is ready to execute. When the IO gets completed, the thread gets notified, and then it can resume the task that it suspended. If this task suspension and resumption sounds familiar, that's because it's a similar concept to what the operating system does with threads. The difference here is that we don't pay the higher memory cost for multiple threads, and we don't have the overhead of context switching. We can switch between tasks much quicker and more efficiently. In order to implement this asyncio model, most languages and frameworks turn to an event loop. In the simplest of terms, an event loop is responsible for taking items from an event queue and handling it. An event could be a change of state on a file when new data is available to read, a timeout occurring, some new data arriving on a socket, etc. The thread goes into a loop and checks for an event it needs to response to. Its response may include executing a callback or some other code that relied on the occurrence of the event. The code currently being executed may generate more events that need to be watched for. When that happens, the loop suspends execution of that code and continues executing other code until the event occurs. There are several ways of implementing the event loop and mechanism for pausing execution, being notified of the completion of the IO task, and then resuming execution. In the next section, we'll take a look at how Python implements the event loop and task execution.

Cooperative Multitasking with Event Loops and Coroutines

In NodeJS programming, the event loop is invisible to the programmer and is implemented within the VM execution engine and in the libev library. But in Python the event loop is explicit. The event loop in Python is responsible for scheduling and executing tasks and callbacks in response to IO events, system events, and application context changes. To get an instance of an event loop, we call the asyncio. get_event_loop method. This method returns an object of abstract_event_loop. As the name implies, abstract_event_loop is an abstract class which has concrete subclasses. The get_event loop determines the appropriate concrete implementation for the platform we're running on and returns it. However, the only methods which we concern ourselves with are those exposed in the abstract_event_loop class. After we get the event_loop instance, we can start it by calling the abstractEventLoop. run_forever method or the AbsractEventLoop. run_until_complete method. If we start an event loop by calling run_forever, then we can stop it by calling AbstractEventLoop. stop. This causes the event loop to exit at the next suitable opportunity. Once an event loop is in the stop state, then we can close it by calling close. This method closes a non-running event loop by clearing the queues and shutting down the executor. In the previous section of this module, I mentioned that the task running in the loop must be suspended when it encounters an IO operation or any other long-running operation that can be offloaded to another actor. While this happens implicitly in some other platforms, in Python the running task itself is responsible for suspending itself and yielding control to the caller so that the caller can run other tasks. When the IO operation completes, the call can then restore the task back to the state it was in before it suspended it and resume execution. This is called cooperative multitasking, and this is where coroutines come in. There are two constructs in Python call coroutines, and it's important to understand which one is being referred to when you hear or read the word coroutine. They are the coroutine function and the coroutine object. For now let's focus on the former. The coroutine function has a special function that can give up control to its caller without losing its state. Here's an example of a coroutine function. The addition of the async keyword turns the say_hello function into a coroutine function, which now makes it suitable for use within an event loop. We create our event_loop instance and then call run_until_complete, passing in the say_hello routine. Whilst the say_hello routine completes, the loop stops, and the loop. close executes. Not much is happening in this example, so let's add just a little bit extra. In this example, you're introduced to the await keyword. The await keyword tells Python to pause the execution of this coroutine at this point, and return control to the event loop until an event occurs. In this case, the event that we need to wait for is a timeout event that'll happen after one second. At this point, if there are other tasks schedule in event_loop, those tasks will start running. After 1 second, the event_loop will detect the timeout event, and then resume the delayed_hello coroutine where it left off, and print the word world. A coroutine can wait on events, return results when completed, or raise exceptions if needed. In Python 3. 4 where a native support for coroutines didn't yet exist, generators were used to implement coroutines, and instead of the await statement, we would use the yield from to transfer control back to the caller. The asyncio. coroutine decorator was also used to indicate that a function was a coroutine function, as opposed to the async keyword, which was introduced from Python 3. 5 on. We won't spend much time in this module on earlier implementations of asyncio, but it's interesting to know that many of the basic asyncio functionality from Python 3. 5 on are also available in Python 3. 4, howbeit in a less-polished fashion. Executing a coroutine function doesn't result in the execution of the instructions within the function block, rather executing a coroutine function returns something called a coroutine object. To execute the coroutine object, you need to wrap it in a future object, and pass it to a running event_loop. In our example, we are not explicitly wrapping the coroutine, but internally the event_loops run_until_complete method will notice that it's receiving a coroutine object and not a future object and will wrap it for us. Now I must mention that this future object is slightly different from the one we saw in the last module, but we'll discuss that in more detail in a few moments. Right now let's try to visualize the state changes and flow of control when executing the coroutine object. When executing the code on the screen, the event_loop starts running after the call to run_until_complete. It then puts the future that wraps the coroutine object into the pending state and begins executing the coroutine code. When it reaches the await instruction, the coroutine is suspended and control returns to the event_loop. A second later, the event_loop resumes execution of the coroutine. Once completed, the future is set to done and the loop is stopped. So now that we've seen how a coroutine works within an event_loop, let's talk about futures and explore other capabilities of coroutines.

More Asyncio Concepts

The asyncio module provides a future object that like the concurrent. futures future object can be used to manage the execution of a function, and from which we can retrieve the results once the function is done executing. Some of the methods both future objects have in common are the cancel method, which is used to cancel the future, the done method, which returns true if the future is completed or canceled, the result method, which returns the result that the future represents, and exception method, which returns any exception that happened during the execution of the future, and the add_done_callback method, which adds a callback to be run when the future becomes done. The major difference between the two future classes is in the blocking behavior of the results and exception methods. The concurrent. futures future object is meant to be used with traditional concurrency, where calling result on a future object will block the calling thread until the result is available. But in the world of asyncio, almost all execution is handled within a single thread, therefore blocking is highly undesirable. Therefore calling the results method on the asyncio future object does not block. It gets the result and returns immediately. If the result is not yet available, because the computation is not yet complete, then the method raises an exception. The same goes for the exception method, which returns immediately or raises an exception if the future is not complete. If you're inside a coroutine, the right way to wait for an asyncio future to complete is to awake the future. The event_loop will pause execution at this point and watch for the future object to be set to done before resuming execution. If you're outside a coroutine, then you can pass the future to an event_loop using the run_until_complete method. In this case, the loop will watch for the completion of the future and then stop after the future gets done. A task is a subclass of future that is used to wrap and manage the execution of a coroutine in an event loop. A coroutine must be wrapped in a task before it can be run on the event_loop. The term task and future are often used interchangeably in documentation and sometimes in this course, because task is a type of future. However, when the term task is mentioned with reference to a Python type, it specifically refers to a class of future that is designed to wrap a coroutine. The ensure_future method is used to wrap a coroutine in a task. The ensure_future method takes in a coroutine or a future object, and if the object passed in is already a future object, it simply returns the object. But if the passed in object is a coroutine, then it creates a future object by calling creating_task on the passed in event_loop instance or the default_event_loop instance. You can also directly call create_task on an event_loop instance to create a task. In any case, creating a task using either the ensure_future or the create_task method has the important side effect of scheduling the task to run on an event_loop. In addition to being able to await future objects, coroutines can also await on other coroutines. This is called chaining coroutines. Coroutine chaining makes it easier to decompose a task into reusable parts. Let's take a look at an example. Here we have the perform_task coroutine that needs to call two other coroutines in order to complete its task. When await subtask1 is run, the perform_task coroutine is suspended, and the subtask1 coroutine is started. When subtask1 completes, the coroutine resumes and continues running until the it has to do the same thing with subtask2. So we see that coroutine chaining allows us to write clean, singularly-focused functions, and compose them together as part of a larger task. It also allows you to await separate sections of your code as needed, which can be really valuable.

Parallel Execution of Tasks

If you have multiple coroutines or tasks, and you want to wait for them all to complete, you can use the asyncio wait function. This function is similar to the concurrent. futures wait function, but it's designed to be used for tasks or coroutines that are running on the event_loop. Therefore it itself is a coroutine. It takes a list of tasks or coroutines, and notifies the event_loop when they're all complete, or when the optional timeout expires. Like the concurrent. futures wait function, the return value is two sets of futures, the done set and the pending set. Also, we can specify when to return, either when any of the tasks completes or raises an exception, or when all the tasks complete. The default is set to the latter. Here's an example of the asyncio wait method in action. Here we have the get_items coroutine, which executes four get_item coroutine calls in a loop and waits for them all to complete. Because we don't have a timeout and the return_when is defaulted to all_completed, the coroutine will only resume when all the tasks are completed, and the pending set will be empty, so we can just ignore it. We know that all the tasks are in the completed state, and we can call the result method without any exceptions being thrown. If we change the code a little bit, and set our timeout to 2 seconds, then we'll have a couple of tasks that aren't completed by then. At this point, we can wait again for the remaining tasks to complete or perform some other action as dictated by your business logic. But let's say what I want to do is cancel the unfinished tasks. I can do so by simply calling the cancel method on each task in the pending set. This ability to set a timeout and respond to tasks that aren't completed by the time the timeout expires is a benefit of using the wait function that you won't get with the gather function, which we'll discuss in a few moments. If you want to have that wait, but with a timeout functionality when you have only coroutine or task, then you can use the wait_for function. You can pass the function the task or coroutine that you want to await and the timeout. You could set the timeout to none in order to simply wait indefinitely until the task completes, but then you might as well just directly await the task. With this function, if the timeout expires before the task is completed, the task gets canceled, and an asyncio timeout error is raised. If instead of waiting for all the tasks to complete before reacting, you want to yield the tasks as they complete, you can use the asyncio. as_completed method. The as_completed method also lets you specify an optional timeout value. If a timeout occurs before all the tasks are done, then a timeout error is raised. With both the wait and the as_completed methods, the return value is a collection of futures. The asyncio. gather method differs in this regard. Instead of returning a collection of futures, the gather method returns a single future that aggregates the result from the past and futures. The results are aggregated in order of the original sequence, not necessarily in order of the results or rival, which is highly beneficial in cases where a business logic requires it. When the return_exceptions parameter is set to False, the gather function immediately propagates the first raised exception to the returned future. Otherwise exceptions in the tasks are treated the same as successful results and gathered in the result list. It should be noted that the gather function has no timeout parameter, and because it returns a single future representing the collection of results, you cannot cancel tasks individually. But if you don't care about these features, then the gather function is a simpler replacement for the wait function, We can take our previous example and replace the wait function with the gather function. And awaiting the future returned by the gather function directly returns the results, removing the need to call a result on individual futures. Note that I'm using the splat operator on the item_coros list to unpack the list into positional arguments for the gather function. Now that we've covered the concepts involved in asyncio, let's use it to do something useful, but first we'll need some library help.

Async IO Libraries

When running coroutines, we want to ensure that we don't block. Instead of blocking, we should yield control back to the event loop, so that it can do other tasks. This poses a challenge for most traditional Python libraries that perform IO. Most of these libraries are not designed to be run an event_loop and yield control, instead they block an IO. As a result, in order to perform the functions that these libraries provide, we need to find alternative libraries that support single-threaded asynchronous behavior. Unfortunately everyone's favorite HTTP client requests doesn't have support asyncio; however, there's an alternative library that does, the aiohttp library. The aiohttp library is an HTTP server and client library that was designed for asyncio. To install aiohttp, you need to run the command pip install aiohttp. Once you have the library installed, creating a server is as easy as creating an instance of web. application, creating a coroutine that responds to the get requests, and then registering that handler coroutine to be run whenever a get request is made to a particular path or paths. There is way more that can be done with the aiohttp library to implement an HTTP server. In fact, the aiohttp. web module provides an entire web framework that rivals Flask, NodeJS, or any other web framework. But exploring this further is beyond the scope of this course. If you're interested in exploring this further, the aiohttp documentation has a really good tutorial that will help you get going. What we're more interested in is the client-side user of the aiohttp library. We can use aiohttp to make HTTP requests in a non-blocking asyncio-compatible manner like in this example. In the coroutine called main, we created an instance of ClientSession using an asynchronous context manager. A asynchronous context manager is a context manager that is able to yield control within its enter and exit methods. The client session object is the main interface for making HTTP requests, and it has coroutine methods like get, post, put, and delete that correspond to the standard HTTP methods. After creating our session object, we called the fetch coroutine, passing in the session object and the URL to get. In our fetch coroutine, we invoke session. get, passing in the URL and use another asynchronous context manager to manage the response object. The response to text method is also a coroutine. So we await its completion, and then return the results to the main coroutine, which prints it. We'll put this into practice soon, so that we get to truly understand what goes on, but first let's get a look at another library that supports asyncio, aiofiles. The aiofiles module is an library that provides an asyncio-enabled alternative to Python's standard file API. The API is very similar to the standard file API, but with support for async and await keywords and constructs. Here's an example of a standard blocking file read versus a non-blocking asynchronous file read. As we can see, the aiofiles version allows you to use it as an asynchronous context manager, and the read method is a coroutine that is non-blocking and awaitable. But aside from that, the API still looks very familiar because of its similarity to its blocking counterpart. Just to illustrate that further, here's an example of a blocking file write using the standard file API versus the asynchronous file API. Installing the aiofiles library can also be done using pip. And now with these two libraries, aiohttp and aiofiles, we can implement an asyncio version of download image in our thumbnail_maker example. Let's switch over to the command prompt or a Git GUI if that's what you prefer, and then we can check out a new branch, which I'll call asyncio. This way we can safely experiment with asyncio and return to our main branch if we need to. For this project, I'm going to install the two libraries I mentioned earlier, so I'll add them to the requirements. txt file, and then run pip or pip3 install -r requirements. txt. Now I'm ready to code. First off, I'll import asyncio, aiohttp, and aiofiles. Now I need to turn my download_image method into a coroutine that takes in an aiohttp ClientSession object and a URL, and downloads the image asynchronously. So I'll add the async keyword and post-fix the method name with coro just to make it easier to identify. I'll also change the passed-in inputs. Now we still need the img_filename and img_filepath definitions, as well as the putting of the img_filename into the img_queue once the download is completed, but most of this code can go. For the dl_size calculation, because we're now in a position where we never modify the dl_size variable from another thread, I don't need to have a lock around it. As an aside, if I was accessing a shared resource that I needed to lock, I would want to use an asyncio lock instead of a threading lock. This is because a threading lock blocks the running thread when it can't acquire the lock, whereas the asyncio lock let's you yield from it. To do the download, we need to call session. get using an asynchronous context manager to manage the response. Now when the response arrives, we want to write it to the img_filepath, and to do so asynchronously, we need to use the aiofiles API in place of the standard file api. So I'm going to open the file in binary write mode using another async with block. And then I'll wait for the response. contents and write it to the file asynchronously. We now have our download file coroutine that downloads each image. Let's create another coroutine. I'll call it the download_images_coro that takes in the img_url list and calls the download_image_coroutine. All we're doing in here is creating the ClientSession object with an asynchronous context manager, and then we loop through the URL list, and call the downloads_image_coro coroutine, passing in the session we just created and the URL. We'll use the await keyword to chain the coroutine. Finally, in our download_images method, in place of looping through the images and doing the download, we'll create an event_loop and pass the download_images coroutine to the run_until_complete method. We should wrap this in a try final block, and close the event loop afterwards. This will execute the coroutine function, wrap the coroutine object in a task, and wait for the task to complete. Now let's go do some cleanup in our make_thumbnails method. We can remove all the queuing code, and simply call the download_images method. The problem here now is that with the old code, we had processes reading items off the queue as they were produced, but here the download_images method completes the downloads before the img_resizing starts. So to fix this, I'll move the invocation of the download_images method until after the processes have been kicked off. This way, as soon as an image is downloaded, one of the processes can pick it up off the queue and begin resizing. I'll also delete this queue. join here. Now we're ready to run this using pytest. So now what we have is that we're performing the IO-bound task of downloading the files asynchronously using a single thread instead of a pool of threads running in parallel. After each download is done, it gets written to a queue, what is a pool of processes waiting to retrieve the items and perform the CPU-bound task of resizing the images. Even though our single-threaded download in this instance is much slower than our pool of threads, it still out performs synchronously downloading the image by a factor of 2x on this machine. More importantly, it's less resource intensive than thread pooling and can better to scale to large numbers of downloads, like downloads in the thousands. If you want to speed this up even further, it's possible to run multiple threads, each with its own event loop, performing single-threaded asyncio, but that's a more advanced topic that won't be covered in this course. So now we've seen how aiohttp and aiofiles enables us to make HTTP requests and perform file IO in an asynchronous fashion. But what about other IO-heavy functions. If we need to, for instance, perform a MySQL query, are we forced to use a blocking MySQL library? Well, we don't have to. There's an asyncio-enabled MySQL library called aiomysql, as well as an aiopg for postgres, an aiocouchdb for Couch DB, and an aiocassandra for Cassandra, and it's not just databases. There's a growing list of libraries and frameworks that support asynchronous operation. The wiki page of the asyncio GitHub titled ThirdParty has a pretty comprehensive list. You should check this out if you need an alternative library for a task, but if there isn't an alternative library available for what you're trying to achieve, all is not lost. It is still possible to use those libraries in an application based on asyncio by using an executor from concurrent. futures to run the code in either a separate thread or process.

Combining Coroutines with Threads and Processes

Blocking in a coroutine function can be very harmful to the system. So if we need to call a blocking function or library, instead of calling it directly, we should delegate that call to an executor. The AbstractEventLoop class provides a method for doing so in an asyncio-friendly way called run_in_executor. The run_in_executor method is a coroutine, therefore it can run on the event_loop without blocking because it can be awaited. The run_in_executor method runs the passed-in blocking function in the passed_in_executor. If the executor is set to none, then by default a ThreadPoolExecutor is used with the default number of threads, which varies based on the number on the number of processor cores available on the machine. The args parameter is a sequence of positional arguments that gets passed in a function. If you need to use keyword args instead, you can use the functools. partial function to create a callable to pass it a run_in_executor. Here's an example of run_in_executor in action using a ThreadPoolExecutor. Here we have a blocking function aptly named blocking_function that we would like to call from within our coroutine. In the _main block, we create our ThreadPoolExecutor instance with three worker threads and our event_loop, and then run the _main coroutine in the event_loop, passing in the loop object and the executor instance. In the _main coroutine, we create six tasks that call the blocking function, and then we can await the completion of those tasks. Because we are calling the function using run_in_executor, the event_loop thread never gets blocked, as the execution gets delegated to the worker threads. And we can have an asyncio future that gets returned, which the event_loop can watch for the completion of the tasks. Once the tasks are complete, we print the results, and now the loop can shut down. The ProcessPoolExecutor works the same way. Here's an example where we use a ProcessPoolExecutor to run a single blocking function. Here the blocking function we're trying to call is the factorial function. We create our ProcessPoolExecutor instance, our event_loop, and we specify the number whose factorial we want to calculate. Like before, we pass all this to the _main coroutine, and in the _main coroutine, we perform the computation using the run_in_executor method. This delegates the function call to the executor instance, and gives us a future object that we can await. When the future is done, execution of the _main coroutine resumes. We print the message and exit the coroutine, allowing the loop to close. SO we see that we can use the run_in_executor method to allow us to run normally-blocking code, which can include not just our own code, but also external libraries inside of an asyncio event_loop.

Concurrency in Python

That concludes the asyncio topics we want to cover in this course. While there are a few things we didn't examine like asyncio synchronization primitives and the stream reader and writer APIs, what we did accomplish is that we learned what single-threaded asynchrony is and why it's beneficial for doing highly-scalable IO work. We also learned about how event loops and coroutines interacts, with coroutines yielding control back to the event loop when it has to wait for a long-running operation to complete, the event loop knowing to resume the coroutine after the operation is completed. We also talked about the terms future and task in the context of asyncio and examined coroutine chaining. We then looked at three different ways to execute IO tasks in parallel on the event loop, and aggregate the results of the tasks. We worked with the aiohttp and aiofiles libraries, two of the many libraries that provide awaitable APIs for performing IO-bound tasks. There's growing support for the asyncio constructs amongst Python libraries. But in cases of blocking a library or some other blocking function that we need to call, we looked at the run_in_executor function for delegating the call to an executor instance. Python is often thought of as a single-threaded language, but there are several avenues for executing tasks concurrently. The threading module allows us to spin up native operating system threads to execute multiple tasks concurrently. The threading API allows us to create thread objects, start them, and join on them. It also provides several synchronization and inter-thread communication mechanisms for when threads need to communicate and coordinate with each other, or for when multiple threads are mutating the same area of memory. But the current implementation of

CPyton has a GIL in order to make Python easier to implement and faster to run for single-threaded programs. And as a result of the GIL, threading is not suitable for CPU-bound tasks. So instead, we have the multiprocessing package. The multiprocessing package uses processes instead of threads as the actors of parallel execution. And the multiprocessing API tries to mimic the threading API as much as possible in order to reduce the amount of distance between the two, and to make switching easier. This is on display right down to the implementation of equivalent synchronization primitives. One of the major areas where there is a difference between the two actors is in the implementation of shared state. Threads automatically share memory with each other, but processes don't. So special accommodations must be made to allow for processes to share state, namely OS-shared memory areas and manager processes. The concurrent. futures module provides a layer of abstraction over both concurrency mechanisms and also introduces us to futures, which represent pending results and also allows us to manage the execution of computations. And lastly, the asyncio module brings single-threaded asynchronous programming into Python as a native first-class citizen of the language with new constructs such as the async and await keywords, asynchronous context managers, and much more. If you would like to go learn more about Python concurrency, the official Python documentation and the Python Module of the Week blob, which is available in book form as a book called The Python 3 Standard Library by Example, both have sections on concurrency and are great resources on the available APIs and how to use them. Aside from that, there are many books, each with videos and blog posts that you can dig into with confidence now that you've gotten started with Python concurrency. My name's Tim Ojo, and thanks for watching.

Course author

Tim Ojo

Tim Ojo is a software developer with a fondness for building scalable backend applications in Java, C#, Python, and Scala. He actively contributes to the developer community by blogging and...

Course info

Level	Beginner
Rating	★★★★⯪ (52)

My rating	★★★★★
Duration	2h 39m
Released	18 Oct 2017

Share course