SCREEN Presentation	**Commentary**
WEB PAGE https://curio.readthedocs.io/en/latest/	**INTRODUCTION TO CURIO** NOTES: Curio written by David Beazley David wrote several books: - Python Essential Reference - Python Cookbook, 3rd Edition (with Brian K. Jones) Curio is a library for performing concurrent I/O and common system programming tasks such as: - Launching subprocesses - Farming work out to thread pools - Farming work out to process pools Curio uses explicit async/await syntax introduced in Python 3.5 Curio uses programming abstractions such as: - Threads - Sockets - Files - Subprocesses - Locks - Signals - Queues This presentation is modeled closely after the tutorial introduction found in the web site shown.

SCREEN Presentation	Commentary
curio_demo_1.py	**Start Hello World** Here is a simple curio hello world program — a task that prints a simple countdown as we wait for our kid to put his or her shoes on: The simple countdown timer is based on the curio sleep rather than standard sleep timer. Comments: - The library **`curio`** is imported - Initiated by **`curio.run()`** - The **`def`** is preceded by **`async`** - `The` **`curio.sleep`** `is preceded by` **`await`** (Run demo) When we run it we see a countdown. Yes, some jolly fun to be sure. Curio is based around the idea of tasks. Tasks are defined as coroutines using **async** functions. To make a task execute, it must run inside the curio kernel. The run() function starts the kernel with an initial task. The kernel runs until there are no more tasks to complete.

e.

SCREEN Presentation	Commentary
curio_demo_2.py	**Add Async Task** Let’s add a few more tasks into the mix — in this case, a kid task. This program illustrates the process of creating and joining with tasks. Here, the parent() task uses the **curio.spawn()** coroutine to launch a new child task. After sleeping briefly, it then launches the **countdown()** task. The **join()** method is used to wait for a task to finish. In this example, the parent first joins with **countdown()** and then with **kid()** before trying to leave. Note that the kid task is sleeping for 1,000 seconds. (Run demo) At this point, the program appears hung. The child is busy for the next 1000 seconds, the parent is blocked on join() and nothing much seems to be happening–this is the mark of all good concurrent programs (hanging that is).

SCREEN Presentation	Commentary
curio_demo_3.py	**External Monitor** Let's change the last part of the program to run the kernel with the monitor enabled. We do this by adding to the **curio.run()**. curio.run(parent, with_monitor=True) (Run program. When program hangs, open the terminal window. Run **monitor**. Enter **ps**.) In the monitor, we can see a list of the active tasks. We can see that the parent is waiting to join and that the kid is sleeping. Actually, we'd like to know more about what's happening. We can get the stack trace of any task using the **where** command. (Enter **w 3** to show the stack trace of the third task and pause.) (Enter w 4 to show the stack trace of the fourth task and pause.) Actually, that kid is just being super annoying. Let's cancel his world. (Enter cancel 4 and show results in both the terminal window and in program output.) Not surprisingly, the parent sure didn't like having their child process abruptly killed out of nowhere like that. The **join()** method returned with a **TaskError** exception to indicate that some kind of problem occurred in the child.

SCREEN Presentation	Commentary

curio_demo_4.py	**Cancel Subtask**
	
	Debugging is an important feature of curio and by using the monitor, we see what's happening as tasks run. We can find out where tasks are blocked and we can cancel any task that we want. However, it's not necessary to do this in the monitor. Change the parent task to include a timeout and a cancellation request like this.
	
	`try:`
	`  await curio.timeout_after(10, kid_task.join)`
	`except curio.TaskTimeout:`
	`  # …`
	`  await kid_task.cancel()`
	`# …`
	
	(Run demo)
	
	When we run this version, the parent will wait 10 seconds for the child to join. If not, the child is forcefully cancelled. Problem solved. Now, if only real life were this easy.

SCREEN Presentation	Commentary

curio_demo_5.py	**Capture Cancel**
	
	Of course, all is not lost in the child. If desired, the child can catch the cancellation request and cleanup. For example:
	
	`try:`
	`    # …`
	`    await curio.sleep(1000)`
	`except curio.CancelledError as xcp:`
	`    # …`
	`    raise`
	
	(Run demo and show output)
	
	By now, we have the basic gist of the curio task model. We can create tasks, join tasks, and cancel tasks. Even if a task appears to be blocked for a long time, it can be cancelled by another task or a timeout. We have a lot of control over the environment.
	
	Note that this is cooperative - not preemptive - multitasking.

SCREEN Presentation	Commentary
curio_demo_6.py	**Task Groups** What kind of kid plays Minecraft alone? Of course, they're going to invite all of their school friends over. Change the kid() function like this: (Show friend function and modifications to the kid function to call it.) In this code, the kid creates a task group and spawns a collection of tasks into it. Now we've got a four-fold problem of tasks sitting around doing nothing useful. We'd think the parent might have a problem with a motley crew like this, but no. If we run the code again, we'll get output like this: (Run demo and show output) Carefully observe how all of those friends just magically went away. That's the defining feature of a TaskGroup. We can spawn tasks into a group and they will either all complete or they'll all get cancelled if any kind of error occurs. Either way, none of those tasks are executing when control-flow leaves the with-block. In this case, the cancellation of child() causes a cancellation to propagate to all of those friend tasks who promptly leave. Again, problem solved.

SCREEN Presentation	Commentary
curio_demo_7.py	**Task Synchronization** Although threads are not used to implement curio, we still might have to worry about task synchronization issues (e.g., if more than one task is working with mutable state). For this purpose, curio provides `Event`, `Lock`, `Semaphore`, and `Condition` objects. For example, let's introduce an event that makes the child wait for the parent's permission to start playing: (Show and run demo) All of the synchronization primitives work the same way that they do in the threading module. The main difference is that all operations must be prefaced by `await`. Thus, to set an event we use `await start_evt.set()` and to wait for an event we use `await start_evt.wait()`.
curio_demo_8.py	**Timeout Capture** All of the synchronization methods also support timeouts. So, if the kid wanted to be rather annoying, they could use a timeout to repeatedly nag like this: (Show modified kid function and run) So the parent still has control but the kid can be more aware of the parent's management if he or she wants to be.

SCREEN Presentation	**Commentary**
curio_demo_9.py	Signals What kind of screen-time obsessed helicopter parent lets their child and friends play Minecraft for a measly 5 seconds? Instead, let's have the parent allow the child to play as much as they want until a Unix signal arrives, indicating that it's time to go. Modify the code to wait for Control-C or a SIGTERM using a SignalEvent like this: (Show revisions to parent code and start running the program.) At this point, nothing is going to happen for awhile. The kids will play for the next 1000 seconds. However, if we press Control-C, we'll see the program initiate it's usual shutdown sequence: (Send the signal [Ctrl-C] and show the output.) In either case, we'll see the parent wake up, do the countdown and proceed to cancel the child. All the friends go home. Very good.

SCREEN Presentation	Commentary
curio_demo_10.py	**Signal Behavior** Signals are a weird affair though. Suppose that the parent discovers that the house is on fire and wants to get the kids out of there fast. As written, a SignalEvent captures the appropriate signal and sets a sticky flag. If the same signal comes in again, nothing much happens. In this code, the shutdown sequence would run to completion no matter how many times we hit Control-C. Everyone dies. Sadness. This problem is easily solved–just delete the event after we're done with it. Like this: (Show modified parent and run again) Run the program again. Now, quickly hit Control-C twice in a row. Boom! Minecraft dies instantly and everyone hurries their way out of there. We'll see the friends, the child, and the parent all making a hasty exit. (Send signal [Ctrl-C] twice and show the results.)

SCREEN Presentation	Commentary
curio_demo_11.py	**Number Crunching and Blocking Operations** Now, suppose for a moment that the kid has decided, for reasons unknown, that building the Millennium Falcon requires computing a sum of larger and larger Fibonacci numbers using an exponential algorithm like this: (Show fib function and modifications to the kid function.) If we run this version, we'll find that the entire kernel becomes unresponsive. For example, signals aren't caught and there appears to be no way to get control back. The problem here is that the kid is hogging the CPU and never yields. Important lesson: curio does not provide preemptive scheduling. If a task decides to compute large Fibonacci numbers or mine bitcoins, everything will block until it's done. Don't do that.

SCREEN Presentation	Commentary
curio_demo_12.py	**Sidestepping the GIL** If we know that work might take awhile, we can have it execute in a separate process. Change the code to use curio.run_in_process() like this: `await curio.run_in_process(fib,n)` In this version, the kernel remains fully responsive because the CPU intensive work is being carried out in a subprocess. We should be able to run the monitor, send the signal, and see the shutdown occur as before. (Run the program, pause, then send the signal [Ctrl-C].) This actually lets the fib function run in a separate core. (Run program again, this time showing the Activity Monitor before sending the signal.) The only limitation of which I am aware is that the task shoved into the separate core must be pickleable.

SCREEN Presentation	Commentary
curio_demo_13.py	**General Blocking Concern** The problem of blocking might also apply to other operations involving I/O. For example, accessing a database or calling out to other libraries. In fact, any I/O operation not preceded by an explicit `await` might block. If we know that blocking is possible, use the `curio.run_in_thread()` coroutine. For example, in this modified code, the kid decides to take a rest after computing each Fibonacci number. For reasons unknown, the kid is calling the blocking `time.sleep()`. To keep it from blocking all tasks and the entire kernel, we've got to constantly remind the kid to do it in a separate thread: (Show modified kid function, then run and signal.) Note: `time.sleep()` has only been used to illustrate blocking in an outside library. Presumably the kid would be doing some more useful here such as watching a famous Fibonacci YouTuber blather on about their pet pugs or something. Curio already has its own sleep function so if we really need to sleep, use that instead.