

Introduction to Asynchronous Python



Hello!

I'm Doug Farrell

I've been developing with Python for quite a while in a few different domains and industries.





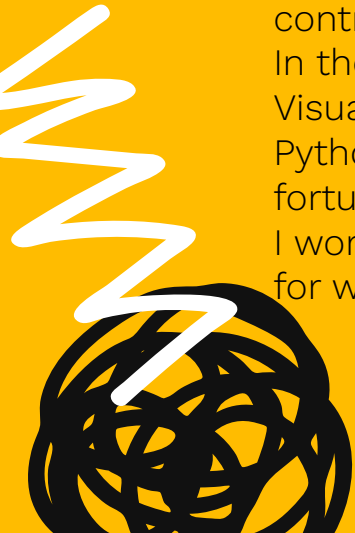
About Me

I've been developing professionally for a long time and got my start on a Radio Shack Color Computer. My BS is in Physics, which means I'm a completely self-taught developer.

I've worked in many industries; process control, embedded machine control, retail software and public Internet applications.

In those roles I've used quite a few languages as well; Fortran, C/C++, Visual Basic, PHP, JavaScript and Python. My personal favorite language is Python as it feels like it fits my brain and the way I think. I've been fortunate enough to use Python in my career for the last two decades.

I work at Cleo, an impactful company providing incredible health services for working families.





What we'll be talking about




Asynchronous World

We live in an asynchronous world and we react to it as people by reacting to events because we're relatively good task switchers.

We can delegate activities to run on their own and switch our personal context between them as needed.

We can drive a car, talk with the passenger and remember the next turn coming up in the route to take.

We can handle these activities because we're reacting to the events of each activity as they happen.



Asynchronous Misconceptions

Asynchronous behavior in a computer program is not parallelism, where things are actually happening at the same time on multiple CPUs.

It's more like task-switching concurrency, where multiple things are getting done, but not necessarily at the same time, and on a single CPU.

Parenting Analogy

You might be a parent, or certainly have parents, so you can relate to what we'll be using as an analogy to work our way towards asynchronous programming with Python.

Being a parent means handling many things at once; watching the kids, balancing the checkbook, moving the laundry to and from the washer and dryer, among lots of other stuff.

We're going to talk about "programming" a parent to handle these things as a way to see how well, or poorly, the different approaches work.



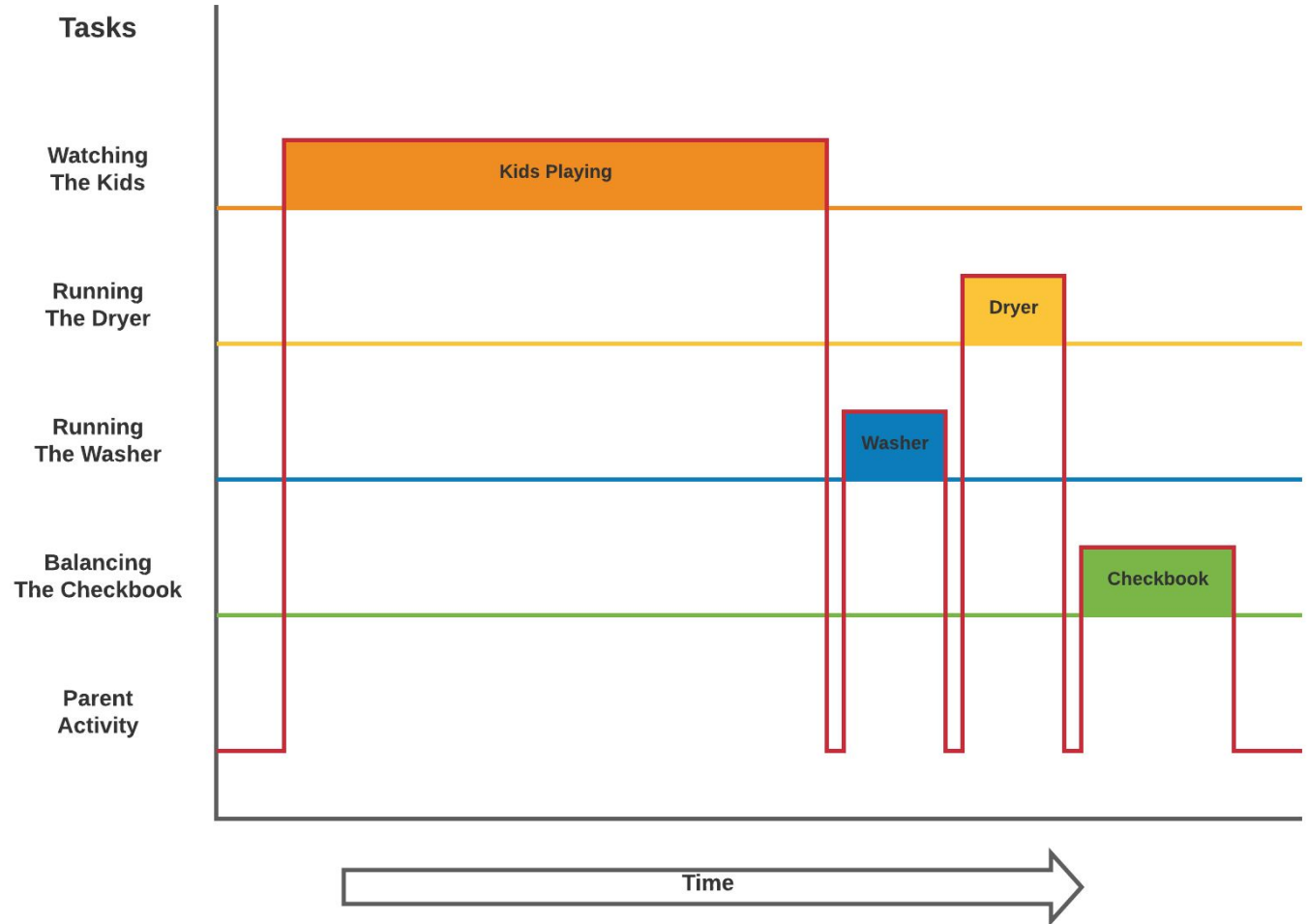


1.

The Synchronous Parent

This parent wants to get everything done, but has to stay with a task all the way through to completion. This means, as a good parent, minding the kids all the way to bedtime when something else can happen

Here we find the synchronous parent attending to various tasks. Because a task has to be completed before starting another, all the tasks are run serially, one after another.



Synchronous Results

From the execution timeline you can see performing the tasks serially makes for a very long day. Parents spend most of their time watching the kids.

Only after the kids go to bed is the parent able to start another task, but even then the serialized nature of this approach means waiting for tasks like running the washing machine to completion before starting the dryer. I think you can see that a few weeks of living like this and most parents would abandon ship.

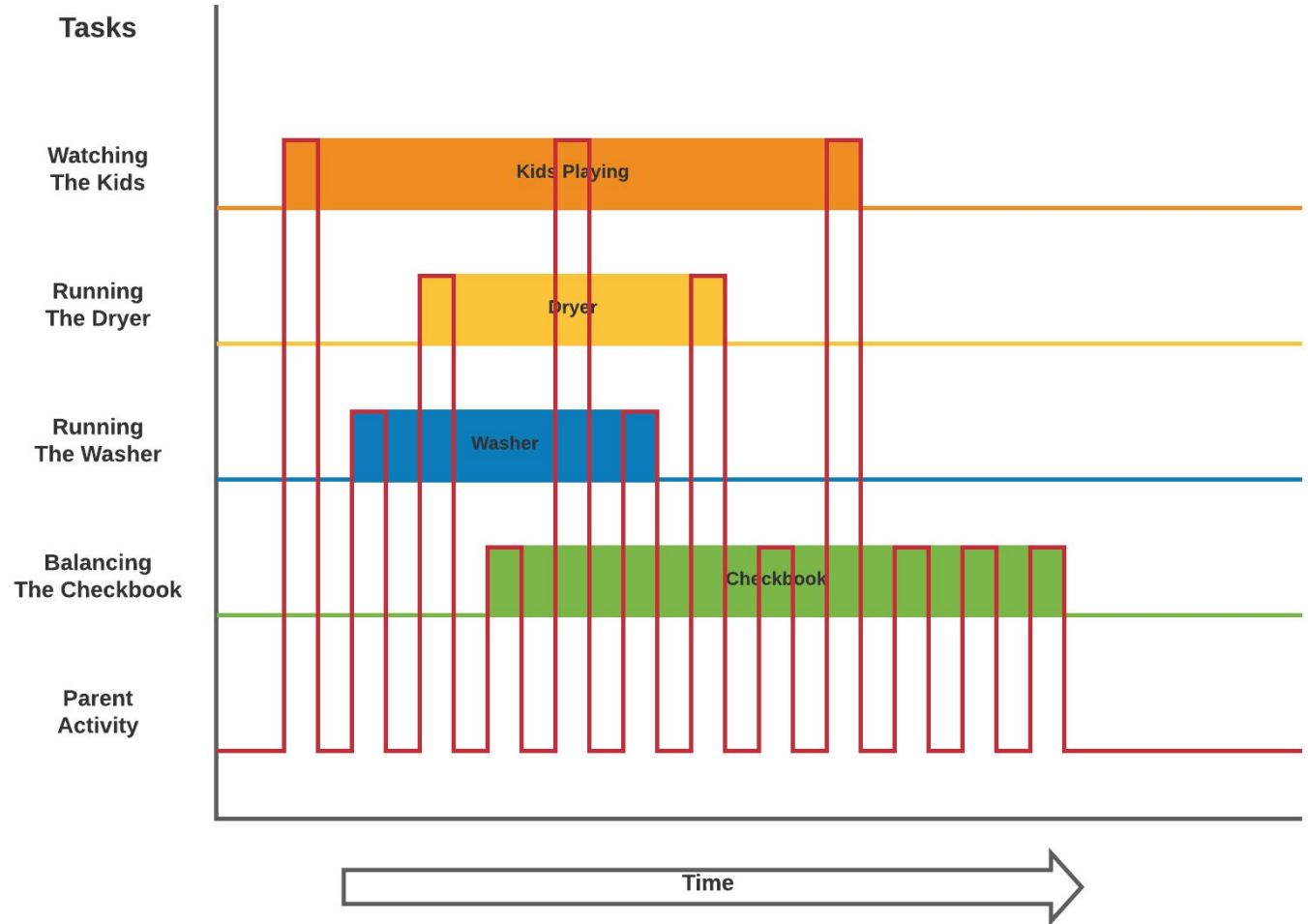


2.

The Polling Parent

This parent wants to get everything done too, and does so by breaking away from any current task on a regular basis to see if any other task needs attention.

The polling parent is able to move all the tasks towards completion, but the activity of polling makes each task taker longer. In addition the delay between activity on a task might be unacceptable, as with watching the kids playing.





Polling Results

The execution timeline shows that breaking away from any particular task on a periodic basis to work on another is a way of making progress on all the tasks.

However, this has some limitations. Depending on the polling frequency a significant amount of time is spent waiting for tasks to complete.

Worse, priority tasks like the kids might not get attention for a multiple of the polling frequency. This fails if a child is hurt and needs attention.

This problem could be partially addressed by increasing the polling frequency, but then more time is spent context switching between tasks.

Again, a couple weeks of this and we're back to the ship and abandonment options.



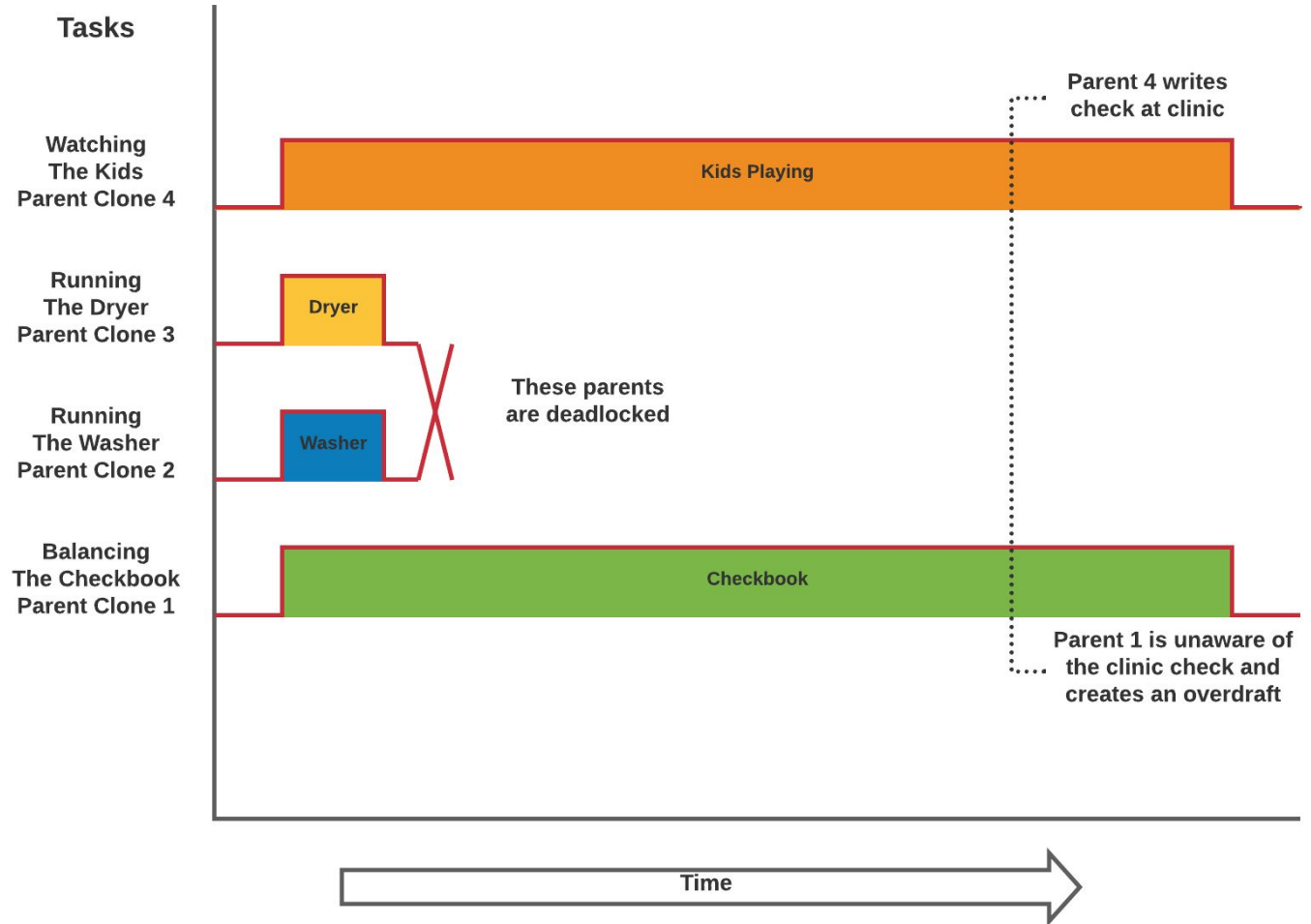


3.

The Threaded Parent

This parent wants to get everything done as well, and does so by realizing the dream of all parents; to clone themselves. Each cloned parent attends to a single task.

The threading model creates parent clones, each assigned a separate task. The parents need to talk with each other in order to avoid deadlocking on resources one or more parents want, and not change a shared resource.



Threading Results

The results of threading looks promising, all the tasks have the full attention of a parent clone and proceed to completion.

However, there are complexities that need to be handled. For instance, if the washing machine parent is using the washer and wants to acquire the dryer, and at the same time the dryer parent is using the dryer and wants to use acquire the washer. Unless handled, these two parents are deadlocked and both tasks are stalled.

Another possible issue is sharing a resource like the family budget. If a child is hurt and has to go to a clinic and that parent has to write a check, how does the checkbook balancing parent know about this and avoid an overdraft?

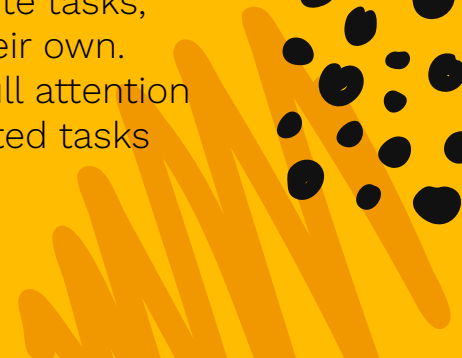



4.

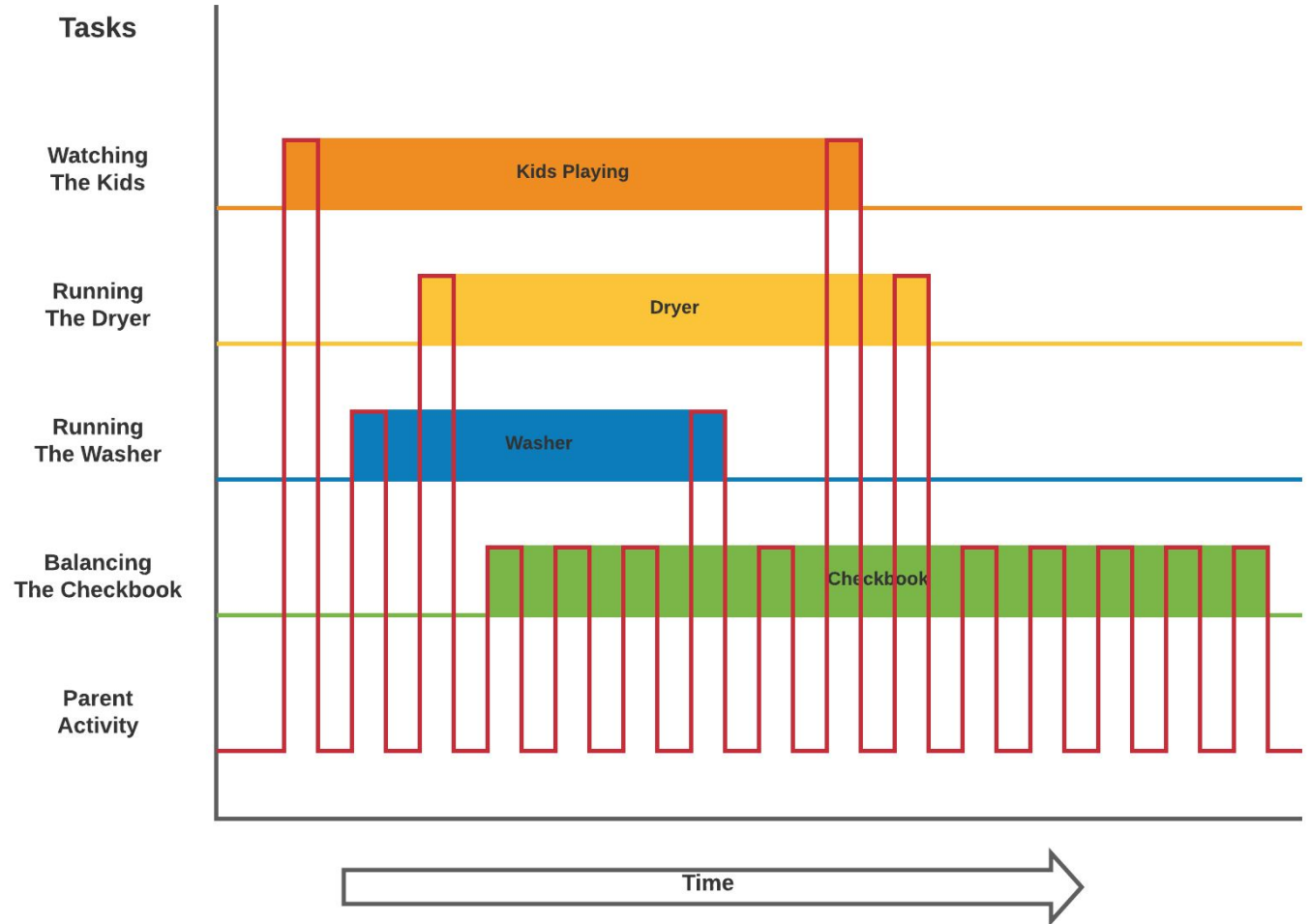
The Asynchronous Parent



Like the other parents, this parent wants to complete tasks, and does so by delegating tasks that can run on their own. This parent can then complete tasks that require full attention to move forward and respond to events the delegated tasks generate.



The asynchronous parent is able to initiate multiple tasks and then move on till those tasks send an event needing attention. The CPU bound task (the checkbook) explicitly cooperates with the event loop to act on the events from the other tasks.



Asynchronous Results

Based on the execution timeline the asynchronous parent handles the tasks pretty well, and models the behavior of a real parent closely. The success of this model is the ability to delegate tasks and respond to events they produce. The events are like the washer and dryer making a tone when the cycles are done, or like the kids getting louder because of something that needs attention. This behavior frees the parent to attend to the checkbook, a task that can't be delegated, but can be paused to break away and check for events.

Asynchronous Python

An asynchronous style of programming is possible in Python by taking advantage of the orders of magnitude differences between CPU and IO bound processing.

CPU bound tasks are like long running calculations, compression algorithms, code that requires the CPU to be executing instructions.

IO bound tasks are like making an HTTP request, reading/writing a file, code that makes a call to the OS, but the CPU has to wait for the IO to complete.

If IO processing can be delegated, and an event raised when the operation is complete, the CPU is freed to process application instructions that can't be delegated.

This is exactly what asynchronous Python applications do, by using the `asyncio` module to create an event loop and modules that support asynchronous IO operations that generate events.

This allows the application to create and manage many relatively long IO operations, while at the same time allowing the CPU to process application instructions.



Example Programs



The example programs follow a pattern that moves towards demonstrations of asynchronous programming with Python.

- * There is an `io_task` and `cpu_task` function that takes time to complete based on the parameter passed to it.
- * A queue contains the tasks to run, along with the parameters to pass to the tasks.
- * There are one or more worker functions that pull tasks from the queue and run them with the associated parameters. The worker(s) will continue to pull work from the task queue until it is empty.
- * The main function is the engine that runs the worker(s) until the task queue is empty.

The end goal is to show how asynchronous programming lets relatively long running IO tasks execute concurrently.

Example Programs 1

The example program contains a single worker pulling work from the queue and invoking a task to perform that work.

The program itself is structured kind of oddly considering what's being done, but this is intentional to build towards multiple workers cooperating to run concurrently.



Example Programs 2

The example program is based on example 1, but has the addition of a second worker. However, nothing has been added to the workers to make them cooperate, so only the first worker performs all the work.



Example Programs 3

The example program has been altered so the two workers can cooperate. This is accomplished by making the worker function a generator function. A generator function can yield control and maintain its state.

The code the generator function yields control to can give control back to the generator function so it can continue its processing.

In this way the worker can perform a context switch and allow the outside controlling code to grant control back to it and other generator functions. With two instances of the worker generator function context switching, they can perform work concurrently.

However, since the work, the `sleep()` function, still blocks the CPU from doing anything else, there's no net gain in efficiency or concurrency.

Example Programs 4

The example program has been altered to use the Python asyncio module. This provides two features the example program can take advantage of:

- To simulate the IO it can use the `asyncio.sleep()` functionality rather than the `time.sleep()`. The `asyncio.sleep()` function can yield control to an event loop.
- The hand-built context manager can be replaced with the `asyncio.run()` function for event handling.

By using the asyncio module the workers will run concurrently and program will run essentially twice as fast because the IO operations are happening asynchronously.

Example Programs 5

The example program has been altered to perform real IO work by downloading the content of a list of URLs. The program does this work using the Requests module.

It's also taken a step backwards because this work happens in a synchronous, non-cooperative manner as the Requests module blocks during IO operations.

Example Programs 6

The example program has been altered to perform real IO work by downloading the content of a list of URLs. It's also been changed to use the aiohttp module which works with asyncio and performs non-blocking, asynchronous IO operations.

With aiohttp integrated into the program, downloading the contents of the URLs happens concurrently and the program executes essentially twice as fast.

Example Programs 7

This is a bonus example program based on example 6. It adds reading files asynchronously to the work the tasks need to perform.

It uses the aiofiles module which works with the asyncio module to read files in an asynchronous manner and cooperate with other asyncio compatible functionality.



Example Programs

The example programs were developed and run in Python version 3.9.6.

You can clone the example program repository from here:

https://github.com/writeson/asynchronous_python_presentation





Asynchronous Use Cases

Pros

If your application is mixture of CPU and IO operations, it could benefit from using the asynchronous model

If your application needs to perform multiple IO operations simultaneously

It can be simpler to develop an asynchronous application over solutions that use polling or threading

Cons

If your application is primarily CPU intensive, there's little benefit using an asynchronous model

The asynchronous model doesn't take advantage of multiple CPU cores

It is more complex to develop an asynchronous application over a synchronous one



Thanks!

Any questions?

You can find me at

@writeson

doug.farrell@gmail.com

[linkedin.com/in/douglasfarrell/](https://www.linkedin.com/in/douglasfarrell/)





Credits



Special thanks to all the people who made and released these awesome resources for free:

- ✕ Presentation template by [SlidesCarnival](#)
 - ✕ Photographs by [Unsplash](#)
- 