We will explore:

* One issue you’ll probably run into (blocking)
* Quick look at some raw solutions
* Focus on a particular library that helps us

**Blocking**

Basics of blocking – waiting for input, or for a long-running task

* 01\_blocking
  + Shows basic example of IO blocking, where program cannot continue until input is there. This can be keyboard input, tailing a file, waiting for a network socket or Rest API call, etc.
* 02\_blocking
  + This one does work in a loop, so it is CPU blocking

The problem with blocking is that we often have tasks that will take a long time, whether they’re waiting for input (IO blocking), or for a long-running task to complete. If you have a data-intensive task, it is quite likely you’ll experience both kinds of blocking.

How do we get around this limitation?

* Somehow we need to have more than one line of code running at a time.
* OR, we need a way to pause one method, go run another method for a while, and then come back to where we were

**Threads**

Python, built-in, has some different ways of dealing with this issue. One of the most common is threading, but there is also multiprocessing and asyncio. Threading is the simplest, but it comes with certain limitations.

* 03\_threads
  + Setting up threads is pretty simple – we just need a method to call
  + So we see code running in parallel, and we can do multiple things at once!
  + So now we just need to solve the problem of getting the threads to talk to each other, right?
* 04\_threads
  + Not so fast. Let’s look at one where we use our old x+=1 task
  + Why is it taking so long? Python has a GIL – the Global Interpreter Lock

So we have some limitations. Threading also runs into issues of resource locking, signaling between threads.

**Other Python solutions**

Multiprocessing is somewhat like threading, but it takes things a step further – tasks are run in a separate process. This means it is not subject to Python’s GIL (technically, it has its own GIL since Python has one per process). So the operating system takes over the process of scheduling how the processes run next to each other – we can actually have multiple lines of code running at the same time! So this makes it good for CPU-bound tasks. This also means each piece of code has its own address space, making it harder to communicate.

We’re not going to focus on CPU-bound tasks tonight, but if you have that issue, Celery is one library that leverages a method of multiprocessing – it starts a process for each of its workers and provides some ways to track your tasks and communicate between them.

Asyncio is the new and different way to rock IO-bound ops, but it is REALLY HARD to get things going. You would have to build a lot of things to make the things happen that we’re about to look at.

**Circuits**

So it is really really hard to take a sequential program and deal with the issues of doing a task (like calling a web API), and continuing to execute while I’m waiting for the result. Let’s look at a different paradigm for dealing with some of these issues: event-driven code.

Easy to conceptualize – jQuery does this in Javascript – somebody clicks on something, and I have an event handler registered on the click event. The underlying work is already done for linking the event up to running the code. Is there something that does that same thing for us in Python?

* 01\_hello\_world
  + Explain components and events
    - Component is any object that has event handlers
    - I can define my own events
    - The method names match the event names
  + The started event fires when the component is fully registered and ready to go
  + App().run() creates an instance of the App component, and then runs it. This happens in our main thread.
  + The hello event fires, so we see the message. We’re done, right? But it doesn’t exit. Ctrl-C doesn’t work in Windows
    - Why isn’t it exiting? *Circuits is still running the component and waiting for events.*
    - Have to kill this from the task manager
    - Do “raise SystemExit(0)” after the fire
      * Why didn’t I get the Hello? *I fired the event, but it didn’t have a chance to execute before I terminated the app forcefully*
    - Do “self.stop()” after the fire
  + Make component composition with Bob and Fred doing Hello
    - **But do the Hello in the started event**
    - “(Bob() + Fred()).run()”
* 02\_components
  + What if I want only Bob to get the woof event?
    - Give Bob a bob channel, and update the event
    - Both still get it… because Fred has the ‘\*’ channel
  + Add a debugger, to show how everything is happening
* 03\_factorial
  + Shows that tasks can operate on separate threads or even processes
* Testing
  + When we are in a program where anything can happen at any time, how do we construct solid tests?
  + We’ll have to test each component separately
  + But how do we handle the event sequences?
  + Pytest example shows using fixtures to set up and run a component, and test results
* Additional examples
  + Server
  + Websockets – **run from the folder**