Deadlocks

One of the pitfalls to the concurrent.futures module is that you can accidentally create deadlocks when the caller to associate with a **Future** is also waiting on the results of another future. This sounds kind of confusing, so let's look at an example:

```
from concurrent.futures import ThreadPoolExecutor


def wait_forever():
    """
    This function will wait forever if there's only one
    thread assigned to the pool
    """
    my_future = executor.submit(zip, [1, 2, 3], [4, 5, 6])
    result = my_future.result()
    print(result)

if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=1)
    executor.submit(wait_forever)
```

Here we import the ThreadPoolExecutor class and create an instance of it. Take note that we set its maximum number of workers to one thread. Then we submit our function, **wait_forever**. Inside of our function, we submit another job to the thread pool that is supposed to zip two lists together, get the result of that operation and print it out. However we've just created a deadlock! The reason is that we are having one Future wait for another Future to finish. Basically we want a pending operation to wait on another pending operation which doesn't work very well.

Let's rewrite the code a bit to make it work:

```
from concurrent.futures import ThreadPoolExecutor


def wait_forever():
    """
```

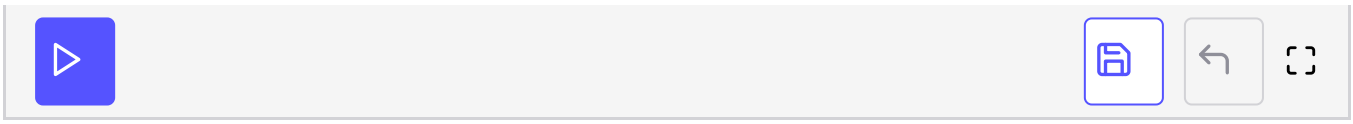
```
    This function will wait forever if there's only one
    thread assigned to the pool
    """

    my_future = executor.submit(zip, [1, 2, 3], [4, 5, 6])

    return my_future

if __name__ == '__main__':
    executor = ThreadPoolExecutor(max_workers=3)
    fut = executor.submit(wait_forever)

    result = fut.result()
    print(list(result.result()))
```



In this case, we just return the inner future from the function and then ask for its result. The result of calling **result** on our returned future is another future that actually contains the result we want, which is a bit confusing. Anyway, if we call the **result** method on this nested future, we get a **zip** object back, so to find out what the actual result is, we wrap the zip with Python's **list** function and print it out.

Wrapping Up

Now you have another neat concurrency tool to use. You can easily create thread or process pools depending on your needs. Should you need to run a process that is network or I/O bound, you can use the thread pool class. If you have a computationally heavy task, then you'll want to use the process pool class instead. Just be careful of calling futures incorrectly or you might get a deadlock.