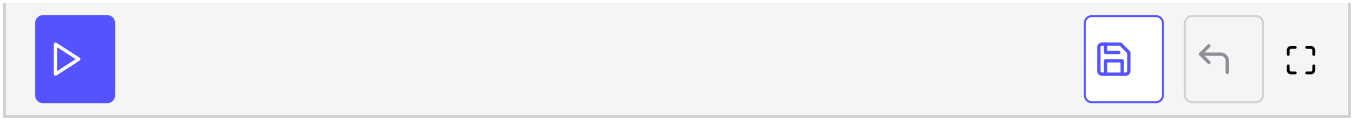
Creating a Pool

Creating a pool of workers is extremely easy when you're using the concurrent.futures module. Let's start out by rewriting our downloading code from the **asyncio** chapter so that it now uses the concurrent.futures module. Here's my version:

```
import os
import urllib.request

from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed


def downloader(url):
    """
    Downloads the specified URL and saves it to disk
    """
    req = urllib.request.urlopen(url)
    filename = os.path.basename(url)
    ext = os.path.splitext(url)[1]
    if not ext:
        raise RuntimeError('URL does not contain an extension')

    with open(filename, 'wb') as file_handle:
        while True:
            chunk = req.read(1024)
            if not chunk:
                break
            file_handle.write(chunk)
    msg = 'Finished downloading {filename}'.format(filename=filename)
    return msg


def main(urls):
    """
    Create a thread pool and download specified urls
    """
    with ThreadPoolExecutor(max_workers=5) as executor:
        futures = [executor.submit(downloader, url) for url in urls]
        for future in as_completed(futures):
            print(future.result())

if __name__ == '__main__':
    urls = ["http://www.irs.gov/pub/irs-pdf/f1040.pdf",
            "http://www.irs.gov/pub/irs-pdf/f1040a.pdf",
            "http://www.irs.gov/pub/irs-pdf/f1040ez.pdf",
            "http://www.irs.gov/pub/irs-pdf/f1040es.pdf"
```

```
            "http://www.irs.gov/pub/irs-pdf/f1040es.pdf",
            "http://www.irs.gov/pub/irs-pdf/f1040sb.pdf"]
    main(urls)
```



First off we do the imports that we need. Then we create our **downloader** function. I went ahead and updated it slightly so it checks to see if the URL has an extension on the end of it. If it doesn't, then we'll raise a **RuntimeError**. Next we create a **main** function, which is where the thread pool gets instantiated. You can actually use Python's **with** statement with the ThreadPoolExecutor and the ProcessPoolExecutor, which is pretty handy.

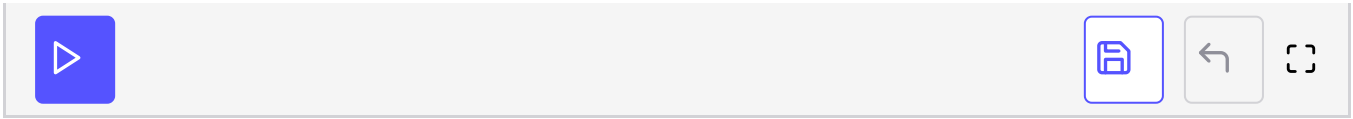
Anyway, we set our pool so that it has five workers. Then we use a list comprehension to create a group of futures (or jobs) and finally we call the **as_complete** function. This handy function is an iterator that yields the futures as they complete. When they complete, we print out the result, which is a string that was returned from our downloader function.

If the function we were using was very computation intensive, then we could easily swap out ThreadPoolExecutor for ProcessPoolExecutor and only have a one line code change.

We can clean this code up a bit by using the concurrent.future's **map** method. Let's rewrite our pool code slightly to take advantage of this:

```
import os
import urllib.request

from concurrent.futures import ThreadPoolExecutor
from concurrent.futures import as_completed


def downloader(url):
    """
    Downloads the specified URL and saves it to disk
    """
    req = urllib.request.urlopen(url)
    filename = os.path.basename(url)
    ext = os.path.splitext(url)[1]
    if not ext:
        raise RuntimeError('URL does not contain an extension')

    with open(filename, 'wb') as file_handle:
        while True:
            chunk = req.read(1024)
            if not chunk:
                break
```

```
            file_handle.write(chunk)
    msg = 'Finished downloading {filename}'.format(filename=filename)
    return msg



def main(urls):
    """
    Create a thread pool and download specified urls
    """
    with ThreadPoolExecutor(max_workers=5) as executor:
        return executor.map(downloader, urls, timeout=60)

if __name__ == '__main__':
    urls = ["http://www.irs.gov/pub/irs-pdf/f1040.pdf",
            "http://www.irs.gov/pub/irs-pdf/f1040a.pdf",
            "http://www.irs.gov/pub/irs-pdf/f1040ez.pdf",
            "http://www.irs.gov/pub/irs-pdf/f1040es.pdf",
            "http://www.irs.gov/pub/irs-pdf/f1040sb.pdf"]
    results = main(urls)
    for result in results:
        print(result)
```



The primary difference here is in the **main** function, which has been reduced by two lines of code. The *map** method is just like Python's map in that it takes a function and an iterable and then calls the function for each item in the iterable. You can also add a timeout for each of your threads so that if one of them hangs, it will get stopped. Lastly, starting in Python 3.5, they added a **chunksize** argument, which can help performance when using the Thread pool when you have a very large iterable. However if you happen to be using the Process pool, the chunksize will have no effect.