

THE POLICE

ASYNCHRONICITY

Introduction to Asynchronous Python

April 20, 2015

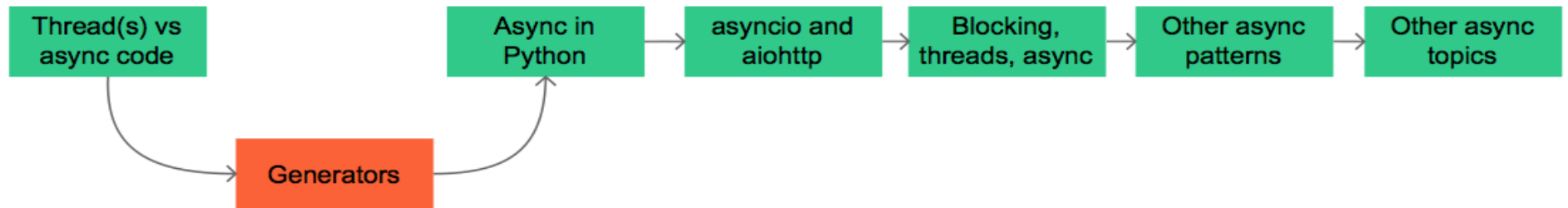
mempy – Scott Umsted

<https://github.com/sumsted/mempy-async>

asynchronicity

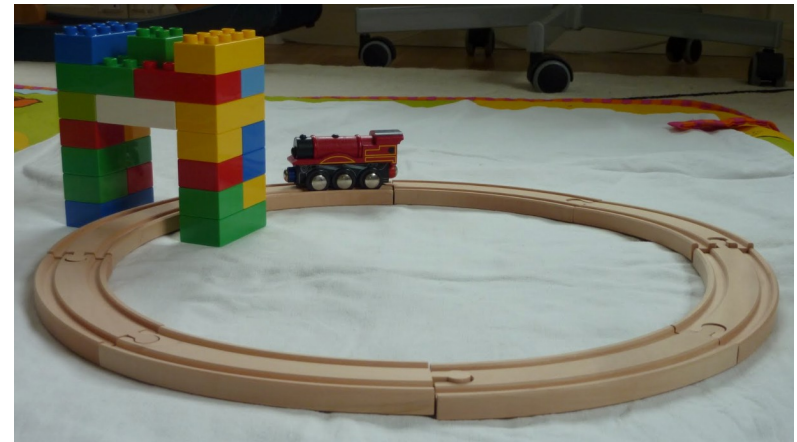
Side One: Sun In Your Face - Nothing In Your Face - O My God - Mother - Miss Godfather - Synchronicity II - Side Two: Every Breath You Take - King Of Pain - Wendon - Love in the US - Every - Turn In The Snow

Asynchronous Python



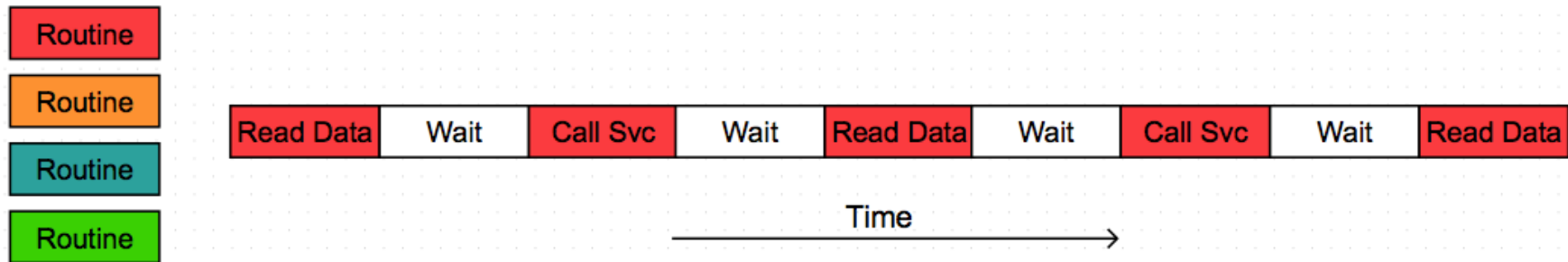
← mark of
asynchronicity

A thread



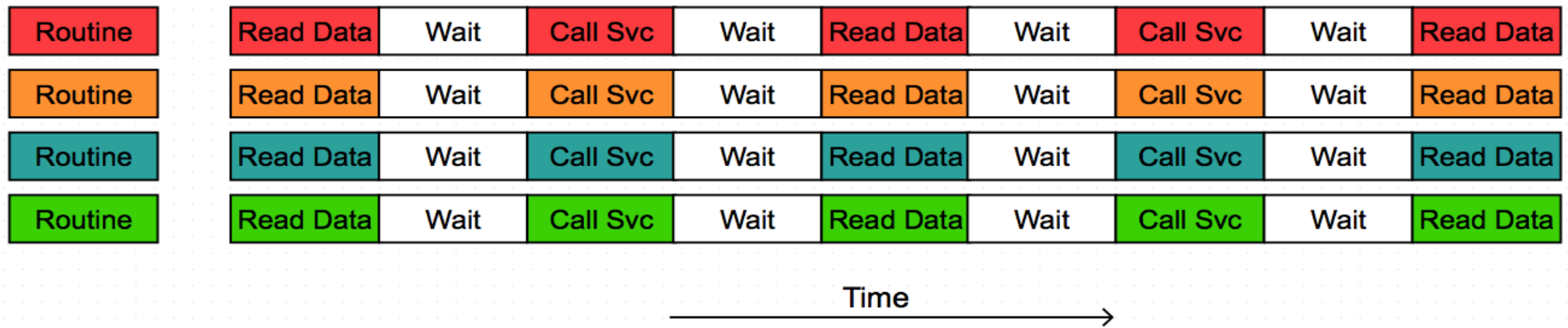
- A thread is like a train
- Length of train is time
- Each car represents an opportunity to execute an instruction
- Empty cars are waste

A thread



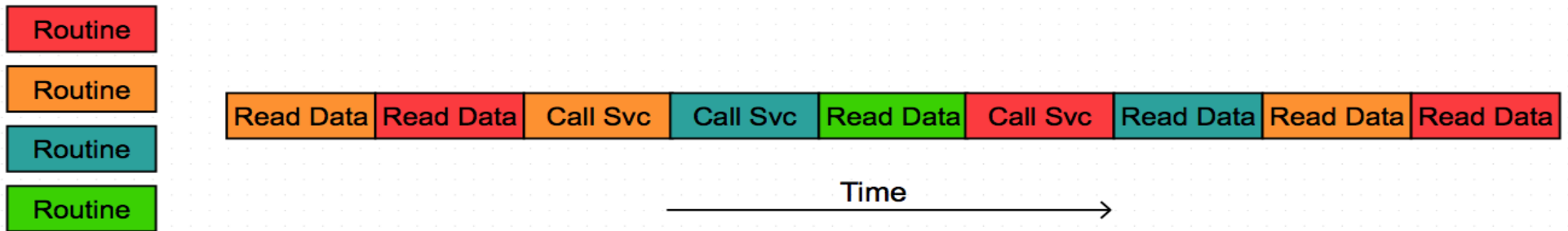
- Single line of execution that processes one instruction at a time
- Blocking of thread occurs on I/O reads
- Waiting for file system, DNS lookup, http server connections and reads

Multiple threads or processes



- Parallel execution of tasks
- More efficient use of time
- But still waste in each thread
- Each thread or process blocks and has a cost
- Python threads and the GIL

Asynchronous



- More efficient use of single thread
- Shelve waiting code so other code may execute
- Fill in the gaps where waits occur



Python generators

- I need a list of numbers 0 to 9999
- I could create a list of numbers in memory
- But its not very efficient

```
module: nogenerator.py
mem:    516,096
utime:  0.004304
stime:  0.000294
cpu:    0.004598
time:   0.004591
```

nogenerator.py

```
1  from resourcehelper import ResourceHelper
2
3  def no_generator(top):
4      result = []
5      i = 0
6      while i < top:
7          result.append(i)
8          i += 1
9      return result
10
11
12 if __name__ == '__main__':
13     rh = ResourceHelper(__file__)
14
15     for i in no_generator(10000):
16         pass
17
18     rh.usage()
```

Python generators

- I could also use a generator pattern
- Much more efficient, but lots of code
- Create an iterable object
- Creates data as needed
- Think range(n)

```
module: generator.py
mem:    4,096
utime:  0.001839
stime:  0.000001
cpu:    0.001840
time:   0.001842
```

generatorlong.py

```
1 from resourcehelper import ResourceHelper
2
3 class generator(object):
4
5     def __init__(self, top):
6         self.top = top
7         self.i = 0
8
9     def __iter__(self):
10        return self
11
12    def __next__(self):
13        c = self.i
14        if self.i < self.top:
15            self.i += 1
16            return c
17        else:
18            raise StopIteration()
19
20 if __name__ == '__main__':
21     rh = ResourceHelper(__file__)
22
23     for i in generator(10000):
24         pass
25
26     rh.usage()
```


Python generators

- Python provides a nice shortcut to create generators
- yield tells thread to return value and shelve code until next call of generator

```
module: generatorlong.py
mem:    4,096
utime:  0.005921
stime:  0.000001
cpu:    0.005922
time:   0.005923
```

```
generator.py
1  from resourcehelper import ResourceHelper
2
3
4  def generator(top):
5      i = 0
6      while i < top:
7          yield i
8          i += 1
9
10
11 if __name__ == '__main__':
12     rh = ResourceHelper(__file__)
13
14     for i in generator(10000):
15         pass
16
17     rh.usage()
```



asyncio and aiohttp

asyncio

- Introduced in 3.4, derived from tulip project
- Deferred pattern using coroutines, futures and tasks as opposed to callbacks
- Thought is that callbacks can get complicated real quick and hard to read
- Single event loop per thread
- Provides transport library to asynchronous managed connections

aiohttp

- Third party library that provides http protocol support to asyncio
- Sits on top of asyncio transport library
- Think of it as a requests for asyncio



Problem: total .gov jobs in zips

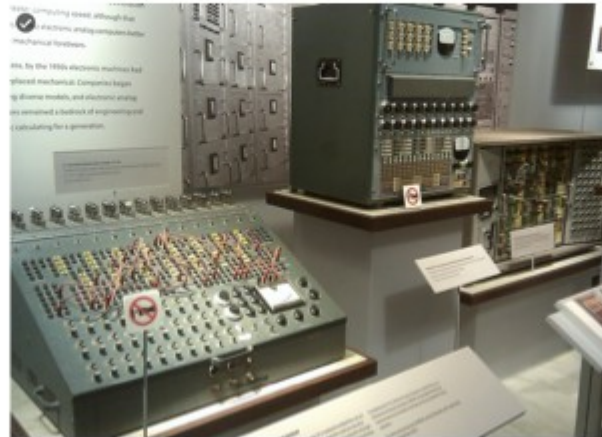
a bunch of zip codes

+

usajobs.gov

$$=$$

number of jobs

[illegible]

1,706

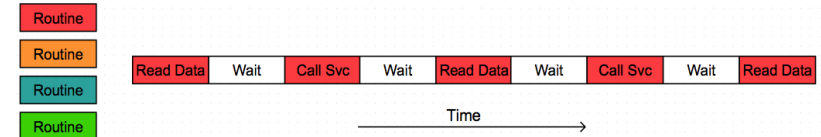
- `total_jobs = 0`
- loop through all zips
 - call `https://data.usajobs.gov/api/jobs?LocationId=' + zip`
 - `total_jobs = total_jobs + response.number_of_jobs`
- print `total_jobs`

Blocking solution

- Single thread, synchronous execution
- Lots of wasted time waiting for usajobs.gov
- Wait for DNS lookup and for a response from server

```
jobs available: 1651

module: block.py
mem:    6,733,824
utime:  3.020994
stime:  0.210687
cpu:    3.231681
time:   41.310599
```



```
block.py
1 import json
2 import requests
3 from resourcehelper import ResourceHelper
4
5
6 def get_jobs(zip):
7     response = requests.get('https://data.usajobs.gov/api')
8     result = json.loads(response.text)
9     total_jobs = int(result['TotalJobs'])
10    return total_jobs
11
12
13 def block_jobs(zips):
14     total = 0
15     for zip in zips:
16         total += get_jobs(zip)
17     return total
18
19
20 if __name__ == '__main__':
21     rh = ResourceHelper(__file__)
22
23     zips = ['79936', '90011', '60629', '90650', '90201',
24            '90250', '90280', '11226', '90805', '91331',
25            '10467', '92683', '75052', '91342', '92704',
26            '75217', '92376', '93307', '10456', '10002',
27            '92345', '60618', '93033', '93550', '95076',
28            '37211', '30043', '11206', '10453', '92154',
29            '92553', '90706', '23464', '11212', '60617',
30            '77429', '93535', '66062', '93257', '30349',
31            '11207', '77494', '75211', '11234', '28269',
32            '92509', '77083', '91335', '85364', '87121',
33
34     jobs = block_jobs(zips)
35     print('\njobs available: %s' % jobs)
36
37     rh.usage()
```

Threaded solution

- Spin a thread for each lookup
- As threads end capture result
- For I/O operations very fast but can use a lot of memory
- For CPU operations GIL will block

```
jobs available: 1651

module: threaded.py
mem:    91,996,160
utime:  2.784083
stime:  1.149400
cpu:    3.933483
time:   2.720289
```

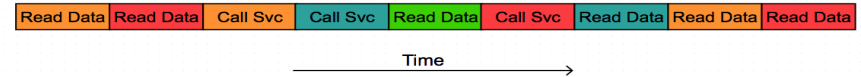


```
threaded.py
1 from threading import Thread
2 import json
3 import requests
4 from resourcehelper import ResourceHelper
5
6
7 class GetJobs(Thread):
8
9     def __init__(self, zip):
10         super(GetJobs, self).__init__()
11         self.zip = zip
12         self.total_jobs = 0
13
14     def run(self):
15         response = requests.get('https://data.usajobs.gov')
16         result = json.loads(response.text)
17         self.total_jobs = int(result['TotalJobs'])
18
19
20 def thread_jobs(zips):
21     total = 0
22     threads = []
23
24     for zip in zips:
25         threads.append(GetJobs(zip))
26         threads[-1].start()
27
28     for t in threads:
29         t.join()
30         total += t.total_jobs
31
32     return total
33
34
35 if __name__ == '__main__':
36     rh = ResourceHelper(__file__)
37
38     zips = ['79936', '90011', '60629', '90650', '90201',
39            '90250', '90280', '11226', '90805', '91331',
```


Async solution

- Event loop is created and passed a coroutine
- coroutine is a generator that wraps a function, it makes it event loop aware
- coroutines return futures
- A future represents a future result, think deferred
- yield from used to handle future of coroutine
- yield from tells the loop to shelve a coroutine until its ready
- coroutines may be stacked on top of one another, results can bubble up

```
jobs available: 1651
module: async.py
mem:      12,185,600
utime:    0.627300
stime:    0.083406
cpu:      0.710706
time:     3.852678
```



```
async.py
1 import asyncio
2 import aiohttp
3 from resourcehelper import ResourceHelper
4
5
6 @asyncio.coroutine
7 def get_jobs(zip):
8     response = yield from aiohttp.request('GET', 'https://data.usa
9     result = yield from response.read_and_close(decode=True)
10    total_jobs = int(result['TotalJobs'])
11    return total_jobs
12
13
14 @asyncio.coroutine
15 def async_jobs(zips):
16     total = 0
17     coroutines = [get_jobs(zip) for zip in zips]
18     for result in asyncio.as_completed(coroutines):
19         total += yield from result
20     return total
21
22
23 if __name__ == '__main__':
24     rh = ResourceHelper(__file__)
25
26     zips = ['79936', '90011', '60629', '90650', '90201', '77084',
27            '92509', '77083', '91335', '85364', '87121', '10468',
28
29     loop = asyncio.get_event_loop()
30     jobs = loop.run_until_complete(async_jobs(zips))
31     print('\njobs available: %s' % jobs)
32
33     rh.usage()
```

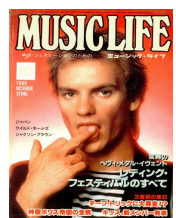
Event Loop

- Is a loop that facilitates the sharing of computing resources by coroutines
- One loop per thread, so typically a single loop
- May be created and retrieved by `.get_event_loop()`, typically in `main()`
- Only the main thread may do this, other threads must create their own and use `.set_event_loop()` before the get
- Ex. `.run_until_complete(future)` to start our loop OR `.run_forever() + .stop()`
- Callbacks within loop using `.call_soon(cb, args)`, `.call_later()`, `.call_at()`

```
36
37     loop = asyncio.get_event_loop()
38     jobs = loop.run_until_complete(async_jobs(zips))
39     print('\njobs available: %s' % jobs)
40
```

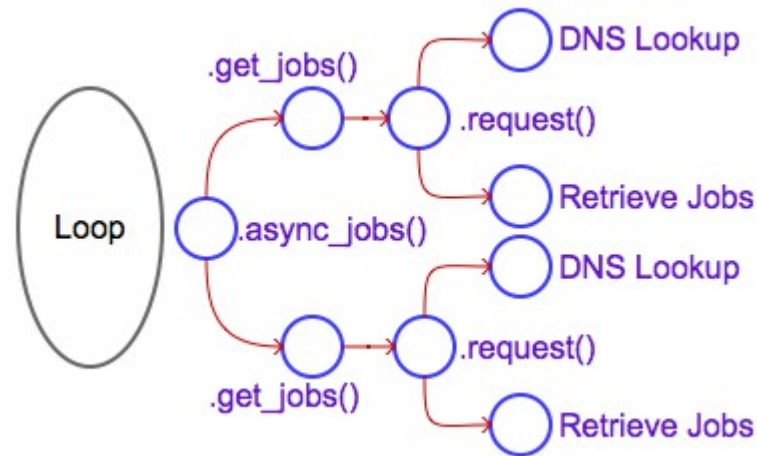
loops.py

```
1  import asyncio
2
3
4  def work_callback(result, loop):
5      print('ending loop for result: {}'.format(result))
6      loop.stop()
7
8  def do_some_work(x, y, cb, loop):
9      r = x + y
10     loop.call_later(3, cb, r, loop)
11
12  if __name__ == '__main__':
13     loop = asyncio.get_event_loop()
14     loop.call_soon(do_some_work, 2, 3, work_callback, loop)
15     jobs = loop.run_forever()
```



Coroutine

- Is a generator that wraps a function, made to be event loop aware
- As a generator they return a computed value
- A future represents a future result, think deferred
- Coroutines return futures, either not done or done with result
- If done code block continues, if not done continue to shelve coroutine
- `yield from` identifies a coroutine
- `yield from` used to handle future of called coroutine
- `yield from` tells the loop to shelve a coroutine until its ready
- “pretend that `yield from` isn't there”



```
6 @asyncio.coroutine
7 def get_jobs(zip):
8     response = yield from aiohttp.request('GET', 'https://dat
9     result = yield from response.read_and_close(decode=True)
10    total_jobs = int(result['TotalJobs'])
11    return total_jobs
12
13
14 @asyncio.coroutine
15 def async_jobs(zips):
16     total = 0
17     coroutines = [get_jobs(zip) for zip in zips]
18     for result in asyncio.as_completed(coroutines):
19         total += yield from result
20     return total
```



Blocking vs Async



block.py

```
1 import json
2 import requests
3 from resourcehelper import ResourceHelper
4
5
6
7 def get_jobs(zip):
8     response = requests.get('https://data.usajob')
9     result = json.loads(response.text)
10    total_jobs = int(result['TotalJobs'])
11    return total_jobs
12
13 |
14
15 def block_jobs(zips):
16     total = 0
17     for zip in zips:
18         total += get_jobs(zip)
19     return total
20
21
22
23 if __name__ == '__main__':
24     rh = ResourceHelper(__file__)
25
26     zips = ['79936', '90011', '60629', '90650',
27            '92509', '77083', '91335', '85364',
28
29
30     jobs = block_jobs(zips)
31     print('\njobs available: %s' % jobs)
32
33     rh.usage()
```

async.py

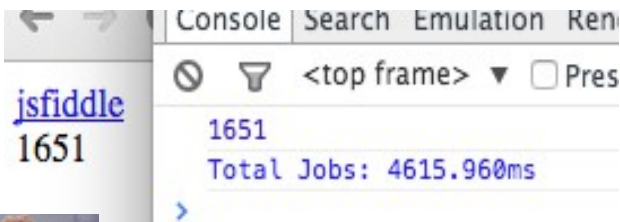
```
1 import asyncio
2 import aiohttp
3 from resourcehelper import ResourceHelper
4
5
6 @asyncio.coroutine
7 def get_jobs(zip):
8     response = yield from aiohttp.request('GET', 'htt
9     result = yield from response.read_and_close(decod
10    total_jobs = int(result['TotalJobs'])
11    return total_jobs
12
13
14 @asyncio.coroutine
15 def async_jobs(zips):
16     total = 0
17     coroutines = [get_jobs(zip) for zip in zips]
18     for result in asyncio.as_completed(coroutines):
19         total += yield from result
20     return total
21
22
23 if __name__ == '__main__':
24     rh = ResourceHelper(__file__)
25
26     zips = ['79936', '90011', '60629', '90650', '9020
27            '92509', '77083', '91335', '85364', '8712
28
29     loop = asyncio.get_event_loop()
30     jobs = loop.run_until_complete(async_jobs(zips))
31     print('\njobs available: %s' % jobs)
32
33     rh.usage()
```


JavaScript

- JavaScript is single threaded
- I/O operations are asynchronous
- Callbacks used to identify follow up code execution
- Easy to end up with multiple nested callbacks
- JQuery deferreds and ES6 promises

```
this.element.addEventListener( 'click', function(){alert('hi');}, false );
```

```
async.html
1 <!DOCTYPE html>
2 <html>
3
4 <head>
5   <link rel="shortcut icon" href="data:image/x-icon;" type="image/x-icon">
6   <script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.3/jquery.min.js"></script>
7 </head>
8
9 <body>
10  <a target="_blank" href="http://jsfiddle.net/scottumsted/866sx7he/1/">jsfiddle</a>
11  <br/>
12  <div id="total_jobs"></div>
13  <script>
14    function start() {
15      console.time('Total Jobs');
16      zips = ['79936', '90011', '60629', '90650', '90201', '77084', '92335', '7852',
17            '92509', '77083', '91335', '85364', '87121', '10468', '90255', '9306'];
18
19      var deferreds = [];
20      for (var i in zips) {
21        var url = 'https://data.usajobs.gov/api/jobs?LocationId=' + zips[i];
22        deferreds.push($.getJSON(url));
23      }
24
25      $.when.apply('$', deferreds).then(function () {
26        var totalJobs = 0;
27        for (var i in arguments) {
28          totalJobs += parseInt(arguments[i][0].TotalJobs);
29        }
30        $('#total_jobs').html(totalJobs);
31        console.log(totalJobs);
32        console.timeEnd('Total Jobs');
33      });
34    }
35
36    $.ready(start);
37  </script>
38 </body>
39
40 </html>
```



Other Resources

- Guido –<https://youtu.be/1coLC-MUCJc>
- <https://docs.python.org/3.4/library/asyncio.html>
- <https://www.python.org/dev/peps/pep-3156/>
- <http://www.drdobbs.com/open-source/the-new-asyncio-module-in-python-34-even/240168401>
- <https://wiki.python.org/moin/Generators>