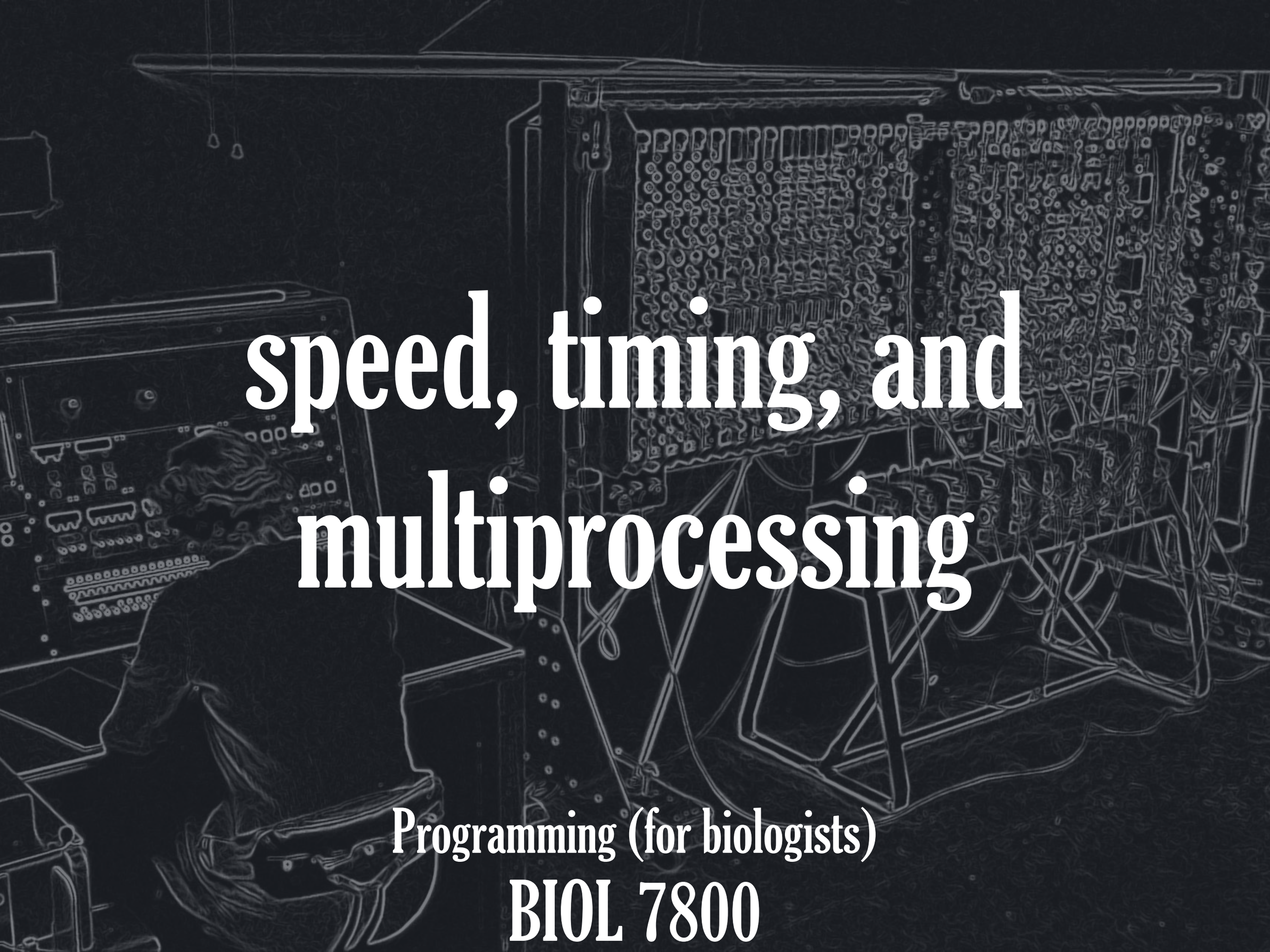


# speed, timing, and multiprocessing

Programming (for biologists)

BIOL 7800

# So far....

We've only talked about making programs functions, not making programs fast...


...and we've run only 1 process at a time

# So far....

We've not talked at all about using multi-CPU resources to run our jobs.


Titan Supercomputer
Oak Ridge National Laboratory

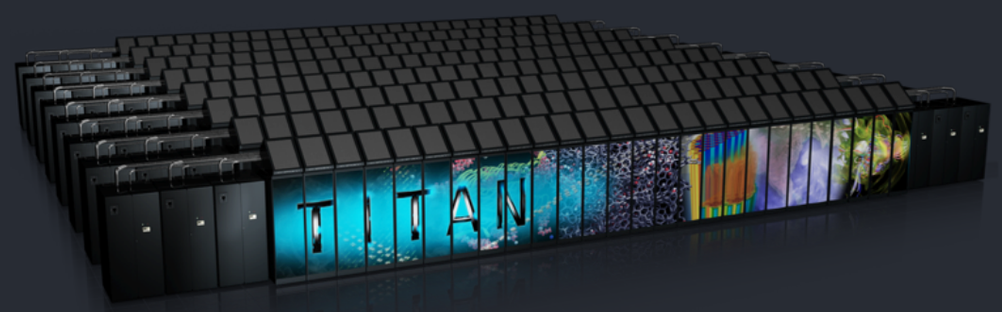
# Speed

is tricky thing…

and can be achieved in a variety of ways

Write more efficient code


Use more CPU resources



# Making things faster

We're going to focus on the following code

example1.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py

```
1   #! /usr/bin/env python
2   # encoding UTF-8
3
4   import random
5
6
7   def sum_lots_of_random_nums():
8       sum_holder = []
9       for elem in range(100):
10          random_numbers = []
11          for cnt in range(1000):
12              # pick 1 random number
13              random_number = random.randrange(0, 100000)
14              # sum those
15              random_numbers.append(random_number)
16          # sum the 1000 random numbers
17          sum_holder.append(sum(random_numbers))
18      # sum the 100 sums of random numbers
19      s = sum(sum_holder)
20      print("The sum is: {:,}".format(s))
21
22
23  def main():
24      sum_lots_of_random_nums()
25
26  if __name__ == '__main__':
27      main()
28
```

File 0 | Project 0 | ✔ No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example1.py | 1:1 | LF | UTF-8 | Python

Does the following 100 times

Does the following 1000 times

Selects 1 random numbers

Adds that to a list

Sums the list of 1000 numbers

Adds that to a list

Sums the list of 100 sums

Pretty prints the output

Example

In: python example1.py

Out: The sum is: 5,001,095,443

# Making things faster

Python provides the `timeit` module for testing speed

You can use it several ways... the easiest is probably

Import timeit

5 fxn calls

setup statement

```
python -m timeit -r 3 -n 5 -v -s 'from example1 import main' 'main()'
```

3 iterations

verbose output

example1.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py

```python
#! /usr/bin/env python
# encoding UTF-8

import random


def sum_lots_of_random_nums():
    sum_holder = []
    for elem in range(100):
        random_numbers = []
        for cnt in range(1000):
            # pick 1 random number
            random_number = random.randrange(0, 100000)
            # sum those
            random_numbers.append(random_number)
        # sum the 1000 random numbers
        sum_holder.append(sum(random_numbers))
    # sum the 100 sums of random numbers
    s = sum(sum_holder)
    print("The sum is: {:,}".format(s))


def main():
    sum_lots_of_random_nums()
```

```
The sum is: 5,017,823,857
The sum is: 4,989,840,976
(…truncated…)

raw times: 0.601 0.6 0.598
5 loops, best of 3: 120 msec per loop
```

# Making things faster

example1.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py

```python
#! /usr/bin/env python
# encoding UTF-8

import random


def sum_lots_of_random_nums():
    sum_holder = []
    for elem in range(100):
        random_numbers = []
        for cnt in range(1000):
            # pick 1 random number
            random_number = random.randrange(0, 100000)
            # sum those
            random_numbers.append(random_number)
        # sum the 1000 random numbers
        sum_holder.append(sum(random_numbers))
    # sum the 100 sums of random numbers
    s = sum(sum_holder)
    print("The sum is: {:,}".format(s))


def main():
    sum_lots_of_random_nums()

if __name__ == '__main__':
    main()

```

File 0 | Project 0 | No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example1.py | 1:1 | LF | UTF-8 | Python

```
raw times: 0.601 0.6 0.598
5 loops, best of 3: 120 msec per loop
```

How can we make this faster?

(without using multiple CPUs)

Increase efficiency!!

# Making things faster

## Approach #1: Big list

example2.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py

```
#! /usr/bin/env python
# encoding UTF-8

import random


def sum_lots_of_random_nums():
    sum_holder = []
    for elem in range(100):
        random_numbers = []
        for cnt in range(1000):
            # pick 1 random number
            random_number = random.randrange(0, 100000)
            # sum those
            random_numbers.append(random_number)
        sum_holder.append(random_numbers)
    # sum the 100 sums of random numbers
    s = sum([sum(elem) for elem in sum_holder])
    print("The sum is: {:,}".format(s))


def main():
    sum_lots_of_random_nums()

if __name__ == '__main__':
    main()

```

File 0 | Project 0 | ✔ No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example2.py | 1:1 | LF | UTF-8 | Python

time to beat

```
raw times: 0.601 0.6 0.598
5 loops, best of 3: 120 msec per loop
```

Put all numbers in lists…

then compute sums…

```
raw times: 0.608 0.607 0.61
5 loops, best of 3: 121 msec per loop
```

Not better

# Making things faster

## Approach #2: itertools

example3.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py

```
#! /usr/bin/env python
# encoding UTF-8

import random
import itertools


def sum_lots_of_random_nums():
    sum_holder = []
    for elem in range(100):
        random_numbers = []
        for cnt in range(1000):
            # pick 1 random number
            random_number = random.randrange(0, 100000)
            # sum those
            random_numbers.append(random_number)
        sum_holder.append(random_numbers)
    # sum the 100 sums of random numbers
    s = sum(list(itertools.chain(*sum_holder)))
    print("The sum is: {:,}".format(s))


def main():
    sum_lots_of_random_nums()

if __name__ == '__main__':
    main()

```

File 1 | Project 0 | ✔ No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example3.py | 19: | LF | UTF-8 | Python

time to beat

```
raw times: 0.601 0.6 0.598
5 loops, best of 3: 120 msec per loop
```

Put all numbers in lists…

then use itertools to unpack lists, and sum

```
raw times: 0.639 0.627 0.638
5 loops, best of 3: 125 msec per loop
```

Not better

# Making things faster

## Approach #3: bigger random draw

example4.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py | example4.py

```
#! /usr/bin/env python
# encoding UTF-8

import random


def sum_lots_of_random_nums():
    sum_holder = []
    for elem in range(100):
        random_numbers = random.sample(list(range(0, 100000)), 1000)
        sum_holder.append(sum(random_numbers))
    # sum the 100 sums of random numbers
    s = sum(sum_holder)
    print("The sum is: {:,}".format(s))


def main():
    sum_lots_of_random_nums()

if __name__ == '__main__':
    main()

```

File 0 | Project 0 | ✔ No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example4.py | 6:1 | LF | UTF-8 | Python

time to beat

```
raw times: 0.601 0.6 0.598
5 loops, best of 3: 120 msec per loop
```

← Select 1000 numbers from a list (e.g. remove a loop)

← Sum list of sum

```
raw times: 1.37 1.36 1.33
5 loops, best of 3: 265 msec per loop
```

Much worse!

2.2x slowdown

**Why?**

# Making things faster

## Approach #4: bigger random draw alt. version

time to beat

```
raw times: 0.601 0.6 0.598
5 loops, best of 3: 120 msec per loop
```

example5.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py | example4.py | example5.py

```
#! /usr/bin/env python
# encoding UTF-8

import random


def sum_lots_of_random_nums():
    sum_holder = []
    # make range once:
    my_list = list(range(0, 100000))
    for elem in range(100):
        random_numbers = random.sample(my_list, 1000)
        sum_holder.append(sum(random_numbers))
    # sum the 100 sums of random numbers
    s = sum(sum_holder)
    print("The sum is: {:,}".format(s))


def main():
    sum_lots_of_random_nums()

if __name__ == '__main__':
    main()

```

File 0 | Project 0 | No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example5.py | 10: | LF | UTF-8 | Python

← Create range of number only 1 time

← Select 1000 numbers from a list (e.g. remove loop)

← Sum list of sum

```
raw times: 0.457 0.443 0.443
5 loops, best of 3: 88.7 msec per loop
```

Waaaay better!

1.3x speedup

New best time!

# Making things faster

## Approach #5: use list comprehension

example6.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py | example4.py | example5.py | example6.py

```
1   #! /usr/bin/env python
2   # encoding UTF-8
3
4   import random
5
6
7   def sum_lots_of_random_nums():
8       my_list = list(range(0, 100000))
9       s = sum([sum(random.sample(my_list, 1000)) for i in range(0, 100)])
10      print("The sum is: {:,}".format(s))
11
12
13  def main():
14      sum_lots_of_random_nums()
15
16  if __name__ == '__main__':
17      main()
18
```

File 0 | Project 0 | ✔ No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example6.py | 7:3 | LF | UTF-8 | Python

time to beat

```
raw times: 0.457 0.443 0.443
5 loops, best of 3: 88.7 msec per loop
```

Create range of number only 1 time

Put all summing in a list comprehsion

```
raw times: 0.466 0.457 0.462
5 loops, best of 3: 91.5 msec per loop
```

Not better

# Making things faster

## Approach #6: use numpy

example7.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py | example4.py | example5.py | example6.py | example7.py

```
1   #! /usr/bin/env python
2   # encoding UTF-8
3
4   import numpy
5
6
7   def sum_lots_of_random_nums():
8       sum_holder = []
9       for elem in range(100):
10          random_numbers = numpy.random.random_integers(0, 100000, 1000)
11          sum_holder.append(sum(random_numbers))
12      # sum the 100 sums of random numbers
13      s = sum(sum_holder)
14      print("The sum is: {:,}".format(s))
15
16
17  def main():
18      sum_lots_of_random_nums()
19
20  if __name__ == '__main__':
21      main()
22
```

File 0 | Project 0 | ✔ No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example7.py | 10: | LF | UTF-8 | Python

time to beat

```
raw times: 0.457 0.443 0.443
5 loops, best of 3: 88.7 msec per loop
```

← Use numpy to randomly sample integer

← Append sum(int) to a list

← Sum list of sum

(use **regular** Python sum)

```
raw times: 0.044 0.0455 0.0454
5 loops, best of 3: 8.8 msec per loop
```

Much better!

10x speedup

New best time!

# Making things faster

## Approach #7: use numpy (with numpy.sum)

example8.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.p | example2.p | example3.p | example4.p | example5.p | example6.p | example7.p | example8.p | example9.p

```
1   #! /usr/bin/env python
2   # encoding UTF-8
3
4   import numpy
5
6
7   def sum_lots_of_random_nums():
8       sum_holder = []
9       for elem in range(100):
10          random_numbers = numpy.random.random_integers(0, 100000, 1000)
11          sum_holder.append(numpy.sum(random_numbers))
12      # sum the 100 sums of random numbers
13      s = sum(sum_holder)
14      print("The sum is: {:,}".format(s))
15
16
17  def main():
18      sum_lots_of_random_nums()
19
20  if __name__ == '__main__':
21      main()
22
```

File 0 | Project 0 | ✔ No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example8.py | 20: | LF | UTF-8 | Python

time to beat

```
raw times: 0.044 0.0455 0.0454
5 loops, best of 3: 8.8 msec per loop
```

← Use numpy to randomly sample integer
← Append numpy.sum(int) to a list
← Sum list of sum
(use **regular** Python sum)

```
raw times: 0.00844 0.00876 0.00837
5 loops, best of 3: 1.67 msec per loop
```

Much better!

5.2x speedup (over 1st numpy version)

71.0x speedup (over 1st python version)

# Making things faster

## Can we go even faster?

## Approach #8: everything numpy

example9.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.p | example2.p | example3.p | example4.p | example5.p | example6.p | example7.p | example8.p | example9.p

```
1   #! /usr/bin/env python
2   # encoding UTF-8
3
4   import numpy
5
6
7   def sum_lots_of_random_nums():
8       s = numpy.sum(numpy.random.randint(0, 100000, (1000, 100)))
9       print("The sum is: {:,}".format(s))
10
11
12  def main():
13      sum_lots_of_random_nums()
14
15  if __name__ == '__main__':
16      main()
17
```

File 0 | Project 0 | ✔ No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example9.py | 5:1 | LF | UTF-8 | Python

time to beat

```
raw times: 0.00844 0.00876 0.00837
5 loops, best of 3: 1.67 msec per loop
```

Select an array of 100 rows of 1000 random numbers and numpy.sum those

```
raw times: 0.00731 0.00662 0.00728
5 loops, best of 3: 1.32 msec per loop
```

Better!

1.2x speedup (over 2nd numpy version)

90x speedup (over 1st python version)

# Making things faster

Now, I've shown you all of those to show you that **how** you write your code is very, very important to how fast it runs

We achieved ~90x speedup by optimizing our code!

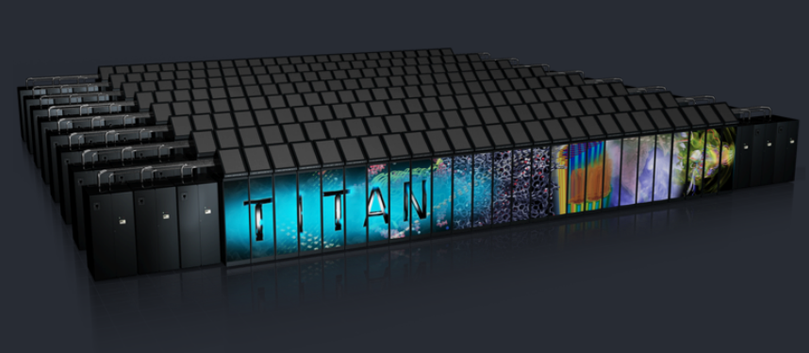
(that's pretty darn amazing)


So being efficient about how you write your code can make a **huge** difference

Write more efficient code

# Making things faster

We can also achieve speedups by taking advantage of multiple CPUs on many (current) computers

| Laptop | Workstation | High-performance Computer |
|---|---|---|
|  |  |   |
| 2 CPU cores | 6-20 CPU cores | 16-20 CPU cores |

One important thing to remember is that physical CPUs are what's (mostly) important

And most Intel chips have both physical and virtual CPUs (via "hyperthreading")

The number of physical CPUs you have is usually 1/2 that of physical + virtual CPUs

# Making things faster

In Python, the **multiprocessing** module helps us parallelize code

And, the **multiprocessing.Pool** class is the easiest way to use **multiprocessing**

```
In:  import multiprocessing
```
← import the **multiprocessing** module

```
In:  def my_function(x):
       return x**x
```
← define some function we want to run in parallel

# of CPUs

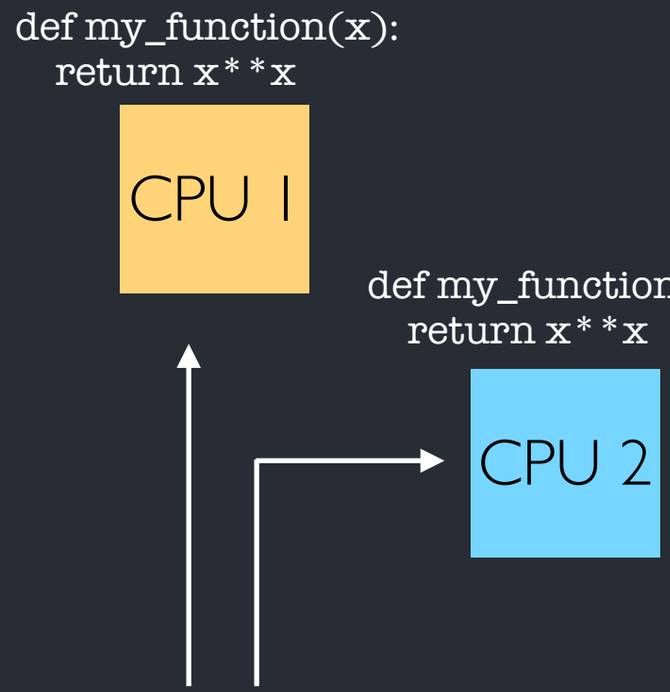
```
In:  pool = multiprocessing.Pool(2)
```
← Create instance of **multiprocessing.Pool** class

```
In:  my_result = pool.map(my_function, [2, 4, 6, 8, 10])
```
← "Map" the values in the list onto the function

```
Out: [4, 256, 46656, 16777216, 10000000000]
```
← You get back a **list** of results, returned by the function, one for each value in the list you "mapped"

# Making things faster

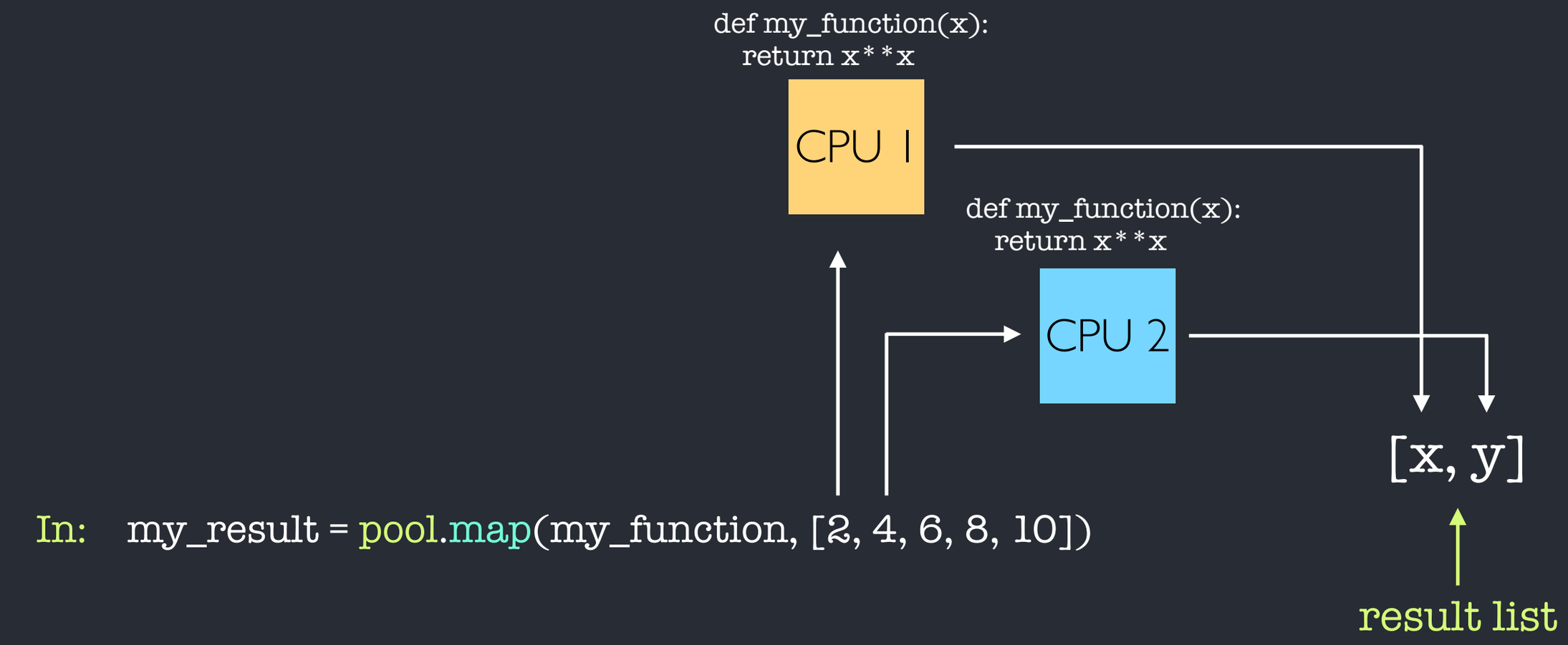
Python creates copies of your function and "loads" one copy on each CPU




```
In:   my_result = pool.map(my_function, [2, 4, 6, 8, 10])
```

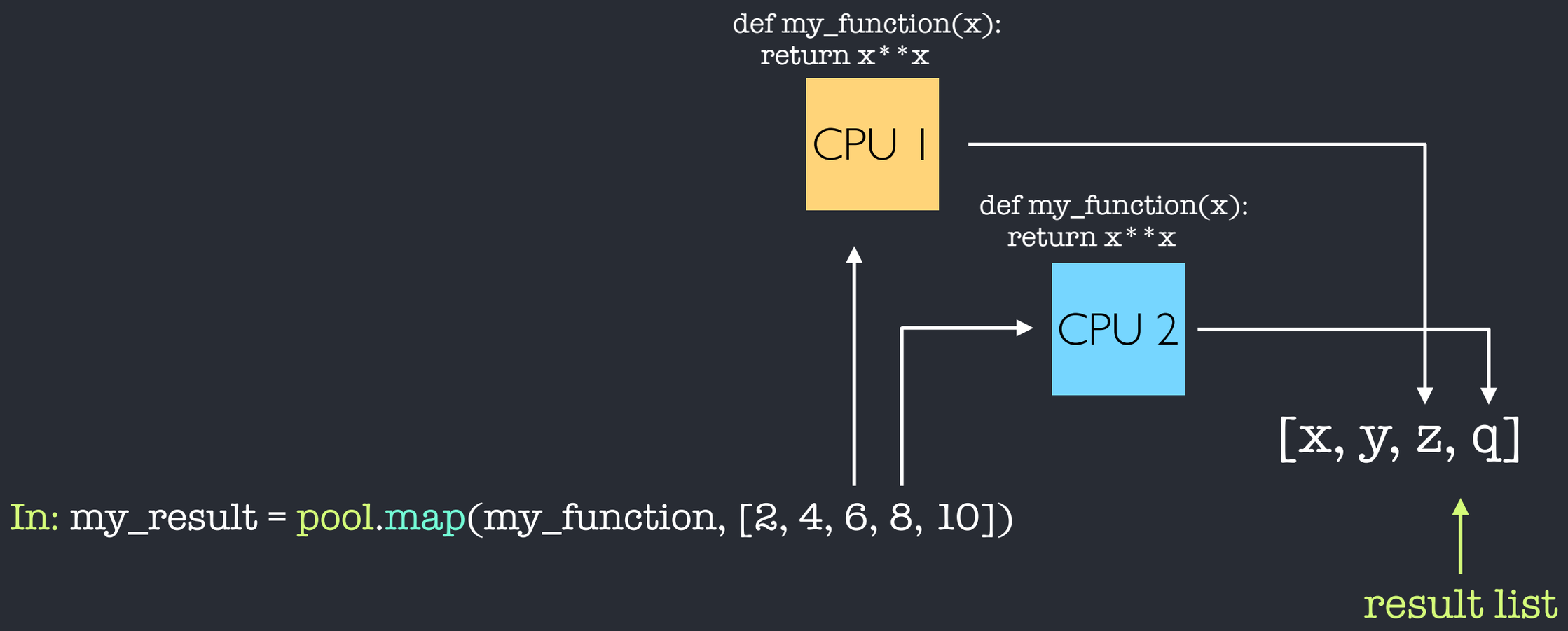
# Making things faster

Each CPU processes the incoming data, and writes the results to a list



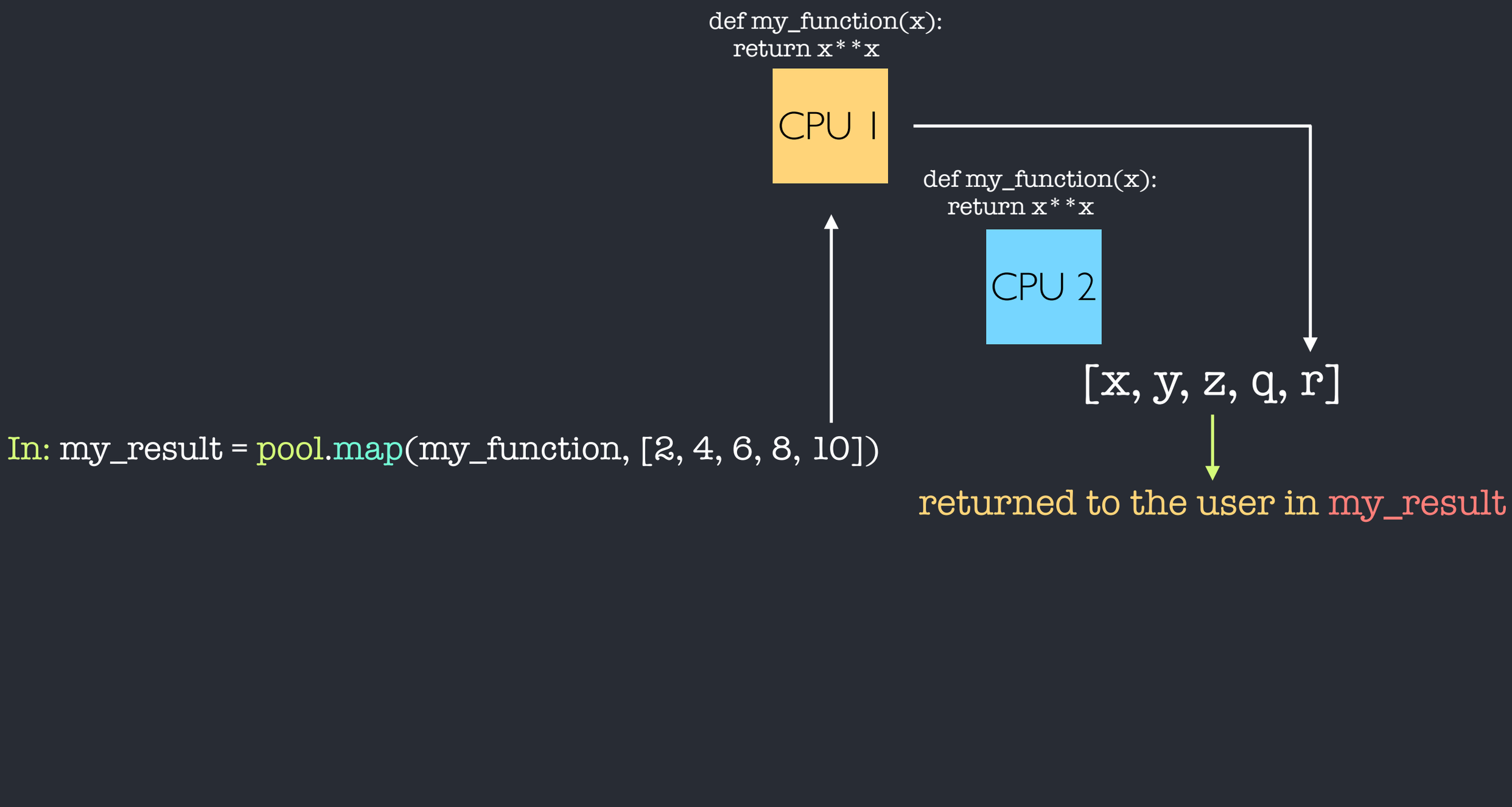
# Making things faster

Execution moves to the next data items, and those results are written to the output list



# Making things faster

Execution moves to the last data item, and those results are written to the output list



# Making things faster

example10.py — /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture21

example1.py | example2.py | example3.py | example4.py | example5.py | example6.py | example7.py | example8.py | example9.py | example10.py

```python
#! /usr/bin/env python
# encoding UTF-8

import random
import multiprocessing


def sum_lots_of_random_nums():
    sum_holder = []
    for elem in range(100):
        random_numbers = []
        for cnt in range(1000):
            # pick 1 random number
            random_number = random.randrange(0, 100000)
            # sum those
            random_numbers.append(random_number)
        # sum the 1000 random numbers
        sum_holder.append(random_numbers)
    return sum_holder


def get_sum(list_item):
    return sum(list_item)


def main():
    random_numbers = sum_lots_of_random_nums()
    pool = multiprocessing.Pool(4)
    sum_of_lists = pool.map(get_sum, random_numbers)
    s = sum(sum_of_lists)
    print("The sum is: {:,}".format(s))
    pool.close()

if __name__ == '__main__':
    main()

```

File 0 | Project 0 | ✔ No Issues | /Users/bcf/Dropbox/Classes/BIOL7800/temp/lecture23/example10.py* | 14:56 | LF | UTF-8 | Python

time to beat

```
raw times: 0.601 0.6 0.598
5 loops, best of 3: 120 msec per loop
```

Make a list of lists

```
[
  [x, y, ..., z],
  [x, y, ..., z],
  ...
  [x, y, ..., z],
]
```

This function just sums a list

We Pool.map the list of lists onto get_sum. Each CPU sums one sublist and returns the sum.

```
raw times: 0.71 0.706 0.705
5 loops, best of 3: 141 msec per loop
```

Much slower!

# Making things faster

## No multiprocessing

time to beat

```
raw times: 0.601 0.6 0.598
5 loops, best of 3: 120 msec per loop
```

## Multiprocessing

```
raw times: 0.71 0.706 0.705
5 loops, best of 3: 141 msec per loop
```

Much slower!

Why, if we are using more CPUs is **multiprocessing** much slower?

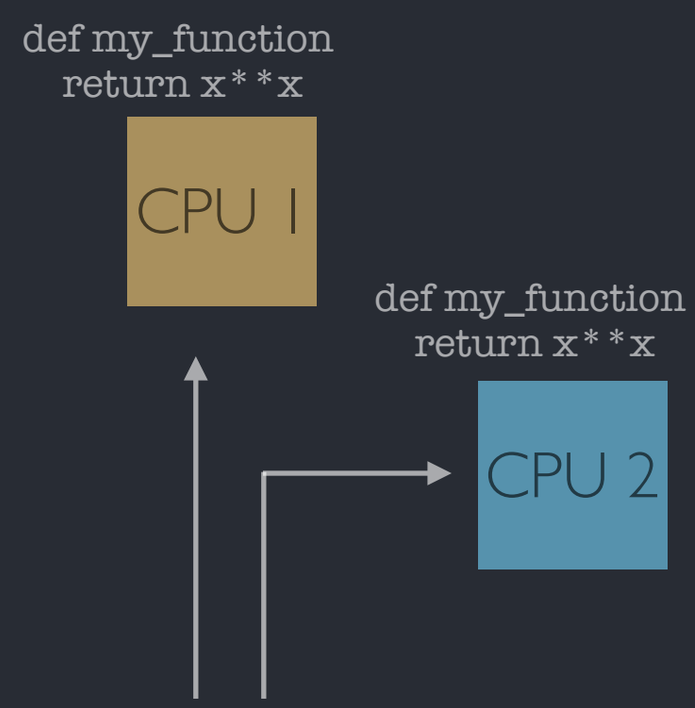
(and we're nowhere near results we saw from numpy)

# Making things faster

Why, if we are using more CPUs is `multiprocessing` much slower?

There's a fair amount of "overhead" setting up multiprocessing

Python creates



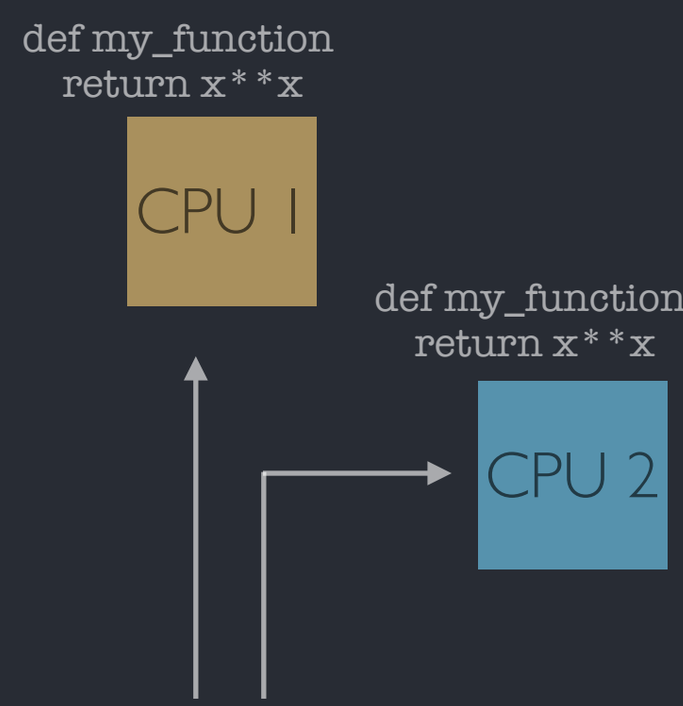

In:

# Making things faster

Why, if we are using more CPUs is multiprocessing much slower?

We often need job components to be more intensive in order to see benefits

Python creates



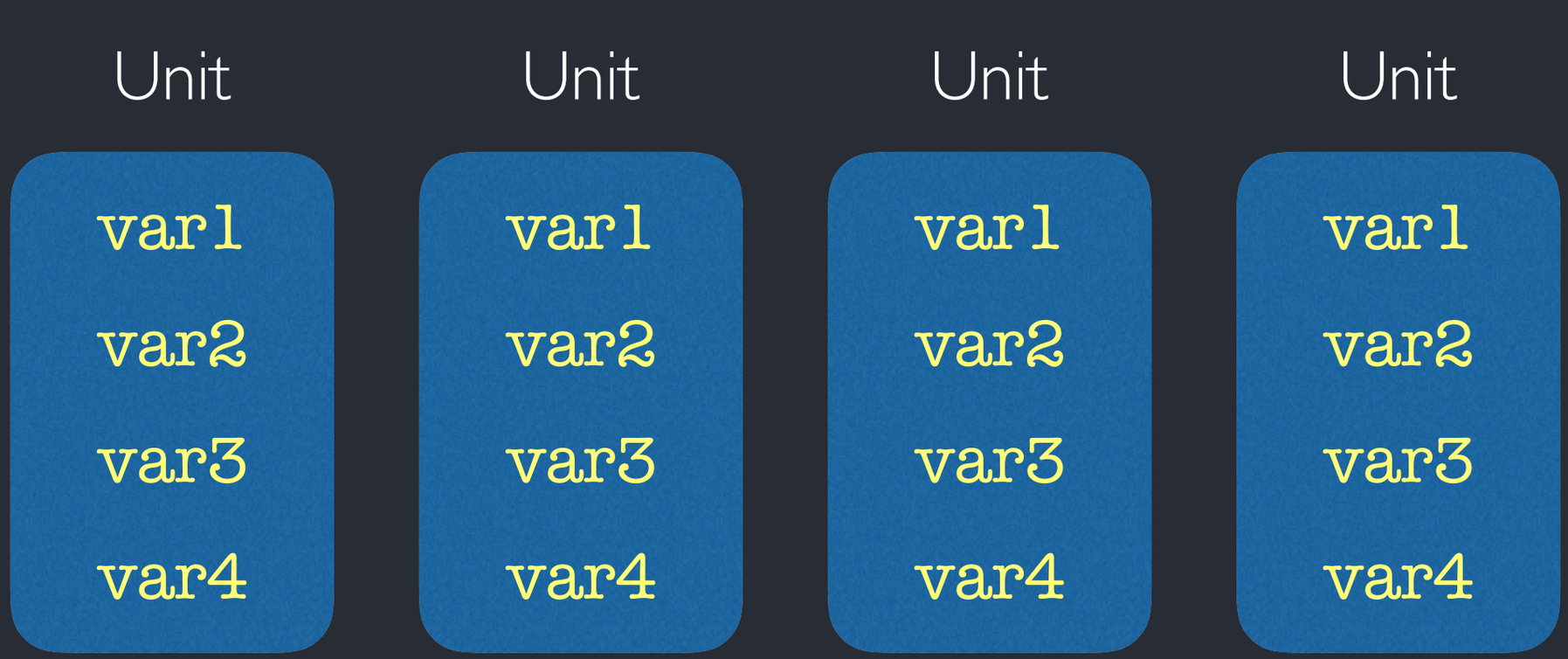

In:

# Making things faster

Getting lots of data in to `multiprocessing.Pool.map()` can be tricky

You need to "package" your data into lists/tuples

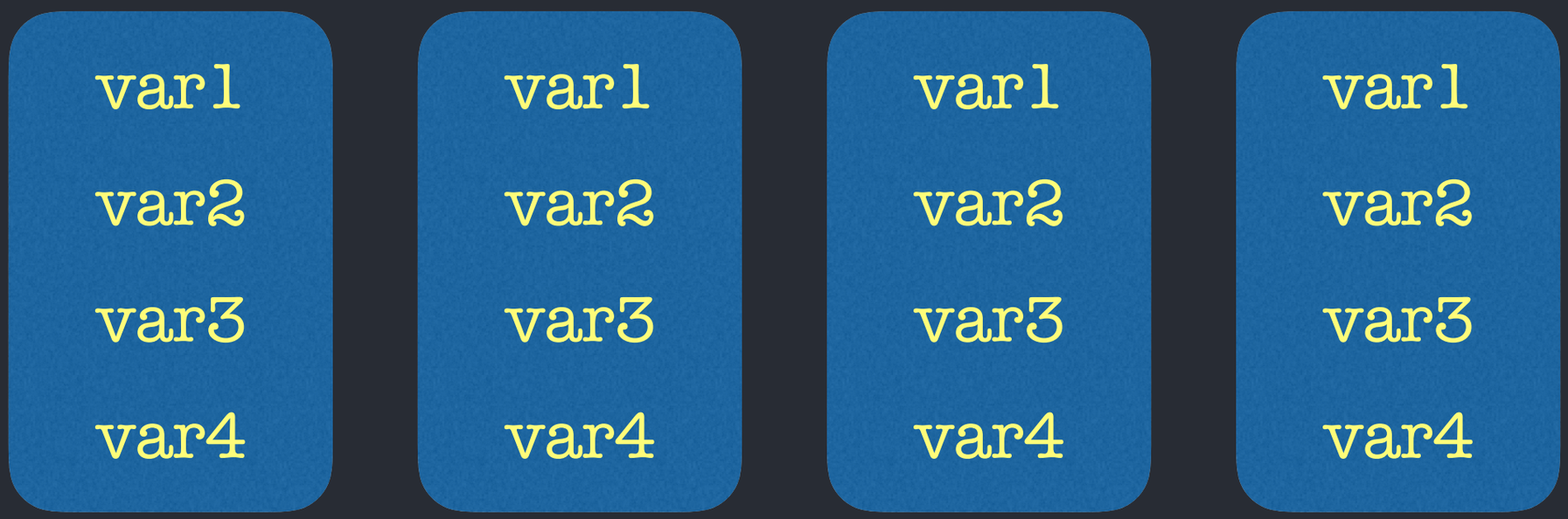
Let's say you have 4 units of "work"



# Making things faster

Getting lots of data in to `multiprocessing.Pool.map()` can be tricky

You have 4 units of "work"




```
In: def my_function(work):
        sum = var1 + var2 + var3 + var4
        return sum
```

← The function we're mapping to can only have one input

So, how can we make 4 work units of 4 variable into 1 work unit?

```
In: my_result = pool.map(my_function, <something here>)
```

# Making things faster

Getting lots of data in to `multiprocessing.Pool.map()` can be tricky

The function we're mapping to can only have one input

```
In: def my_function(work):
        var1, var2, var3, var4 = work
        sum = var1 + var2 + var3 + var4
        return sum
```

and unpack them

We can package data as 4 list/tuple items, then pass those to mapped function

```
In: work = (
              (var1, var2, var3, var4),
              (var1, var2, var3, var4),
              (var1, var2, var3, var4),
              (var1, var2, var3, var4)
            )
```

unit 1
unit 2
unit 3
unit 4

```
In: my_result = pool.map(my_function, work)
```

# Making things faster

Getting lots of data out of multiprocessing.Pool.map() can also be tricky

```
In: def my_function(work):
        var1, var2, var3 = work
        var1 += 1
        var2 += 2
        var3 += 3
        return ????
```

← The mapped function can only return one result

How can we deal with this?

```
In: work = (
              (var1, var2, var3, var4),    unit 1
              (var1, var2, var3, var4),    unit 2
              (var1, var2, var3, var4),    unit 3
            )
```

```
In: my_result = pool.map(my_function, work)
```

# Making things faster

Getting lots of data out of `multiprocessing.Pool.map()` can also be tricky

```
In: def my_function(work):
        var1, var2, var3 = work
        var1 += 1
        var2 += 2
        var3 += 3
        return [var1, var2, var3]
```

The mapped function can only return one result

← We can package those data into a single tuple or list

```
In: work = (
              (var1, var2, var3, var4),
              (var1, var2, var3, var4),
              (var1, var2, var3, var4),
            )
```

unit 1
unit 2
unit 3

```
In: my_result = pool.map(my_function, work)
In: for elem in my_result:
        var1, var2, var3 = elem
        # do stuff
```

← And then unpack them

# Making things faster

How do you determine the number of CPUs on a system?

It's a little hard…

Because hyperthreading is hard to detect

## Approach #1

```
def get_args():
   parser = argparse.ArgumentParser(
      description="""Do stuff in parallel""",
   )
   parser.add_argument(
      "--cores",
      type=int,
      default=1,
      help="Process in parallel using --cores"
   )

def main():
    args = get_args()
    pool = multiprocessing.Pool(args.cores)
    my_result = pool.map(my_function, [2, 4, 6, 8, 10])
```

← You can ask user for -cores

Then instantiate multiprocessing.Pool with args.cores

# Making things faster

How do you determine the number of CPUs on a system?

It's a little hard…

Because hyperthreading is hard to detect

## Approach #2

```
def main():
    cores = multiprocessing.cpu_count()

    pool = multiprocessing.Pool(cores)

    my_result = pool.map(my_function, [2, 4, 6, 8, 10])
```

You can use multiprocessing to "count" your CPU cores

Then instantiate multiprocessing.Pool with cores

This will return the # of "logical" CPUs (physical + virtual)