# Process Pools

First, let's see what you remember about processes.

Suppose we make two processes to execute some function F:

```
p = multiprocessing.Process(F, <args for p>)
q = multiprocessing.Process(F, <args for q>)
p.start( )
q.start( )
```

Which process will finish first?

A) p  B) q  C) can't tell  D) It depends on the args

# How many times will "bob" print?

```
def Printer( ):
        print( "bob" )

def spawnProcesses( ):
        print( "bob" )
        for i in range(0, 2):
                p = multiprocessing.Process(target=Printer)
                p.start()

def main( ):
        for i in range(0, 4):
                p = multiprocessing.Process(target=spawnProcesses)
                p.start()
```

A) 2 B) 4 C) 12 D) 16

Because many of the advantages of multiprocessing come from using multiple processes to do part of one big task, it is very common to want to start up multiple processes to work on the same function, just with different arguments.  Process *Pools* do this automatically.

We create n processes with

```
p = multiprocessing.Pool(processes=n)
```

We apply them to function *func* with

```
p.map(func, [arg1, arg2, ..., argn] )
```

The function *func* you call in this way can only have one argument (though that may be a tuple).

Note that *n* here is the number of processes.

For example,

```
def printer( args):
    stringToPrint, numTimes = args
    for i in range(numTimes):
        print(stringToPrint)

def main():
        p = multiprocessing.Pool(processes=2)
        p.map(printer, [("She loves me.", 5),
                        ("She loves me not.", 4)])
```

<FirstPoolExample.py>

The Pool.map method takes a function to call and a list of arguments, one for each process being created. If *func* returns a value, Pool.map returns a list of the n values returned by that process. You usually need to do something to put together these n partial results into the final result.

<SecondPoolExample.py>
<ThirdPoolExample.py>

In that third example, why do the gains from adding more processes taper off?

A) Because the computer is tired.
B) Because I am tired.
C) Because there aren't enough processors in the computer.
D) Because coordinating lots of processes also takes time.

Throwing more processes at a problem doesn't necessarily mean that you'll get faster computations.  There is a balance between savings from having multiple processes and costs from having to manage all of them.