Multiprocessing

Processes

- Run multiple independent copies of the Python interpreter
  - In separate processes
  - Possible on different machines
- How to synchronize or cooperate among the different processes
  - By having each process to send messages to one another

Message Passing

Each instance of Python is independent

Programs just send and receive messages

Looks simple enough

Two issues:

- **What is a message**
- **What is the transport mechanism**

Messages

- What kind of messages should we send
  - A message is nothing more than just a bunch of bytes
  - BUT sending a bunch of bytes and having to reassemble it again is a pain
- What we want to do is to send a whole object, and the receiver will get that object
  - This is what we call “serialization”
    - Reduce an object to a bunch of formatted bytes,
    - Send this bytes
    - Reassemble it again to the same object

process1.py

Create a new Folder “mprocess” Under CS355

```
import multiprocessing as mp
import os

def info(title):
        print(title)
        print(' module name:', __name__)
        print(' parent process:', os.getppid())
        print(' process id:', os.getpid())

def f(name):
        info('function f')
        print('hello', name)

if __name__ == '__main__':
        info('main line')
        p = mp.Process(target=f, args=('bob',))
        p.start()
        p.join()
```

NOTICE:
We are getting pid

Remember perf1.py

```
import threading
import time

ntimes = 100000000

def count(n):
        while n > 0:
                n -= 1

# sequential execution
t1_start = time.time()
count(ntimes)
count(ntimes)
t1_end = time.time()
print("Sequential Execution time = ",
t1_end - t1_start)
```

Convert this to using Process

```
# the threaded execution
t1_start = time.time()
thread1 = threading.Thread(target=count,
args=(ntimes,))
thread2 = threading.Thread(target=count,
args=(ntimes,))
thread1.start()
thread2.start()
thread1.join()
thread2.join()
t1_end = time.time()
print("Threaded Execution time = ", t1_end -
t1_start)
```

Some error in running

Just change Thread to Process

https://docs.python.org/3.7/library/multiprocessing.html#the-spawn-and-forkserver-start-methods

You need this enclosure

```
if __name__ == '__main__':
```

This allows the newly spawned Python interpreter to safely import the module and then run the module's `count()` function.

```
if __name__ == '__main__':
        freeze_support()
        set_start_method('spawn')

        p = Process(target=foo)

        p.start()
```

Depending on the platform, `multiprocessing` supports three ways to start a process. These *start methods* are

spawn

The parent process starts a fresh python interpreter process.
The child process will only inherit those resources necessary to
run the process objects `run()` method. In particular, unnecessary file descriptors and
handles from the parent process will not be inherited.
Starting a process using this method is rather slow compared to using *fork* or *forkserver*.
Available on Unix and Windows. The default on Windows.

fork

The parent process uses `os.fork()` to fork the Python interpreter.
The child process, when it begins, is effectively identical to the parent process.
All resources of the parent are inherited by the child process.
Note that safely forking a multithreaded process is problematic.
Available on Unix only. The default on Unix.

forkserver

When the program starts and selects the *forkserver* start method, a server process is started.
From then on, whenever a new process is needed, the parent process connects
to the server and requests that it fork a new process. The fork server process is single
threaded so it is safe for it to use `os.fork()`. No unnecessary resources are inherited.
Available on Unix platforms which support passing file descriptors over Unix pipes.

Data communication using Pipe

Subprocess2.py

```
from subprocess import Popen, PIPE

p1 = Popen(['python','child2.py'],
stdin=PIPE, stdout=PIPE)

# send data to subprocess
p1.stdin.write('this is from master'.encode())
p1.stdin.close() # cause a flush

# read data from subprocess
print("reading from child")
ss = p1.stdout.readline()
print(ss)

print("exit main")
```

child2.py

```
# this is child process
import sys

s1 = input()
sys.stderr.write("child process running
here\n")
sys.stdout.write("child responding: this is
from child")
sys.stdout.close()
sys.stderr.write('end of child process\n')
```

Critical detail: ALWAYS launch in main as shown.
This is required for Windows and optional for Linux or Mac

```
import mprocess1

if __name__ == '__main__':

      p1 = mprocess1.CountdownProcess('p1',10)
      p1.start()

      p2 = mprocess1.CountdownProcess('p2', 15)
      p2.start()
```

Other Process Features

- Joining a process (waits for termination – something you forgot to do)
  - P.start()
  - P.join()
- Making a daemonic process
  - Make the process to run by itself in the background – no frontend
  - How to terminate this process?
  - P.daemon = True
  - P.start()
- Terminating a process
  - P.terminate()

With multiprocessing there is no shared memory – mostly true

- Every process is completely isolated – mostly true
- So there is no locking or synching – mostly true
- Everything now is exchanging messages
  - Pipes
    - Leave this as an exercise
  - Message Queues
    - This is what we will focus
    - You have seen examples of use
  - Message exchange using network IO
    - May work on this

Message Queues

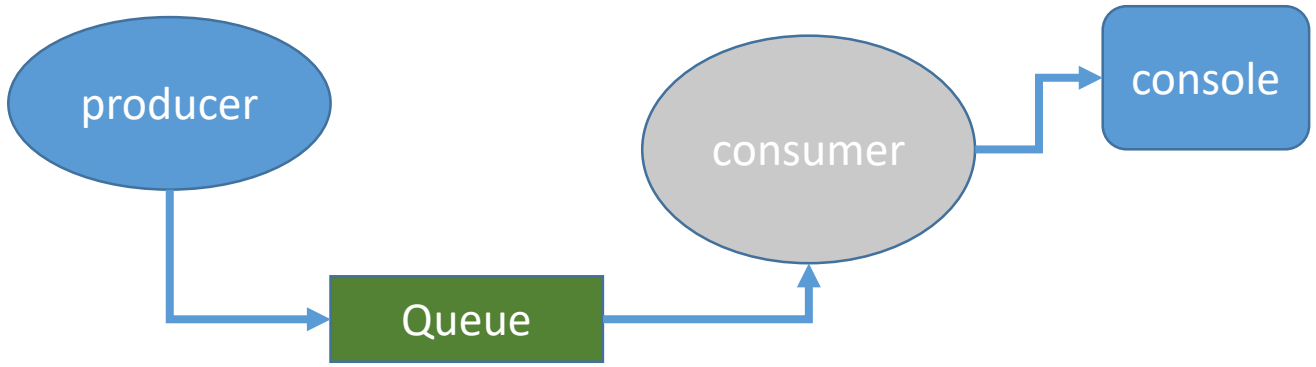
- There is a queue in multiprocessing module
- Programming interface is the same

```
from multiprocessing import Queue

q = Queue()
q.put(item)
item = q.get()
```

Queue Implementation

- Queues are implemented on top of pipes
  - There's a feeder thread running behind the scenes
  - Pickling is done for you in passing objects
  - **Only objects compatible with pickle can be queued**
- Putting an item on a queue returns immediately
  - The feeder thread works on its own to take the item and send it to consumers

```python
import multiprocessing as mp

def consumer(input_q):
while True:
        # get an item from the queue
        item = input_q.get()
        # process the item
        print(mp.current_process().name, ":: ",item)
        # signal completion
        input_q.task_done()

def producer(sequence, output_q):
        for item in sequence:
                # put item on the queue
                output_q.put(item)
```



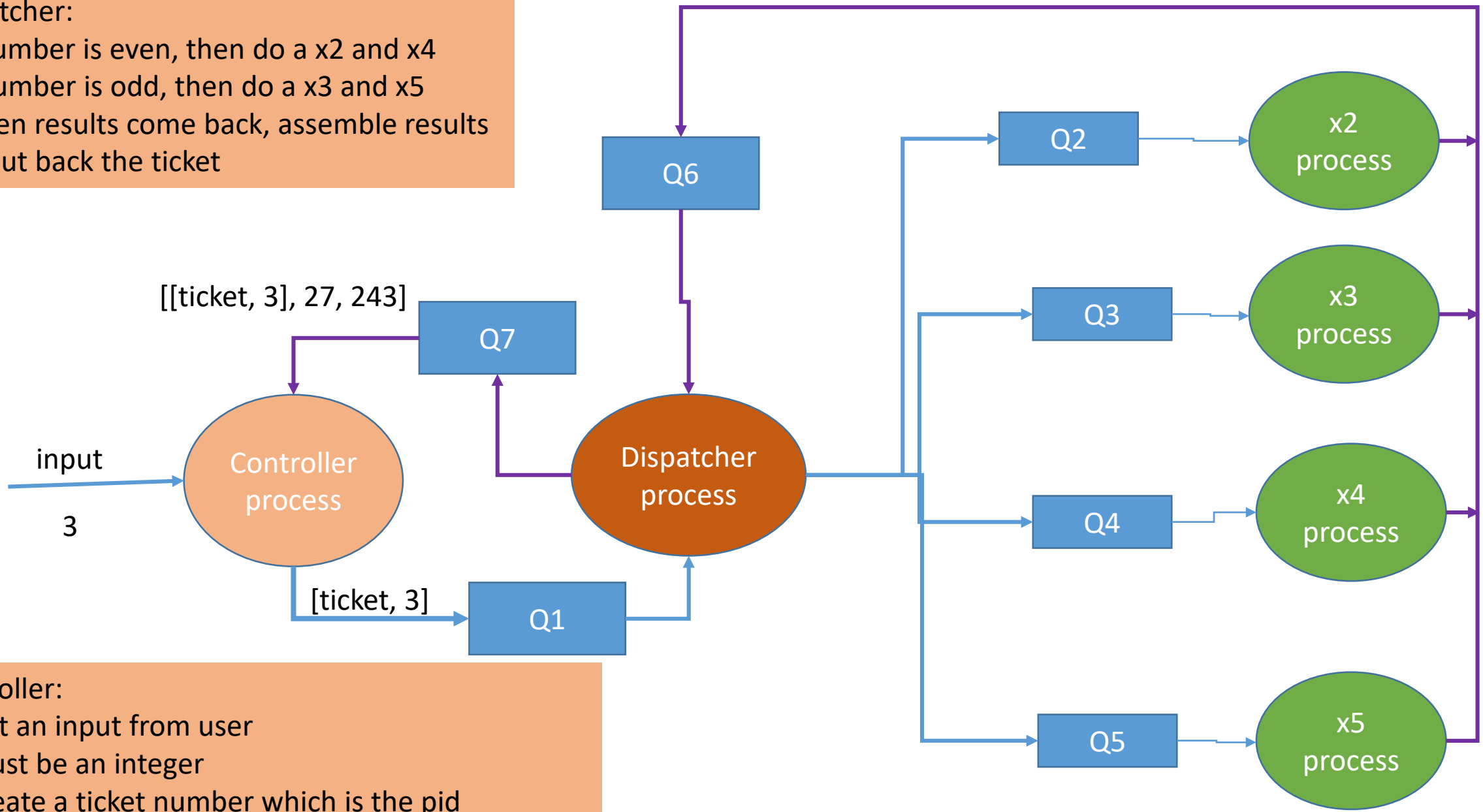
mprocessQueue3.py

```python
if __name__ == '__main__':
# create the queue
qq = mp.JoinableQueue()

# launch the consumer process
consumer_p = mp.Process(target=consumer,
args=(qq,))
consumer_p.daemon = True
consumer_p.start()

#run the producer function on some data
sequence = range(50)
producer(sequence, qq)

# wait for consumer to finish
qq.join()

# finally terminate the daemon process
consumer_p.terminate()
```

Progamming with Queues

```
import multiprocessing as mp

def powerof(doq, resultq, aa):
    alist = doq.get()
    aresult = [aa[0]]
    [aresult.append(i**aa[1]) for i in alist]
    resultq.put(aresult)

if __name__ == '__main__':
    do_queue = mp.Queue()
    result_queue = mp.Queue()

    aalist = [['x^2',2], ['x^3', 3], ['x^4', 4]]

    [do_queue.put([1,2,3,4]) for qq in aalist]

    procs = [mp.Process(target=powerof, args=(do_queue, result_queue, nn)) for nn in aalist]

    [(aproc.start(),aproc.join()) for aproc in procs]

    while not result_queue.empty():
        print("Result:", result_queue.get())
```

Dispatcher:
if number is even, then do a x2 and x4
If number is odd, then do a x3 and x5
When results come back, assemble results and put back the ticket



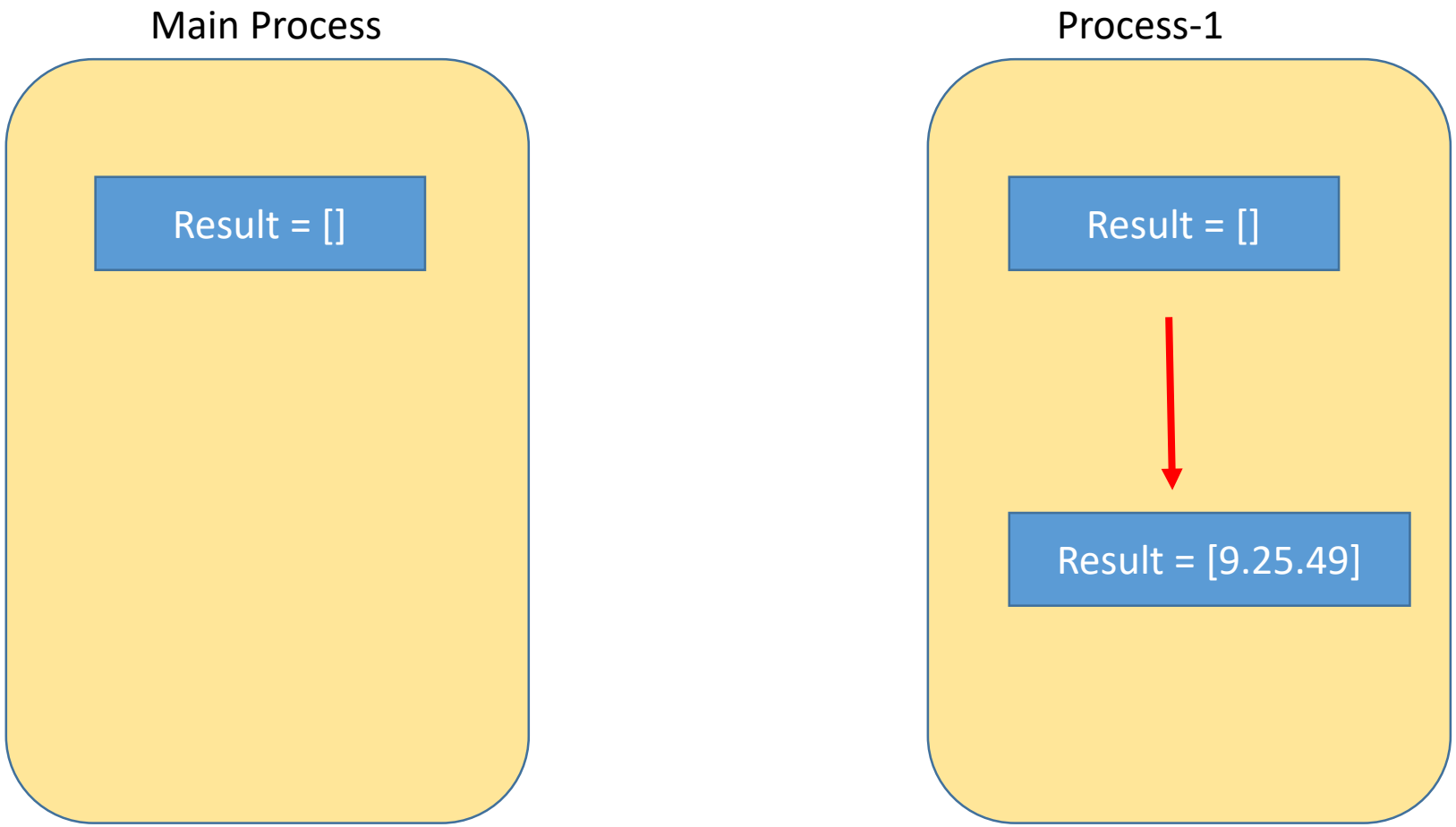
Controller:
Get an input from user
Must be an integer
Create a ticket number which is the pid
send the ticket and number to Q1
Display the results

mprocessShared1.py

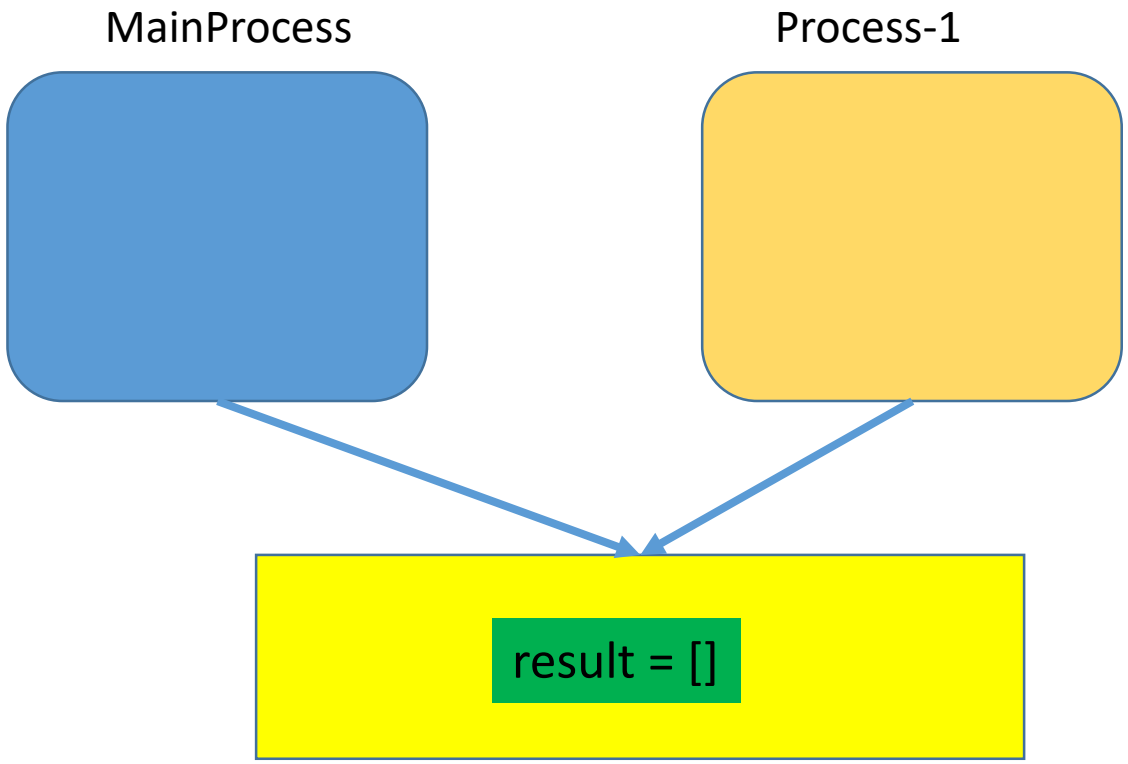
```
import multiprocessing as mp

result = []

def do_square(sequence):
        global result
        for n in sequence:
                result.append(n*n)
        print(mp.current_process().name,"::in:: ",result)

if __name__ == '__main__':
        numbers = [3,5,7]
        pp = mp.Process(target=do_square, args=(numbers,))

        pp.start()
        pp.join()

        print(mp.current_process().name,"::out:: ",result)
```

Explain the result

Not in shared memory


Main Process
Result = []
Process-1
Result = []
Result = [9.25.49]

What we need is shared memory among the processes



mprocessShared2.py

```
import multiprocessing as mp

def do_square(sequence, result):
	for index,n in enumerate(sequence):
		result[index] = n*n
	print(mp.current_process().name,"::in:: ",result[:])

if __name__ == '__main__':
	numbers = [3,5,7]
	# define the shared object as array
	result = mp.Array('i',3)
	pp = mp.Process(target=do_square, args=(numbers,result))

	pp.start()
	pp.join()

	print(mp.current_process().name,"::out:: ",result[:])
```

Other shared objects

- Queue is a shared object
- Other shared objects like Value will be left as an exercise

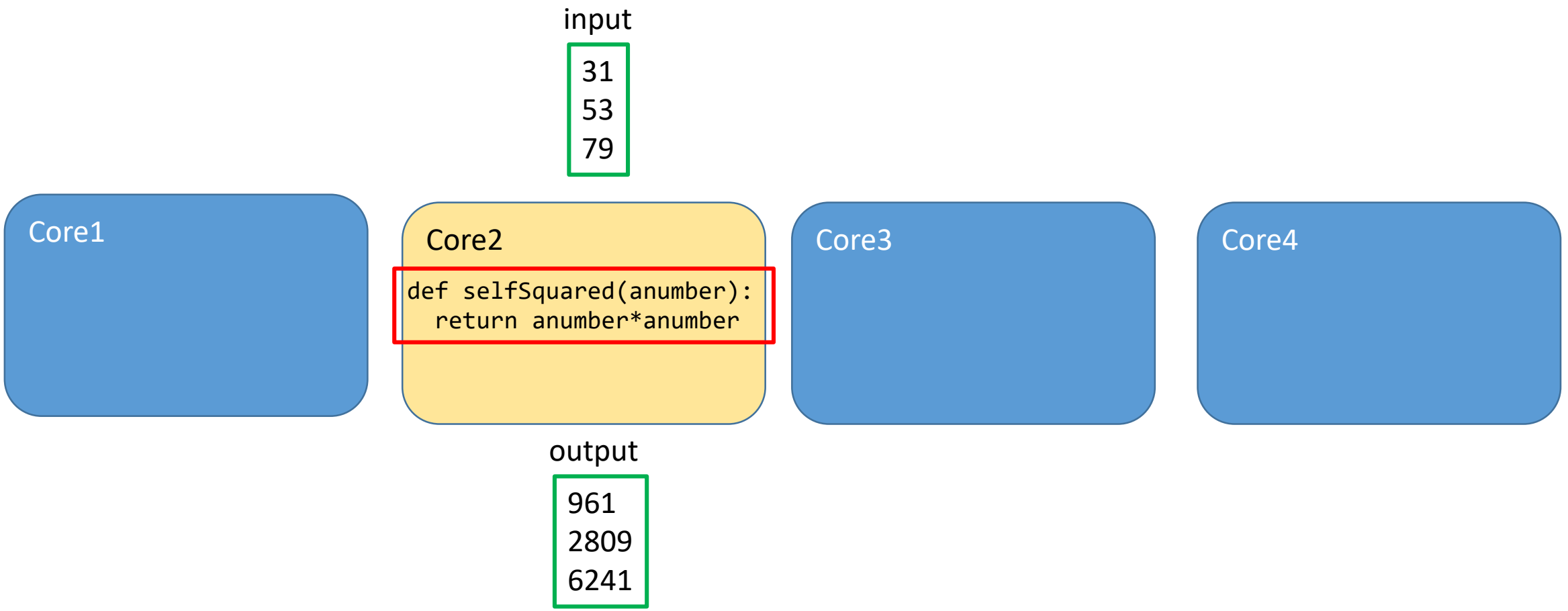
The Pool Class

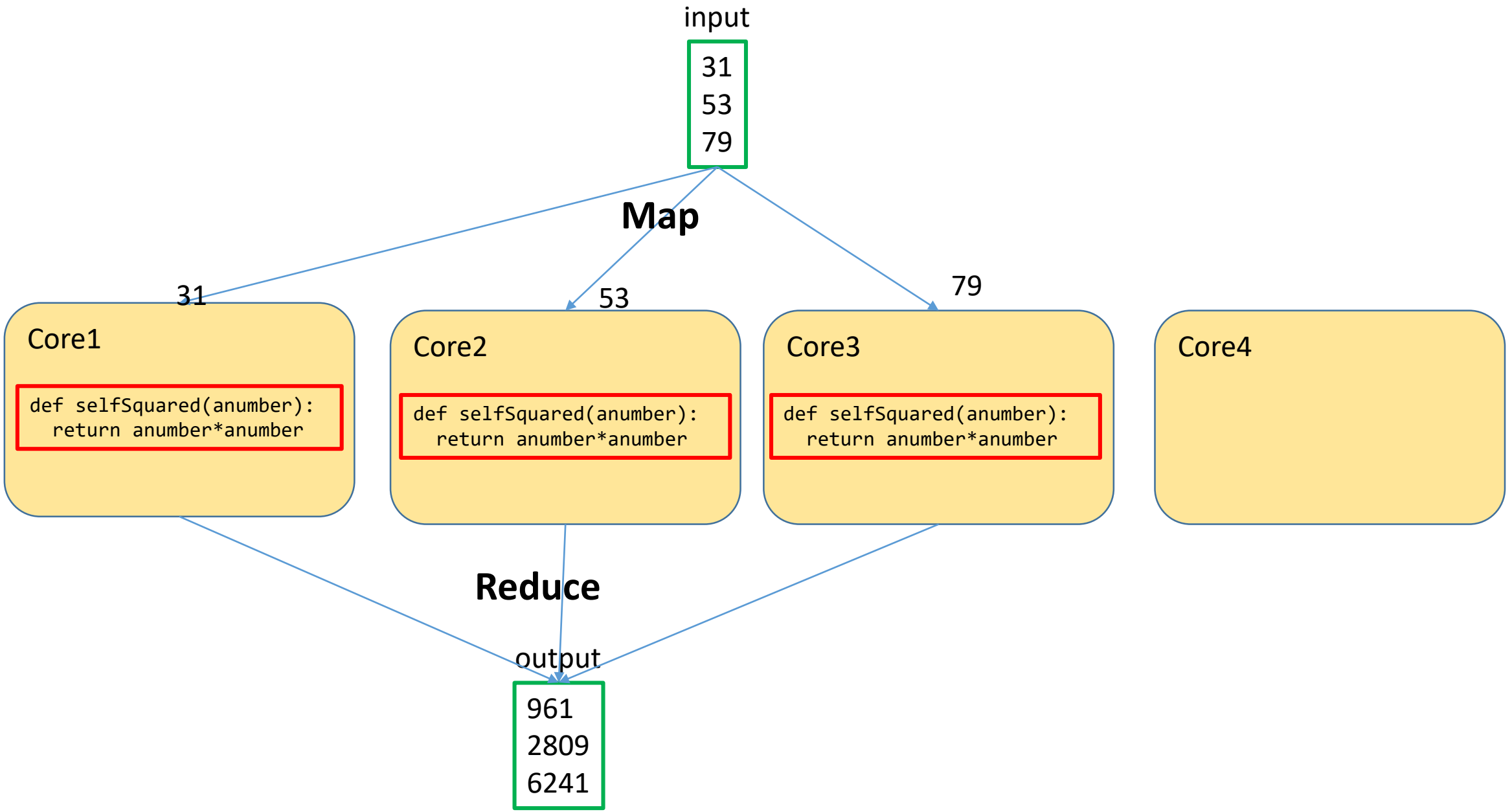
What is happening?

Each process only works on One item in the list

```
import multiprocessing as mp

def selfSquared(anumber):
        print(mp.current_process().name, ':: number = ',anumber)
        return anumber*anumber

if __name__ == '__main__':
        print('cpu_count = ',mp.cpu_count())
        numberSequence = [31,53,79]
        pool = mp.Pool(processes=3)
        print(mp.current_process().name, ':: ',pool.map(selfSquared, numberSequence))
```

- **Create an instance of Pool with 3 processes**
- **Use the map method to map a function and an iterable to each process**
- **Finally, print the result**


input
31
53
79
Core1
Core2
def selfSquared(anumber):
return anumber*anumber
Core3
Core4
output
961
2809
6241


input
31
53
79
Map
31
53
79
Core1
def selfSquared(anumber):
return anumber*anumber
Core2
def selfSquared(anumber):
return anumber*anumber
Core3
def selfSquared(anumber):
return anumber*anumber
Core4
Reduce
output
961
2809
6241

```
import os

def toFahrenheit(tempC):
       print(os.getpid(), '::toF:: ', tempC)
       return ((float(9)/5)* tempC + 32)

def toCentigrade(tempF):
       print(os.getpid(), '::toC:: ', tempF)
       return ((float(5)/9)* (tempF-32))

temps = (36, 37,38,39)
F = map(toFahrenheit, temps)
F1 = list(F)
C = map(toCentigrade, F1)
C1 = list(C)

print(F)
print(F1)
print("==============")
print(F1)
print(C1)
```

normal map and reduce

mapReduce1.py

All computation done within One process possibly with Threads..

Pool of processes

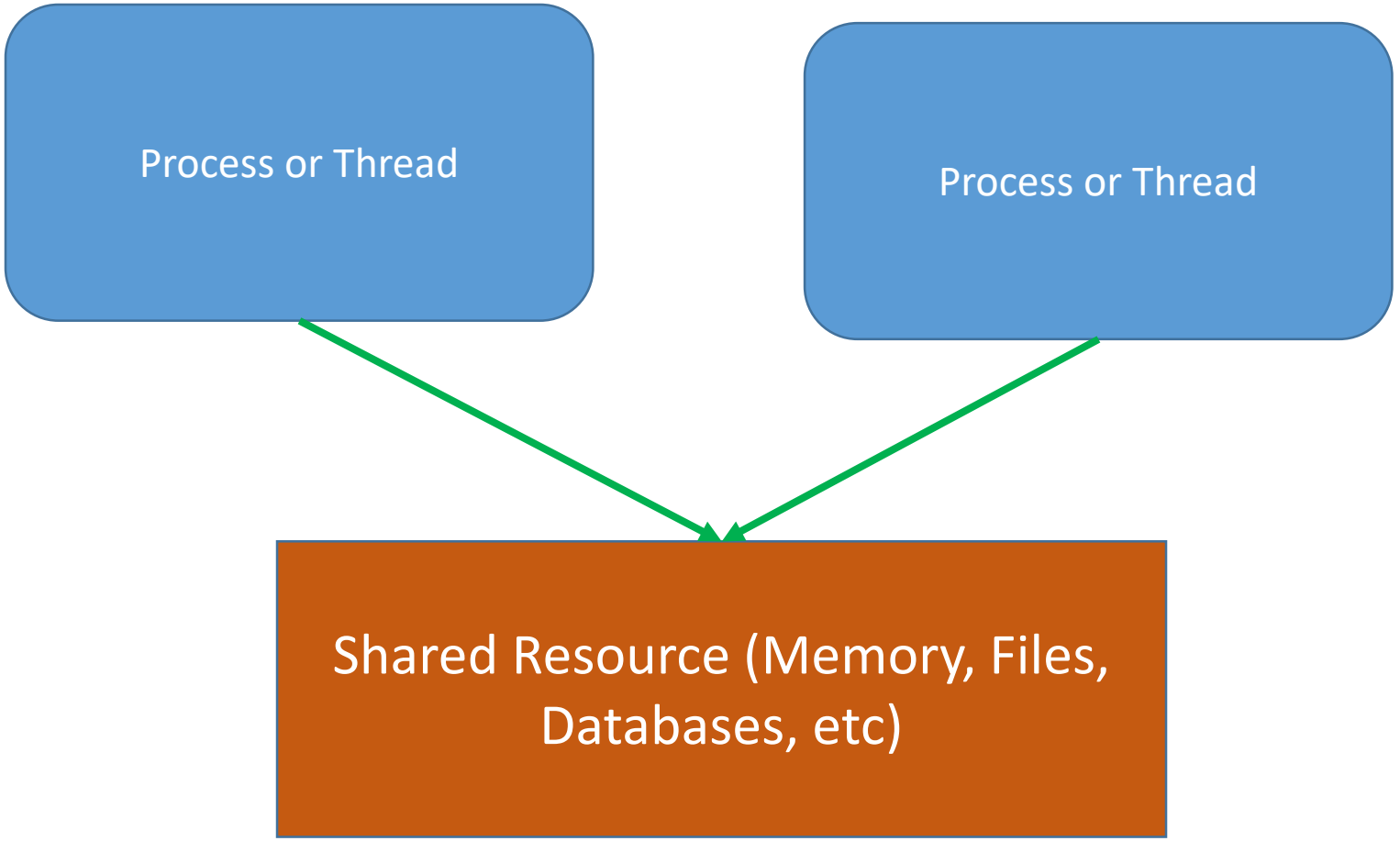
Map, filter and reduce

- Python provides several functions to enable a functional approach to programming.
- Functional programming is all about expressions.
- Expression oriented functions of Python:
  - map(aFunction, aSequence)
  - filter(aFunction, aSequence)
  - reduce(aFunction, aSequence)
  - lambda
  - list comprehension

Benefits of map

- Since it's a built-in, **map** is always available and always works the same way.
- It also has some performance benefit because it is usually faster than a manually coded **for** loop.
- On top of those, **map** can be used in more advance way. For example, given multiple sequence arguments, it sends items taken form sequences in parallel as distinct arguments to the function

Concurrent access in multiprocessing



```
import time
from multiprocessing import Process, Value

def func(val):
    for i in range(50):
        time.sleep(0.01)
        val.value += 1

if __name__ == '__main__':
    v = Value('i', 0)
    procs = [Process(target=func, args=(v,)) for i in range(10)]

    for p in procs:
        p.start()
    for p in procs:
        p.join()

    print("value = ", v.value)
```

Run program a few time.

You should see different Values for each run

NOW, add a lock

mprocessLock1.py

```
import time
from multiprocessing import Process, Value,
Lock

class Counter(object):
        def __init__(self, initval=0):
        self.val = Value('i', initval)
        self.lock = Lock()

        def increment(self):
                with self.lock:
                        self.val.value += 1

        def value(self):
                with self.lock:
                        return self.val.value

        def func(counter):
                for i in range(50):
                        time.sleep(0.01)
                        counter.increment()
```

```
if __name__ == '__main__':
counter = Counter(0)
procs = [Process(target=func, args=(counter,))
for i in range(10)]

for p in procs: p.start()
for p in procs: p.join()

print("Counter = ",counter.value())
```

Summary of things learned

- How to use multiprocessing module
  - To target regular functions
  - Communicate between processes using Queues and Pipes
  - Concurrency control using synchronization primitives for Threads
  - And other stuff
- Hopefully have expanded awareness of how Python works under the covers as well as some of the pitfalls and tradeoffs.
- Workings with the operating system