Parallel Computing in Python

By Hayley Roy Gill & John Fantell

Python: An Overview

- Dynamically-typed, interpreted language
- Applications: Rapid prototyping, Machine Learning, Data Science, Data Visualization, Web Development, Computer Vision, Simulations, IoT
- 3rd Most Popular Programming Language 2019 [1]
- De-facto Programming Language for Deep Learning (TensorFlow) and Computer Vision (OpenCV)

Terminology

- Process
  - Program in execution
  - Managed by the operating system
- CPU (Central Processing Unit)
  - Executes instructions of a computer program
  - Typically have several independent cores
  - Each core can run one process
- Thread
  - Wikipedia: “Smallest sequence of programmed instructions that can be managed independently by a scheduler”
  - Component of a process

# of CPU Cores: 8 **# of Threads:** 16

Threading Implementations

- Time-slicing:
  - CPU rapidly switches between different threads to maximize performance
    - CPU utilization, Throughput, Turnaround time, Response time, Waiting time
  - Perception that threads are operating in parallel
- Hardware Thread
  - “Allows single core to interleave memory references and operations” [2]
  - Simultaneous multithreading (SMT): “Share the functional units in a single core” [2]

Multithreading VS Multiprocessing

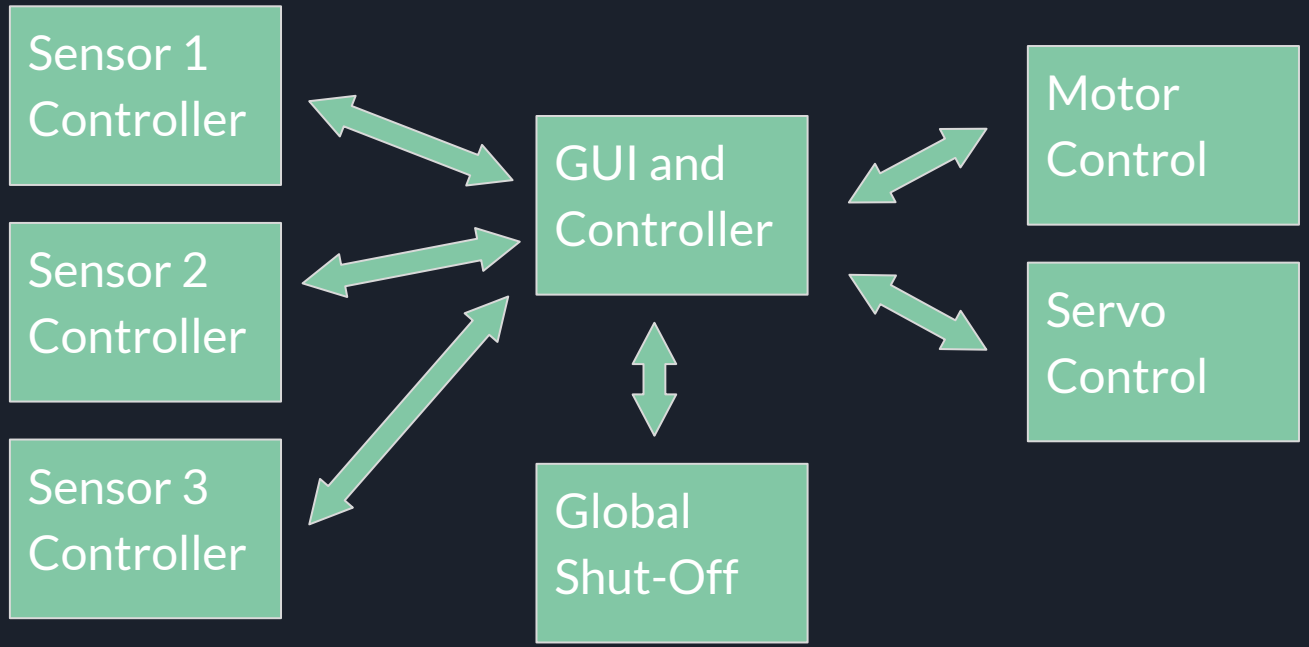
- Multithreading
  - Rapidly alternate between different threads using one core
  - Especially useful when different threads:
    - Are not computationally demanding
    - Must wait for input from another source such as from a sensor sampling request or downloading data

- Multiprocessing
  - Runs processes simultaneously across multiple cores,
  - Useful for situations when you want to:
    - Maximize hardware usage to decrease speed
    - Computing multiple interdependent, computationally demanding operations

Multithreading

- Advantages
  - Low overhead
  - Faster to task-switching than multiprocessing
  - All threads access same memory space (IPC not required)
  - Increases responsiveness of system
- Disadvantages
  - Not as flexible as processes
    - No distributed computing
  - Threads not independent

Multithreading - GUI Example



Multiprocessing

- Advantages
  - Multiple processes operating concurrently
  - Can process high volumes of data
  - If one processor fails, the others will not be affected
  - Each processor can share peripherals
- Disadvantages
  - Large amount of overhead
  - Separate memory for each process
    - Often requires use of Inter-Process Communication
  - Only applicable for CPUs with more than one processor (core)
  - Deadlocks

Demo #1: threading.Thread and multiprocessing.Process

```python
import threading
import multiprocessing
import time

def Fibonacci_driver(n):
    if n<0:
        print("Invalid Input")
    elif n==1:
        return 0
    elif n==2:
        return 1
    else:
        return Fibonacci_driver(n-1)+Fibonacci_driver(n-2)

def Fibonacci(n):
    print(f"Fib {n} is {Fibonacci_driver(n)}")
```

Demo #1: threading.Thread and multiprocessing.Process

```
18 if __name__ == "__main__":
19     num_tasks = 5
20
21     # creating thread
22     start = time.time()
23     threads_list = []
24     for i in range(num_tasks):
25         threads_list.append(threading.Thread(target=Fibonacci, args=(40,)))
26
27     for i in range(num_tasks):
28         threads_list[i].start()
29
30     for i in range(num_tasks):
31         threads_list[i].join()
32
33     end = time.time()
34     thread_duration = end-start
35     print(f"Threads completed their tasks in {thread_duration} s!\n")
36
```

Demo #1: threading.Thread and multiprocessing.Process

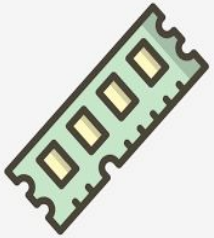
```
37      # creating process
38      start = time.time()
39      processes_list = []
40
41      for i in range(num_tasks):
42          processes_list.append(multiprocessing.Process(target=Fibonacci, args=(40,)) )
43
44      for i in range(num_tasks):
45          processes_list[i].start()
46
47      for i in range(num_tasks):
48          processes_list[i].join()
49
50      end = time.time()
51      process_duration = end-start
52      print(f"Processes completed their tasks in {process_duration} s!\n")
53
54      if process_duration < thread_duration:
55          print("Multiprocessing wins!")
56      else:
57          print("Multithreading wins!")
```

Demo #1: threading.Thread and multiprocessing.Process

```
(dl4cv) Johns-MacBook-Pro-2:Desktop johnfantell$ python speed_test.py
Fib 40 is 63245986
Fib 40 is 63245986
Fib 40 is 63245986
Fib 40 is 63245986
Fib 40 is 63245986
Threads completed their tasks in 167.1014928817749 s!

Fib 40 is 63245986
Fib 40 is 63245986
Fib 40 is 63245986
Fib 40 is 63245986
Fib 40 is 63245986
Processes completed their tasks in 55.684601068496704 s!

Multiprocessing wins!
```

Demo #2: multiprocessing.Pool

```
17  if __name__ == "__main__":
18      num_tasks = 5
19
20      # creating process
21      start = time.time()
22      with Pool(num_tasks) as p:
23          print(p.map(Fibonacci_driver, [40, 40, 40, 40, 40]))
24
25      p.join()
26      end = time.time()
27      process_duration = end-start
28      print(f"Processes completed their tasks in {process_duration} s!\n")
```



Initial Memory Setup for Python Multiprocessing

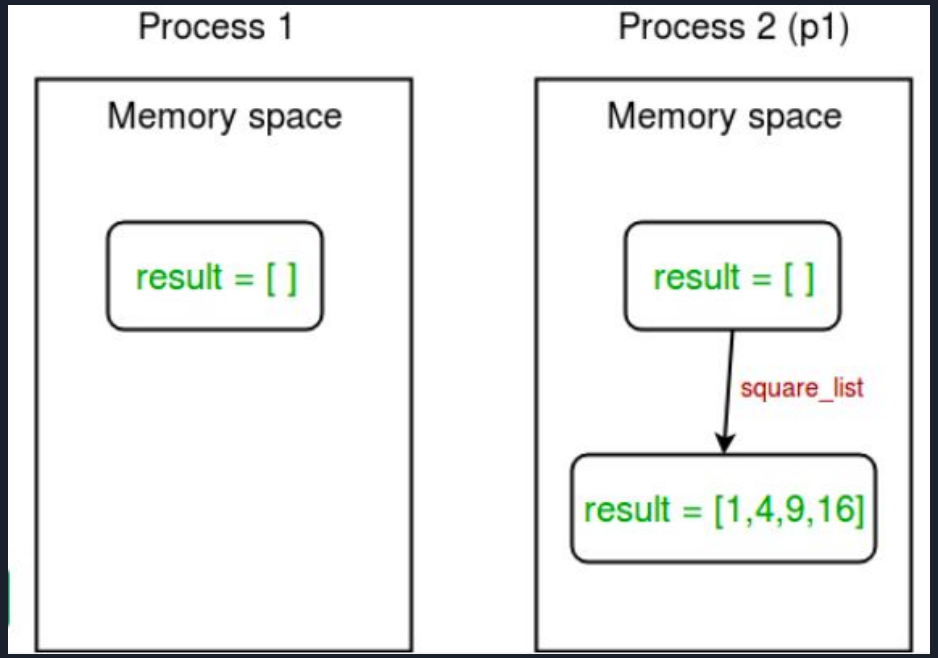
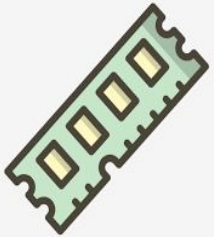


Image Source: [5]

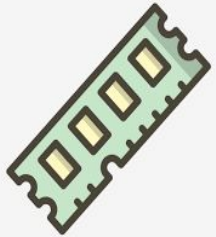


Initial Memory Setup for Python Multiprocessing - Code Excerpt

```
import multiprocessing

# empty list with global scope
result = []

def square_list(mylist):
    """
    function to square a given list
    """
    global result
    # append squares of mylist to global list result
    for num in mylist:
        result.append(num * num)
    # print global list result
    print("Result(in process p1): {}".format(result))

if __name__ == "__main__":
    # input list
    mylist = [1,2,3,4]

    # creating new process
    p1 = multiprocessing.Process(target=square_list, args=(mylist,))
    # starting process
    p1.start()
    # wait until process is finished
    p1.join()

    # print global result list
    print("Result(in main program): {}".format(result))
```

```
Result(in process p1): [1, 4, 9, 16]
Result(in main program): []
```

Image Source: [5]



Multiprocessing Shared Memory

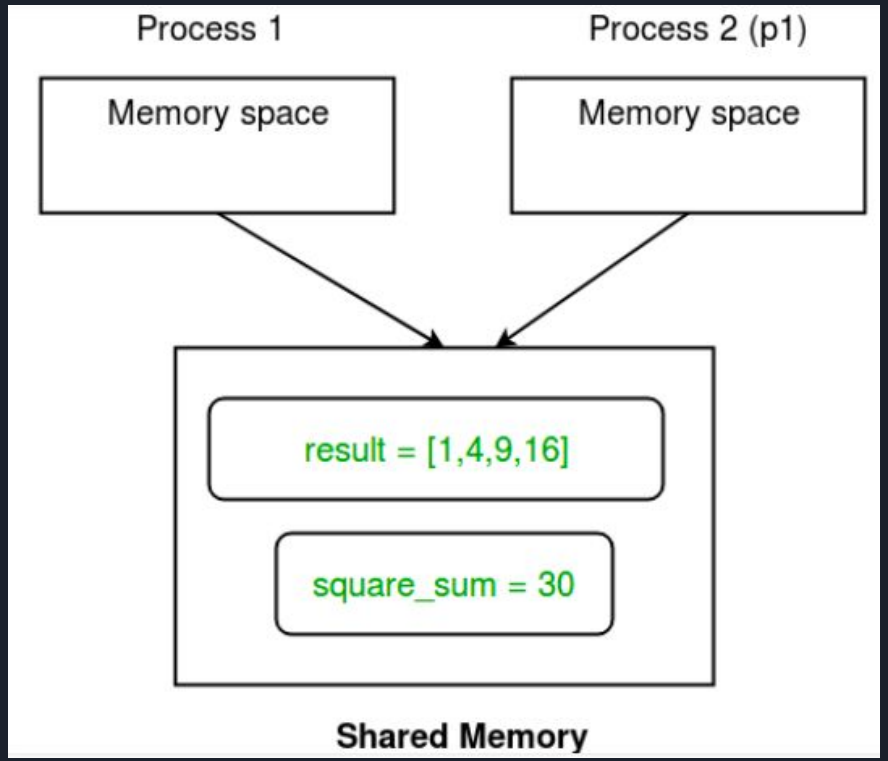
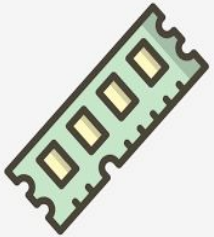


Image Source: [5]



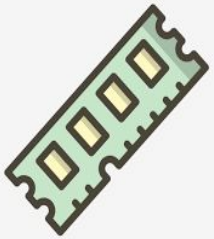
Multiprocessing Shared Memory - Code Excerpt

```
result = multiprocessing.Array('i', 4)
```

```
square_sum = multiprocessing.Value('i')
```

```
p1 = multiprocessing.Process(target=square_list, args=(mylist, result, square_sum))
```

Image Source: [5]



Server Processes

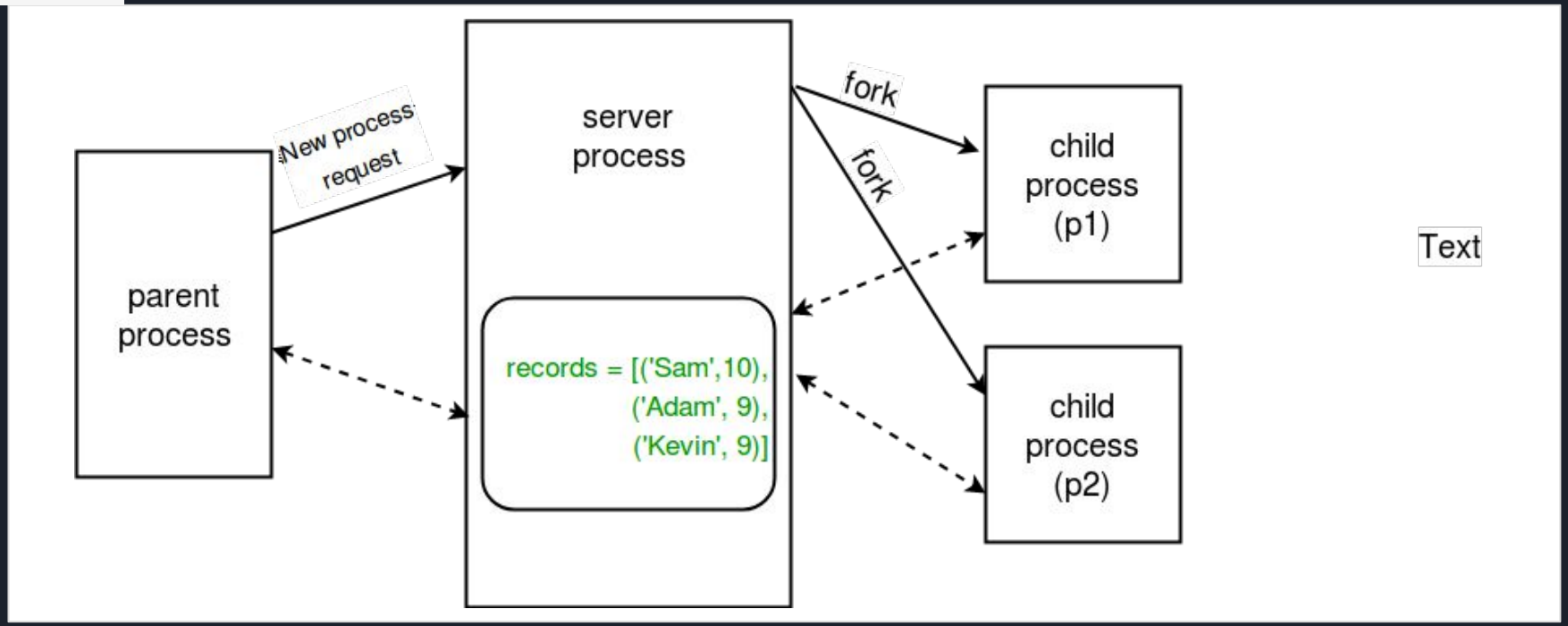


Image Source: [5]

Demo #3: threading.Lock and shared memory

```python
from threading import Thread, Lock
import time

def f(l, i):
    l.acquire()
    try:
        if i%2 == 0:
            time.sleep(.5)
            global_list.append(i)
        else:
            global_list.append(i)
    finally:
        l.release()

```

```python
from threading import Thread, Lock
import time

def f(l, i):
    # l.acquire()
    # try:
    if i%2 == 0:
        time.sleep(.5)
        global_list.append(i)
    else:
        global_list.append(i)
    # finally:
    #   l.release()
```

Lock: “Once a process or thread has acquired a lock, subsequent attempts to acquire it from any process or thread will block until it is released”

Demo #3: threading.Lock and shared memory

```
15  if __name__ == '__main__':
16      global_list = []
17      numbers = 100
18      lock = Lock()
19
20      threads_list = []
21      for i in range(numbers):
22          threads_list.append(Thread(target=f, args=(lock,i+1)))
23
24      for i in range(numbers):
25          threads_list[i].start()
26
27      for i in range(numbers):
28          threads_list[i].join()
29
30      print(global_list)
31      print()
```

Demo #3: threading.Lock and shared memory

```
Using No Lock
```

List not in order

```
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41,
43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81,
83, 85, 87, 89, 91, 93, 95, 97, 99, 2, 48, 8, 20, 12, 24, 26, 30, 10, 16, 64, 34
, 38, 42, 66, 46, 54, 32, 18, 36, 44, 28, 6, 56, 22, 50, 40, 14, 58, 60, 62, 4,
52, 68, 70, 72, 74, 76, 90, 80, 82, 92, 96, 98, 100, 86, 84, 94, 78, 88]

(DL4CV) Johns-MacBook-Pro-2:Desktop johnfantell$ python demo_3.py
Using Lock
```

List in order

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,
83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]
```

Demo #4: multiprocessing.Lock and multiprocessing.Manager

```
15  if __name__ == '__main__':
16      global_list = Manager().list()
17      numbers = 100
18      lock = Lock()
19
20      processes_list = []
21      for i in range(numbers):
22          processes_list.append(Process(target=f, args=(lock,i+1)))
23
24      for i in range(numbers):
25          processes_list[i].start()
26
27      for i in range(numbers):
28          processes_list[i].join()
29
30      print(global_list)
31      print()
```

Demo #4: multiprocessing.Lock and multiprocessing.Manager

```
Using No Lock

[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, 37, 39, 41,
43, 45, 47, 49, 51, 53, 55, 57, 59, 61, 63, 65, 67, 69, 71, 73, 75, 77, 79, 81,
83, 85, 87, 89, 91, 93, 95, 97, 99, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24,
26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54, 56, 58, 60, 62, 64,
66, 68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96, 98, 100]

(DL4CV) Johns-MacBook-Pro-2:Desktop johnfantell$ python demo_4.py
Using Lock

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22,
23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62,
63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82,
83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

(DL4CV) Johns-MacBook-Pro-2:Desktop johnfantell$
```

List not in order

List in order

Demo #5: multiprocessing.Pipe

Pipe: pair of connection objects that facilitate communication between two processes

```
# Created by Mike Pennington
# Source: https://stackoverflow.com/questions/8463008/
# multiprocessing-pipe-vs-queue
from multiprocessing import Process, Pipe
import time

def reader_proc(pipe):
    ## Read from the pipe; this will be spawned as a separate Process
    p_output, p_input = pipe
    p_input.close()    # We are only reading
    while True:
        msg = p_output.recv()    # Read from the output pipe and do nothing
        if msg=='DONE':
            break

def writer(count, p_input):
    for ii in range(0, count):
        p_input.send(ii)             # Write 'count' numbers into the input pipe
    p_input.send('DONE')

if __name__=='__main__':
    for count in [10**4, 10**5, 10**6]:
        # Pipes are unidirectional with two endpoints:  p_input ------> p_output
        p_output, p_input = Pipe()  # writer() writes to p_input from _this_ process
        reader_p = Process(target=reader_proc, args=((p_output, p_input),))
        reader_p.daemon = True
        reader_p.start()     # Launch the reader process

        p_output.close()       # We no longer need this part of the Pipe()
        _start = time.time()
        writer(count, p_input) # Send a lot of stuff to reader_proc()
        p_input.close()
        reader_p.join()
        print("Sending {0} numbers to Pipe() took {1} seconds".format(count,
            (time.time() - _start)))
```

Demo #6: multiprocessing.Queue

Queue is built on top of Pipe (Slower than Pipe)

Use Queue if multiple consumers and producers

```
# Created by Mike Pennington
# Source: https://stackoverflow.com/questions/11515944/
# how-to-use-multiprocessing-queue-in-python

from multiprocessing import Process, Queue
import time
import sys

def reader_proc(queue):
    ## Read from the queue; this will be spawned as a separate Process
    while True:
        msg = queue.get()         # Read from the queue and do nothing
        if (msg == 'DONE'):
            break

def writer(count, queue):
    ## Write to the queue
    for ii in range(0, count):
        queue.put(ii)             # Write 'count' numbers into the queue
    queue.put('DONE')

if __name__=='__main__':
    pqueue = Queue() # writer() writes to pqueue from _this_ process
    for count in [10**4, 10**5, 10**6]:
        ### reader_proc() reads from pqueue as a separate process
        reader_p = Process(target=reader_proc, args=((pqueue),))
        reader_p.daemon = True
        reader_p.start()        # Launch reader_proc() as a separate python process

        _start = time.time()
        writer(count, pqueue)    # Send a lot of stuff to reader()
        reader_p.join()         # Wait for the reader to finish
        print("Sending {0} numbers to Queue() took {1} seconds".format(count,
            (time.time() - _start)))
```

Multithreading Issues in Python

- GIL: Global Interpreter Lock [3]
  - Prevents multiple threads from reading or writing Python objects (CPython bytecode)
  - Required to make Python memory safe
    - Python's reference implementation CPython is not memory safe
  - Prevents true multithreading
  - I/O, image processing, Numpy happen outside of GIL and not affected
- Degraded performance [3]
  - Two threads calling a function may be twice as slow as one thread calling one function
  - I/O bound threads might be scheduled ahead of CPU-bound threads
  - Prevents signal (asynchronous notification) delivery

References

1. Chan, Rosalie. “The 10 Most Popular Programming Languages, According to the 'Facebook for Programmers'.” *Business Insider*, Business Insider, 22 Jan. 2019, www.businessinsider.com/the-10-most-popular-programming-languages-according-to-github-2018-10#3-python-8.
2. Gropp, William. “Lecture 16: Threads .” *Cs.illinois.edu/~Wgropp*, University of Illinois at Urbana-Champaign, www.cs.illinois.edu/~wgropp. Slides 13-14.
3. “Global Interpreter Lock.” Python Wiki, wiki.python.org/moin/GlobalInterpreterLock.
4. “Multiprocessing - Process-Based Parallelism.” Python 3.4.10 Documentation, https://docs.python.org/3.4/library/multiprocessing.html?highlight=process.
5. “Multiprocessing in Python: Set 2 (Communication between Processes).” GeeksforGeeks, 9 Feb. 2018, www.geeksforgeeks.org/multiprocessing-python-set-2/.