Programación Concurrente con Python

Antonio J. Tomeu[1]

[1]Departamento de Ingeniería Informática
Universidad de Cádiz

PCTR, 2018

Contenido

1. Elementos (muy) Básicos de Python
2. Programación Multihebra con Python
3. La pesadilla del GIL
4. Programación Multiproceso con Python

¿Qué es Pyhon?

- Es un lenguaje de programación interpretado y multiplatoforma
- Incorpora orientación a objetos, programación funcional y concurrencia/paralelismo
- Con un modelo de tipado dinámico
- El ámbito de las estructuras de control y de las subrutinas se define mediante indentación
- Incorpora funcionalidades añadiendo módulos (como `threading` o `multiprocessing`)

Elementos Básicos de Python

Código 1: ./code/discriminante.py

```
1  #esto es un comentario
2  #un modulo se importa asi:
3  import math
4
5  #el tipado de las variables es dinamico...
6  a=int(input('coeficiente cuadratico? '))
7  b=int(input('coeficiente lineal? '))
8  c=int(input('coeficiente libre? '))
9  discriminante=(b**2)-4*a*c
10 print('discriminante=', discriminante)
11 #la indentacion determina el ambito de las estructuras de
       control
12 if discriminante>0:
13   raiz=math.sqrt(discriminante)
14   print(raiz)
15 else:
16   print('discriminante negativo...')
```

Bucles for: cálculo de π con Código Secuencial

Código 2: ./code/pimontecarlo.py

```
1  import math
2  import random
3
4  dardos=int(input('cuantos dardos? '))
5  aciertos=0
6
7  for cont in range(dardos):
8    cx=random.random()
9    cy=random.random()
10   if math.sqrt((cx**2)+(cy**2))<1:
11     aciertos=aciertos+1
12
13 piaprox=4.0*(aciertos/dardos)
14 print(piaprox)
```

Tomando Tiempos de Ejecución

Código 3: ./code/prod_escalar.py

```
1  #numpy is a module for scientific computation with Python
2  import numpy as np
3  from timeit import default_timer as timer
4
5  def pow(a, b, c):
6      for i in range(a.size):
7          c[i] = a[i] ** b[i]
8
9  def main():
10     vec_size = 320000
11     a = b = np.array(np.random.sample(vec_size),
           dtype=np.float32)
12     c = np.zeros(vec_size, dtype=np.float32)
13
14     start = timer()
15     pow(a, b, c)
16     duration = timer() - start
17     print(duration)
18
19 if __name__ == '__main__':
20     main()
```

Concurrencia con Hebras: La Clase Thread I

Código 4: ./code/threads.py

```
1  from threading import Thread
2  import time
3
4  class myThread(Thread):
5    def init (self):
6      Thread.init(self)
7      self.niter=niter;
8
9    def run(self):
10     for cont in range(10):
11       print(cont)
12     print(self.getName())
13     print(self.isDaemon())
14     print(self.isAlive())
15
16 def main():
17   hebra1=myThread()
18   hebra2=myThread()
19   print('abriendo co-rutina...')
```

Concurrencia con Hebras: La Clase Thread II

```
20      hebra1.start()
21      hebra2.start()
22      hebra1.join()
23      hebra2.join()
24      print('cerrando co-rutina...')
25
26  if __name__ == '__main__':
27          main()
```

Condiciones de Concurso I

Código 5: ./code/race_condition.py

```
1  from threading import Thread
2  import time
3
4  #shared will be within a race condition
5  shared_cont = 0
6  niter       = 1000000
7
8  class myThread(Thread):
9
10   def init (self):
11     Thread.init(self)
12     self.niter=niter;
13
14   def run(self):
15     #doing shared_cont visible for threads
16     global shared_cont
17     for cont in range(niter):
18       #the race condition...
19       shared_cont+=1
```

Condiciones de Concurso II

```
20
21  def main():
22    hebra1=myThread()
23    hebra2=myThread()
24    hebra1.start()
25    hebra2.start()
26    hebra1.join()
27    hebra2.join()
28    print(shared_cont)
29
30  if __name__ == '__main__':
31      main()
```

Compartiendo Objetos I

Código 6: ./code/race_condition_object.py

```
1  import threading
2  import time
3
4  class myCounter:
5    def __init__(self):
6      self.val=0
7    def inc(self):
8      self.val+=1
9    def value(self):
10     return self.val
11
12 def myThread(ref, iters):
13     for cont in range(iters):
14         ref.inc()
15
16
17 if __name__ == '__main__':
18     myThreads=[]
19     nproc=int(input('tasks?'))
```

Compartiendo Objetos II

```
20      iters=int(input('iterations?'))
21      cont=myCounter()
22      for i in range(nproc):
23          th= threading.Thread(target=myThread, args=(cont,
                iters))
24          myThreads.append(th)
25      for i in myThreads: i.start()
26      for i in myThreads: i.join()
27      print(cont.value())
```

Control con Cerrojos: La Clase Lock I

Código 7: ./code/race_condition_lock.py

```
1  #necessary packages are imported
2  from threading import Thread
3  from threading import Lock
4  import time
5
6  shared_cont = 0
7  lock        = Lock()
8
9  #code for threads...
10 def myThread(iter):
11     global shared_cont
12     for i in range(iter):
13         #increment is done with locks...
14         lock.acquire()
15         shared_cont+=1
16         lock.release()
17
18 if  __name__ == '__main__':
19     iter     = int(input('iterations?'))
```

Control con Cerrojos: La Clase Lock II

```
20      myThread1 = Thread(target=myThread, args=(iter,))
21      myThread2 = Thread(target=myThread, args=(iter,))
22      start=time.time()
23      #now, the race condition...
24      myThread1.start()
25      myThread2.start()
26      myThread1.join()
27      myThread2.join()
28      end=time.time()-start
29      print('final value: ',shared_cont)
30      print('seconds... : ',end)
```

Ejecución a Futuro con Ejecutores I

Código 8: ./code/primes_with_pool.py

```
1  #looking for primes with parallel threads and data partition
2  import math
3  import threading
4  import concurrent.futures
5  import time
6
7  shared_cont = 0
8  mylock      = threading.Lock()
9
10 def isPrime(n):
11   if(n<=1): return False
12   for i in range(2, int(math.sqrt(n)+1)):
13     if(n%i==0):
14       return False
15       break
16   return True
17
18 def codeForThread(linf, lsup):
19   global shared_cont
```

Ejecución a Futuro con Ejecutores II

```
20      local_cont = 0
21      for cont in range(linf, lsup):
22        if isPrime(cont):
23          local_cont+=1
24      mylock.acquire()
25      shared_cont+=local_cont
26      mylock.release()
27
28  if __name__ == '__main__':
29      niter    = int(input('range?'))
30      nthreads = int(input('threads?'))
31      linf     = int(0)
32      #now, the data partition...
33      twindow  = int(niter/nthreads)
34      lsup     = twindow
35      start    = time.time()
36      #creatin the thread pool...
37      with
            concurrent.futures.ThreadPoolExecutor(max_workers=nthreads)
            as myExec:
38        for i in range(nthreads):
39          myExec.submit(codeForThread, linf, lsup)
```

Ejecución a Futuro con Ejecutores III

```
40          linf = lsup
41          lsup = lsup+twindow
42
43      print(shared_cont)
44      print(time.time()-start)
```

Monitores

- Es posible disponer de monitores SC en Python
- Se estructuran en torno a la clase `Condition`
- Un objeto de clase `Condition` soporta un cerrojo y un **único** `wait-set`
- Es un modelo similar al de monitores en Java con el API de concurrencia estándar
- Es por tanto obligado el uso de condiciones de guarda

Monitor Productor-Consumidor I

Código 9: ./code/monitor_p_c.py

```
1  from threading import Thread, Condition
2  import time
3
4  items = []
5  condition = Condition()
6
7  class Consumer(Thread):
8
9      def __init__(self):
10         Thread.__init__(self)
11
12     def consume(self):
13         global condition
14         global items
15
16         condition.acquire()
17         while len(items) == 0:
18             condition.wait()
19             print('Consumer notify : no item to consume')
```

Monitor Productor-Consumidor II

```
20            items.pop()
21
22            print('Consumer notify : consumed 1 item')
23            print('Consumer notify : items to consume are',
                  len(items))
24
25            condition.notify()
26            condition.release()
27
28        def run(self):
29            for i in range(20):
30                time.sleep(10)
31                self.consume()
32
33
34    class Producer(Thread):
35
36        def __init__(self):
37            Thread.__init__(self)
38
39        def produce(self):
40            global condition
```

Monitor Productor-Consumidor III

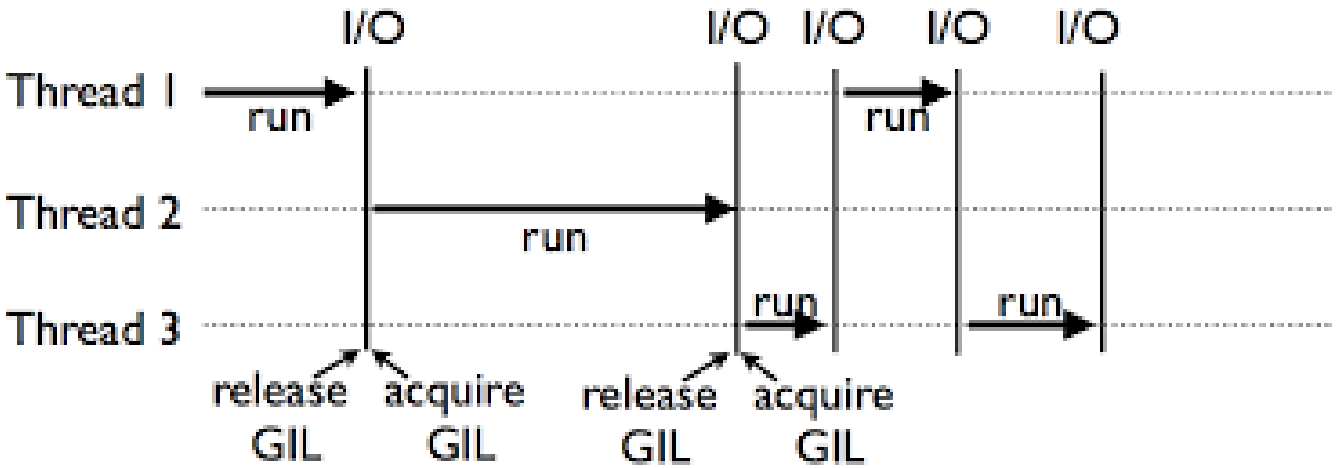
```
41          global items
42
43          condition.acquire()
44          while len(items) == 10:
45              condition.wait()
46              print('Producer notify : items produced are',
                    len(items))
47              print('Producer notify : stop the production')
48          items.append(1)
49
50          print('Producer notify : total items produced',
                len(items))
51
52          condition.notify()
53          condition.release()
54
55      def run(self):
56          for i in range(20):
57              time.sleep(5)
58              self.produce()
59
60  if __name__ == '__main__':
```

Monitor Productor-Consumidor IV

```
61      producer = Producer()
62      consumer = Consumer()
63
64      producer.start()
65      consumer.start()
66
67      producer.join()
68      consumer.join()
```

Y sin embargo, este modelo de hebras... es un bluff

- Python utiliza el Global Intepreter Lock (GIL)
- Lo cuál imposibilita un paralelismo real...
- Obliga a las hebras a ejecutarse secuencialmente
- No es posible lograr *speedups* mayores a uno



¿Y por qué es un bluff...? I

- Veamos por qué el multihebrado de Python es bueno para nada
- Recuerde el problema de búsqueda de números primos en un rango dado
- Estaba resuelto con división de datos entre hebras paralelas
- A partir de él, haga un sencillo experimento: fije un rango (por ejemplo 6 millones), ejecute el programa para un número de tareas $n = 1, 2, 4, 8$ y anote los tiempos
- Analice los tiempos, calcule los *speedups*... y saque sus propias conclusiones

Resultado I

- Ejecución en una plataforma Intel(R) Corei5(TM)-4440 a 3.10GHz con 8.00 GB de RAM
- S.O.: Linux Fedora 28
- 4 cores físicos sin hyperthreading

Cuadro: Tiempos y *Speedups* para rango 6×10^6

Threads	Tiempo (seg.)	Speedup
1	60.91	1.00
2	61.13	0.99
4	60.83	1.00
8	61.75	0.98

Concurrencia con Procesos

- Alternativa a `threading` y el GIL
- Utilizando el módulo `multiprocessing`
- Libre del GIL $\rightarrow$ concurrencia real
- Pero a diferencia de las hebras...
- ... los procesos no comparten memoria
- Se necesitan medios específicos (IPC) para compartirla

Cómo Instanciar un Procesos Rápidamente

- Utilizando el constructor de la clase `Process`
- Recibe dos parámetros:
- Una función/método con el código que el proceso debe ejecutar y
- La lista de parámetros que la función/método necesita
- Ejemplo:

```
multiprocessing.Process(target=miFuncion, args=(arg1,
arg2,..., argn))
```

Creando Procesos Concurrentes

Código 10: ./code/processes.py

```
1  import multiprocessing
2  import time
3
4  def myProcess(iter):
5      global shared_cont
6      for cont in range(iter):
7          print('say hello...')
8
9
10 if __name__ == '__main__':
11     myProcs=[]
12     nproc=int(input('tasks?'))
13     iters=int(input('iterations?'))
14     start=time.time()
15     for i in range(nproc):
16         proc= multiprocessing.Process(target=myProcess,
               args=(iters,))
17         myProcs.append(proc)
18     for i in myProcs: i.start()
19     for i in myProcs: i.join()
20     end=time.time()
21     print('end of execution')
```

Compartir Memoria ahora no es tan fácil I

Código 11: ./code/processes2.py

```
1  import multiprocessing
2  import time
3
4  shared_cont=0
5
6  def myProcess(iter):
7      global shared_cont
8      for cont in range(iter):
9          shared_cont=shared_cont+1
10
11 if __name__ == '__main__':
12     myProcs=[]
13     nproc=int(input('tasks?'))
14     iters=int(input('iterations?'))
15     start=time.time()
16     for i in range(nproc):
17         proc= multiprocessing.Process(target=myProcess,
               args=(iters,))
18         myProcs.append(proc)
```

Compartir Memoria ahora no es tan fácil II

```
19        for i in myProcs: i.start()
20        for i in myProcs: i.join()
21        end=time.time()
22        print(shared_cont)
23        print(shared_cont+1)
24        print('Como explica el output mostrado?')
```

Comunicación Entre Procesos

- Existen dos técnicas para comunicar procesos
- No se basan en el paradigma de memoria compartida
- Se basan en el paradigma de paso de mensajes
- Esto requiere algún tipo de estructura de datos *ad-hoc* como los cauces (pipes) o las colas. Básicamente son memorias accesibles para varios procesos, que deben crearse explícitamente, y que suelen residir en el espacio del *kernel*
- Más detalles sobre el paso de mensajes en Principles of Concurrent and Distributed Programming (M. Ben-Ari), capítulo 8 o en Programación Concurrente (Palma et. al), capítulo 7 y siguientes

Comunicación con *Pipes*

- Un cauce (pipe) es una zona de memoria del kernel
- Puede ser leída y escrita por varios procesos
- Se basan en el paradigma de paso de mensajes
- La comunicación utiliza send-receive

Ejemplo de Uso de Pipes I

Código 12: ./code/multiprocessing_pipes.py

```
1  import multiprocessing
2
3  #tarea para el proceso 1; se escriben en el pipe numeros
4  #del 1 al 10
5  def create_items(pipe):
6      output_pipe, _ = pipe
7      for item in range(10):
8          output_pipe.send(item)
9      output_pipe.close()
10
11 #tarea para el proceso 2; se leen los datos del pipe,
12 #se elevan al cuadrado, y se vuelven a escribir
13 def multiply_items(pipe1, pipe2):
14     close, input_pipe = pipe1
15     close.close()
16     output_pipe, _ = pipe2
17     try:
18         while True:
19             item = input_pipe.recv()
```

Ejemplo de Uso de Pipes II

```
20              output_pipe.send(item*item)
21      except EOFError:
22          output_pipe.close()
23
24  if __name__ == '__main__':
25      #creacion y ejecucion del proceso 1
26      pipe1 = multiprocessing.Pipe(True)
27      process_pipe1 = multiprocessing.Process(
28          target=create_items, args=(pipe1,)
29      )
30
31      #creacion y ejecucion del proceso 2
32      pipe2 = multiprocessing.Pipe(True)
33      process_pipe2 = multiprocessing.Process(
34          target=multiply_items, args=(pipe1, pipe2)
35      )
36
37      process_pipe1.start()
38      process_pipe2.start()
39
40      pipe1[0].close()
41      pipe2[0].close()
```

Ejemplo de Uso de Pipes III

```
42
43      #el proceso principal imprime el contenido del pipe
44      try:
45          while True:
46              print(pipe2[1].recv())
47      except EOFError:
48          print('Programa principal finalizando...')
```

Comunicación con Colas

- Una cola es una estructura de datos FIFO compartida entre procesos
- Se gestiona con los métodos put y get
- Es muy parecido a utilizar contenedores autosincronizados de Java

Ejemplo de Uso de Colas: el Productor-Consumidor I

Código 13: ./code/multiprocessing_queues.py

```
1  import multiprocessing
2  import random
3  import time
4
5  class Producer(multiprocessing.Process):
6
7      def __init__(self, queue):
8          multiprocessing.Process.__init__(self)
9          self.queue = queue
10
11     def run(self):
12         for _ in range(10):
13             item = random.randint(0, 256)
14             self.queue.put(item)
15             print('Process Producer : item %d append to queue %
                   s' % (
16                 item, self.name
17             ))
18             time.sleep(1)
```

Ejemplo de Uso de Colas: el Productor-Consumidor II

```
19              print('The size of queue is %s' %
                    self.queue.qsize())
20
21
22  class Consumer(multiprocessing.Process):
23
24      def __init__(self, queue):
25          multiprocessing.Process.__init__(self)
26          self.queue = queue
27
28      def run(self):
29          while True:
30              if self.queue.empty():
31                  print('The queue is empty')
32                  break
33              else:
34                  time.sleep(2)
35                  item = self.queue.get()
36                  print('Process Consumer : item %d popped from
                        %s\n' % (
37                      item, self.name
38                  ))
```

Ejemplo de Uso de Colas: el Productor-Consumidor III

```
39                  time.sleep(1)
40
41  if __name__ == '__main__':
42      queue = multiprocessing.Queue()
43
44      process_producer = Producer(queue)
45      process_consumer = Consumer(queue)
46
47      #process_producer.start()
48      process_consumer.start()
49
50      #process_producer.join()
51      process_consumer.join()
```

Procesos y Ejecutores

- El módulo `multiprocessing` permite utilizar un ejecutor de procesos
- El ejecutor es dimensionable
- Utiliza el método `map()` para enviar procesos al ejecutor

Ejemplo de Uso un Ejecutor de Procesos I

Código 14: ./code/processes_pol.py

```
1  import multiprocessing
2
3
4  def function_square(data):
5      result = data*data
6      return result
7
8  if __name__ == '__main__':
9      inputs = list(range(100))
10     pool = multiprocessing.Pool(processes=4)
11     pool_output = pool.map(function_square, inputs)
12     pool.close()
13     pool.join()
14     print('Pool:', pool_output)
```

Sincronización de Procesos

- Es posible sincronizar procesos de la forma estándar
- El módulo `multiprocessing` ofrece las clases `Lock`, `Barrier` y `Condition`
- Sin embargo, puesto que Python con procesos no utiliza un paradigma de memoria compartida, sino un paradigma de paso mensajes, el modelo de sincronización estándar con estas clases apenas se utiliza.

Bibliografía

Palach, Jan

Parallel Programming with Python

Packt Publising, 2014

Zaccone, Giancarlo

Python Parallel Programming Cookbook

Packt Publising, 2015