Procesos en Python

Hernández Lozano Juan Pablo Mosqueda García Raúl Isaid

Ruiz Puga Ingrid Pamela Veleros Vega Luis Alfonso

October 2020

En este trabajo se documenta las herramientas que se usan en Python para la creación, implementación y uso de subprocesos. En python, los programas se ejecutan línea por línea, ejecutando solo un proceso a la vez. Los hilos permiten que múltiples procesos fluyan independientemente unos de otros. El subprocesamiento con múltiples procesadores permite que los programas ejecuten múltiples procesos simultáneamente.

1. Threading

Un hilo (thead) es un flujo de control secuencial dentro de un programa, usar hilos permite que un programa ejecute múltiples operaciones aparentemente simultáneamente en el mismo espacio de proceso. La forma más sencilla de usar un Thread es crear una instancia con un función de destino y llamar a start() para que comience a funcionar. Para la creación de hilos se necesita importar la siguiente librería: `import threading` . La forma mas sencilla de hacer un hilo es la siguiente:

```
import logging
import threading
import time

def thread_function(name):
    logging.info("Thread %s: starting", name)
    time.sleep(2)
    logging.info("Thread %s: finishing", name)

if __name__ == "__main__":
    format = "%(asctime)s: %(message)s"
    logging.basicConfig(format=format, level=logging.INFO,
                        datefmt="%H:%M:%S")

    logging.info("Main    : before creating thread")
    x = threading.Thread(target=thread_function, args=(1,))
    logging.info("Main    : before running thread")
```

```
    x.start()
    logging.info("Main    : wait for the thread to finish")
    # x.join()
    logging.info("Main    : all done")
```

Donde las librerias `logging` y `time` encapsulan el hilo de forma que su ejecución se agradable visualmente. Podemos ver que de hecho, el hilo se genera hasta la siguiente linea:

```
    x = threading.Thread(target=thread_function, args=(1,))
```

El cual considera los Atributos target y args, donde target es el tipo de Hilo que se creará, y los argumentos estarán sujetas al tipo de hílo. En este caso el hílo solo ejecutará un $'1'$.

Posteriormente se emplea el método:

```
    x.start()
```

Para inciar/ejecutar el hílo que creamos anteriormente.

2. Multiprocessing

El paquete de multiprocesamiento `multiprocessing` perimite la creación y manejo de los subprocesos en Python, también ofrece concurrencia local y remota.

El multiprocesamiento funciona creando un objeto del `process` y luego llamando a su método `start()` como se muestra a continuación:

```
from multiprocessing import Process


def greeting():
    print 'hello world'

if __name__ == '__main__':
    p = Process(target=greeting)
    p.start()
    p.join()
```

El multiprocesamiento admite dos tipos de canales de comunicación entre procesos: Tubos (pipes) que se utilizan principalmente para comunicación entre procesos, y colas(queue) se utilizan para pasar datos entre procesos:

```
#ejemplo cola (queue)
def is_even(numbers, q):
    for n in numbers:
        if n % 2 == 0:
            q.put(n)

if __name__ == "__main__":

    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=is_even, args=(range(20), q)
    )

```

```
12      p.start()
13      p.join()
14
15      while q:
16          print(q.get())
17
18  #ejemplo tubo (pipe)
19  def f(conn):
20      conn.send(['hello world'])
21      conn.close()
22
23  if __name__ == '__main__':
24      parent_conn, child_conn = Pipe()
25      p = Process(target=f, args=(child_conn,))
26      p.start()
27      print parent_conn.recv()
28      p.join()
```

También podemos asegurarnos de que solo un proceso de ejecute en un momento dado con el método `Lock()` o aplicar el método `release` para terminar el Lock

A continuación se listan algunos métodos y objetos de la clase *multiprocessing* y sus utilidades:

- `Process`: objetos que representan la actividad ejecutada en un proceso separado.
- `run()` método que representa la actividad del proceso
- `Start()` empieza la actividad del proceso
- `is_alive()` regresa si el proceso está en ejecución
- `pid` regresa el id del proceso
- `terminate()` termina el proceso
- `Pipe()` pasar un mensaje entre dos procesos
- `join()` espera hasta que un proceso haya completado su trabajo y termine