Unidad 5

Programación Asíncrona

Manual del Estudiante

CLOUD FORMACIÓN

Maria del Carmen Sierra Fernández

Contenido

Nivel II: Representación, proceso y visualización de Datos 2
Unidad 5: Programación Asíncrona 2
¿Qué es un proceso asíncrono? 2
Objeto Thread (hilos) 4
Identificar el Thread 5
Usando el módulo logging 6
Crear una subclase de Thread 6
Ciclo de vida de un hilo o thread 7
Hilos Demonios 8
Controlar la ejecución de varios demonios 10
Multiprocesamiento 12
Pool 13
Intercambiando objetos entre procesos 14
Sincronización entre procesos 16
Compartiendo estado entre procesos 16
Memoria compartida 16
Proceso servidor (*Server process*) 17
asyncio — E/S Asíncrona 18
Corutinas 18
Funciones 21
Temporizadores. 24
Programación de un hilo en el tiempo. 24
Programación repetitiva de un hilo. 25
Referencias 26

Nivel II: Representación, proceso y visualización de Datos

Unidad 5: Programación Asíncrona

Nuestras aplicaciones, en general tienen un comportamiento síncrono, es decir que un proceso, no va a permitir ejecutar otro hasta que este haya finalizado. Por ende, esto genera una cola o tráfico dentro de nuestra aplicación. Por ejemplo:

Los estudiantes de una Universidad tienen que adjuntar su comprobante de pago y registrarlo en el sistema. El flujo sería el siguiente:

1. El estudiante adjunta su comprobante de pago.
2. El sistema valida el comprobante.
3. El sistema sube el comprobante (guardar copia del comprobante).
4. El sistema envía notificación del pago registrado

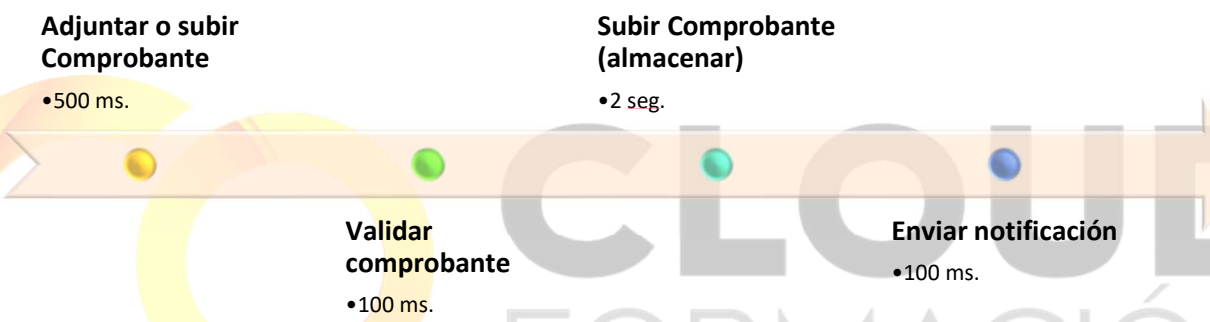


Figura 1: Ejecución del proceso síncrono

Todo el proceso demora 2.7 segundos y la tarea más demora en procesarse es la de subir el comprobante. El tiempo del proceso no está mal, pero si consideramos que tenemos alrededor de 3.000 estudiantes intentando subir su comprobante al mismo tiempo, puede ser un problema.

El proceso se ejecuta como una cola o tráfico: comienza a ejecutarse y el hilo de ejecución no será liberado para otro usuario, hasta que la primera ejecución haya sido completada. Es decir, el primero que llega, es el primero que sale.

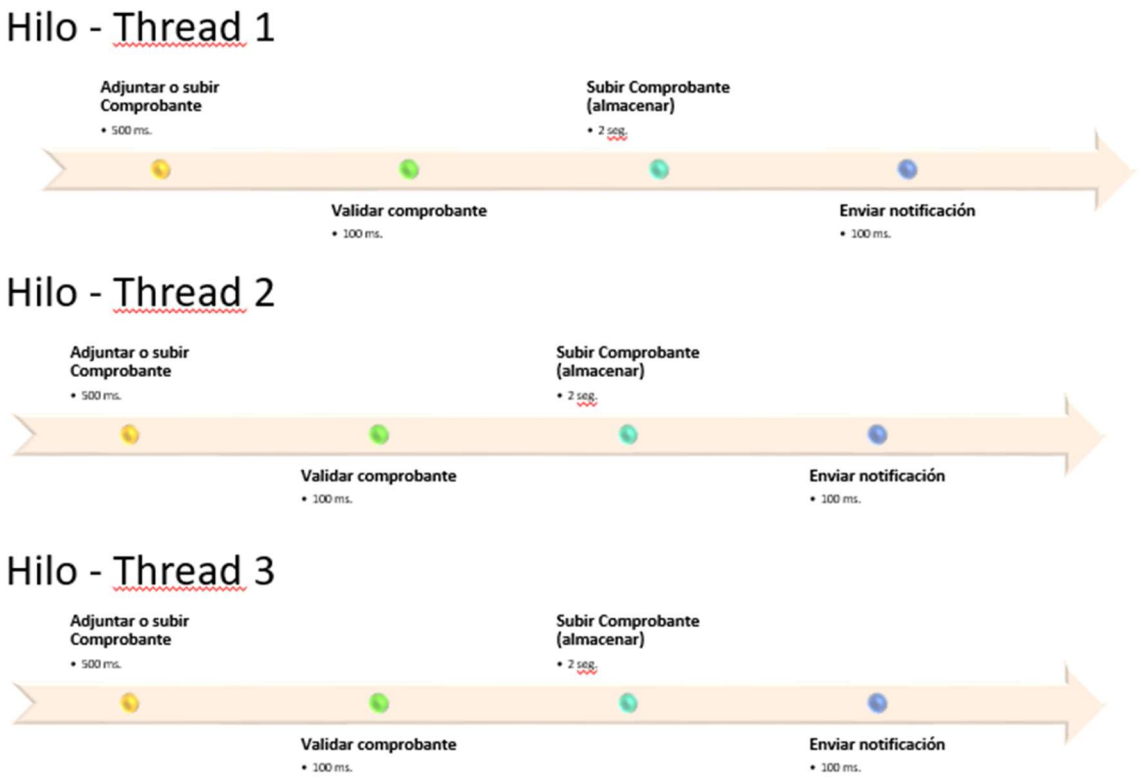
En nuestro ejemplo, el proceso que más tiempo tarda en ejecutarse es la subida del comprobante a nuestro sistema. A este proceso se le conoce como **código bloqueante** debido a que genera los picos de espera.

¿Qué es un proceso asíncrono?

Un proceso asíncrono es aquel que nos permite ejecutar varias tareas simultáneas en nuestras aplicaciones. La ejecución de hilos se trata de un recurso muy utilizado en programación para agilizar la ejecución de las tareas.

Los hilos son una técnica de programación que permite que una aplicación ejecute simultáneamente varias operaciones en el mismo espacio de proceso, en inglés se denomina `Threading`. A cada flujo de ejecución que se origina durante el procesamiento se le denomina hilo (`Thread`) o subproceso, pudiendo realizar o no una misma tarea.

El modo más sencillo para usar un hilo es instanciar un objeto de la clase `Thread` con una función objetivo y hacer una llamada a su método `start()`.



Existen multitud de situaciones en las que la utilización de hilos puede resultar interesante, por ejemplo: una aplicación que realice varias descargas de ficheros en paralelo, una aplicación que pueda abrir o guardar un documento de un gran tamaño mientras se está editando otro, aplicaciones que permitan lanzar varias operaciones de búsqueda al mismo tiempo, aplicaciones que prueben el funcionamiento de un conjunto de sistemas simultáneamente, etc.

Ejecutar varios hilos o subprocesos es similar a ejecutar varios programas diferentes al mismo tiempo, pero con algunas ventajas:

- Los hilos en ejecución de un proceso comparten el mismo espacio de datos que el hilo principal y pueden, por tanto, tener acceso a la misma información o comunicarse entre sí más fácilmente que si estuvieran en procesos separados.
- Ejecutar un proceso de varios hilos suele requerir menos recursos de memoria que ejecutar lo equivalente en procesos separados.

Permite simplificar el diseño de las aplicaciones que necesitan ejecutar varias operaciones de manera concurrente.

Para cada hilo de un proceso existe un puntero que realiza el seguimiento de las instrucciones que se ejecutan en cada momento. Además, la ejecución de un hilo se puede detener temporalmente o de manera indefinida. En general, un proceso sigue en ejecución cuando al menos uno de sus hilos permanece activo, es decir, cuando el último hilo concluye su cometido, termina el proceso, liberándose en ese momento todos los recursos utilizados.

Hay dos formas de definir un hilo: la primera, consiste en pasar al método constructor un objeto invocable, como una función, que es llamada cuando se inicia la ejecución del hilo y, la segunda, radica en crear una subclase de `Thread` en la que se reescribe el método `run()` y/o el constructor `__init__()`.

Objeto Thread (hilos)

En Python un objeto `thread`, es la representación de un hilo, representa una determinada operación que se ejecuta como un subproceso independiente.

```
import threading
def worker(alumno):
    """funcion que realiza el trabajo en el thread"""
    print('Alumno adjuntando comprobante....')
    return
threads = list()
for i in range(3):
    t = threading.Thread(target=worker(i))
    threads.append(t)
    t.start()
Alumno adjuntando comprobante....
Alumno adjuntando comprobante....
Alumno adjuntando comprobante....
```

A los `threads` podemos pasarle parámetros que después son usados por la función objetivo. Podemos utilizar los argumentos `args` y `kwargs` en el constructor o cualquier tipo de objeto puede ser pasado como parámetro a un `thread`.

```
import threading

def worker(alumno):
    """funcion que realiza el trabajo en el thread"""
    print(f'Alumno {alumno} adjuntando comprobante....')
    return
threads = list()
for i in range(1,4):
    t = threading.Thread(target=worker, args=(i,))
```

```
        threads.append(t)
        t.start()
Alumno 1 adjuntando comprobante....
Alumno 2 adjuntando comprobante....
Alumno 3 adjuntando comprobante....
```

Identificar el Thread

Cada instancia de la clase `Thread` tiene un nombre asignado por defecto. Aunque, también podemos utilizar argumentos para nombrar los `threads` que creamos. Nombrar los `threads` puede ser útil, por ejemplo, a la hora de clarificar nuestro código.

```
import threading
import time

def worker():
    print(threading.currentThread().getName(), 'Iniciando...')
    print(threading.currentThread().getName(), 'En pausa...')
    time.sleep(2)
    print(threading.currentThread().getName(),
'Finalizando...')

def servicio():
    print(threading.currentThread().getName(), 'Iniciando...')
    print(threading.currentThread().getName(),
'Finalizando...')

# Asignamos un nombre al thread
t = threading.Thread(target=servicio, name='Servicio')
# Asignamos un nombre al thread
w = threading.Thread(target=worker, name='Worker')
# Thread con su nombre por defecto
z = threading.Thread(target=worker)
w.start()
z.start()
t.start()
Worker Iniciando...
Worker En pausa...
Thread-42 Iniciando...
Thread-42 En pausa...
Servicio Iniciando...
Servicio Finalizando...
WorkerThread-42 Finalizando...
```

En el ejemplo anterior, hemos utilizado `time.sleep(2)`, para bloquear o pausar el hilo `worker`.

Usando el módulo logging

Si vamos a depurar o crear un log relacionado con los `threads`, podemos utilizar el módulo `logging` para ello.

```
import threading
import logging
import time

logging.basicConfig( level=logging.DEBUG,
    format='[%(levelname)s] - %(threadName)-10s : %(message)s')
def worker():
    logging.debug('Iniciando...')
    time.sleep(2)
    logging.debug('Finalizando...')
w = threading.Thread(target=worker, name='Worker')
w.start()
```

```
[DEBUG] - Worker     : Iniciando...
[DEBUG] - Worker     : Finalizando...
```

Como podemos ver, el módulo `logging` soporta la inclusión del nombre del hilo de forma nativa.

Crear una subclase de Thread

Hasta ahora hemos visto una forma de definir hilos, pasando al método constructor un objeto invocable, como una función, que es llamada cuando se inicia la ejecución del hilo. La segunda forma, es creando una subclase de `thread` en la que se reescribe el método `run()` y/o el constructor `__init__()`.

```
import threading

class MiHilo(threading.Thread):

    def __init__(self,num):
        threading.Thread.__init__(self);
        self.num = num;

    def run(self):  # mínimo debe tener un método run
        print("    Ejecutando el hilo:  ",self.num)

# Programa Principal

print("Ejecutando el hilo principal...")
for i in range(0,3):
    t = MiHilo(i);
```

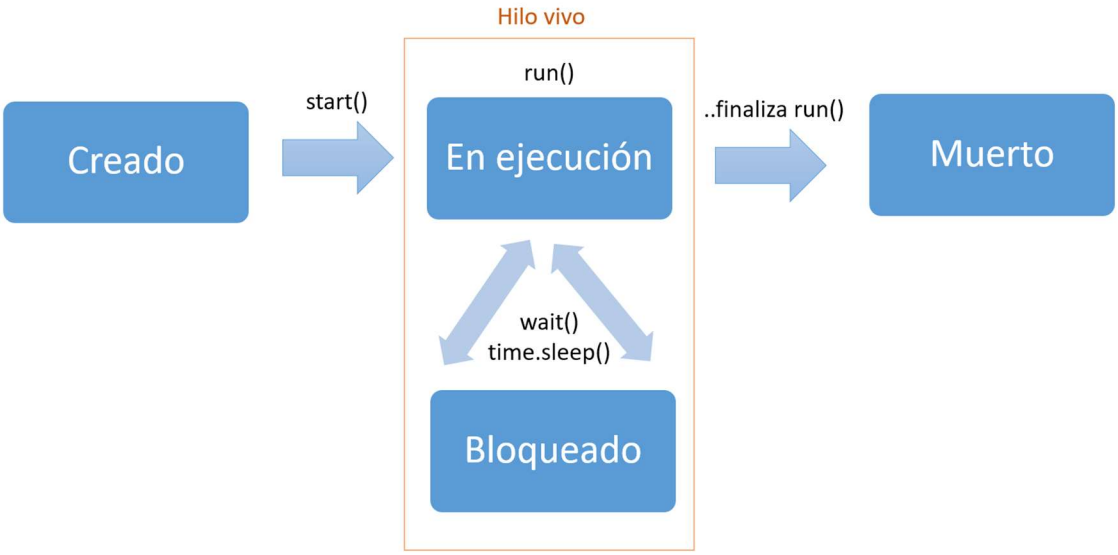
```
t.start();
t.join();
```

`run()` es el método que representa la actividad del hilo y `join()` espera a que todos los hilos terminen para que finalice el hilo principal.

Ciclo de vida de un hilo o thread

Un hilo tiene un ciclo de vida que va desde su creación hasta su terminación. Durante su ciclo de vida cada uno de los hilos o tareas de una aplicación puede estar en diferentes estados, algunos de los cuales se indican a continuación:

- **Creado o nacimiento:** Cuando se acaba de crear un hilo, este continúa en este estado hasta que se invoca el método `start()` del hilo. Cuando un `thread` está en este estado, es sólo un objeto `Thread` vacío o nulo. No se han asignado recursos del sistema todavía para el `thread`. Así, cuando un `thread` está en este estado, lo único que se puede hacer es arrancarlo con `start()`.
- **Listo:** Cuando se invoca el método `start()` del hilo, se dice que está en estado listo.
- **Ejecutable:** cuando el método `start()` se ejecuta, crea los recursos del sistema necesarios para ejecutar el `thread`, programa el `thread` para ejecutarse, y llama al método `run()` del `thread` que se ejecuta en forma secuencial. En este punto el `thread` está en el estado ejecutable. Se denomina así puesto que todavía no ha empezado a ejecutarse.
- **En ejecución:** Un hilo en estado de listo de la más alta prioridad, pasa al estado de ejecución, cuando se le asignan los recursos de un procesador, o sea cuando inicia su ejecución. Aquí el `thread` está en ejecución. Cada hilo tiene su prioridad, hilos con alta prioridad se ejecutan preferencialmente sobre los hilos de baja prioridad.
- **No ejecutable o Bloqueado:** Un hilo continúa la ejecución de su método run(), hasta que pasa al estado de no ejecutable, este se puede originar por los siguientes eventos:
  - Se invoca a `time.sleep()`.
  - El `thread` utiliza su método `wait()` para esperar una condición variable.
- **Muerto:** Un hilo pasa al estado de muerto cuando se termina su método run()



Podemos utilizar el método `is_alive()` para conocer si un hilo está vivo o no. El método `isAlive()` es de versiones anteriores.

Hilos Demonios

Un hilo puede ser marcado como un hilo demonio. El significado de esta marca es que la totalidad del programa de Python finalizará cuando solo queden hilos demonio. El valor inicial es heredado del hilo creador.

El hilo principal del programa puede finalizar, aunque uno o más hilos hijos no hayan terminado su tarea. Es importante destacar, que cuando finalice el hilo principal también lo harán estos hilos especiales llamados **demonios**. Si existen hilos no-demonios, el hilo principal esperará a que éstos concluyan su trabajo. Los demonios son útiles para programas que realizan operaciones de monitorización o de chequeo de recursos, servicios, aplicaciones, etc.

Para declarar un hilo como demonio se asigna **True** al argumento **daemon** al crear el objeto `Thread`, o bien, se establece dicho valor con posterioridad con el método `set_daemon()`.

En el siguiente ejemplo utilizamos dos hilos: un hilo escribirá en un archivo y mientras el otro hilo (el demonio) estará comprobando el tamaño del archivo cada cierto tiempo. Cuando el hilo encargado de escribir termina, todo el programa llega a su fin a pesar de que el contador del demonio no ha alcanzado el valor límite.

```
import time, os, threading

def chequear(nombre):
    '''Chequea tamaño de archivo'''
    contador = 0
    tam = 0
    while contador < 100:
        contador += 1
```

```
            if os.path.exists(nombre):
                estado = os.stat(nombre)
                tam = estado.st_size

            print(threading.current_thread().getName(),
                  contador,
                  tam,
                  'bytes')

            time.sleep(0.1)

    def escribir(nombre):
        '''Escribe en archivo'''
        contador = 1
        while contador <= 10:
            with open(nombre, 'a') as archivo:
                archivo.write('1')
                print(threading.current_thread().getName(),
                      contador)
                time.sleep(0.3)
                contador += 1

    nombre = 'archivo.txt'
    if os.path.exists(nombre):
        os.remove(nombre)

    hilo1 = threading.Thread(name='chequear',
                             target=chequear,
                             args=(nombre,),
                             daemon=True)

    hilo2 = threading.Thread(name='escribir',
                             target=escribir,
                             args=(nombre,))
    hilo1.start()
    hilo2.start()
```

Para hacer que el hilo principal espere a que el hilo demonio complete su trabajo, utilizaremos el método `join()` con dicho hilo. El método `is_alive()` también es útil para conocer si un hilo está o no activo.

```
    import time, os, threading

    def chequear(nombre):
        '''Chequea tamaño de archivo'''
        contador = 0
        tam = 0
        while contador < 100:
            contador += 1
```

```
            if os.path.exists(nombre):
                estado = os.stat(nombre)
                tam = estado.st_size

            print(threading.current_thread().getName(),
                  contador,
                  tam,
                  'bytes')

            time.sleep(0.1)

    def escribir(nombre):
        '''Escribe en archivo'''
        contador = 1
        while contador <= 10:
            with open(nombre, 'a') as archivo:
                archivo.write('1')
                print(threading.current_thread().getName(),
                      contador)
                time.sleep(0.3)
                contador += 1

    nombre = 'archivo.txt'
    if os.path.exists(nombre):
        os.remove(nombre)

    hilo1 = threading.Thread(name='chequear',
                             target=chequear,
                             args=(nombre,),
                             daemon=True)

    hilo2 = threading.Thread(name='escribir',
                             target=escribir,
                             args=(nombre,))
    hilo1.start()
    hilo2.start()

    hilo1.join()
    print(hilo1.is_alive())
```

Controlar la ejecución de varios demonios

Cuando un programa utiliza un gran número de demonios y se quiere que el hilo principal espere a que todos los demonios terminen su ejecución, utilizaremos el método `join()` con cada demonio. Para hacer el seguimiento de los hilos activos se puede emplear `enumerate()` pero teniendo en cuenta que dentro de la lista que devuelve se incluye el hilo principal. Con este hilo hay que tener cuidado porque no acepta ciertas operaciones: no se puede obtener su nombre con `getName()` o utilizar el método `join()`.

En el ejemplo que mostramos a continuación, se utiliza la función `threading.main_thread()` para identificar al hilo principal. Después, se recorren todos los hilos activos para ejecutar `join()`, excluyendo al principal.

```
import threading

def contar(numero):
    contador = 0
    while contador < 10:
        contador += 1
        print(numero, threading.get_ident(), contador)

for numero in range(1, 11):
    hilo = threading.Thread(target=contar, args=(numero,),
                            daemon=True)
    hilo.start()

# Obtiene hilo principal

hilo_ppal = threading.main_thread()

# Recorre hilos activos para controlar estado de su ejecución

for hilo in threading.enumerate():

# Si el hilo es hilo_ppal continua al siguiente hilo activo

    if hilo is hilo_ppal:
        continue

    # Se obtiene información hilo actual y núm. hilos activos

    print("Nombre: ", hilo.getName(),
          "Id: ", hilo.ident,
          "Demonio: ", hilo.isDaemon(),
          "Activo: ",threading.active_count())

    # El programa esperará a que este hilo finalice:

    hilo.join()
```

Multiprocesamiento

En la actualidad la mayoría de los lenguajes de programación soportan la programación concurrente y la programación en paralelo, pudiendo así ejecutar diferentes tareas en diferentes procesadores, sin embargo, con Python esto no es así. Con Python al utilizar `Threads` nunca seremos capaces de lograr un verdadero paralelismo, ya que el lenguaje está diseñado para que un `thread` y solo un `thread` pueda ejecutarse a la vez.

El GIL, Global Interepter Lock de Python, permite que sólo un `thread` tome el control del intérprete, es decir, que solo un `thread` puede estar en ejecución a la vez.

Los hilos son una buena alternativa para lograr concurrencia en tareas de entrada y salida (por ejemplo, lectura y escritura de archivos en disco), caso único en donde la simultaneidad es real. Sin embargo, pueden ser útiles también para evitar bloqueos simples (por más que no haya simultaneidad real), por ejemplo, para separar operaciones pesadas del bucle principal de una aplicación gráfica.

`multiprocessing` es un paquete que permite crear nuevos procesos utilizando un API similar a la del módulo `threading`. Debido a que utiliza subprocesos en lugar de hilos (`threads`), permite llevar a cabo varias operaciones concurrentes sin las limitaciones del Global Interpreter Lock (GIL). Corre en sistemas Unix y Windows.

Veamos un ejemplo:

```
from multiprocessing import Process

def say_hello(name):
    print("Hello, %s!" % name)

if __name__ == '__main__':
    p = Process(target=say_hello, args=("world",))
    p.start()
    p.join()
```

Observamos que la creación de un subproceso es similar a la creación de un hilo, en vez de utiliza `Thread`, utilizamos `Process`. A diferencia de los hilos, es muy importante que los subprocesos se ejecuten subordinados al bloque creado por:

```
if __name__ == '__main__'.
```

Veamos ahora como ejecutar varios procesos:

```
import os
from multiprocessing import Process

def info(title):
    print(title)
```

```
    print('Nombre del módulo:', __name__)
    print('Proceso padre:', os.getppid())
    print('ID del proceso', os.getpid())

def tarea(tarea_id):
    print('Tarea ', tarea_id)
    # time.sleep(tarea_id)


if __name__ == '__main__':
    info('Iniciamos los procesos')
    subproceso1 = Process(target=tarea, args=(1,))
    subproceso2 = Process(target=tarea, args=(2,))
    subproceso3 = Process(target=tarea, args=(3,))
    subproceso1.start()
    subproceso2.start()
    subproceso3.start()
    subproceso1.join()
    subproceso2.join()
    subproceso3.join()
    print('Los subprocesos han finalizado...')
```

Pool

Ofrece un medio conveniente para paralelizar la ejecución de una función a través de múltiples valores de entrada, distribuyendo los datos de entrada entre los procesos (paralelismo de datos). El siguiente ejemplo demuestra la práctica común de definir tales funciones en un módulo para que los procesos secundarios puedan importar ese módulo con éxito.

Veamos un ejemplo, donde obtenemos el contenido de diferentes urls:

```
import os
import requests
from multiprocessing.pool import ThreadPool

PATH_FILE = os.getcwd()


def download_url(url):
    print("downloading: ", url)

    PATH_DATA = PATH_FILE + "/data"
    os.chdir(PATH_DATA )

    file_name_start_pos = url.rfind("/") + 1
    file_name = url[file_name_start_pos:]

    r = requests.get(url, stream=True)
```

```
        if r.status_code == requests.codes.ok:
            with open(file_name, 'wb') as f:
                for data in r:
                    f.write(data)
            return url


    urls = ["https://jsonplaceholder.typicode.com/posts",
            "https://jsonplaceholder.typicode.com/comments",
            "https://jsonplaceholder.typicode.com/photos",
            "https://jsonplaceholder.typicode.com/todos",
            "https://jsonplaceholder.typicode.com/albums",
            ]

    # Ejecuta n procesos. Cada llamada toma un elemento de urls

    results = ThreadPool(6).imap_unordered(download_url, urls)
    for r in results:
        print(r)
```

Intercambiando objetos entre procesos

`multiprocessing` admite dos tipos de canales de comunicación entre procesos: Colas (queues) y tuberías (pipe).

Colas (queues): La clase Queue es prácticamente un clon de queue.Queue. Queue devuelve una cola de proceso compartida implementada mediante una tubería y algunos bloqueos o semáforos. Cuando un proceso coloca por primera vez un elemento en la cola, se inicia un hilo alimentador que transfiere objetos desde un búfer a la tubería.

Por ejemplo:

```
    from multiprocessing import Process, Queue

    def f(q):
        urls = ["https://jsonplaceholder.typicode.com/posts",
                "https://jsonplaceholder.typicode.com/comments",
                "https://jsonplaceholder.typicode.com/photos",
                "https://jsonplaceholder.typicode.com/todos",
                "https://jsonplaceholder.typicode.com/albums",
                ]

        q.put(urls)

    if __name__ == '__main__':
        q = Queue()
        p = Process(target=f, args=(q,))
        p.start()
        print(q.get())
```

```
    p.join()
```

Las colas (`queues`) son hilos y procesos seguro.

Tuberías (Pipes): La función `Pipe()` retorna un par de objetos de conexión (`conn1, conn2`) que representan los extremos de una tubería (`pipe`), por defecto, es un dúplex (`bidireccional`).

- Si `duplex` es `True` (el valor predeterminado), entonces la tubería (`pipe`) es bidireccional.
- Si `duplex` es `False`, entonces la tubería es unidireccional: `conn1` solo se puede usar para recibir mensajes y `conn2` solo se puede usar para enviar mensajes.

Por ejemplo:

```
from multiprocessing import Process, Pipe

def f(conn):
    urls = ["https://jsonplaceholder.typicode.com/posts",
            "https://jsonplaceholder.typicode.com/comments",
            "https://jsonplaceholder.typicode.com/photos",
            "https://jsonplaceholder.typicode.com/todos",
            "https://jsonplaceholder.typicode.com/albums",
           ]
    conn.send(urls)
    conn.close()

if __name__ == '__main__':
    parent_conn, child_conn = Pipe()
    p = Process(target=f, args=(child_conn,))
    p.start()
    print(parent_conn.recv())
    p.join()
```

Los dos objetos de conexión retornados por `Pipe()` representan los dos extremos de la tubería (`pipe`). Cada objeto de conexión tiene los métodos `send()` y `recv()` (entre otros). Lo que enviemos a través de uno de estos objetos (mediante su método send) se recibirá en el otro (mediante su método recv) y viceversa. El método `recv()` bloquea el proceso actual hasta que se recibe algo enviado por el método `send()`. Por eso, el uso una tubería implica que debemos establecer un protocolo de comunicación entre los dos extremos, para que cada `recv()` se corresponda con un `send()` en el otro lado.

Tenga en cuenta que los datos en una tubería pueden corromperse si dos procesos (o hilos) intentan leer o escribir en el mismo extremo de la tubería al mismo tiempo. Por supuesto, no hay riesgo de corrupción por procesos que utilizan diferentes extremos de la tubería (`pipe`) al mismo tiempo.

Cuando utilizar Pipe() o Queue():

- Pipe() sólo puede tener dos puntos finales
- Queue() puede tener multiples productores y consumidores.

En conclusión, si necesitamos más de dos puntos de comunicación, utilizamos Queue y si necesitamos rendimiento, utilizamos pipe() que es mas rápido que queue.

Sincronización entre procesos

`multiprocessing` contiene equivalentes de todas las sincronizaciones primitivas de `threading`. Por ejemplo, se puede usar un candado (`lock`) para garantizar que solo un proceso imprima la salida estándar a la vez:

```
from multiprocessing import Process, Lock

def f(l, i):
    l.acquire()
    try:
        print('hello world', i)
    finally:
        l.release()

if __name__ == '__main__':
    lock = Lock()

    for num in range(10):
        Process(target=f, args=(lock, num)).start()
```

Sin usar el candado (`lock`) de salida de los diferentes procesos, es probable que todo se mezcle.

Compartiendo estado entre procesos

Cuando se realiza una programación concurrente, generalmente es mejor evitar el uso del estado compartido en la medida de lo posible. Esto es particularmente cierto cuando se utilizan múltiples procesos.

Sin embargo, si necesitamos usar algunos datos compartidos `multiprocessing` proporciona un par de maneras de hacerlo.

Memoria compartida

Los datos se pueden almacenar en un mapa de memoria compartida usando `Value` o `Array`. Por ejemplo, el siguiente código:

```
from multiprocessing import Process, Value, Array

def f(n, a):
    n.value = 3.1415927
    for i in range(len(a)):
```

```
            a[i] = -a[i]

    if __name__ == '__main__':
        num = Value('d', 0.0)
        arr = Array('i', range(10))

        p = Process(target=f, args=(num, arr))
        p.start()
        p.join()

        print(num.value)
        print(arr[:])
    3.1415927
    [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

Los argumentos `d` e `i` utilizados cuando creamos `num` y `arr` son `typecodes` del tipo utilizado por el módulo array: `d` indica un de tipo `float` e `i` indica un entero con signo. Estos objetos compartidos serán seguros para procesos y subprocesos.

Para una mayor flexibilidad en el uso de la memoria compartida, se puede usar el módulo `multiprocessing.sharedctypes` que admite la creación arbitraria de objetos `ctypes` asignados desde la memoria compartida.

Proceso servidor (*Server process*)

Un objeto de administrador retornado por `Manager()` controla un proceso de servidor que contiene objetos de Python y permite que otros procesos los manipulen usando `proxies`.

Un administrador retornado por `Manager()` soportará tipos de clases como list, dict, Namespace, Lock, RLock, Semaphore, BoundedSemaphore, Condition, Event, Barrier, Queue, Value y Array.

Por ejemplo:

```
    from multiprocessing import Process, Manager

    def f(d, l):
        d[1] = '1'
        d['2'] = 2
        d[0.25] = None
        l.reverse()

    if __name__ == '__main__':
        with Manager() as manager:
            d = manager.dict()
            l = manager.list(range(10))

            p = Process(target=f, args=(d, l))
            p.start()
```

```
    p.join()

    print(d)
    print(l)
```

asyncio — E/S Asíncrona

- `asyncio` es una biblioteca para escribir código concurrente utilizando la sintaxis: `async/await`.
- `asyncio` es utilizado como base en múltiples frameworks asíncronos de Python y provee un alto rendimiento en redes y servidores web, bibliotecas de conexión de base de datos, colas de tareas distribuidas, etc.
- `asyncio` suele encajar perfectamente para operaciones con límite de E/S y código de red estructurado de alto nivel.

`asyncio` provee un conjunto de APIs de alto nivel para:

- ejecutar corutinas de Python de manera concurrente y tener control total sobre su ejecución;
- realizar redes E/S y comunicación entre procesos (IPC);
- controlar subprocesos;
- distribuir tareas a través de colas;
- sincronizar código concurrente;

Adicionalmente, existen APIs de bajo nivel para desarrolladores de bibliotecas y frameworks para:

- crear y administrar bucles de eventos, los cuales proveen APIs asíncronas para redes, ejecutando subprocesos, gestionando señales del sistema operativo, etc;
- implementar protocolos eficientes utilizando transportes;
- Bibliotecas puente basadas en retrollamadas y código con sintaxis async/wait.

Corutinas

`Coroutines` se declara con la sintaxis `async/await` es la forma preferida de escribir aplicaciones `asyncio`.

Para ejecutar una corrutina, `asyncio` proporciona tres mecanismos principales:

- La función `asyncio.run()` para ejecutar la función de punto de entrada de nivel superior `main()`.
- Esperando en una corrutina.
- La función `asyncio.create_task ()` para ejecutar corutinas simultáneamente como tareas `asyncio`.

asyncio.run(): En el siguiente código se imprime `Hola`, espera 1 segundo y luego imprime `Mundo`:

```
import asyncio

async def main():
    print('Hello ...')
    await asyncio.sleep(1)
    print('... World!')
```

Si utilizamos Notebook Jupyter, la ejecutaremos con:

```
await main()
Hello ...
... World!
```

En otro IDLE, sustituya `await main()` por:

```
asyncio.run(main())
```

Esto sucede, porque `asyncio` no se puede llamar cuando otro bucle de eventos `asyncio` se está ejecutando en el mismo hilo. El caso es que el Notebook Jupyter (IPython) ya está ejecutando un bucle de eventos.

Es importante destacar, que no podemos llamar a una corrutina directamente, porque no la ejecutará:

```
main()
<coroutine object main at 0x0000016F7373B140>
```

Esperando en una corrutina:

El siguiente fragmento de código imprimirá `"hola"` después de esperar 1 segundo, y luego imprimirá `"mundo"` después de esperar otros 2 segundos:

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    print(f"started at {time.strftime('%X')}")

    await say_after(1, 'hello')
    await say_after(2, 'world')

    print(f"finished at {time.strftime('%X')}")
```

```
# asyncio.run(main())
await main()
```

asyncio.create_task(): En el siguiente ejemplo, ejecutaremos dos corrutinas `say_after` al mismo tiempo:

```
import asyncio
import time

async def say_after(delay, what):
    await asyncio.sleep(delay)
    print(what)

async def main():
    task1 = asyncio.create_task(say_after(1, 'hello'))
    task2 = asyncio.create_task(say_after(2, 'world'))
    print(f"started at {time.strftime('%X')}")

    # Wait until both tasks are completed (should take
    # around 2 seconds.)
    await task1
    await task2

    print(f"finished at {time.strftime('%X')}")

#asyncio.run(main())

await main()
```

```
started at 19:35:54
hello
world
finished at 19:35:56
```

Awaitables

Decimos que un objeto es un objeto `awaitable` o en espera si se puede usar en una expresión `await`. Muchas API de `asyncio` están diseñadas para aceptar esperables.

Hay tres tipos principales de objetos en espera: coroutines, Tasks y futures.

Coroutines: Las corrutinas de Python son `awaitables` y, por lo tanto, pueden esperar a otras corrutinas.

En esta documentación, el término `corrutina` se puede utilizar de dos formas estrechamente relacionados:

- una función de corrutina: una función `async def`;
- un objeto de corrutina: un objeto devuelto llamando a una función de corrutina.

Tasks: Las tareas se utilizan para programar corrutinas concurrentes. Cuando una corrutina se incluye en una tarea con funciones como `asyncio.create_task()`, la corrutina se programa automáticamente para ejecutarse pronto.

Futures: es un objeto especial `awaitable` de bajo nivel que representa un resultado de una operación asincrónica.

Cuando se espera un objeto `Futures`, significa que la corrutina esperará hasta que el `Futures` se resuelva en algún otro lugar. Se necesitan objetos `futures` en `asyncio` para permitir que el código basado en `callback` se use con `async/await`.

Normalmente no es necesario crear objetos Future en el código de nivel aplicación. Se pueden esperar objetos futures, a veces expuestos por bibliotecas y algunas API `asyncio`:

```
async def main():
    await function_that_returns_a_future_object()

    # this is also valid:
    await asyncio.gather(
        function_that_returns_a_future_object(),
        some_python_coroutine()
    )
```

Un buen ejemplo de una función de bajo nivel que devuelve un objeto Future es `loop.run_in_executor()`.

Funciones

Sleeping: La función `asyncio.sleep`, retorna una corutina que bloquea el proceso actual los segundos indicados en el parámetro `delay`, Si se proporciona un valor en `result`, es devuelto por la función cuando se completa la espera.

Sintaxis:

```
coroutine asyncio.sleep(delay, result=None, *, loop=None)
```

Ejemplo:

```
import asyncio
import datetime

async def display_date():
    loop = asyncio.get_running_loop()
    end_time = loop.time() + 5.0
    while True:
```

```
            print(datetime.datetime.now())
            if (loop.time() + 1.0) >= end_time:
                break
            await asyncio.sleep(1)

    asyncio.run(display_date())
```

gather(): La función `gather()` ejecuta objetos awaitables, en la secuencia proporcionada en aws, de forma concurrente, y espera a que finalicen. Si el objeto awaitable es una co-rutina se programa automáticamente como un Task.

Sintaxis:

```
    asyncio.gather(*aws, loop=None, return_exceptions=False)
```

Una vez que todos los `awaitables` se acaban de ejecutar, la función retorna una lista con los valores retornados por cada `awaitable`, en el mismo orden en el cual son pasados en `aws`.

Si `return_exceptions` es `False`, la primera excepción que se lance se propaga inmediatamente a las tareas que estén esperando, pero los demás `awaitables` se continuarán ejecutando. Si es `True`, la excepción es tratada como si fuese el resultado del `awaitable`.

```
    import asyncio

    async def factorial(name, number):
        f = 1
        for i in range(2, number + 1):
            print(f"Task {name}: Compute factorial({i})...")
            await asyncio.sleep(1)
            f *= i
        print(f"Task {name}: factorial({number}) = {f}")

    async def main():
        # Schedule three calls *concurrently*:
        await asyncio.gather(
            factorial("A", 2),
            factorial("B", 3),
            factorial("C", 4),    )

    #asyncio.run(main())
    await main()

    Task A: Compute factorial(2)...
    Task B: Compute factorial(2)...
    Task C: Compute factorial(2)...
    Task A: factorial(2) = 2
    Task B: Compute factorial(3)...
```

```
Task C: Compute factorial(3)...
Task B: factorial(3) = 6
Task C: Compute factorial(4)...
Task C: factorial(4) = 24
```

wait_for(): ejecuta un `awaitable` y espera a que se complete o lanza una excepción de tipo `asyncio.TimeoutError` si la espera sobrepasa los segundos especificados en `timeout`.

Sintaxis:

```
asyncio.wait_for(aw, timeout, *, loop=None)
```

Ejemplo:

```
async def eternity():
    # Sleep for one hour
    await asyncio.sleep(3600)
    print('yay!')

async def main():
    # Wait for at most 1 second
    try:
        await asyncio.wait_for(eternity(), timeout=1.0)
    except asyncio.TimeoutError:
        print('timeout!')

#asyncio.run(main())
await main()
```

La función `asyncio.wait()` ejecuta un conjunto de objetos `awaitable` concurrentemente y espera hasta que se cumple la condición especificada en `return_when`.

Sintaxis:

```
asyncio.wait(aws, *, loop=None, timeout=None,
                                return_when=ALL_COMPLETED)
```

Posibles valores del parámetro `return_when`:

- `asyncio.FIRST_COMPLETED`, la función retorna cuando se complete o se cancele alguno de los `awaitables`.
- `asyncio.FIRST_EXCEPTION`, la función retorna cuando alguno de los `awaitables` lance una excepción, o todos finalicen normalmente.
- `asyncio.ALL_COMPLETED`, la función retorna cuando todos los `awaitables` se completan o se cancelan todos.

Al contrario que `wait_for()`, `wait()` no cancela las tareas cuando se rebasa el tiempo límite especificado en `timeout`. Esta función no acepta corutinas como argumento, por tanto, deben ser encapsuladas en un `Task`.

Cuando la función retorna, devuelve dos conjuntos de tareas: (finalizadas, pendientes)

Veamos un ejemplo:

```
async def get(n):
    await asyncio.sleep(n)
    return n

async def main():
    tareas = {
              asyncio.create_task(get(2)),
              asyncio.create_task(get(1)),
              asyncio.create_task(get(3))
              }
    done, pending = await asyncio.wait(tareas,
                          return_when =
                     asyncio.FIRST_COMPLETED)
    print(f'Tareas finalizadas: {done}')
    print(f'Tareas pendientes: {pending}')

asyncio.run(main())
```

```
Tareas finalizadas: {<Task finished name='Task-20' coro=<get() done, defined
at <ipython-input-11-4c6ad3698642>:1> result=1>}

Tareas pendientes: {<Task pending name='Task-21' coro=<get() running at
<ipython-input-11-4c6ad3698642>:2>          wait_for=<Future          pending
cb=[<TaskWakeupMethWrapper object at 0x000001C1F2DC84F0>()]>>, <Task pending
name='Task-19' coro=<get() running at <ipython-input-11-4c6ad3698642>:2>
wait_for=<Future     pending     cb=[<TaskWakeupMethWrapper     object     at
0x000001C1F2CD4850>()]>>}
```

Temporizadores.

Hemos visto que los hilos se ejecutan después de invocar el método `start()` y están vivos mientras se está ejecutando su código. Habrá ocasiones donde nos interesa comenzar la ejecución de un hilo en un momento dado del tiempo, o poder repetir su ejecución periódicamente.

Programación de un hilo en el tiempo.

Un temporizador (`Timer`) es un tipo de hilo especial que permite programar su ejecución en el tiempo. La clase `threading.Timer` hereda de `threading.Thread`, y añade un método `cancel()` que permite cancelar su programación. En el constructor de `Timer` también podemos pasar argumentos.

Veamos primero un ejemplo de un temporizador que programa un hilo para que muestre la hora actual dentro de 5 segundos:

```
import threading, time

def mostrarHora():
    print("Se ejecuta en:", time.strftime('%H:%M:%S'))

hilo = threading.Timer(5, mostrarHora);
print("Lanzamos el hilo en:", time.strftime('%H:%M:%S'))
hilo.start()
Lanzamos el hilo en: 11:35:02
Se ejecuta en: 11:35:07
```

El constructor de `Timer` recibe como primer argumento el tiempo de espera, en segundos, para ejecutar el hilo después de invocar `start()`. Se pueden usar fracciones de segundo para este tiempo de espera.

Podemos pasar argumentos a un `Timer` al igual que a un `Thread`:

```
threading.Timer(espera, función, args=None, kwargs=None)
```

En el siguiente ejemplo se pasa un mensaje al hilo:

```
funcion = lambda msg: print(msg)
hilo = threading.Timer(3,funcion, args=("Hola mundo!",) )
hilo.start()
```

Pasados tres segundos se imprimirá el mensaje.

Programación repetitiva de un hilo.

El módulo `threading` no ofrece ninguna clase para programar hilos de forma repetitiva, pero podemos diseñar una clase como la siguiente:

```
import threading, time

class RepeatingTimer():

    def __init__(self, espera, intervalo, funcion):
        self.intervalo = intervalo
        self.funcion = funcion
        self.cancelado = False
        self.thread = threading.Timer(espera,
                                      self.__handlerFunction__)

    def __handlerFunction__(self):
        self.funcion();
        if not self.cancelado:
            self.thread = threading.Timer(self.intervalo,
```

```
                                        self.__handlerFunction__)
                self.thread.start()

        def start(self):
            self.thread.start()
            self.thread.cancel()
```

Ahora crearemos una tarea repetitiva que vaya incrementando un contador hasta llegar a 10:

```
contador = 0

def tarea():
    global contador
    contador = contador + 1
    if contador == 10:
        hilo.cancel()
    else:
        print(contador)

hilo = RepeatingTimer(0, 1, tarea)
hilo.start()
```

Referencias

https://recursospython.com/guias-y-manuales/multiprocessing-tareas-concurrentes-con-procesos/