Python Asyncio Coroutines with examples

Presentation · December 2016

DOI: 10.13140/RG.2.2.17439.36003

1 author:

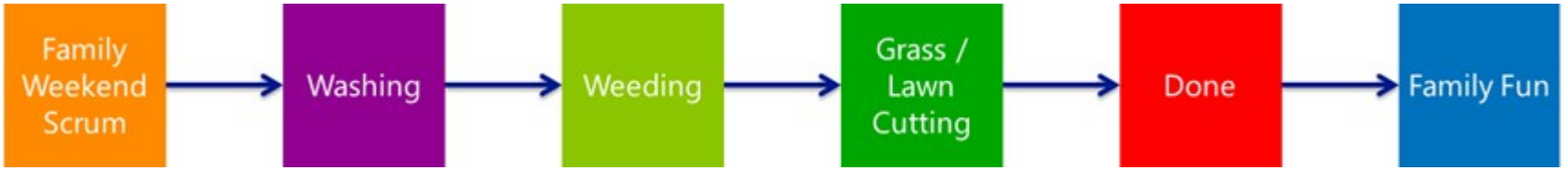
David Pineda
University of Chile
12 PUBLICATIONS **177** CITATIONS

SEE PROFILE

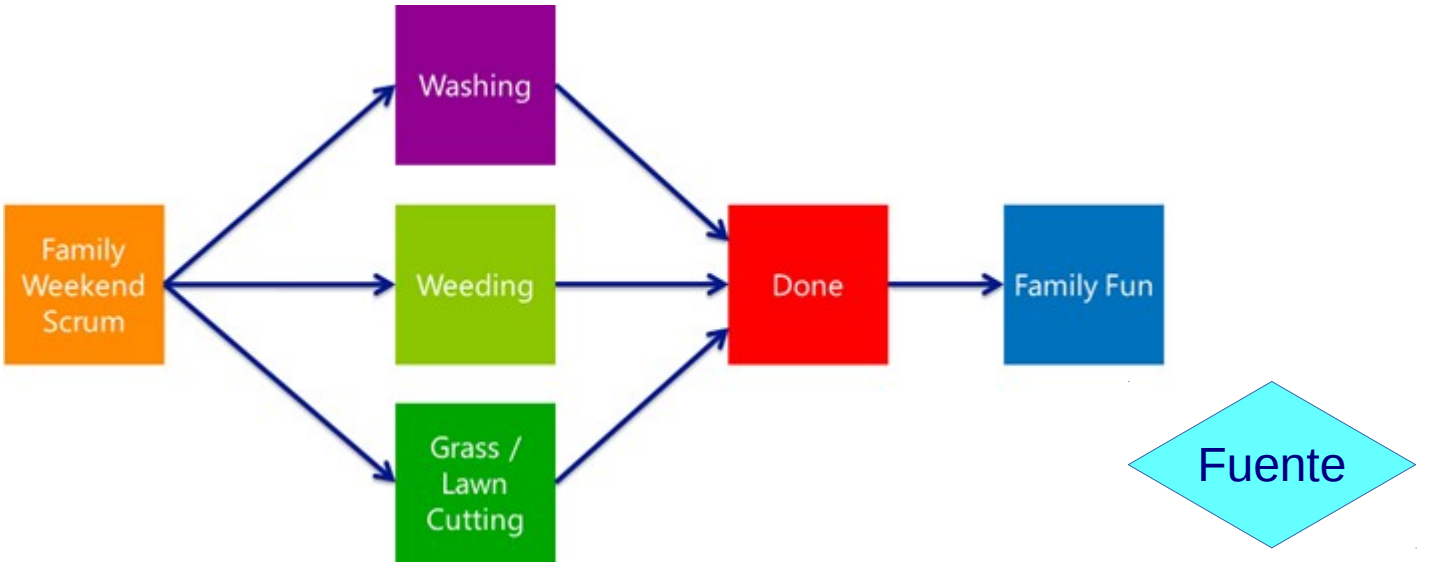
Corrutinas Asíncronas en Python (>3.4)

David Pineda Osorio
dahalpi@gmail.com
@pineiden
Diciembre 2016

No todo es secuencial...

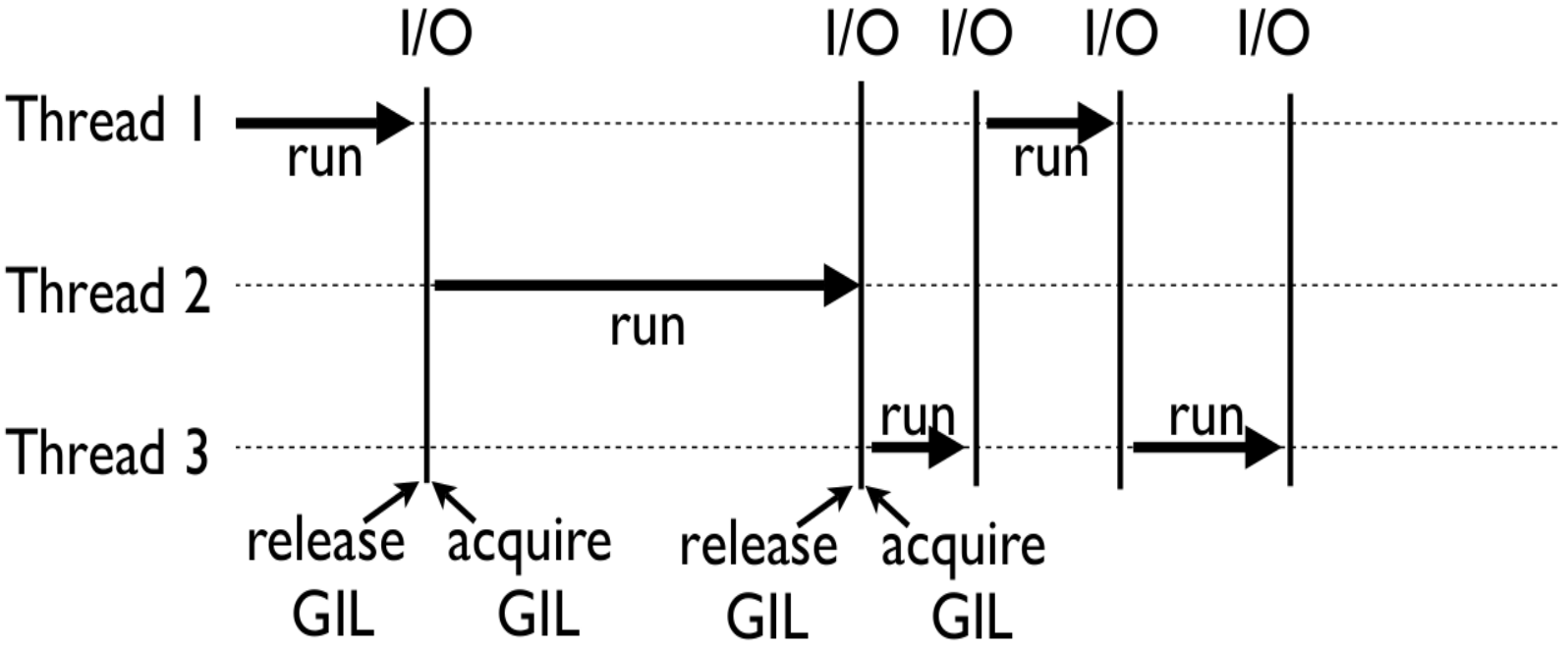


- Hay cosas que ocurren de manera aleatoria e independiente
- Es posible ocupar la CPU de manera intensiva



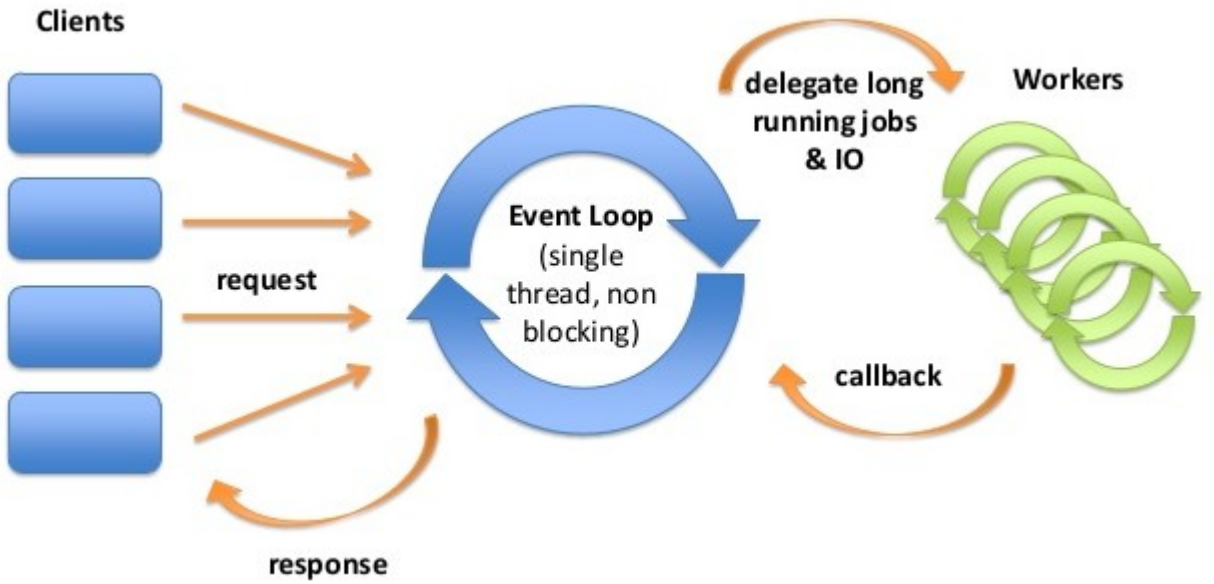
El tradicional GIL de Python

- Es un administrador general que gestiona la secuencialidad de una ejecución de python



El EventLoop de Asyncio

- Da un paso más allá, tomando las riendas de la ejecución de las corrutinas, permitiendo realizar multitareas mediante la gestión de envío y recepción de mensajes de eventos.



Fuente

Definición de asyncio en PEP

- PEP 492 https://www.python.org/dev/peps/pep-0492/
- PEP 3156 https://www.python.org/dev/peps/pep-3156/
- PEP 3153 https://www.python.org/dev/peps/pep-3153/
- PEP 525 https://www.python.org/dev/peps/pep-0525/
- PEP 492 https://www.python.org/dev/peps/pep-0492/
- PEP 530 https://www.python.org/dev/peps/pep-0530/

Definen la síntaxis y principales funcionalidades que se habilitan con asyncio.

Principales Características

- Un eventloop con diversas implementaciones para especificaciones de sistema
- Abstracciones para una capa de Transporte y una de Protocolo
- Soporte networking TCP, UDP, UnixSocket, SSL, subprocesos, pipes, etc
- Síntaxis basada en “yield from” o “await”
- Cancelación de Futures o manager de corrutinas
- Interface para trabajo con threads y procesos

Elementos Básicos a Considerar

- EventLoop: el gestor de ejecución
- Corrutinas: la función con síntaxis especial
- Futures, Tasks: el gestor de ejecución de cada corrutina
- Conexiones: Objetos de red como sockets de distintos tipos
- Streams: elementos de comunicación
- Subprocesos: herramientas para uso de multiprocesos
- Colas (Queues): Objetos para enviar elementos entre corrutinas

En python 3.4

La llamada sería:

```
import asyncio
@asyncio
def coro_fun(input):
    yield from accion()

loop=asyncio.get_event_loop()
loop.run_until_complete(coro_fun(1))
```

En python >3.5

La llamada sería:

```
import asyncio


async def coro_fun(input):
    r=await f(input)
    return r
loop=asyncio.get_event_loop()
loop.run_until_complete(coro_fun(1))
```

Ejemplo 1

- Lo más simple, una corrutina ejecutada en el event-loop:

```
import asyncio

async def holacoro():
	for i in range(3):
		await asyncio.sleep(1)
		print("Hola %d" % i)

if __name__ == "__main__":
	loop = asyncio.get_event_loop()
	#creamos tarea y la asociamos al loop, ejecutandola
	loop.run_until_complete(holacoro())
```

Ejemplo 2

- Corrutinas ejecutadas secuencialmente

```
import asyncio

async def holacoro():
	for i in range(3):
		await asyncio.sleep(1)
		print("Hola %d" % i)

async def chaocoro():
	for i in range(3):
		await asyncio.sleep(2)
		print("Chao %d" % i)

async def doscoro():
	await holacoro()
	await chaocoro()



if __name__ == "__main__":
	loop = asyncio.get_event_loop()
	#creamos tarea y la asociamos al loop, ejecutandola
	loop.run_until_complete(doscoro())
```

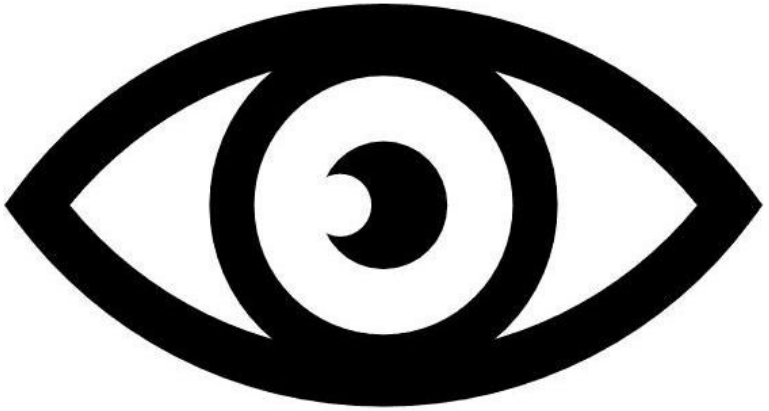
Ejemplo 3

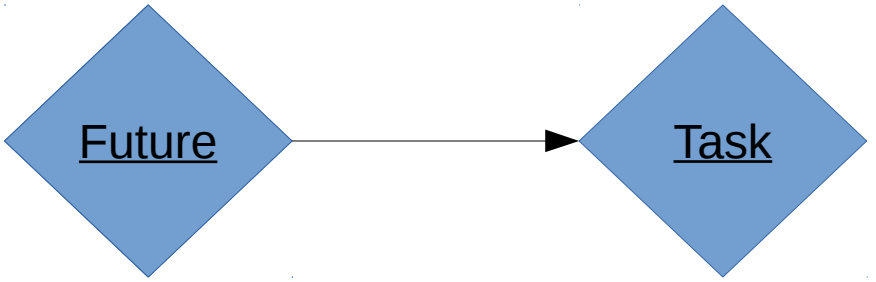
- Corrutinas ejecutadas independientes

```
import asyncio

async def holacoro():
	for i in range(3):
		await asyncio.sleep(1)
		print("Hola %d" % i)

async def chaocoro():
	for i in range(3):
		await asyncio.sleep(2)
		print("Chao %d" % i)

if __name__ == "__main__":
	tasks=[
	asyncio.ensure_future(holacoro()),
	asyncio.ensure_future(chaocoro())
	]
	loop = asyncio.get_event_loop()
	#creamos tarea y la asociamos al loop, ejecutandola
	loop.run_until_complete(
		asyncio.gather(*tasks)
	)
```

Vamos con calma, ¿Qué estamos haciendo?: Un algoritmo

1) Crear corrutinas
2) Agendar la ejecución de corrutinas mediante Tasks
3) Iniciar un eventloop
4) Asociar Tasks a EventLoop



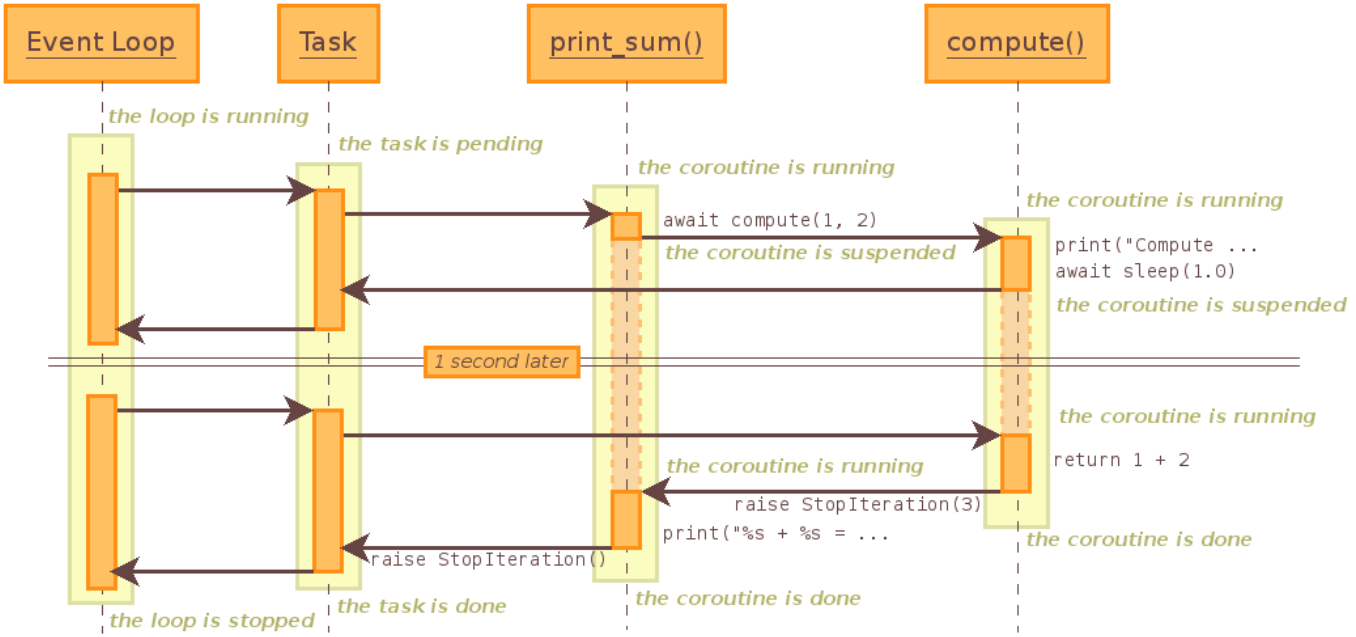


Veamos la documentación de Asyncio

- MUY RECOMENDABLE ESTUDIARLA:

  https://docs.python.org/3/library/asyncio.html

- Future es una clase que provee una serie de métodos que permiten asociar la ejecución de una corrutina a un eventloop.
- Task es una clase que hereda de Future y añade algunas funcionalidades especiales para la gestión de corrutinas.



Ejemplo 4

```
import asyncio

async def holacoro():
    print("Hola %d" % 1)
    await asyncio.sleep(1)

def renew(*args):
    task=loop.create_task(holacoro())
    task.add_done_callback(renew)


task=loop.create_task(holacoro())
task.add_done_callback(renew)

loop=asyncio.get_event_loop()

try:
     loop.run_forever()
except KeyboardInterrupt:
     print('Loop stopped')
```

Ejemplo 5

```
import asyncio

async def holacoro():
    print("Hola %d" % 1)
    await asyncio.sleep(1)

def renew(*args):
    task=loop.create_task(holacoro())
    task.add_done_callback(renew)


task=loop.create_task(holacoro())
task.add_done_callback(renew)

loop=asyncio.get_event_loop()

try:
     loop.run_forever()
except KeyboardInterrupt:
     print('Loop stopped')
```

Ejemplo 6: La potencia asyncio+multiprocessing

- Realiza Tasks de manera independiente en procesadores distintos

```
if __name__ == "__main__":
    loop = asyncio.get_event_loop()
    executor=concurrent.futures.ProcessPoolExecutor()
    v=1
    b=1
    gprint("Entrando a loop event")
    future1=loop.run_in_executor(
        executor,
        functools.partial(hola_task,v))
    future2=loop.run_in_executor(
        executor,
        functools.partial(chao_task,b))
    tasks=[future1,future2]
    loop.run_until_complete(asyncio.gather(*tasks))
```

Otros usos

- Muy potente para el desarrollo de sistemas de comunicación en tiempo real.

1) Chats
2) Networking entre sockets
3) Mensajería
4) Clusters

Más info

- Dave Baezly
  http://www.dabeaz.com/python.html
- Bret Cannon

  http://www.snarky.ca/how-the-heck-does-async-await-work-in-python-3-5
- Mi gitlab https://gitlab.com/pineiden
- Modulos adicionales hechos con asyncio

  https://github.com/python/asyncio/wiki/ThirdParty