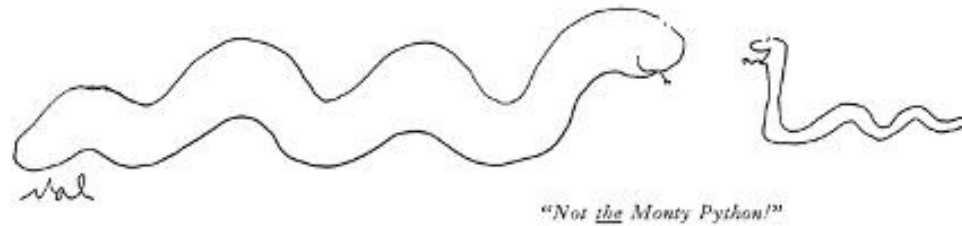
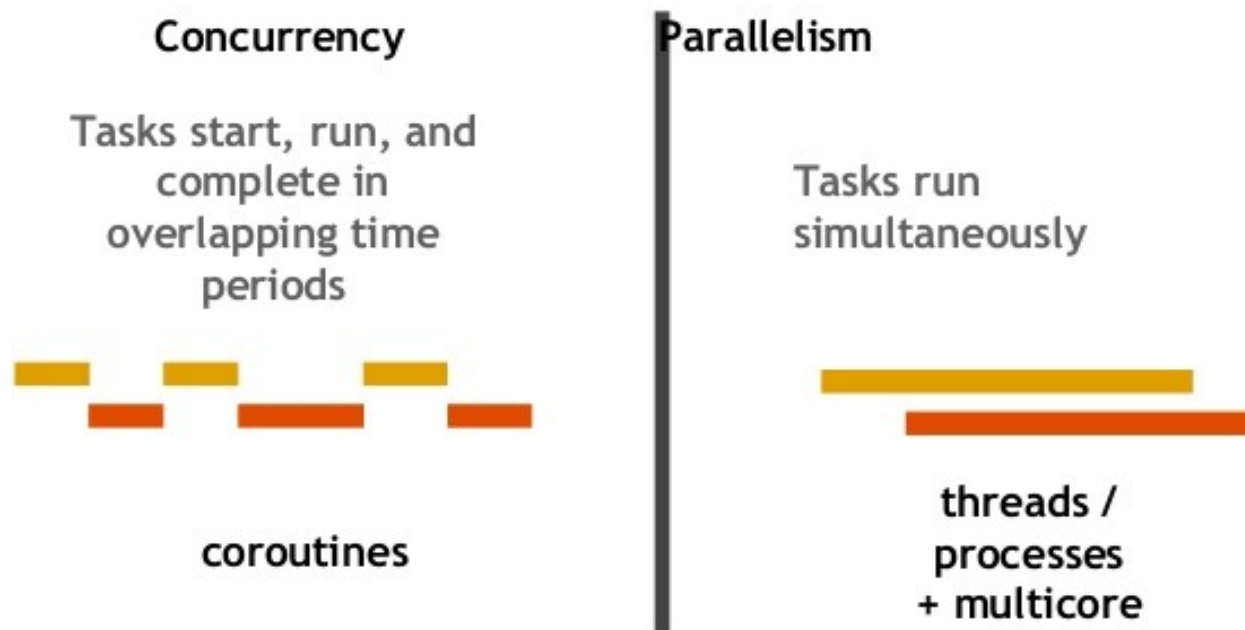


# Python asyncio & Friends



# O que é Async ?

- É um padrão de programação concorrente



# O que é Async ?





# Como funciona o Python?

- O SO gere todo o trabalho de multi-tasking
- Em CPython, a GIL (global interpreter lock) impede concorrência usando múltiplas cores – o interpretador está locked a um core.
- Asyncio:
  - Sem intervenção do SO
  - Um processo, uma thread

# Xadrez – Mestre a jogar contra 50 oponentes



- Mestre leva 30 segundos a fazer uma jogada
- Jogador Normal leva 5 minutos
- Quanto tempo levariam os 50 jogos sincronamente (media de 30 movimentos)?
  - $30 \times 0,5 + 30 \times 5 = 165$  min por jogo
  - $165 \text{ min} \times 50 = 5,7$  dias
- Se considerarmos que o Mestre leva 1 min a mudar de contexto
  - $30 \times 0,5 + 30 \times 1 = 45$  min por ronda
  - $45 \text{ min} \times 30 = 22\text{h}30$




# Como suportar Async?

- Funções Async precisam de poder suspender e resumir.
- Uma função que entre num periodo de espera deve ser suspensa, e só resumir quando a espera tiver terminado
- Como implementar suspensão/resumir em Python:
  - Callback's
  - Geradores
  - Async/Await (Python 3.5+)



# Programação de tarefas assíncronas

- Os frameworks async precisam de um scheduler, normalmente chamado de “event loop”
- O loop toma conta de todas as tarefas em execução
- Quando uma função suspende, devolve o control ao loop, que procura por outra tarefa que lhe suceda
- A isto chama-se “cooperative multi-tasking”



```
import time

def hello():
    print('Hello')
    time.sleep(3)
    print('World')

if __name__ == '__main__':
    for _ in range(3):
        hello()
```

```
1 import asyncio
2 loop = asyncio.get_event_loop()
3
4 async def hello():
5     print('Hello')
6     await asyncio.sleep(3)
7     print('World')
8
9 async def mainloop():
10     t = []
11     for _ in range(3):
12         t.append(hello())
13     await asyncio.gather(*t)
14
15 if __name__ == '__main__':
16     loop.run_until_complete(mainloop())
```

# Sync vs Async

---





# Problemas...

- Tarefas intensivas (CPU) que demorem muito tempo precisam libertar periodicamente o CPU para evitar que as outras tarefas bloqueiem.
  - Solução: Pode-se fazer um `await asyncio.sleep(0)` periodicamente.
- Várias bibliotecas nativas não são compatíveis!
  - `Socket, select, subprocess, os.waitpid, threading, multiprocessing, time.sleep`
  - Existem bibliotecas alternativas que suportam async

Como assim não  
posso usar Socket ?

- Stream based

```
1  import asyncio
2
3  async def echo_server(reader, writer):
4      while True:
5          data = await reader.read(100)
6          if not data:
7              break
8          writer.write(data)
9          await writer.drain() # Flow control
10         writer.close()
11
12
13  async def main(host, port):
14      server = await asyncio.start_server(echo_server, host, port)
15      await server.serve_forever()
16
17  loop = asyncio.get_event_loop()
18  loop.run_until_complete(main('127.0.0.1', 5000))
19  loop.close()
```

Como assim não  
posso usar Socket?

- Callback based

```
1  import asyncio
2
3  class EchoProtocol(asyncio.Protocol):
4      def connection_made(self, transport):
5          self.transport = transport
6      def data_received(self, data):
7          self.transport.write(data)
8
9  async def main(host, port):
10     loop = asyncio.get_running_loop()
11     server = await loop.create_server(EchoProtocol, host, port)
12     await server.serve_forever()
13
14  loop = asyncio.get_event_loop()
15  loop.run_until_complete(main('127.0.0.1', 5000))
16  loop.close()
```



All code:

<http://tiny.cc/oug7vz>

Original:

[https://gist.github.com/dgomes/cd8a7e46967b5c  
fbccc474b748662ad8](https://gist.github.com/dgomes/cd8a7e46967b5cfbccc474b748662ad8)





# one more thing...

Ninguém desenvolve um sistema distribuído do início (a menos que exista uma MUITO BOA razão para isso)

É usual criar-se uma pilha (stack) a partir de várias peças/projectos existentes.

Vamos analisar alguns....



# NGINX

- Nginx is a web server which can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache.



# Flask

- Flask is a micro web framework written in Python. It is classified as a microframework because it does not require particular tools or libraries. It has no database abstraction layer, form validation, or any other components where pre-existing third-party libraries provide common functions.





# RabbitMQ

- RabbitMQ is an open-source message-broker software that originally implemented the Advanced Message Queuing Protocol and has since been extended with a plug-in architecture to support Streaming Text Oriented Messaging Protocol, Message Queuing Telemetry Transport, and other protocols.





# redis

- Redis is an in-memory data structure project implementing a **distributed, in-memory key-value database** with optional durability. Redis supports different kinds of abstract data structures, such as strings, lists, maps, sets, sorted sets, HyperLog Logs, bitmaps, streams, and spatial indexes.





# HAProxy

- HAProxy is free, open source software that provides a high availability load balancer and proxy server for TCP and HTTP-based applications that spreads requests across multiple servers. It is written in C and has a reputation for being fast and efficient.





# É tudo Python !?

- <https://stackshare.io/stacks>