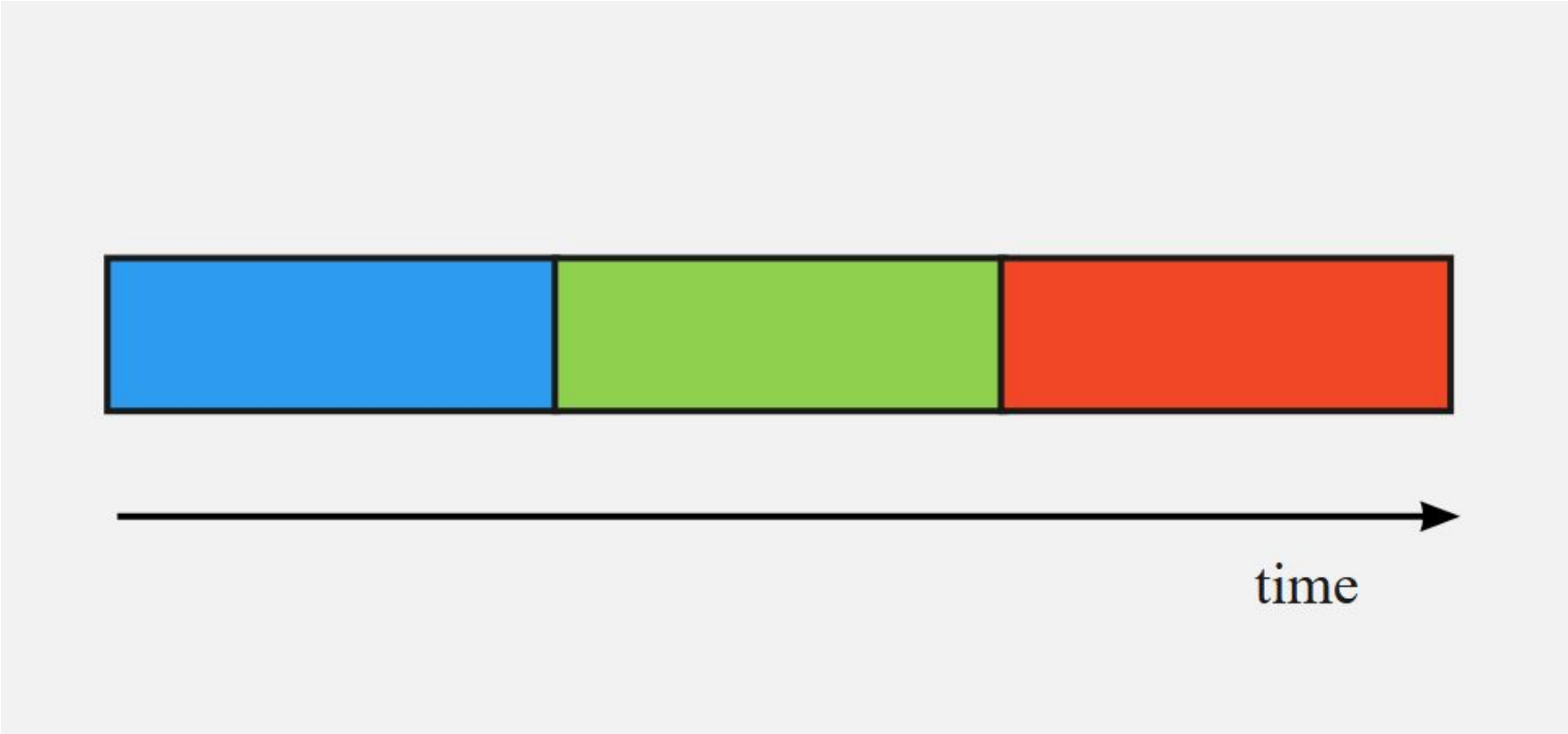
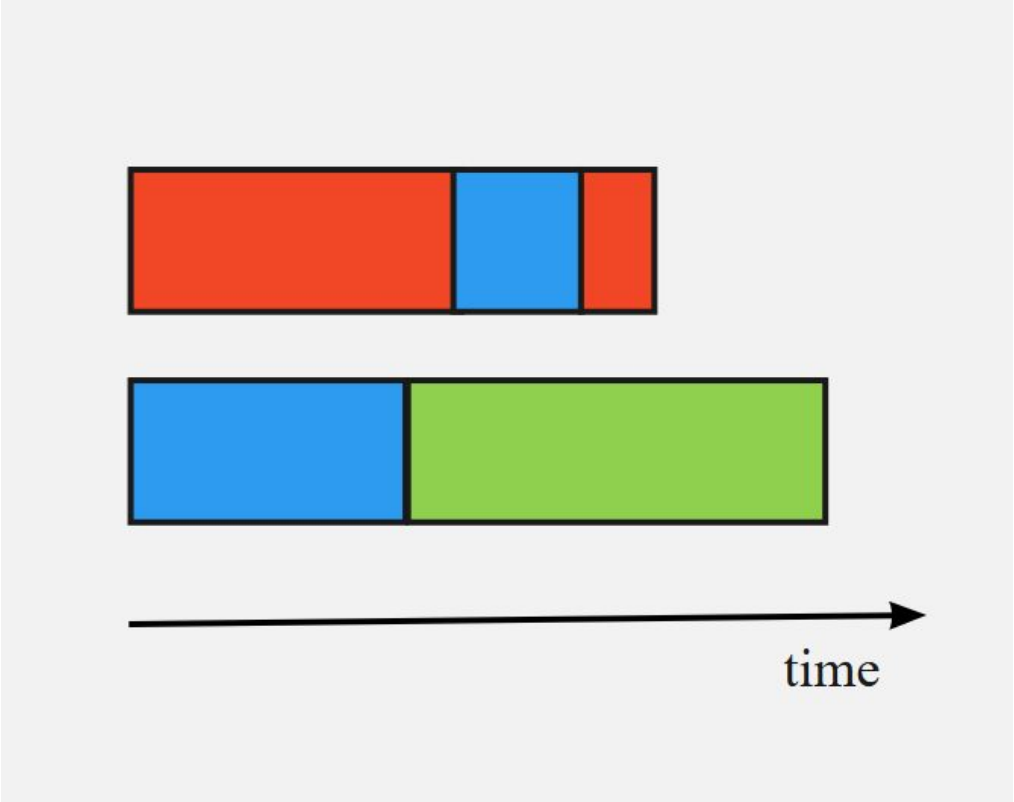
Python

Асинхронность

Синхронное выполнение



Асинхронное выполнение

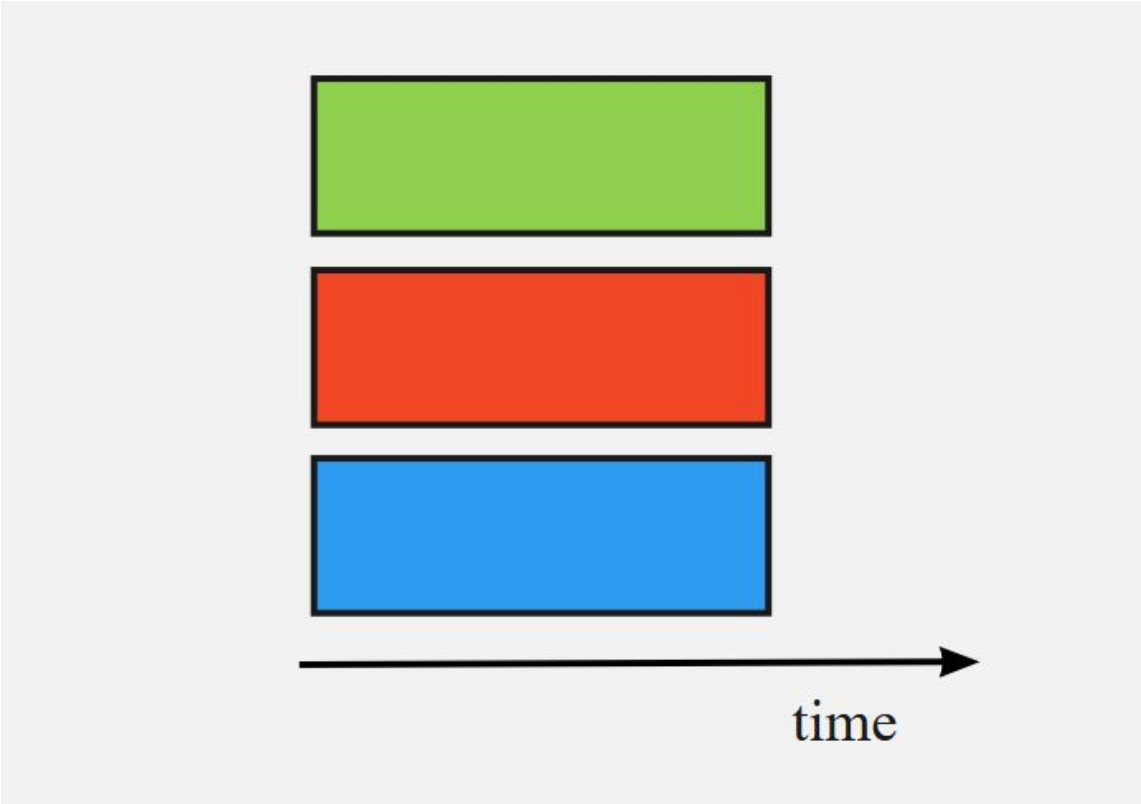


Виды выполнения задач

Синхронное - задачи выполняются друг за другом

Асинхронное - задачи могут запускаться и завершаться независимо друг от друга.

Параллельное выполнение



Конкурентное выполнение

Виды выполнения задач

Конкурентное - задачи выполняются совместно.

Параллельное - задачи выполняются параллельно. По сути, является формой конкурентности. Зависит от оборудования.

Асинхронность в python

1. Процессы
    - запуск нескольких экземпляров скрипта в терминале
    - библиотека multiprocessing
2. Потоки (библиотека threading)

В обоих случаях ОС контролирует распределение ресурсов процессора.
Потоки ограничены выполнением на 1 ядре процессора.

3. Asyncio

Процессы

```
import multiprocessing
import time
import random

def worker(number):
    sleep = random.randrange(1, 10)
    time.sleep(sleep)
    print(f'I am Worker {number}, I slept for {sleep} seconds')

if __name__ == '__main__':
    for i in range(5):
        p = multiprocessing.Process(target=worker, args=(i,))
        p.start()
```

Потоки

```
import threading
import time
import random

def worker(number):
    sleep = random.randrange(1, 10)
    time.sleep(sleep)
    print(f'I am Worker {number}, I slept for {sleep} seconds')

if __name__ == '__main__':
    for i in range(5):
        t = threading.Thread(target=worker, args=(i,))
        t.start()
```

Процессы/потоки

```
> I am Worker 3, I slept for 1 seconds
> I am Worker 1, I slept for 3 seconds
> I am Worker 0, I slept for 3 seconds
> I am Worker 2, I slept for 8 seconds
> I am Worker 4, I slept for 9 seconds
```

Asyncio

Несколько основных понятия в asyncio:

1. Event loop (цикл событий) - управляет выполнением различных задач: регистрирует поступление и запускает в подходящий момент
2. Корутины - специальные функции, похожие на генераторы, от которых ожидают, что они будут отдавать управление обратно в цикл событий.
3. Future (футуры) - объекты, в которых хранится текущий результат выполнения какой-либо задачи.

Специальный тип футуры Task позволяет запускать корутины в цикле событий.

Корутины

Обычно корутина – это асинхронная функция. Но может быть также специальным объектом.

```
async def say_after(delay, what):
    ...
```

await

Ключевое слово. Указывает, что при выполнении следующего за ним выражения возможно переключение потока выполнения.

```
await say_after(1, 'hello')
```

Применяется только к awaitable выражениям.

- Корутины
- Объекты, у которых реализован специальный метод __await__

Пример

```
import asyncio


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


asyncio.run(say_after(5, 'hello'))
```

Пример

```
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())
```

Пример

```
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')}")

    await task1
    await task2

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

Возвращаемые значения

```
async def nested():
    print('nested')
    return 42

async def main():
    nested()
    result = await nested()
    print(await nested())

asyncio.run(main())
```

Функции asyncio

`sleep(delay, result=None)` - останавливает выполнение на `delay` секунд. Возвращает `result` - если задано. Всегда приостанавливает текущую задачу.

`await asyncio.sleep(0)` - простой способ передать исполнение другим ожидающим задачам.

Функции asyncio

gather(*aws, return_exceptions=False) - запускает все awaitable-объекты, переданные в aws (автоматически оборачивает корутины в task). Возвращает список значений, после выполнения всех задач.

```
result = await asyncio.gather(nested(), nested())
print(result)
```

```
> [42, 42]
```

Функции asyncio

wait(aws, timeout=None, return_when=ALL_COMPLETED) - возвращает завершенные и ожидающие задачи. return_when может принимать значения: FIRST_COMPLETED, ALL_COMPLETED, FIRST_EXCEPTION

```
done, pending = await asyncio.wait([
        asyncio.create_task(say_after(1, 'hello')),
        asyncio.create_task(say_after(2, 'world'))
    ], return_when=asyncio.FIRST_COMPLETED
)
```

Упражнение 1

Реализовать асинхронную функцию проверки, что переданное число является простым. Выводить сообщение на каждой итерации проверки числа.

Проверить 2 числа. В процессе работы контекст выполнения должен переключаться между проверкой этих чисел.

Упражнение 2

Реализовать функцию, которая будет возвращать содержимое файла, не блокируя выполнение других задач.

Упражнение 3

Реализовать функцию, которая будет возвращать содержимое файла, не блокируя выполнение других задач и не превышая ограничение на скорость чтения (передается параметром в функцию).

Домашнее задание

Необходимо написать программу, реализующую сеанс одновременной игры в шахматы. Играет гроссмейстер и 8 любителей. Гроссмейстер делает ход за 3-5 секунд, любители за 30-50 секунд. Первыми ходят любители. Для простоты считаем, что партия продолжается 20 ходов.

На каждый ход любителя выводить сообщение, что он сделал ход и его имя. На каждый ход гроссмейстера, аналогично и выводить имя любителя, против которого он играет.
Выводить сообщение об окончании партии (с именем шахматиста) и об окончании всех партий.