In [1]:
```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

Python3

Asyncio, coroutines, tasks

MIPT 2020

Здесь будут рассмотрены основные понятия библиотеки asyncio. Материалы и примеры сильно опираются на документацию и версию питона 3.8

https://docs.python.org/3/library/asyncio-task.html (https://docs.python.org/3/library/asyncio-task.html)

Мотивация

asyncio предоставляет особый нелинейный подход к созданию программ. Да, в питоне есть GIL, который запрещает использовать более одного потока, но тем не менее можно экономить время на различных видах блокировок - input/output, connections, sleep, **manual**. Последнее входит в парадигму, с помощью которой можно писать программы, основываясь на так называемых событиях

Coros and tasks

Coroutines

Корутины объявляются через async/await и позволяют конкурентно выполнять задания

In [53]:
```
import asyncio

async def main():
    print('hello')
    await asyncio.sleep(2)
    print('world')

# Чтобы запустить main() извне, нужно вызвать run,
# он создаст event loop и будет работать с корутинами
# В ноутбуке event loop уже настроен, поэтому можно считать, что run кто-то выполнил за нас

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

```
hello
world
```

В базовом варианте `async def main()` означает, что `main` - функция корутины, `main()` - собственно объект корутины. Замечу, что строчка без await не сделает ничего

In [4]:
```
main()
```

Out[4]:
```
<coroutine object main at 0x7f5ce45b0040>
```

Похоже на генераторы

In [55]:
```
def gen():
    yield 1
gen()
```

Out[55]:
```
<generator object gen at 0x7f5cd5b7d270>
```

Использование корутин ТОЛЬКО при помощи async/await ничем не отличается от линейного исполнения

In [59]:
```python
import time

async def say_after(delay, what):
    print('something')
    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')}")

await main()
```

```
started at 09:24:00
something
hello
something
world
finished at 09:24:03
```

Здесь мы ждем все 3 секунды

Tasks

Есть и другой **awaitable** объект (то есть к которому валидно приписывать await) - `Task` . Он создается с помощью `asyncio.create_task` и означает то, что мы положили в event loop новую задачу, которая реально будет выполнена в неизвестный момент времени, но доисполняется, когда будет вызов await. Если говорить о последнем, то в строчке await мы всегда дожидаемся конца исполнения послестоящего объекта.

In [61]:
```python
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')}")

await main()
```

```
started at 09:26:03
something
something
hello
finished at 09:26:04
world
```

Вопрос на засыпку - что будет, если убрать `await task1` ?

Futures

Эти объекты означают **конечный результат** исполнения асинхронной операции. Это низкоуровневый примитив, обычно нет смысла использовать его в коде

Хороший пример исполнения - `asyncio.gather`

In [62]:
```
async def printer(num):
    await asyncio.sleep(num)
    print(num)

future = asyncio.gather(
    printer(1),
    printer(2),
)
future
await future
```

Out[62]:
```
<_GatheringFuture pending>

1
2
```

Out[62]:
```
[None, None]
```

Функциональность модуля asyncio

asyncio.gather

In [63]:
```
async def even_thrower(num):
    await asyncio.sleep(num)
    if num % 2 == 0:
        raise ArithmeticError
    return num

try:
    await asyncio.gather(
        even_thrower(1),
        even_thrower(2),
        even_thrower(3),
    )
except ArithmeticError:
    print("error occured, everything canceled")
```

```
error occured, everything canceled
```

In [64]:
```
await asyncio.gather(
    even_thrower(1),
    even_thrower(2),
    even_thrower(3),
    return_exceptions=True,
)
```

Out[64]:
```
[1, ArithmeticError(), 3]
```

asyncio.shield

Защищает корутину от того, чтобы она была отменена

Если вызывающая корутина была отменена, то это не спасет внутреннюю даже под shield

In [65]:
```
async def canceller(coro):
    coro.cancel()

task = asyncio.create_task(asyncio.sleep(2))

await canceller(task)
try:
    await task
except asyncio.CancelledError:
    print('Cancelled')
```

```
Cancelled
```

In [66]:
```
async def cancelled_coro():
    task = asyncio.create_task(asyncio.sleep(2, result=3))

    await canceller(asyncio.shield(task))
    return await task

print(await cancelled_coro())
```

```
3
```

asyncio.wait_for

Не дает выполняться корутине дольше таймаута. Кидает `TimeoutError`. При этом сначала завершается корутина, затем кидается исключение

In [75]:
```python
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!')

await main()
```

```
timeout!
```

asyncio.wait

Выполняет сет корутин до `return_when`, который является параметром функции. По дефолту `return_when=ALL_COMPLETED`

Возвращает 2 сета - выполненных и еще нет корутин (при этом важно заметить, что возвращают связанные `Task`, а не старые корутины)

In [86]:
```
task1 = asyncio.create_task(printer(1))
coros = {printer(3), printer(2), task1}
done, pending = await asyncio.wait(coros, return_when=asyncio.FIRST_COMPLETED)
print(done, pending)
```

```
1
{<Task finished name='Task-194' coro=<printer() done, defined at <ipython-input-62-4a425ba1d080>:1> result=None>} {<T
ask pending name='Task-196' coro=<printer() running at <ipython-input-62-4a425ba1d080>:2> wait_for=<Future pending cb
=[<TaskWakeupMethWrapper object at 0x7f5cd635a7f0>()]>>, <Task pending name='Task-195' coro=<printer() running at <ip
ython-input-62-4a425ba1d080>:2> wait_for=<Future pending cb=[<TaskWakeupMethWrapper object at 0x7f5cd635a910>()]>>}
2
3
```

In [87]:
```
for task in done:
    print(task.result())
```

```
None
```

Замечу, что остальные корутины в этот момент не отменяются, а также что при выполнении множественных корутин всегда для корутины мгновенно создается `Task`

asyncio.as_completed

Аналогично `wait`, только создает итератор в порядке выполнения корутин

In [89]:
```
coros = {printer(3), printer(2), printer(1)}
for coro in asyncio.as_completed(coros):
    print(f'waiting next {coro}')
    await coro
    print('next coroutine done')
```

```
waiting next <coroutine object as_completed.<locals>._wait_for_one at 0x7f5cd5bdb2c0>
1
next coroutine done
waiting next <coroutine object as_completed.<locals>._wait_for_one at 0x7f5cd5bdbd40>
2
next coroutine done
waiting next <coroutine object as_completed.<locals>._wait_for_one at 0x7f5cd5bdb2c0>
3
next coroutine done
```

Также есть однопоточные (не thread-safe) примитивы синхронизации, позволяющие настраивать общение корутин

Всего их 4 вида - `Lock, Semaphore, Event, Condition` . Последний - объединение lock + event

asyncio.Lock

In [106]:
```
lock = asyncio.Lock()

async def solo_waiter(num):
    async with lock:
        await asyncio.sleep(num)
    return num

print(f'Before start {time.strftime("%X")}')
for coro in asyncio.as_completed({solo_waiter(i) for i in range(1, 4)}):
    res = await coro
    print(f'Coro result {res}, time: {time.strftime("%X")}')
```

```
Before start 10:14:31
Coro result 1, time: 10:14:32
Coro result 3, time: 10:14:35
Coro result 2, time: 10:14:37
```

asyncio.Semaphore

In [105]:
```
sem = asyncio.Semaphore(2)

async def solo_waiter(num):
    async with sem:
        await asyncio.sleep(num)
    return num

print(f'Before start {time.strftime("%X")}')
for coro in asyncio.as_completed({solo_waiter(i) for i in range(1, 4)}):
    res = await coro
    print(f'Coro result {res}, time: {time.strftime("%X")}')
```

```
Before start 10:13:17
Coro result 1, time: 10:13:18
Coro result 2, time: 10:13:19
Coro result 3, time: 10:13:21
```

asyncio.Event

In [102]:
```
async def waiter(event):
    print('waiting for it ...')
    await event.wait()
    print('... got it!')

async def main():
    # Create an Event object.
    event = asyncio.Event()

    # Spawn a Task to wait until 'event' is set.
    waiter_task = asyncio.create_task(waiter(event))

    # Sleep for 1 second and set the event.
    await asyncio.sleep(1)
    event.set()

    # Wait until the waiter task is finished.
    await waiter_task

await main()
```

```
waiting for it ...
... got it!
```

In [ ]: