Асинхронные возможности Python

Асинхронность - это последовательное выполнение произвольных кусков кода. Никакой параллельности механизм асихронности не даёт, однако позволяет переключаться между вычислительными блоками для постепенного выполнения каждого

Будем потихоньку изобретать логику асинхронного программирования, усложняя примерчики
Стартанём с поочерёдно выполняемых кусков кода в Генератора

```
>>> def fun():
...     yield 'One'
...     yield 'Two'
...
>>> def run():
...     yield from fun()
...     yield from fun()
...
>>> for i in run():
...     print(i)
...
One
Two
One
Two
```

```
>>> def fun(n):
...     yield 'One'
...     yield 'Two'
...     return n
...
>>> def run():
...     res1 = yield from fun(1)
...     res2 = yield from fun(2)
...     print(res1, res2)
...
>>> list(run())
1 2
['One', 'Two', 'One', 'Two']
```

```
>>> def fun(n):
...     yield 'One'
...     yield 'Two'
...     return n
...
>>> def run():
...     res1 = yield from fun(1)
...     res2 = yield from fun(2)
...     return res1, res2
...
>>> r = run()
>>> next(r)
'One'
>>> next(r)
'Two'
>>> next(r)
'One'
>>> next(r)
'Two'
>>> next(r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: (1, 2)
```

Следующий шаг - передача параметра в генератор через .send()

```
>>> def task(res):
...     print('start', res)
...     while(True):
...         res = yield f'/{res}/'
...
>>>
>>> T = task(100500)
>>> next(T)
start 100500
'/100500/'
>>> T.send(42)
'/42/'
>>> next(T) # Что то же самое, что и T.send(None)
'/None/'
```

До этого порядок исполнения нам явно был понятен (вспоминаем, как работают yield в генераторах - просто возвращаемся в код и продолжаем работу)

Теперь сделаем подобие Асинхронности, как исполнения произвольных частей кода между yield-ами
Справедливости ради, в наших примерах всё ещё однозначно определяется порядок. Мы позже поговорим, где реально произвольное (как бы) исполнение кусков кода будет

```
>>> def mult():
...     x = yield "Give me X"
...     y = yield "Give me Y"
...     return x * y
...
>>> def task():
...     res = 0
...     while True:
...         res = yield from mult()
...         yield res
...
>>>
>>> core = task()
>>> next(core)
'Give me X'
>>> for i in range(1, 10):
...     print(core.send(i))
...
Give me Y
2 # поймали 1 и 2, произведение 2, она и напечаталась
# вот тут как бы на yield res приходит вызов core.send(3), значение там никуда не пишется, поэтому мы работу-то продолжаем, но тройку в расчётах как бы теряем
Give me X
Give me Y
20 # поймали 4 и 5, произведение 20
Give me X
Give me Y
56 # поймали 7 и 8, произведение 56
Give me X
```

Вот тут явно видно, что мы ловим значение, которое не считалось

```
>>> def mult():
...     x = yield "Give me X"
...     y = yield "Give me Y"
...     return x * y
...
>>> def task():
...     res = 0
...     while True:
...         res = yield from mult()
...         res2 = yield res
...         print('Got', res2)
...
>>> core = task()
>>> next(core)
'Give me X'
>>> for i in range(1, 10):
...     print(core.send(i))
...
Give me Y
2
Got 3
Give me X
Give me Y
20
Got 6
Give me X
Give me Y
56
Got 9
Give me X
```

Теперь ещё чуть больше приблизимся к асинхронности: сделаем образующий цикл (main loop), который будет давать задачи вразнобой
Пока что всё ещё можем явно сказать порядок выполнения. Вообще, асинхронность нам и не говорит, что этот порядок неисчислим и случаен, главное, что мы между кусками кода прыгаем. Просто я явно указываю на то, что мы всё ещё можем явно предугадать исполнение

```
>>> def mult(name):
...     x = yield f"{name}: Give me X"
...     y = yield f"{name}: Give me Y"
...     return x * y
...
>>> def task(name):
...     while True:
...         value = yield from mult(name)
...         yield f"[{name}]: {value}"
...

>>> tasks = task('One'), task('Two') # Создали два потока вычислений (два самостоятельных генератора)
>>> next(tasks[0]), next(tasks[1]) # Запустили их
('One: Give me X', 'Two: Give me X')
>>> for i in range(20): # вот этот образующий цикл нам как бы случайно переключает управление между кусками кода
...     n = not i % 3
...     print(tasks[n].send(i))
...
Two: Give me Y
One: Give me Y
[One]: 2
[Two]: 0
One: Give me X
One: Give me Y
Two: Give me X
[One]: 35
One: Give me X
Two: Give me Y
One: Give me Y
[One]: 110
[Two]: 108
One: Give me X
One: Give me Y
Two: Give me X
[One]: 224
One: Give me X
Two: Give me Y
One: Give me Y
```

Там у нас тоже некоторые значения отсеиваются чисто на продолжение работы (yield без присваивания, как в примерах выше), посмотрим, что и куда приходит вообще

```
>>> def mult(name):
...     x = yield f"{name}: Give me X"
...     y = yield f"{name}: Give me Y, ({x=})"
...     return x * y
...
>>> def task(name):
...     while True:
...         value = yield from mult(name)
...         yield f"[{name}]: {value}"
...
...
>>> tasks = task('One'), task('Two')
>>> next(tasks[0]), next(tasks[1])
('One: Give me X', 'Two: Give me X')
>>>
>>> for i in range(20):
...     n = not i % 3
...     print(tasks[n].send(i))
...
Two: Give me Y, (x=0)
One: Give me Y, (x=1)
[One]: 2
[Two]: 0
One: Give me X
One: Give me Y, (x=5)
Two: Give me X
[One]: 35
One: Give me X
Two: Give me Y, (x=9)
One: Give me Y, (x=10)
[One]: 110
[Two]: 108
One: Give me X
One: Give me Y, (x=14)
Two: Give me X
[One]: 224
One: Give me X
Two: Give me Y, (x=18)
One: Give me Y, (x=19)
```

Ещё больше асинхрона: будем крутить три генератора, считающих суммы n произведений (у каждого своё n)

Что в примерчике примечательного:

1. Во-первых, yield может ничего не возвращать. Т.е. формально он возвращает нам None, а потом мы в него входим через send со значением, которое обрабатывается
2. Во-вторых, чтобы чередовать обработку генераторов, используем деку, чтобы в ней организовать очередь (с одного конца забираем в обработку, после обработки в другой конец возвращаем) От этого в результате у нас генераторы стоят в порядке окончания работы (сравните адреса при Start и Done)

```
>>> from collections import deque
>>> def mult():
...     return (yield) * (yield)
...
>>> def task(num):
...     res = 0
...     for i in range(num):
...         res += yield from mult()
...     return res
...
>>> def loop(*tasks):
...     queue, res = deque(tasks), []
...     print("Start:", *queue, sep="\n\t")
...     for task in tasks:
...         next(task)
...     while queue:
...         task = queue.popleft()
...         try:
...             task.send(randint(3, 10))
...         except StopIteration as E:
...             res.append((task, E.value))
...         else:
...             queue.append(task)
...     return res
...
>>> print("Done:", *loop(task(10), task(3), task(5)), sep="\n\t")
Start:
        <generator object task at 0x7f3272db73e0>
        <generator object task at 0x7f3272db7bc0>
        <generator object task at 0x7f3272db7e60>
Done:
        (<generator object task at 0x7f3272db7bc0>, 136)
        (<generator object task at 0x7f3272db7e60>, 144)
        (<generator object task at 0x7f3272db73e0>, 504)
>>> print("Done:", *loop(task(10), task(100), task(50)), sep="\n\t")
Start:
```

<generator object task at 0x7f3272db73e0>
<generator object task at 0x7f3272db7bc0>
<generator object task at 0x7f3272db7e60>
Done:
(<generator object task at 0x7f3272db73e0>, 422)
(<generator object task at 0x7f3272db7e60>, 2020)
(<generator object task at 0x7f3272db7bc0>, 3728)

По сути своей, мы разобрались, как асинхронность работает под капотом: Образующий цикл раздаёт управление на разные вычислительные блоки. которые работают независимо (или зависимо) друг от друга

Теперь обсудим непосредственно то, как с асинхронностью работают. А в питончике, когда с чем-то надо работать, на это уже давно придумали свой модуль. Имя ему ***asyncio***

В асинхронности придумали кучу фишек по синхронизации состояний этих асинхронных процессов, по задержкам и структурам, которыми можно пользоваться в работе. Мы затронем самую базу работы

Наш модуль самостоятельно реализовывает образующий цикл. Нам вообще не надо париться, каким образом и какими алгоритмами выбирается другое вычисление. Главное, что у него есть это множество потоков обработки, между которыми мы будем переключаться

Каждая функция, которую мы хотим асинхронно обрабатывать, называется ***корутина*** и обозначается ключевым словом ***async***

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

Внутри корутин \ основной стартовой корутины (с которой начнётся обработка асинхронной работы) для вызова корутины испольуется ключевое слово ***await***. Именно в await-точках у нас может произойти переключение работы: мы как бы забрасываем в общую очередь корутин образующего цикла новую задачку (ту самую корутину, которую хотим обработать), а после отдаём управление, и образующий цикл по своим каким-то логикам включает в работу какую-то из корутин

Включение корутин образующим циклом НЕ случайное. У него под капотом алгоритм, оценивающий встроенные в модуль корутины (типа вышеуказанной asyncio.sleep, которая делает задержку работы вызвавшей её корутины на заданное время) и наши корутинки и в каком-то неизвестном нам, но вполне строгом порядке для одного и того же кода вызывает их

Для запуска асинхронной программы используется команда ***asyncio.run(arg)***, где аргумент - стартовая корутина

Давайте рассмотрим простую программку, печатающую асинхронно фразу

```
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())
```

```
started at 17:13:52
hello
world
finished at 17:13:55
```

Для удобства работы хочется взаимодействовать как бы с объектами - задачками -, которые можно передавать везде и в нужном месте уже запускать. Для такой логики работы в модуле есть возможность преобразования корутины в, буквально, Задачу, которую можно будет передать куда-то или запустить выполнение.

Создание Задачки из корутины делается методом ***create_task(coroutine)***

```
import asyncio

async def nested():
    return 42

async def main():
    task = asyncio.create_task(nested())
    await task

asyncio.run(main())
```

Однако в обоих случаях мы запускаем наши корутины/задачи последовательно друг относительно друга. А хотелось бы, н-р, подать на вход все задачи одновременно, а то, как асинхронно они будут выполняться, уже не так важно для нас

Для одновременного “запуска” корутин в ***asyncio*** есть метод ***gather***, получающий набор корутин в аргументах, и асинхронный менеджер ***TaskGroup***
Они оба осуществляют одновременный запуск корутин. При этом независимо от того, в каком порядке завершились корутины, оба возвращают списки завершённых корутин в изначальном порядке

```
import asyncio

async def late(delay, msg):
    await asyncio.sleep(delay)
    print(msg)
    return delay

async def main():
    res = await asyncio.gather(
        late(3, "A"),
        late(1, "B"),
        late(2, "C"),
    )
    print(res)

asyncio.run(main())
```

```
B
C
A
[3, 1, 2]
```

```
import asyncio

async def late(delay, msg):
    await asyncio.sleep(delay)
    print(msg)
    return delay

async def main():
    async with asyncio.TaskGroup() as tg:
        tg.create_task(late(3, "A"))
        tg.create_task(late(1, "B"))
        tg.create_task(late(2, "C"))
    print("Done")

asyncio.run(main())
```

```
B
C
A
Done
```