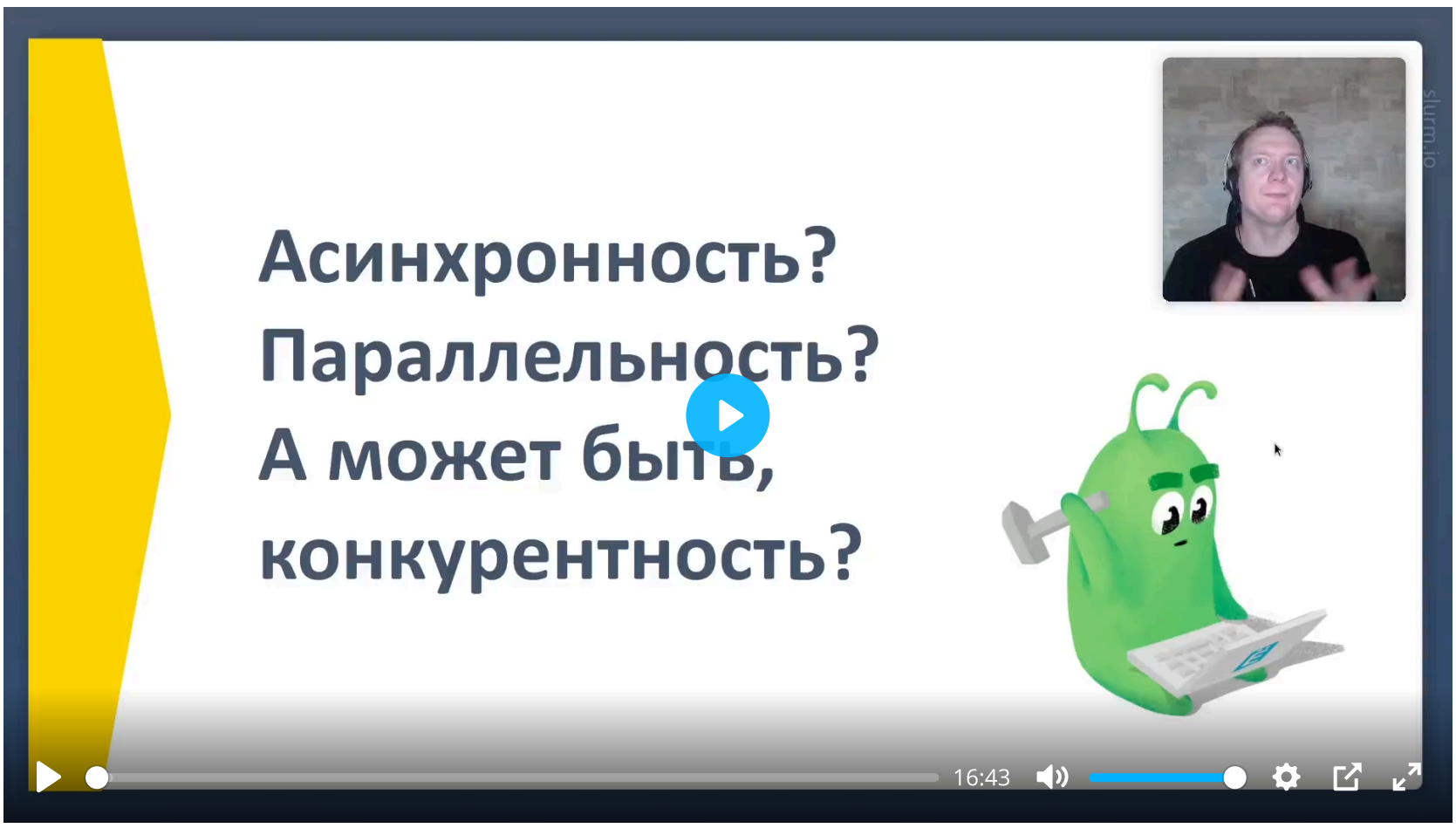
2.3 Параллельность и асинхронность в Python



Текстовая расшифровка видео:

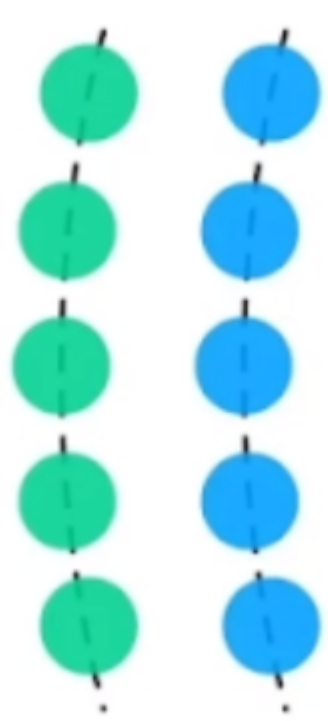
АСИНХРОННОСТЬ? ПАРАЛЛЕЛЬНОСТЬ? А МОЖЕТ БЫТЬ, КОНКУРЕНТНОСТЬ?

План:

- О терминах;
- Корутины и event loop;
- Событийно-ориентированное программирование;
- Классические генераторы.

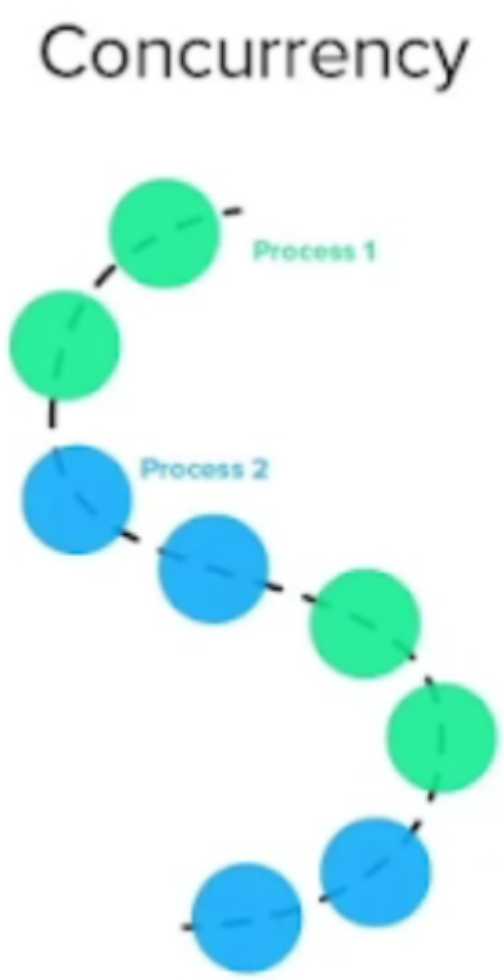
О терминах

Параллельность – это когда два фрагмента кода выполняются физически одновременно, например, на разных ядрах CPU.



Асинхронность – это когда код выполняется непоследовательно, как написан в файле, а в виде реакций (callback’ов) на события. Это может происходить как параллельно, так и нет.

Конкурентность – это общее название ситуации, когда несколько задач выполняются не в строго определенном порядке, а иногда и с перехлестом по времени.



Корутины и event loop

Основные положения:

1) Асинхронность может быть реализована с помощью **параллельности** или **ручного переключения** контекста в коде, сохраняя состояние.

2) Исторически в Python есть такая концепция – концепция генераторов. Существует ключевой метод «Yeld», с помощью которого мы буквально можем передавать контекст (одно значение из генератора мы можем перенести в другое место, сохранив состояние этого генератора на том месте, где он остановился).

3) **«Кооперативная многозадачность»** – фрагменты кода (корутины или сопрограммы), которые **самостоятельно определяют, когда передавать управление** друг другу и не зависят от внешнего системного планировщика.

4) Нужна дополнительная сущность, чтобы **связывать фрагменты кода с событиями**, которая называется **«event loop»**

5) **Недостаток** – долгоиграющая процедура не под контролем event loop вешает все процессы приложения.

Событийно-ориентированное программирование

Основные положения:

1) Пока корутина **ждет внешнее событие**, контекст переключается на другую.

2) Сообщение о переключении контекста может содержать в себе **данные** для другой корутины.

3) В современной реализации асинхронности в Python **обычные и асинхронные функции не являются взаимозаменяемыми**.

4) Существуют альтернативные реализации для старых версий – **Gevent, Eventlet, Tornado** и др.

Классические генераторы

До привлечения новых современных концепций с async/await уже была возможность делать и использовать корутины, чтобы слать данные в две стороны.

```
1. import random
2.
3. def multiply_gen(lst):
4.   multiplier = 2
5.   for i in lst:
6.       multiplier = (yield multiplier) or multiplier
7.       print(f"Multiplier is now {multiplier}")
8.
9. data = [random.randint(-10, 10) for _ in range(10)]
10. mult_data = multiply_gen(data)
11. for i, entry in enumerate(zip(data, mult_data)):
12.    print(f"{i}: {entry} {entry[0] * entry[1]}")
13.    if i == 4:
14.       mult_data.send(3)
15.       print(f"{i}: {entry} {entry[0] * entry[1]}")
```

Функция «yield» позволяет превращать функцию в генератор и может из нее вернуть значение, сохраняя ее внутреннее состояние.

Функция «yield» может не только возвращать значения, но и принимать. Есть возможность использовать у генератора метод «.send» для того, чтобы в место вызова «yield» прислать новое значение, которое генератор может использовать для своих целей.

Данный код, предоставленный в качестве примера, работает:

Мы перебираем элементы в генераторе и в середине, если доходим до элемента с №4, с помощью «.send» можем отправить в генератор значение и поменять множитель.

Данный код достаточно тяжелый для понимания. Представить себе продакшн, написанный в таком коде сложно. Тем не менее почти все, что написано в Python поверх асинхронки (новый API) реализовано поверх данной парадигмы.

Дэвид Бизли – инженер-программист написал лекцию о генераторах и корутинах.

Как вам урок? 😡 😐 🔥

Далее ›