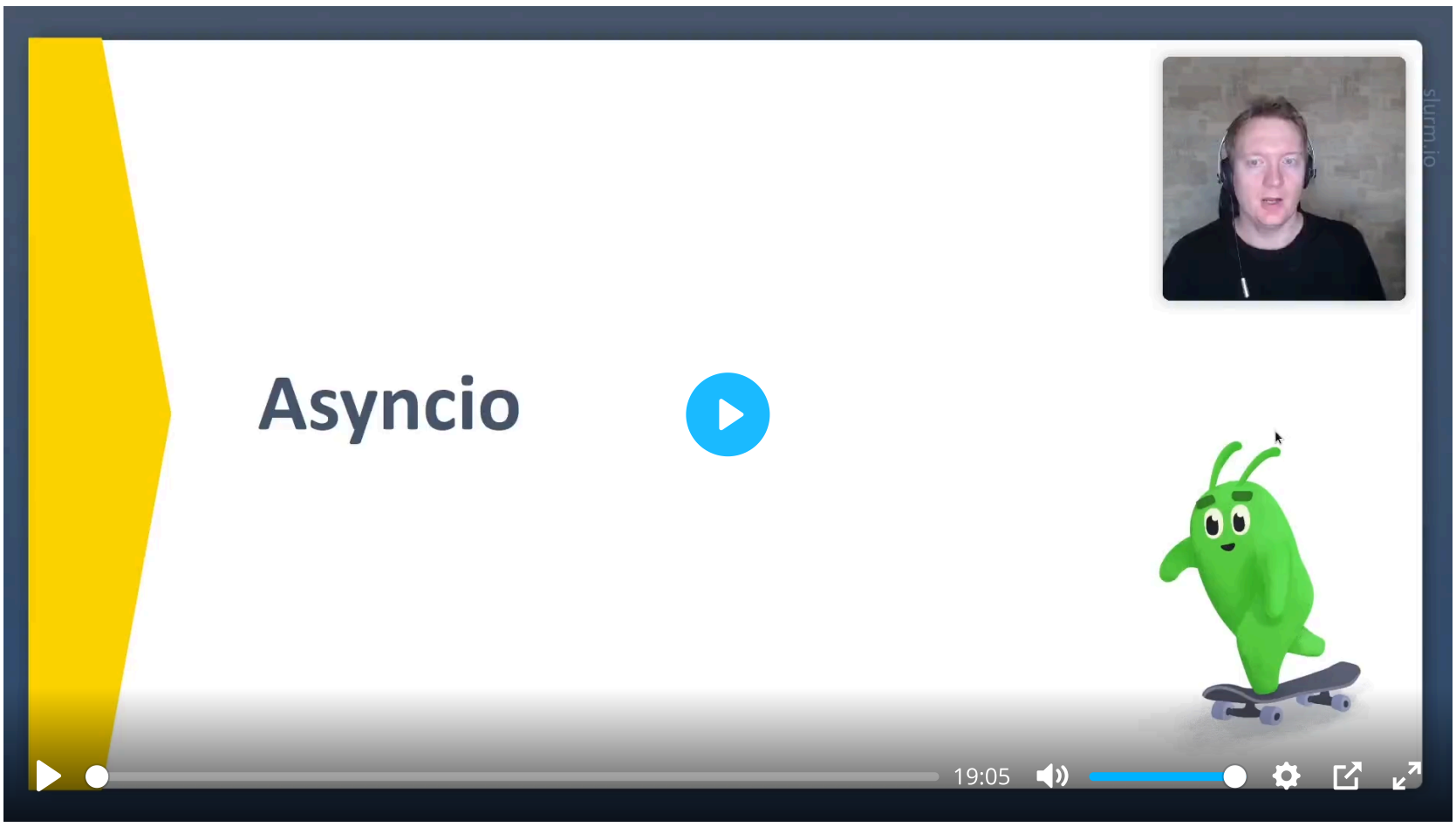
2.3 Параллельность и асинхронность в Python



Текстовая расшифровка видео:

ASYNCIO

План:

- Asyncio;
- Ключевые слова async/await;
- Когда использовать await?;
- Решение проблем.

Asyncio

Сегодня мы рассмотрим «великий и ужасный» модуль, который приносит в Python асинхронность, а именно, – «Asyncio».

Сложность его заключается в том, что он дает поверх API.

Проблема с асинхронностью во многих языках, включая Python, заключается в появлении двух видов разных функций (синхронная/асинхронная, корутина/функция). Это приводит к тому, что стандартной библиотеке приходится иметь для разных операций несколько реализаций, несколько имплементаций – асинхронных и синхронных.

Так, в модуль Asyncio добавили многочисленные реализации, которые можно использовать в контексте асинхронного программирования, например:

```
1. import asyncio
2.
3. asyncio.Queue()  # асинхронная очередь
4. asyncio.sleep(10)  # асинхронный “сон"
5. # асинхронный subprocess
6. asyncio.create_subprocess_exec()
7. asyncio.Lock()  # асинхронный мьютекс
8. # ручное добавление корутины в event loop
9. asyncio.ensure_future()
10. # дождаться окончания работы списка корутин
11. asyncio.gather()
```

- Асинхронную очередь (asyncio queue), при работе с которой можно переключать асинхронный контекст в ивент-лупе (event loop).
- Асинхронный сон (asyncio sleep), реализация которого заключается в возможности переключения между корутинами (одна корутина «спит», пока мы работаем в другой).
- Асинхронный subprocess, реализация которого заключается в возможности запуска подпроцесс в асинхронном режиме.
- Асинхронный мьютекс (asyncio lock), реализация которого заключается в возможности выбора порядка действий для использования синхронных примитивов, так как нам изначально не известен порядок переключения контекста.
- Asyncio.ensure_future (дождаться окончания работы списка корутин) – дополнительный примитив, который позволяет добавить в ивент-луп какое-либо задание
- Asyncio.gather – дополнительный примитив, позволяющий дожидаться ответа от всех корутин, стоящих в очереди.

Если мы пишем асинхронный код, нам стоит заглянуть в модуль Asyncio и проверить наличие того, что могло бы принести пользу в реализации асинхронной версии.

Ключевые слова async/await

Если говорить про написание современного асинхронного кода на питоне, то можно привести простой пример:

```
1. import asyncio
2.
3. async def hello(name):
4.   return "Hello, {}!".format(name)
5.
6. async def call_vasya():
7.   greeting = await hello("Vasya")
8.   return greeting
9.
10. loop = asyncio.get_event_loop()
11. print(loop.run_until_complete(call_vasya()))
```

Основные два понятия, которые нужны в асинхронном коде – это корутины и ивент-лупы. Корутины и асинхронные функции мы можем задавать с помощью ключевого слова «async». Если простую функцию мы задаём с помощью «def», то асинхронную – «async def». С помощью «async» мы задаём, какие из них асинхронные, а с помощью «await» мы создаём event в loop на ожидание.

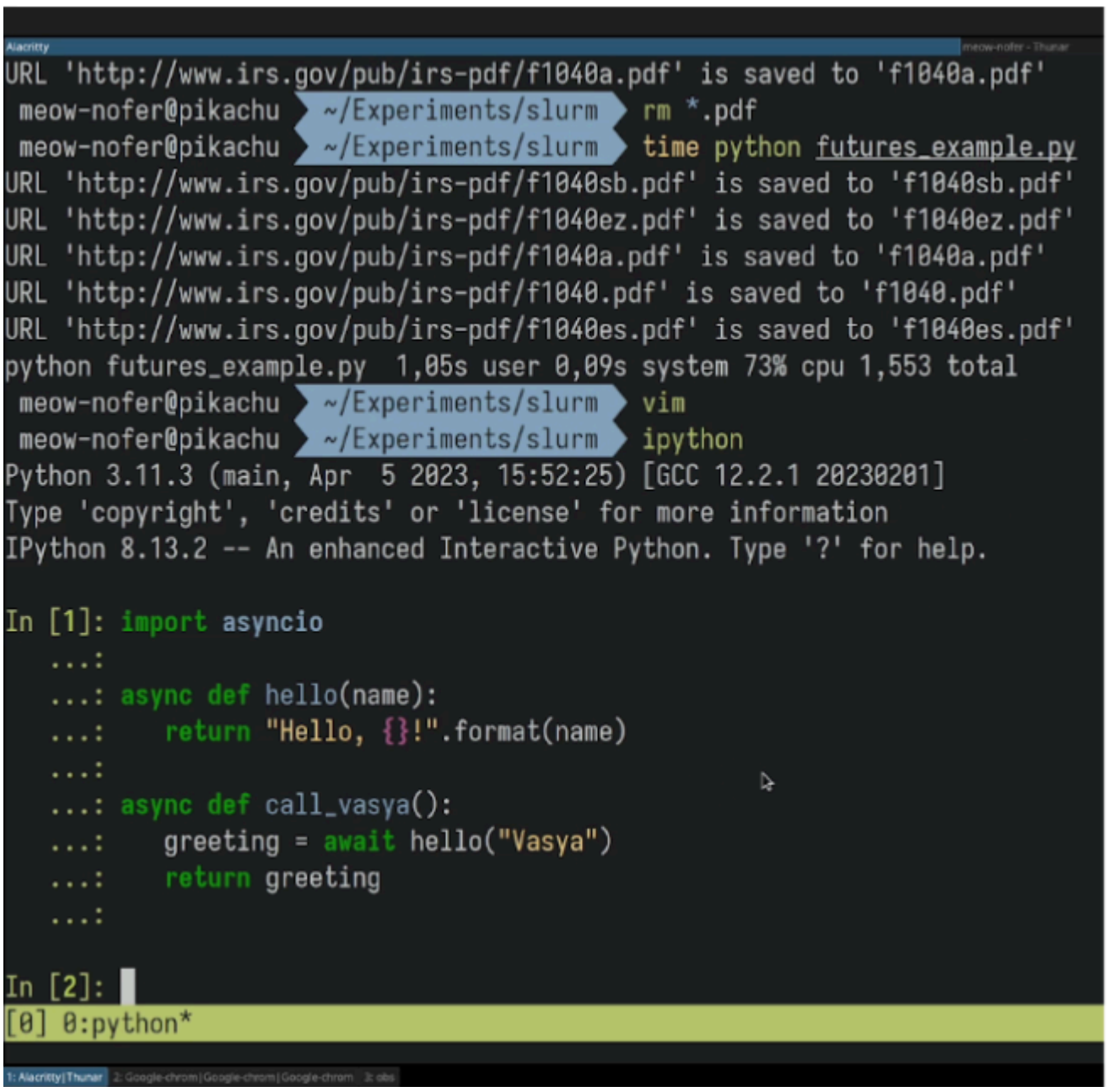
Обратим внимание на пример (см. выше):

У нас есть две крутины – «hello» (которая возвращает какую-либо строчку с именем) и «call_vasya» (которая просит запуститься корутину «hello»), то есть она говорит в event loop: «запусти корутину “hello”, когда она отработает, запусти меня обратно». Когда это все отрабатывается, мы возвращаем значение в «greeting» и возвращаем из корутины «call_vasya» – «greeting».

Помимо этого, здесь происходит явное создание event loop. При наличии корутины, мы создаем event loop – asyncio.get_event_loop (). С помощью метода «run_until_complete» мы вызываем данную корутину, раскручивая цепочку вызовов. Например:

- вызывается «call_vasya»;
- «call_vasya» вызывает «hello» и с аргументами добавляет события в event loop;
- event loop, видя, что есть корутина на выполнение «hello», выполняет ее;
- по завершении event loop видит корутину, которая ждет результат выполнения той и возвращает контекст, тем самым «call_vasya» завершается.

Мы можем самостоятельно запустить этот код в интерпретаторе:



Можно создать такой же event loop, который мы привели в пример ранее. В современном Python есть более простая обертка для этого – «asyncio.run».

Обратите внимание: если мы вызовем функцию напрямую (например, вызовем функцию «hello»), то у нас это не получится. Причина – вызов в обход event loop (отсутствие ивента, на который функция должна среагировать):

Более того, если забыть написать «await», то возникнет специальный тип ошибки «warning», говорящий о существовании корутины, не находящейся под await. Это говорит о проблеме в архитектуре. Функции важно запускать в event loop.

Концепция «async/await» была создана для того, чтобы код, который мы пишем в асинхронном виде, был похож на обычный синхронный код. При этом, если будет сложная структура, например, работа с сетевыми данными, дисками и т.д., в случае с «async/await» мы сможем одновременно делать большее количество операций и обслужить разное количество tasks'ов, а также реализовать кооперативную многозадачность. В контексте Python все это будет работать в рамках одного потока.

Когда использовать await?

С общей перспективы мы можем выделить следующее:

- Чтение и запись байтов в любое устройство осуществляется через системный вызов.
- Каждый раз, когда ответ на запрос зависит от внешних (по отношению к коду) обстоятельств, мы можем смело переключить контекст на другие задачи.
- Возможность переключать контекст вручную без оглядки на механизм потоков позволяет утилизировать ресурсы системы намного эффективнее, чем если бы они работали через системный планировщик.
- Если поддержка переключения контекста не реализована в библиотеке, то есть шанс подвесить все приложение, но на случай, когда все плохо, в Python есть специальная обертка.

Когда мы говорили о потоках, мы рассматривали executing в concurrent futures. Так, у объекта loop есть метод «run_in_executor». В асинхронной программе (на asyncio) вы можете создать через concurrent futures экземпляр executor'a и с помощью «run_in_executor» какую-либо функцию отправить на запуск в отдельный поток или процесс таким образом, чтобы на нее можно было навесить «await».

Не рекомендуется: использовать в переменной один и тот же сохраненный объект event loop из разных потоков. Это является антипаттерном! У каждого потока должен быть собственный экземпляр event loop!

Решение проблем

Существует задача:

Как запустить дочерний процесс, оставив возможность посылать ему данные на стандартный ввод, читая его стандартный вывод?

В Python до версии 3.4 эта задача была нерешаема. Asyncio же помогает решить ее.

Решение:

- С помощью «Create_subprocess_exect» делаем процесс;
- Делаем колбэки (callback) и обработчики на запись и чтение данных, комбинируя их по удобству:

```
        line = await stream.readline()
        if not line:
            break
        callback(line)

async def main():

    proc = await asyncio.subprocess.create_subprocess_exec(
        'sed', r's/o/o\n/g',
        stdin=asyncio.subprocess.PIPE,
        stdout=asyncio.subprocess.PIPE,
    )

    await asyncio.wait([
        enqueue(sys.stdin.buffer, proc.stdin),
        dequeue(proc.stdout, sys.stdout.buffer.write),
    ])

    # I'm not completely sure the call to `communicate` is necessary
    (stdout_data, stderr_data) = await proc.communicate()
    await proc.wait()

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    rc = loop.run_until_complete(main())
    loop.close()
```

```
"""

import asyncio
import sys
import typing as T

async def enqueue(values: T.Iterable[bytes], stream: asyncio.StreamWriter):

    for line in values:
        stream.write(line)
        # Yield to the asyncio loop
        await stream.drain()

    # Once we've exhausted values, we need to close the async stream to signal to
    # the subprocess that it can exit
    stream.close()

async def dequeue(stream: asyncio.StreamReader, callback: T.Callable[[bytes], None]):

    while True:
        line = await stream.readline()
        if not line:
            break
        callback(line)

async def main():

    proc = await asyncio.subprocess.create_subprocess_exec(
```

Если вас интересует исследование задач касательно работы с определенной базой или устройством с помощью asyncio, то в Python на Github создали общую группу проектов под названием «Aio-libs». Там вы можете найти асинхронные обертки для чего бы то ни было для Python.

Ссылка: https://github.com/aio-libs

Asyncio имеет ряд сложностей, однако все они постепенно решаются (чтением документации, разбором примеров и т.д.). Asyncio имеет низкоуровневый интерфейс. Asyncio неплохо помогает в решении многих задач и широко используется в применении.