1. Теория
2. Практика
3. Сопрограмму можно рассматривать как обычную функцию Python. Слово **async** определяет сопрограмму.

import asyncio

async def cor(number):

return number + 1

result = cor(5)

print(result, type(result))

-------------------------------

<coroutine object cor at 0x000001fa1b5119c0> <class 'coroutine'>

Код выше возвращает объект сопрограммы, а не результат.

1. Сопрограмма наделена сверхспособностью - приостанавливаться, встретив операцию, для выполнения которой нужно заметное время. По завершении такой длительной операции сопрограмму можно “пробудить”, после чего она продолжит выполнение. Пока приостановленная сопрограмма ждет завершения операции, мы можем выполнять другой код. Такое выполнение другого кода во время ожидания и обеспечивает конкурентность внутри приложения. Можно также одновременно выполнять несколько длительных операций, что еще больше повышает производительность приложения.

Сопрограммы не выполняются, если их вызвать напрямую. Вместо этого возвращается объект сопрограммы, который будет выполнен позже. Чтобы выполнить сопрограмму, мы должны явно передать ее **циклу событий**. Можно воспользоваться функцией **asyncio.run():**

import asyncio

import random

async def cor(num):

n = random.randint(1, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

result = asyncio.run(cor(5))

print(result)

--------------------

task 5 - sec. 4

Код вернул результат сопрограммы.

Функция **asyncio.run()** делает несколько важных вещей: создает новое событие, выполняет сопрограмму и возвращает результат, останавливает и завершает цикл событий. Она – главная точка входа в приложение asyncio.

1. Слово **await** приостанавливает сопрограмму на время выполнения длительной операции. Использование ключевого слова **await** приводит к выполнению сопрограммы, а не просто к возврату объекта сопрограммы, как при прямом вызове (пункт 1). Кроме того, выражение **await** приостанавливает объемлющую сопрограмму до того момента, как сопрограмма, которую мы ждем, завершится и вернет результат. А после этого мы получим доступ к возвращенному результату, а объемлющая сопрограмма пробудится и обработает результат.

import asyncio

import random

async def func(num):

n = random.randint(1, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

async def main():

res\_a = await func(1)

res\_b = await func(2)

print(res\_a)

print(res\_b)

asyncio.run(main())

--------------------

task 1 - sec. 7

task 2 - sec. 7

--------------------

time: 0:00:14.013180

В примере выше две копии сопрограммы работают последовательно.

1. Чтобы выполнить код конкурентно нужны задачи. Задача – это обертка вокруг сопрограммы, которая планирует выполнение ее в цикле событий как можно раньше. И планирование, и выполнение происходят в *неблокирующем режиме*, т.е., создав задачу, можно сразу приступить к выполнению другого кода, пока эта задача работает в фоне.

Для создания задачи служит функция **asyncio.create\_task**(). Ей передается подлежащая выполнению сопрограмма, а в ответ она немедленно возвращает объект задачи. Этот объект можно включить в выражение **await**, которое извлечет возвращенное значение по завершении задачи.

Каждое обращение к **create\_task** возвращает управление немедленно!

import asyncio

import random

async def func(num):

n = random.randint(1, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

async def main():

a = asyncio.create\_task(func(1))

b = asyncio.create\_task(func(2))

res\_a = await a

res\_b = await b

print(res\_a)

print(res\_b)

asyncio.run(main())

--------------------

task 1 - sec. 5

task 2 - sec. 3

--------------------

time: 0:00:05.012068

В примере выше две задачи на основе сопрограмм выполняются конкурентно. Сопрограмма main дойдя до внутренних сопрограмм приостанавливается и ждет выполнение всех задач. Внутренние сопрограммы выполняются конкурентно. Далее main возобнавляет работу. Задачи не работают конкурентно, конкурентно задачи только спят!

1. У каждого объекта задачи есть метод **cancel**, который можно вызвать, если требуется остановить задачу. В результате снятия задача возбудит исключение CancelledError, когда мы ждем ее с помощью await. Это исключения можно обработать.

import asyncio

from asyncio import CancelledError

import random

async def func(num):

n = random.randint(7, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

async def main():

a = asyncio.create\_task(func(1))

cnt\_sec = 0

while not a.done():

await asyncio.sleep(1)

cnt\_sec += 1

print(cnt\_sec)

if cnt\_sec == 5:

a.cancel()

try:

res = await a

except CancelledError:

res = 'Stop by timeout > 5 secs.'

asyncio.run(main())

--------------------

Stop by timeout > 5 secs.

--------------------

time: 0:00:06.053051

В примере выше запускаем задачу, которая выполняется более 5 секунд, по прошествию 5 секунд останавливаем эту задачу. Вызов **cancel** прерывает задачу только если она уже находится в точке ожидания (await) или когда дойдет до следующей такой точки. В цикле while проверяем состояние задачи.

1. Автоматическая остановка задачи по таймауту с помощью функции **wait\_for().**

async def func(num):

n = random.randint(7, 10)

await asyncio.sleep(n)

return f'task {num} - sec. {n}'

async def main():

T = 2

a = asyncio.create\_task(func(1))

aw = asyncio.wait\_for(a, timeout=T)

try:

res = await aw

except TimeoutError:

res = f'Stop by timeout > {T} secs.'

asyncio.run(main())

--------------------

Stop by timeout > 2 secs.

--------------------

time: 0:00:02.008888

Функция **asyncio.shield()** предотвращает снятие задачи.