Асинхронное программирование

Цель занятия

После освоения темы вы:

- узнаете подходы и принципы асинхронного программирования;
- сможете писать код для асинхронных приложений на Python;
- сможете использовать возможности библиотеки `asyncio` для реализации асинхронности.

План занятия

1. Асинхронное программирование как альтернатива потокам
2. Введение в генераторы
3. Генераторы как сопрограммы
4. Синтаксис `async/await` и `awaitable`-объекты
5. Модуль `asyncio`: основные функции
6. Модуль `asyncio`: работа с сетью
7. Модуль `asyncio`: работа с потоками и процессами
8. Асинхронная загрузка данных с помощью библиотеки `httpx`

Используемые термины

Сопрограмма — функция, которая может приостанавливать свою работу в определенные моменты и возобновлять ее позже.

Цикл событий — цикл, который создает сопрограммы, управляет их выполнением, принимая решения о том, какой из них следует передать управление.

Конспект занятия

1. Асинхронное программирование как альтернатива потокам

Использования потоков в большинстве случаев достаточно для реализации многозадачности. Но у этого решения есть свои минусы.

Рассмотрим минусы использования потоков.

1. Переключение между потоками занимает заметное время.
   Переключение потоков требует от операционной системы сохранения текущего состояния выполняющегося потока, переключения на другой поток, восстановления сохраненного состояния. Указанные действия реализованы универсальным способом, что обходится недешево с точки зрения затрачиваемого времени. Если счетную функцию разбить на несколько частей и запустить в разных потоках, ее выполнение может стать в несколько раз дольше.
2. Мы не контролируем моменты, в которые происходит переключение между потоками, поэтому может возникать состояние гонки.

Обе проблемы связаны с тем, что переключение между потоками выполняется средствами операционной системы. То есть если не полагаться на средства операционной системы, а реализовать переключение между задачами самостоятельно, это может решить обе проблемы.

Есть несколько вариантов замены потоков.

1. **«Зеленые потоки»** (green threads). Реализованы в некоторых языках программирования на уровне виртуальной машины или стандартной библиотеки (Java, Go, Erlang...). В Python реализованы в дополнительных библиотеках (`greenlet`), требуют явного переключения.
   Идея состоит в том, что мы сами делаем потоки.
2. **Обратные вызовы** (callback — игра слов, «обратный звонок») — функции, которые «регистрируются» для вызова позже при наступлении какого-либо события.
3. **Сопрограммы** (coroutines) — функции, которые могут приостанавливать свою работу в определенные моменты и возобновлять ее позже. Мы в основном сосредоточимся на рассмотрении сопрограмм.

Программу с обратными вызовами мы уже рассматривали ранее, когда делали сервер, который мог обрабатывать несколько соединений одновременно. Разные

действия мы разнесли в разные функции, которые вызываем в определенный момент. Например, если сокет сервера готов читать данные, это означает, что пришло новое подключение. Мы вызываем функцию, которая обрабатывает подключение и возвращает клиента. Аналогично происходит обработка готовности сокета клиента.

```
from select import select
with socket.create_server(('127.0.0.1', 5000)) as server:
    monitoring = [server]
    while True:
        ready, *_ = select(monitoring, [], [])
        for sock in ready:
            if sock is server:
                new_client = accept_connection(sock)
                monitoring.append(new_client)
            else:
                if not proceed_message(sock):
                    sock.close()
                    monitoring.remove(sock)
```

В рассматриваемом примере такой подход оправдан и не вызывает проблем. В более сложных программах использование обратных вызовов может привести к запутанному коду. Рассмотрим структуру такой программы. Есть кнопка, при нажатии на которую мы должны вызвать функцию. Вызываемая функция делает запрос в сеть, который может быть достаточно длительным. Мы не хотим, чтобы весь интерфейс «подвис» на время выполнения запроса, поэтому делаем запрос. А когда он будет готов, вызываем еще одну функцию у запроса.

```
button.on_click(handle_click)

def handle_click():
    ...
    request = make_request(url)
    request.as_completed(handle_respnose)
    ...

def handle_request(response):
    ...
```

Получается, что мы дробим логику программы на отдельные фрагменты, которые описаны в разных местах и вызываются в разные моменты. Когда таких фрагментов становится много, становится сложно контролировать и понимать программу.

2. Введение в генераторы

Генераторные выражения используются для создания списков, множеств, словарей. Также генераторные выражения могут использоваться и сами по себе.

Рассмотрим пример генераторного выражения. В следующем фрагменте мы создаем объект `g` — генераторное выражение:

```
>>> g = (x ** 2 for x in range(1, 10))
>>> g
<generator object <genexpr> at 0x7f3bf0c673e0>
>>> type(g)
<class 'generator'>
```

Список методов созданного объекта содержит методы `__iter__` и `__next__`:

```
>>> dir(g)
[... '__iter__' ... '__next__' ...,
 'close', 'gi_code', 'gi_frame',
 'gi_running', 'gi_yieldfrom',
 'send', 'throw']
```

То есть генераторное выражение является итератором и может использоваться в любом контексте, где требуется итератор. С помощью функции `next` можно получить следующее значение из итератора, а с помощью `list` сделать список:

```
>>> next(g)
1
>>> next(g)
4
>>> next(g)
9
>>> list(g)
[16, 25, 36, 49, 64, 81]
```

В генераторном выражении мы формировали квадраты чисел от 1 до 9. Получив первые три значения и передав генератор в конструктор списка, мы получаем оставшиеся значения. То есть по сути генератор «одноразовый» — выдав один раз значения, генератор больше их не выдает и переходит к следующим.

Генераторное выражение можно напрямую передавать в функцию, которая требует какого-либо итерируемого объекта. Таким образом, можно получить минимальное значение, кратное трем, из списка:

```
>>> a = [6, 5, 2, 8, 9, 1, 11, 7, 4]
>>> min(x for x in a if x % 3 == 0)
6
```

Сортировать все четные значения:

```
>>> sorted(x for x in a if x % 2 == 0)
[2, 4, 6, 8]
```

Определить, больше ли числа 10:

```
>>> any(x > 10 for x in a)
True
```

Определить, все ли числа меньше 15:

```
>>> all(x < 15 for x in a)
True
```

Получить первое значение больше 10:

```
>>> next(x for x in a if x > 10)
11
```

Получить первое значение больше 15. Если таких значений нет, генератор выдаст специальное исключение `StopIteration`:

```
>>> next(x for x in a if x > 15)
Traceback (most recent call last):
  ...
StopIteration
```

В более сложных случаях используют генераторную функцию.

Генераторная функция — это функция, в которой присутствует оператор `yield`. При вызове такой функции создается объект-генератор:

```
>>> def f():
...     yield 0
...
>>> f()
<generator object f at 0x7f3bf0b7c7b0>
```

Оператор `yield` позволяет функции отдать значение и приостановиться, пока не потребуется следующее значение.

Созданный объект-генератор ведет себя точно так же, как и генераторное выражение. Если у нас есть функция, которая выдает несколько раз одно и то же значение (а оператор `yield` это и делает), то с помощью функции `next` мы можем получить эти значения. Если мы вызвали `next`, а тело функции закончилось, мы получаем исключение `StopIteration`:

```
>>> def f():
...     yield 1
...     yield 2
...     yield 3
...
>>> g = f()
>>> next(g)
1
>>> next(g)
```

```
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

В таком виде генераторы используются редко. Чаще генераторная функция используется в контексте, когда мы ожидаем итератор. Например, можно сделать список и пройти по элементам списка:

```
>>> list(f())
[1, 2, 3]
>>> for x in f():
...     print(x * 2)
...
2
4
6
```

Обычно генераторы используются, когда необходимо создать функцию, выдающую некую серию значений. Если функция должна вернуть серию значений, мы можем:

- создать в функции список и вернуть его;
- написать класс-итератор;
- сделать функцию-генератор.

Создание функции-генератора имеет некоторые преимущества перед другими подходами:

- меньше кода;
- не потребляет лишние ресурсы на создание списка;
- более универсальная (возможно, нам нужен не список, а множество или просто итератор);
- может быть бесконечной.

Рассмотрим пример генератора чисел Фибоначчи. Такой генератор — бесконечный. Для его остановки вводится условие вовне функции:

```
def fib():
    a, b = 0, 1
    while True:
        yield b
        a, b = b, a + b


for x in fib():
```

```
    if x > 1_000_000:
        break
    print(x)
```

Попытка создать список из бесконечного генератора может очень быстро «убить» систему:

```
>>> # Не надо этого делать!
>>> list(fib())
```
🤯

В примере функция `list` не остановится до тех пор, пока не получит исключение `StopIteration`. А этого не произойдет, поскольку генератор бесконечный.

Наиболее интересный способ использования генераторов — конвейер генераторов. При объединении генераторов в «конвейер» мы получаем фактически однопроходную программу, но при этом имеем возможность на уровне кода разделить разные части логики.

Рассмотрим простую задачу из ЕГЭ по информатике (обычно эту задачу решают не так). В файле содержится последовательность целых чисел. Рассматривается множество элементов последовательности, которые удовлетворяют следующим условиям:

- сумма цифр числа кратна 5;
- троичная запись числа не заканчивается на 00.

Найдите количество таких чисел и максимальное из них.

Возможное решение с использованием генераторов:

```
numbers = map(int, open(filename))
filtered_numbers = (x for x in numbers if sum(map(int, str(x))) %
5 == 0)
result = [x for x in filtered_numbers if x % 9 != 0]
print(len(result), max(result))
```

Само чтение данных из файла мы осуществляем с помощью функции `map`, которая тоже является генератором: функция выдает значения по одному, читая их из файла. Далее объект `numbers` мы передаем в генератор, который проходит по всем цифрам, находит их сумму и определяет, кратна ли она 5. Далее мы создаем список `result`.

Важно: пока мы не дошли до третьей строки программы, мы ничего не читали из файла, мы лишь создавали генераторы. Генераторы запускаются тогда, когда создается список `result`.

Выполнение программы выглядит следующим образом. Из главного модуля `main` мы создаем список `result`, конструктор списка обращается к генератору `filtered_numbers` для выдачи следующего значения. Генератор `filtered_numbers` обращается к генератору `numbers`, который выдает все значения.

То есть мы получаем программу, которая работает за один проход: один раз читает данные и проходит по ним. Но в коде это разделено, как будто мы делаем несколько разных операций. Такой подход позволяет сделать программу более эффективной и разделить логику программы на отдельные операции.

Фактически в примере было сделано два прохода. Сначала мы сделали список, а потом нашли в нем максимум.

3. Генераторы как сопрограммы

Генератор может прерывать работу не только для того, чтобы выдать значение, но и для приостановки и последующего возобновления работы функции. Генератор может выдавать значение после оператора `yield`, приостанавливать работу, а позже ее возобновлять.

Рассмотрим пример использования оператора `yield` для приостановки функции в стандартной библиотеке. Модуль `contextlib` содержит декоратор `contextmanager`, который позволяет из функции-генератора сделать контекстный менеджер. Такая функция будет содержать только одну инструкцию `yield` в тот момент, когда контекстный менеджер должен приостановиться. Код, записанный до оператора `yield`, выполняется и входит в контекстный менеджер. Код, следующий после `yield`, выполняется уже при выходе.

Рассматриваемый пример достаточно условный. В нем мы обрабатываем соединение с базой данных так, чтобы при возникновении исключений откатить выполняемые действия. Если исключений не произошло, все действия записываем в базу данных. В конце закрываем базу данных. Для работы с базой данных в программе используем специальный объект `cur`:

```
from contextlib import contextmanager
import sqlite3

@contextmanager
def get_cursor() -> sqlite3.Cursor:
    con = sqlite3.connect(DB_FILE)
    cur = con.cursor()
```

```
    try:
        yield cur
    except:
        con.rollback()
        raise
    else:
        con.commit()
    finally:
        con.close()
```

Базовой функциональности генераторов уже достаточно, чтобы написать простейшую «асинхронную сопрограмму».

Вспомним сервер, который обрабатывает множество соединений. Попробуем написать этот код асинхронно.

Так эта программа была реализована ранее:

```
def accept_connection(server_socket):
    client_socket, addr = server_socket.accept()
    print(f'Connection from {addr}.')
    return client_socket

def proceed_message(client_socket):
    addr = client_socket.getpeername()
    try:
        data = client_socket.recv(4096)
    except OSError:
        print(f'connection from {addr} caused error:', e)
        return False
    if not data:
        print(f'{addr} left us')
        return False

    print(f'{addr}: {data.decode("utf-8").strip()}')
    client_socket.sendall(data.upper())
    return True
```

При использовании генератора мы можем объединить работу с клиентом в одну функцию. Вначале мы также получаем информацию о клиентском сокете и выдаем ее с помощью оператора `yield`. При этом функция «замирает» и ждет, когда ее снова вызовут.

Далее мы в цикле получаем данные от клиента и обрабатываем их. В конце тела цикла мы также записываем оператор `yield`, но не указываем никаких значений, поскольку выдавать ничего не требуется. Оператор `yield` нужен, чтобы приостановиться перед следующей итерацией цикла, на которую программа пойдет при появлении в сокете данных для чтения:

```
def handle_connection(server_socket):
    client_socket, addr = server_socket.accept()
    print(f'Connection from {addr}.')
    yield client_socket

    while True:
        try:
            data = client_socket.recv(4096)
        except OSError:
            print(f'connection from {addr} caused error:', e)
            return
        if not data:
            print(f'{addr} left us')
            return

        print(f'{addr}: {data.decode("utf-8").strip()}')
        client_socket.sendall(data.upper())
        yield
```

Также следует изменить главный скрипт. Раньше он выглядел так:

```
with socket.create_server(('127.0.0.1', 5000),
                          reuse_port=True) as server:
    monitoring = [server]
    while True:
        ready, *_ = select(monitoring, [], [])
        for sock in ready:
            if sock is server:
                new_client = accept_connection(sock)
                monitoring.append(new_client)
            else:
                if not proceed_message(sock):
                    sock.close()
                    monitoring.remove(sock)
```

С использованием сопрограмм главный скрипт будет выглядеть сложнее. Необходимо добавить в скрипт информацию о выполняемых сопрограммах — словарь `coroutines`.

В коде точно так же, как и ранее с помощью `select` мы выбираем сокеты, готовые на чтение. Если серверный сокет готов на чтение, мы получили новое соединение и вызываем функцию `handle_connection`, которая создает объект-генератор — в данном случае сопрограмму `coro`.

Далее с помощью функции `next` мы вызываем сопрограмму и получаем значение, которое она выдает — это будет клиентский сокет. Его мы добавляем в список объектов для мониторинга, саму сопрограмму мы добавляем в словарь.

Словарь хранит пары сокет-сопрограмма. При получении данных из клиентского сокета мы вызываем сопрограмму.

При получении исключения `StopIteration` мы все очищаем и удаляем:

```
with socket.create_server(('127.0.0.1', 5000),
                          reuse_port=True) as server:
    monitoring = [server]
    coroutines = {}
    while True:
        ready, *_ = select(monitoring, [], [])
        for sock in ready:
            if sock is server:
                coro = handle_connection(sock)
                new_client = next(coro)
                monitoring.append(new_client)
                coroutines[new_client] = coro
            else:
                try:
                    next(coroutines[sock])
                except StopIteration:
                    sock.close()
                    monitoring.remove(sock)
                    coroutines.pop(sock)
```

Цикл, который создают сопрограммы и который управляет их выполнением, принимая решения о том, какой из них следует передать управление, называется «**цикл событий**» (event loop). Цикл событий мы обычно не пишем сами, на практике он уже есть в библиотеке.

Оператор `yield` работает в обе стороны — то есть генератор может не только выдавать данные, но и получать. Такое действие выполняется с помощью присваивания:

```
def echo_gen():
    msg = None
    while True:
        msg = yield msg
```

Примеры использования простого echo-генератора, написанного в программе выше:

```
>>> eg = echo_gen()
>>> next(eg)
>>> eg.send('Hello')
'Hello'
>>> eg.send(0)
0
>>> next(eg)   # = eg.send(None)
>>> eg.close()
>>> next(eg)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Важно инициализировать генератор командой `next`. После этого генератор остановится (оператор `yield`). Далее мы можем посылать в генератор значения командой `send`, которые генератор будет нам возвращать. Метод `close` генератора принудительно завершает его работу.

Остановимся подробнее на методе `close`. Программа ниже содержит генератор, который должен выдать три значения:

```
def gen():
    try:
        yield 1
        yield 2
        yield 3
    except BaseException as e:
        print(type(e))
    finally:
        print('Finally')

g = gen()
print(next(g))
print(next(g))
```

При запуске такой программы генератор не закончит свою работу, так как он должен выдать три значения. В этом случае метод `close` будет вызван автоматически, если программа завершилась до выхода из сопрограммы.

Программа выведет следующее:

```
1
2
GeneratorExit
Finally
```

Метод `close` создает исключение `GeneratorExit`, которое перехватывается программой, блок `finally` также выполняется.

Рассмотрим пример подсчета среднего значения «на ходу»:

```
def mean():
    summa = 0
    count = 0
    while True:
        mean = summa / count if count else None
        x = yield mean
        summa += x
```

```
        count += 1
```

Пример использования такой программы приведен ниже. У нас есть последовательность чисел. Мы считаем отдельно среднее значение для четных и нечетных значений с помощью двух экземпляров генератора:

```
m1 = mean()
next(m1)
m2 = mean()
next(m2)
for x in some_numbers:
    if x % 2 == 0:
        mean_even = m1.send(x)
    else:
        mean_odd = m2.send(x)

print(mean_even)
print(mean_odd)
```

Более сложный пример подобного использования генератора: чтение данных из файла и запись содержимого по нескольким другим файлам «на лету».

Разберем случай использования функции `yield` и `return` вместе. Когда генератор заканчивает свою работу и выдает исключение `StopIteration`, это исключение несет внутри себя то значение, которое возвращает функция:

```
def f():
    yield 1
    return 2
>>> g = f()
>>> next(g)
1
>>> next(g)
Traceback (most recent call last):
  ...
StopIteration: 2
```

Используя это, можно упростить программу для подсчета среднего значения. Если генератор принимает значение `None`, это сигнал к остановке. Также в программу добавлена логика работы с исключением `StopIteration`. Основной цикл программы останется таким же, а в конце мы выведем полученные средние значения:

```
def mean():
    summa = 0
    count = 0
    while True:
        x = yield
        if x is None:
```

```
            return summa / count
        summa += x
        count += 1

def get_value(gen):
    try:
        next(gen)
    except StopIteration as e:
        return e.value

...
print(get_value(m1))
print(get_value(m2))
```

Наиболее удобно использовать совместно оператор `yield from` и `return`.

Оператор `yield from` делегирует работу другому генератору.

```
def gen1():
    yield 1
    yield 2
    yield 3

def gen2():
    yield from gen1()
```

При работе такой программы:

- значения, выдаваемые из `gen1` с помощью `yield`, будут выданы из `gen2`;
- значения, передаваемые в `gen2` с помощью `send`, будут переданы в `gen1`;
- исключения, переданные в `gen2` с помощью `throw`, будут переданы в `gen1`.

В программе `gen2` — посредник, который передает значения в обе стороны. Если же `gen1` имеет оператор `return`, то функция выдаст именно возвращаемое значение:

```
def gen1():
    yield 1
    yield 2
    return 3


def gen2():
    res = yield from gen1()
    assert res == 3
    ...
```

Пример применения подобного кода — программа для диалога с пользователем. Для уточняющих вопросов можно использовать второй генератор.

Еще один пример использования оператора `yield from` — рекурсивный генератор.

В примере рассмотрен генератор, создающий все перестановки букв строки. Мы проверяем: если дошли до конца строки, то возвращаем получаемый результат. В примере это остановка работы функции.

Далее в цикле мы перебираем все буквы в слове. Если буква еще не встречалась в перестановке, добавляем букву к перестановке и строим перестановки с помощью рекурсии:

```
def permutations(s, i=0, a=''):
    if i == len(s):
        yield a
        return

    for c in s:
        if c not in a:
            yield from permutations(s, i + 1, a + c)
```

Поскольку программа является генератором, все значения будут выдаваться «на ходу», но мы можем поместить их в список:

```
>>> list(permutations('abc'))
['abc', 'acb', 'bac', 'bca', 'cab', 'cba']
```

Все описанные особенности делают генератор полноценной сопрограммой:

- может приостанавливать выполнение;
- может выдавать промежуточные значения;
- может получать значения извне;
- может получать извне информацию об исключениях;
- может вызвать другую сопрограмму неблокирующим способом;
- есть отдельный механизм возврата окончательного результата;
- есть возможность корректного принудительного завершения.

4. Синтаксис `async/await` и `awaitable`-объекты

Некоторое время асинхронное программирование в Python задействовало генераторы в качестве сопрограмм (PEP 342, описывающий двусторонний интерфейс yield, был принят еще в версии 2.5). В версии 3.5 был добавлен специальный синтаксис для сопрограмм.

Сопрограмма определяется с помощью ключевых слов `async def`, и этим существенно отличается от обычной программы визуально. Для того чтобы вызвать другую сопрограмму, используется ключевое слово `await`:

```
async def read_data(db):
    data = await db.fetch('SELECT ...')
```

```
    ...
```

Ключевое слово `await` почти эквивалентно `yield from` со следующими отличиями:

- после `await` должен следовать awaitable-объект — объект, реализующий метод `__await__`, возвращающий итератор;
- после `yield from` должен следовать непосредственно итератор;
- оператор `await` можно использовать только в функции, помеченной словом `async`;
- функция, помеченная `async`, возвращает сопрограмму (coroutine), которая является awaitable-объектом.

Частая ошибка — забыть написать `await`, например:

```
async def read_data(db):
    data = db.fetch('SELECT ...')
    print(type(data))
    ...
```

Программа сработает следующим образом: в переменную `data` будут записаны не данные, а объект `coroutine`. Поскольку это достаточно частая ошибка, Python выдаст соответствующее предупреждение:

```
<class 'coroutine'>
RuntimeWarning: coroutine 'db.fetch' was never awaited
```

Вместе с синтаксисом `async/await` в Python появились некоторые специальные методы с буквой «a» в начале. Например, асинхронный итератор реализует методы `__aiter__` и `__anext__`.

Рассмотрим пример асинхронного итератора из документации Python — класс `AsyncIterable`, реализующий асинхронный итератор:

```
class AsyncIterable:
    def __aiter__(self):
        return self

    async def __anext__(self):
        data = await self.fetch_data()
        if data:
            return data
        else:
            raise StopAsyncIteration

    async def fetch_data(self):
        ...
```

Для использования асинхронного итератора применяется цикл `for` с ключевым словом `async`:

```
async for row in Cursor():
    print(row)
```

Асинхронный итератор можно создать с помощью асинхронной функции-генератора. То есть если в асинхронной функции есть слово `yield`, эта функция становится асинхронным итератором:

```
async def get_links(url):
    async with httpx.AsyncClient() as client:
        r = await client.get(url)
        for tag in parse_html(r.text):
            if tag.name == 'a':
                yield tag.get('href')

async def main():
    ...
    async for link in get_links(url):
        ...
```

Асинхронные контекстные менеджеры реализуют методы `__aenter__` и `__aexit__`, которые используются в блоке `with` с ключевым словом `async`:

```
class AsyncContextManager:
    async def __aenter__(self):
        await log('entering context')

    async def __aexit__(self, exc_type, exc, tb):
        await log('exiting context')
```

5. Модуль `asyncio`: основные функции

Сопрограммы нельзя вызвать непосредственно, с ними должен работать цикл событий. Внедрение асинхронного кода в уже существующий проект, который работает на синхронном коде — сложная задача. Если мы пишем асинхронную программу, то эта программа должна быть асинхронной от начала и до конца.

Можно написать цикл событий самостоятельно, используя методы `send`, `close`, `throw`, а также перехватывая `StopIteration`. Но обычно это делается один раз при разработке библиотеки. При написании прикладного кода используется уже готовая библиотека. Стандартная библиотека Python содержит модуль `asyncio`, реализующий цикл событий и ряд полезных инструментов асинхронного программирования. Существуют и другие асинхронные библиотеки, но мы их рассматривать не будем.

На практике работа выглядит следующим образом — функция `run` запускает сопрограмму. В теории это несколько сложнее — `run` запускает цикл событий, который запускает сопрограмму:

```
import asyncio

async def main():
    print('hello')
    await asyncio.sleep(1)
    print('world')

asyncio.run(main())
```

Для асинхронной программы всегда нужна главная функция, зачастую это функция `main`. Она вызывает другие сопрограммы. В примере функция `main` запускает `asyncio.sleep`, которая приостанавливает работу. Важное отличие `asyncio.sleep` от `time.sleep` — сопрограмма возвращает управление циклу событий, то есть дает возможность поработать другим сопрограммам.

Использование функции `run` является обязательным, вызов функции `main` только создает сопрограмму:

```
>>> main()
<coroutine object main at 0x...>
```

Функция `create_task` позволяет запускать несколько задач одновременно.

В примере ниже рассмотрена асинхронная функция, которая выдает текст с некоторыми задержками. Мы создаем две задачи, и обязательно для каждой сопрограммы нам нужно прописать `await`. То есть требуется отделить создание сопрограммы и ожидание ее выполнения. Если `await` не прописать, сопрограммы не запустятся. Мы должны явно передать управление циклу событий.

```
import asyncio

async def say(text, times):
    for _ in range(times):
        print(text)
        await asyncio.sleep(0.1)


async def main():
    task1 = asyncio.create_task(say('Hello', 5))
    task2 = asyncio.create_task(say('Good bye', 5))
    await task1
    await task2

asyncio.run(main())
```

Рассмотрим различные варианты запуска нескольких сопрограмм. Первый способ — с помощью двух инструкций `await`. Такой способ ничем не отличается от синхронного выполнения — сопрограммы будут выполняться по очереди. Сначала одна, а когда она закончится — другая:

```
await say('Hello', 5)
await say('Good bye', 5)

task1 = asyncio.create_task(say('Hello', 5))
task2 = asyncio.create_task(say('Good bye', 5))
for cor in asyncio.as_completed([task1, taks2]):
    result = await cor
```

Если запускаемые сопрограммы содержат возможность вернуть управление циклу событий, то внутри них могут быть запущены другие сопрограммы.

Модуль `asyncio` содержит функцию `as_completed`, которая похожа на одноименную функцию из модуля `concurrent.futures`. Функция `asyncio.as_completed` получает список задач и по мере их завершения выдает сами сопрограммы. Мы должны обязательно прописать `await`, чтобы получить результат. Синтаксически это выглядит так, как будто мы чего-то ждем в программе.

Также есть функция `asyncio.gather`, в которую мы передаем сопрограммы, а она возвращает список результатов. Нужно аккуратно работать с этой функций. Дело в том, что если одна из сопрограмм сгенерирует исключение, то даже, если мы перехватим исключение, результат не получим. У функции `gather` есть дополнительный параметр, который позволяет вернуть исключение в списке получаемых результатов. В целом обработка исключений в асинхронном коде будет сложнее.

```
results = await asyncio.gather(
    say('Hello', 5),
    say('Good bye', 5)
)
```

В Python 3.11 появилась новая возможность создавать группы задач с помощью асинхронного контекстного менеджера. Задачи создаются с помощью метода `create_task`, но который относится к объекту `TaskGroup`. При выходе из контекстного менеджера созданные задачи будут запущены. А если точнее, получен их результат:

```
# 3.11
async with asyncio.TaskGroup() as tg:
    tg.create_task(say('Hello', 5))
    tg.create_task(say('Good bye', 5))
```

Также имеется возможность использовать таймауты. В программе ниже приведена функция `timeout`, которая является контекстным менеджером. Внутри контекстного менеджера записан код, который будет прерван, если произойдет таймаут (то есть, если истечет время и будет сгенерировано исключение TimeoutError из модуля asyncio). Важно, что это исключение должно произойти:

```
try:
    async with asyncio.timeout(10):
        await long_running_task()
except asyncio.TimeoutError:
    print("The long operation timed out, but we've handled it.")
```

Есть более удобная функция `wait_for`, которая позволяет запустить сопрограмму с некоторым таймаутом. При возникновении таймаута функция выдает исключение `TimeoutError`. Этот способ позволяет запустить с таймаутом одну сопрограмму:

```
try:
    await asyncio.wait_for(eternity(), timeout=1.0)
except asyncio.TimeoutError:
    print('timeout!')
```

Модуль `asyncio` также содержит все необходимые средства синхронизации кода:

- `asyncio.Lock`
- `asyncio.Semaphore`
- `asyncio.Event`
- `asyncio.Condition`
- `asyncio.Barrier`
- `asyncio.Queue`
- `asyncio.sleep(0)`

В отличие от многопоточного программирования указанные инструменты не столь востребованы. Причина — асинхронный код позволяет сделать остановки в тех местах, где это необходимо. Таким образом, исключается состояние гонки.

Стоит обратить внимание на вызов функции `asyncio.sleep(0)`. Такой вызов не дает никакого ожидания. Это скорее возможность переключиться на другие сопрограммы, готовые поработать. Если таких сопрограмм нет, то код продолжит работать дальше.

Запуск в терминале `python -m asyncio` открывает интерактивную консоль, в которой можно использовать асинхронные функции. Модуль `asyncio` импортируется автоматически — цикл событий уже запущен, и мы можем в нем писать `await` и запускать сопрограммы:

```
$ python -m asyncio
asyncio REPL 3.11.0 (main, Nov  6 2022, 18:11:16) [GCC 9.4.0] on
linux
Use "await" directly instead of "asyncio.run()".
Type "help", "copyright", "credits" or "license" for more
information.
>>> import asyncio
>>> # ^ эту строчку я не писал
>>> await asyncio.sleep(1)
>>>
```

6. Модуль `asyncio`: работа с сетью

Модуль asyncio предоставляет классы потоков `StreamReader` и `StreamWriter`, реализующие асинхронные операции чтения-записи данных. Документация не рекомендует работать с этими классами напрямую, а использовать функции `start_server` и `open_connection`. Рассмотрим на примерах, как это работает.

В примере представлен код программы echo-сервера:

```
import asyncio

async def handle_echo(reader, writer):
    data = await reader.read(100)
    message = data.decode()
    addr = writer.get_extra_info('peername')

    print(f"Received {message!r} from {addr!r}")

    print(f"Send: {message!r}")
    writer.write(data)
    await writer.drain()

    print("Close the connection")
    writer.close()

async def main():
    server = await asyncio.start_server(
        handle_echo, '127.0.0.1', 8888)

    addrs = ', '.join(str(sock.getsockname()) for sock in
server.sockets)
    print(f'Serving on {addrs}')

    async with server:
        await server.serve_forever()

asyncio.run(main())
```

Начнем рассмотрение с функции `main`. Внутри функции мы создаем сервер с помощью асинхронной функции `start_server`. Первым аргументом `start_server` принимает функцию, которая будет вызвана для обработки каждого входящего соединения. Саму функцию мы не вызываем, а лишь записываем ее название, поскольку серверу нужно будет создать много сопрограмм для обработки каждого соединения. То есть сервер должен иметь возможность создавать эти сопрограммы самостоятельно.

Функция `handle_echo` принимает на вход объекты классов `StreamReader` и `StreamWriter`. В следующих строках функции `main` мы смотрим, на каком адресе работает сервер. После этого используем сервер в асинхронном контекстном менеджере, вызывая метод `serve_forever` — вход в бесконечный цикл обработки входящих соединений.

Далее рассмотрим функцию `handle_echo`, обрабатывающую соединение. Мы читаем данные из объекта `reader`, используя `await`. Если данных нет, управление передается циклу событий. Затем получаем сообщение, раскодируем его, печатаем на экран. Метод `writer.write` записывает данные в буфер, но не отправляет их. Это удобно, когда требуется собрать какие-то данные по кусочкам. После вызывается асинхронный метод `writer.drain`.

Ниже рассмотрен пример программы для клиента. Программа включает асинхронную функцию `tcp_echo_client`, в которой для подключения к серверу, мы вызываем асинхронную функцию `open_connection`, возвращающую два объекта — `reader`, `writer`. С помощью этих объектов мы можем читать и писать данные в сокет. После этого происходит запись данных, вызов асинхронной функции `drain`, чтение данных и закрытие соединения.

Функция `tcp_echo_client` вызывается с помощью метода `run`:

```
import asyncio

async def tcp_echo_client(message):
    reader, writer = await asyncio.open_connection(
        '127.0.0.1', 8888)

    print(f'Send: {message!r}')
    writer.write(message.encode())
    await writer.drain()

    data = await reader.read(100)
    print(f'Received: {data.decode()!r}')
```

```
    print('Close the connection')
    writer.close()

asyncio.run(tcp_echo_client('Hello World!'))
```

Еще один пример — программа для проверки доступности доменов с помощью `asyncio`. В этой программе интересна обработка исключений. В модуле `asyncio` есть свои исключения. Например, `CancelledError`, которые иногда требуется перехватывать и обрабатывать.

Заранее создав все сопрограммы, мы запускаем их с помощью `as_completed` и смотрим время выполнения:

```
import asyncio
import time

async def check_free(domain):
    try:
        await asyncio.open_connection(
            'python' + domain, 80)
    except (OSError, asyncio.CancelledError):
        return domain, True
    else:
        return domain, False

async def main():
    domains = [line.strip() for line in open('domains.csv')]
    start = time.perf_counter()
    tasks = [check_free(d) for d in domains]
    for cor in asyncio.as_completed(tasks):
        domain, result = await cor
        free = 'free' if result else 'not free'
        print(f'python{domain} is {free}. '
              f'{time.perf_counter() - start:0.2f}')
```

Для данного примера разница между многопоточным и асинхронным программированием будет несущественна, поскольку время работы определяется самым медленным соединением и не зависит от используемого подхода программирования.

7. Модуль `asyncio`: работа с потоками и процессами

Все сопрограммы, как и цикл событий, по умолчанию работают в одном потоке. Однако модуль asyncio позволяет запускать отдельные процессы и потоки.

Запуск внешней программы выполняется с помощью методов:

- `create_subprocess_shell`, который запускает программу в командной оболочке (например, различные утилиты с передачей им аргументов командной строки);
- `create_subprocess_exec`, который запускает программу без терминала.

В примере выполняется запуск программы архивации некоего большого файла. Пример интересен тем, что мы не просто запускаем программу, но мы отслеживаем процесс и взаимодействуем с ним. Знак «`-`» после названия утилиты `zip` говорит о том, что утилита будет выводить архивируемые данные в стандартный поток вывода. Такая программа будет работать, только если она запущена не в терминале.

Для параметра `stdout` задается значение `asyncio.subprocess.PIPE`, которое означает, что есть несколько процессов, связанных друг с другом как конвейер. Данные, которые выдает один процесс, являются входными данными для другого. Другим процессом будет сама программа, и с помощью `process.stdout.read` мы можем асинхронно читать данные по мере их накопления в выводе:

```
process = await asyncio.create_subprocess_shell(
    "zip - some_big_file",
    stdout=asyncio.subprocess.PIPE
)
while True:
    buf = await process.stdout.read(1024)
    if not buf:
        break
```

В этот же пример можно добавить обработку данных или, например, отправку их по сети на ходу.

Метод `to_thread` запускает функцию в отдельном потоке, возвращает сопрограмму. Функция полезна для запуска неасинхронных функций, которые долго работают (например, чтение данных из файла, которое не поддерживается модулем `asyncio`):

```
import csv

def read_database(filename):
    with open(filename) as csv_file:
        reader = csv.DictReader(csv_file)
        return list(reader)

async def main():
    ...
    data = await asyncio.to_thread(read_database, 'db.csv')
    ...
```

Запускаемые потоки «бесшовно» интегрируются в асинхронную программу через `await`.

Модуль `asyncio` также поддерживает интеграцию с исполнителями из модуля `concurrent.futures`. Например, с помощью метода `run_in_executor` мы передаем выполнение отдельной функции исполнителю отдельных потоков `ThreadPoolExecutor`, который сам выполнит распределение по потокам, если это необходимо.

Чтобы получить такую возможность реализации, необходимо получить объект текущего цикла событий, получаемый как `loop = asyncio.get_running_loop()`.

Аналогично работают и процессы:

```
import asyncio
import concurrent.futures

def blocking_io():
    with open('/dev/urandom', 'rb') as f:
        return f.read(100)

def cpu_bound():
    return sum(i * i for i in range(10 ** 7))

async def main():
    loop = asyncio.get_running_loop()

    with concurrent.futures.ThreadPoolExecutor() as executor:
        result = await loop.run_in_executor(
            executor, blocking_io)

    with concurrent.futures.ProcessPoolExecutor() as executor:
        result = await loop.run_in_executor(
            executor, cpu_bound)

if __name__ == '__main__':
    asyncio.run(main())
```

Функция `run_in_executor` позволяет передать в функцию только позиционные аргументы. Чтобы передать именованные аргументы, нужно воспользоваться `functools.partial`. Эта функция создает новую функцию на основе уже имеющейся и передает ей некоторые аргументы:

```
from functools import partial

print_oneline = partial(print, end='')
```

8. Асинхронная загрузка данных с помощью библиотеки `httpx`

Базовый пример использования библиотеки `httpx` в асинхронном коде:

```
>>> async with httpx.AsyncClient() as client:
...     r = await client.get('https://www.example.com/')
...
>>> r
<Response [200 OK]>
```

Мы создаем специальный асинхронный клиент, который работает как контекстный менеджер. У этого клиента будут методы, которые копируют методы-запросы `get`, `post`, `put` и другие. Указанные методы будут асинхронными, так как использовать их мы будем с помощью ключевого слова `await`. Метод `get` в примере вернет обычный объект `Response`.

При загрузке большого объема данных при медленном соединении, можно читать данные по мере их поступления. Для этого у клиента есть метод `stream`, который тоже работает как асинхронный контекстный менеджер и внутри реализует асинхронный итератор:

```
async with httpx.AsyncClient() as client:
    async with client.stream('GET', 'https://www.example.com/') as
response:
        async for chunk in response.aiter_bytes():
            ...
```

Рассмотрим пример программы: необходимо найти «путь» по ссылкам в Википедии от статьи «Python» до статьи «Карл Маркс».

Начнем с загрузки ссылок с помощью API (Application Programming Interface). Википедия имеет API, который позволяет получать данные в удобном для автоматизированной обработки виде.

Важно! Чтобы узнать, как пользоваться API для того или иного сервиса, нужно обратиться к документации этого сервиса.

В программе мы делаем запрос на адрес https://ru.wikipedia.org/w/api.php. В параметрах запроса необходимо указать, что мы хотим сделать, какую страницу хотим получить и в каком формате.

Функция `get_links` получает создаваемого клиента первым аргументом. Это необходимо, чтобы в дальнейшем создать асинхронного клиента и все запросы делать через него. После этого мы получаем ответ и делаем из него словарь с помощью метода `json` (сам формат json по сути представляет упрощенный вариант

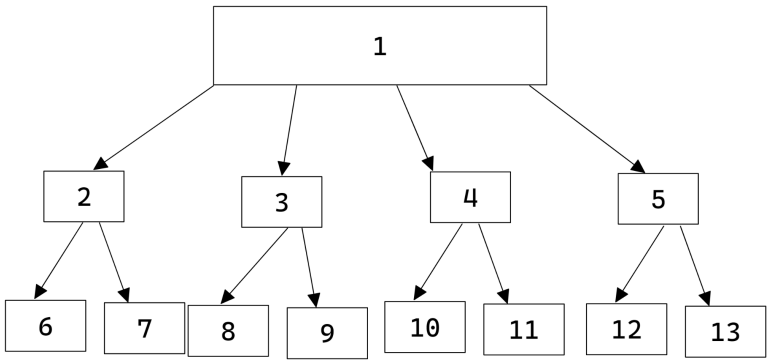
словаря). Если в результате ничего не получилось, мы будем возвращать пустой результат — для данной программы это не критично.

Формат возвращаемого объекта определяется API Википедии. Если ничего не получилось вернуть, программа также вернет пустой кортеж.

```
async def get_links(client: httpx.AsyncClient, page_name: str):
    url = 'https://ru.wikipedia.org/w/api.php'
    page_name = page_name.replace(' ', '_')
    params = {'action': 'parse', 'page': page_name, 'format':
'json'}
    response = await client.get(url, params=params)
    try:
        page = response.json()
    except json.JSONDecodeError:
        return ()

    try:
        return (link['*'] for link in page['parse']['links'])
    except KeyError:
        return ()
```

Для поиска пути воспользуемся алгоритмом «поиска в ширину». Суть алгоритма: мы берем элемент, который имеет ссылки на другие элементы, далее проходим по элементам-ссылкам. У следующих элементов также получаем ссылки и так далее. В алгоритме важен порядок прохождения элементов: мы следуем по уровням как показано на рисунке.



На практике нам будет удобно использовать очередь, в которую мы помещаем ссылки и последовательно их обрабатываем. Основная функция, которая выполняет работу, может выглядеть следующим образом:

```
link_source = {}
queue = asyncio.Queue()
done = asyncio.Event()

source = 'Python'
```

```
target = 'Карл Маркс'

async def worker(client: httpx.AsyncClient):
    while True:
        page_name = await queue.get()
        print('processing', page_name)
        for link in await get_links(client, page_name):
            if link == target:
                done.set()
            if link in link_source:
                continue
            if any(x.isdigit() for x in link):
                continue
            link_source[link] = page_name
            queue.put_nowait(link)
```

Поскольку в программе мы не работаем с потоками, можно использовать глобальные переменные. В программе мы создаем словарь, в который будем записывать источник для каждой ссылки. Также мы создаем очередь и объект класса `Event`, который нужен, чтобы сигнализировать о том, что путь найден и задача решена.

Асинхронная подпрограмма `worker` в бесконечном цикле получает название страницы из очереди, выводит некую информацию на экран (необходимо, чтобы видеть, что программа работает), получает все ссылки со страницы. Если текущая ссылка та, что мы ищем, устанавливает флаг `done`. Если текущая ссылка уже была, мы ее пропускаем.

В программе использован дополнительный код, связанный с обработкой статей Википедии, привязанных к датам. Через такие статьи можно получить путь от одной страницы к другой, но нам неинтересно это, поэтому программа игнорирует статьи, содержащие цифры в названии.

После обработки источник ссылки записывается в очередь. Очередь имеет асинхронный метод `get`, который ждет, когда значение в очереди появится. Для помещения ссылки в очередь используем в программе метод `put_nowait`, который помещает значение в очередь сразу.

Далее после завершения процесса нужно показать найденный путь, для этого мы проходим по словарю:

```
def print_path():
    node = target
    path = [node]
    while node != source:
```

```
        node = link_source[node]
        path.append(node)
    print(*reversed(path), sep=' -> ')
```

В главной функции мы создаем клиента, который будет выполнять загрузку данных. Также мы создаем 20 задач `worker` (число 20 выбрано условно, можно поэкспериментировать и изменить значение как в большую, так и в меньшую сторону). Затем в очередь помещаем первую ссылку и ждем, когда программа приостановит работу (флаг `done` установлен). Далее останавливаем всех `worker`, дожидаемся ее выполнения, получаем и выводим результат на экран:

```
async def main():
    async with httpx.AsyncClient() as client:
        workers = [
            asyncio.create_task(worker(client))
            for _ in range(20)
        ]
        queue.put_nowait(source)
        await done.wait()

        for w in workers:
            w.cancel()
        await asyncio.gather(*workers, return_exceptions=True)

    print_path()
```

После работы программы мы получили результат:

```
Python -> Apple -> Антиутопия -> Карл Маркс
```

Программа работала около 1 минуты, было обработано около 10 000 ссылок.

Дополнительные материалы для самостоятельного изучения

1. The yield statement
2. asyncio — Asynchronous I/O