Асинхронность в Python: генераторы.

Наш рассказ об асинхронности в Python был бы неполным, если бы мы не посмотрели на то, как исторически развивалось это направление. Так что приготовьтесь.

И начнём мы с того, что повторим, что такое генераторы.
Генераторы — это функции, внутри которых используется ключевое слово yield. Как только интерпретатор видит это слово внутри функции, он компилирует её в байткод не как обычную функцию, а как генератор. В результате мы получаем объект, который автоматически поддерживает протокол итератора. Посмотрим на примере:

```
In[4]: from typing import Iterator
  ...:
  ...:
  ...: def simple_gen() -> Iterator[str]:
  ...:     yield 'something'
  ...:
In[5]: type(simple_gen)
Out[5]: function
In[6]: g = simple_gen()
In[7]: type(g)
Out[7]: generator
```

Определим простейший генератор, который возвращает строку. Кстати, простые генераторы, которые что-то возвращают, — аннотируются так. Обратите внимание, что сам объект simple_gen — всё ещё функция, но вот то, что она возвращает, является генератором. В этом легко убедиться, проверив тип объекта, который нам вернули.

Как мы знаем, объект является итератором, когда он реализует два дандер- (или по другому «магических») метода: iter и next. Мы легко можем убедиться, что эти методы у объекта есть — давайте проверим.

```
In[15]: g = simple_gen()
In[16]: g.__iter__()
Out[16]: <generator object simple_gen at 0x1047dccf0>
In[17]: g.__next__()
Out[17]: 'something'
In[18]: g.__next__()
Traceback (most recent call last):
  File "/Users/andrei/PycharmProjects/python_advanced/venv/lib/python3.8/site-packages
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-18-42e506b10868>", line 1, in <module>
    g.__next__()
StopIteration
```

Метод iter должен вернуть объект итератора, а метод next — следующее значение. Ещё этот метод можно вызвать через функцию next. При исчерпании итератор должен выкинуть ошибку StopIteration — так мы поймём, что больше в итераторе ничего нет. Так, кстати, работает for loop с нашими коллекциями, например, со списками. Под капотом создаётся итератор, у него вызывается метод next до тех пор, пока не выскочит ошибка.

Хорошо, это технические детали. Но зачем они нужны? Генераторы позволяют нам производить так называемые «ленивые вычисления».
Посмотрим на пример:

```
In[24]: def heavy_computation() -> Iterator[int]:
   ...:     for i in range(100000000000000000000000000000000000000000):
   ...:         yield i ** 2
   ...:
In[25]: g = heavy_computation()
In[26]: for i in range(10):
   ...:     print(next(g))
   ...:
0
1
4
9
16
25
36
49
64
81
```

У нас есть функция, которая делает некоторые потенциально долгие вычисления. И мы не знаем, потребуются ли нам все они или только часть.

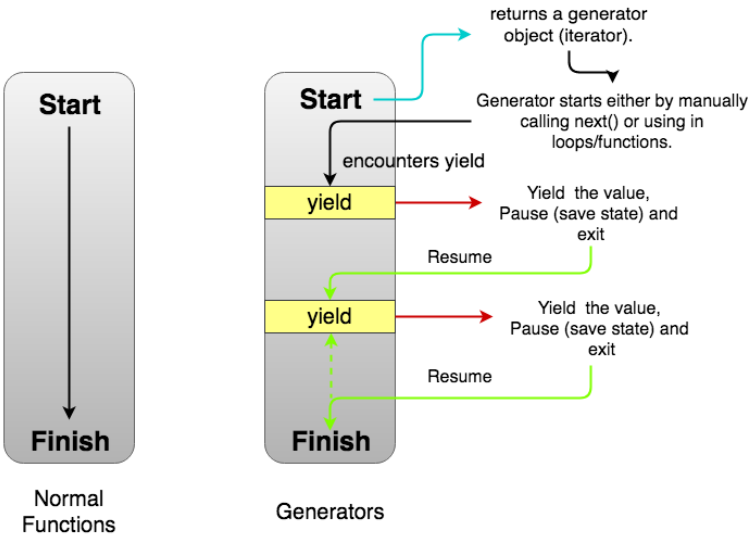
Скажем, первому юзеру хочется узнать степени двойки до 10, а следующему, например, работнику Google, — степени двойки до 64. Используя генераторы, мы можем эффективно, с точки зрения памяти и вычислений, организовать это — производить вычисление только в тот момент, когда это действительно необходимо. Мы не храним эти огромные последовательности в памяти и не высчитываем всё это заранее.

Кроме того, генераторы между вызовами метода next() сохраняют своё состояние. Например, в цикле мы были на индексе 2. Сохранили это состояние, ждём вызова next и продолжаем работу дальше уже на индексе 3. Как мы потом узнаем, это свойство, в действительности, ключевое у генераторов в контексте асинхронного программирования. Ещё раз: давайте не путать функции «генератор» и «объект генератора». Функция — это то, что мы вызываем, чтобы получить объект генератора. Объект генератора — это то, у чего мы можем вызвать метод next.

С повторением на этом всё, пока вроде бы ничего принципиально нового. Движемся дальше.

Как мы заметили, жизнь генератора более насыщена разными событиями, нежели жизнь простой функции. Его создают, запускают и исчерпывают. И если вдруг нам дадут некий генератор в пользование (не функцию, а именно объект) — неплохо бы знать, в каком он состоянии. Каждый генератор в своей жизни переживает четыре этапа (почти как человек):

- **GEN_CREATED** — мы только что создали объект генератора, вызвав функцию генератора.
- **GEN_RUNNING** — мы вызвали метод next, и генератор в данный момент исполняется интерпретатором.
- **GEN_SUSPENDED** — генератор вернул что-то после yield и уснул.
- **GEN_CLOSED** — генератор закончил выполнение (исчерпал себя) и уснул.



Чтобы узнать состояние генератора, нужно воспользоваться специальным методом из библиотеки inspect. Эта библиотека позволяет, помимо работы с генераторами, узнать много интересного и полезного об объектах в Python. За подробностями отсылаю вас в документацию. Вы найдёте ссылку под видео.

```
In[27]: def gen_states() -> Iterator[int]:
   ...:     for i in range(3):
   ...:         yield i
   ...:
In[28]: import inspect
In[29]: g = gen_states()
In[30]: inspect.getgeneratorstate(g)
Out[30]: 'GEN_CREATED'
In[31]: next(g)
Out[31]: 0
In[32]: inspect.getgeneratorstate(g)
Out[32]: 'GEN_SUSPENDED'
In[33]: next(g)
Out[33]: 1
In[34]: next(g)
Out[34]: 2
In[35]: next(g)
Traceback (most recent call last):
  File "/Users/andrei/PycharmProjects/python_advanced/venv
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-35-e734f8aca5ac>", line 1, in <modu
    next(g)
StopIteration
In[36]: inspect.getgeneratorstate(g)
Out[36]: 'GEN_CLOSED'
```

Итак, напишем простой генератор, который возвращает три числа. Импортируем модуль инспект. Далее давайте создадим генератор, у него ожидаем состояние created. Вызываем next — состояние suspend. Поймать генератор в состоянии running — нетривиальная задача; нужно будет писать код с тредами, так что пропустим это. И, наконец, исчерпаем генератор — генератор в состоянии close.

Но как мы только что сказали, генератор может быть закрыт — что это значит? У объектов генератора есть три специальных метода: send, throw и close.

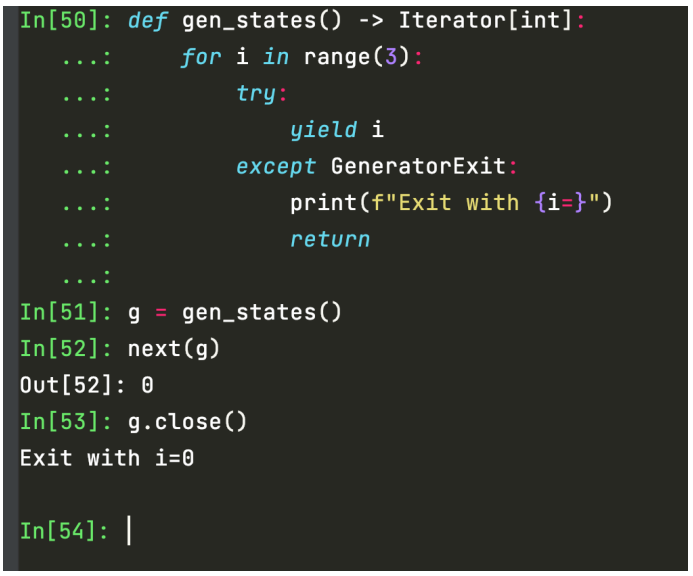
```
dir(g)
Out[37]:
...
 'close',
...
 'send',
 'throw']
```

Метод send мы рассмотрим чуть позже, метод throw не будем разбирать, так как он неважен для нашей темы. Ссылку смотрите под видео. И, наконец, метод close — скажем пару слов о нём:

```
In[39]: next(g)
Out[39]: 0
In[40]: inspect.getgeneratorstate(g)
Out[40]: 'GEN_SUSPENDED'
In[41]: g.close()
In[42]: inspect.getgeneratorstate(g)
Out[42]: 'GEN_CLOSED'
In[43]: next(g)
Traceback (most recent call last):
  File "/Users/andrei/PycharmProjects/python_advanced/venv/lib/pyt
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "<ipython-input-43-e734f8aca5ac>", line 1, in <module>
    next(g)
StopIteration
```

Инициализируем генератор повторно и переведём его в состояние suspend. Как только мы вызовем у генератора метод close, мы как бы переступим через пару ступенек и сразу окажемся в конце итератора. Как видим, у генератора соответствующее состояние, и при попытке получить следующее значение выкидывается ошибка. Зачем нужен этот метод?

Метод позволяет генератору отловить специальную ошибку — GeneratorExit. И, отловив её, сделать необходимые действия по, например, освобождению ресурсов.
Посмотрим на примере:



Создадим простой генератор и внутри него обернём yield statement в try-except, где будем ловить ошибку generatorexit. Как видим, так мы можем сделать некоторые действия по освобождению ресурсов перед тем, как генератор будет уничтожен сборщиком мусора.

В целом в методе close мало интересного, кроме того, что он позволяет управлять состоянием генератора — это нужно запомнить.

Теперь давайте рассмотрим метод send.

Оказывается, генератор может не только вернуть что-то, но и это что-то принять в себя! То есть мы можем отправить объект в генератор, который проинициализирован и уже во всю работает.

Синтаксис такой:

```
something = yield
```

То есть, вместо того, чтобы сделать yield чего-то из генератора, мы присваиваем значение выражения в переменную, которую используем дальше в коде.

Давайте посмотрим, как это можно сделать:

```
def gen_with_send() -> Iterator[int]:
    while True:
        item = yield
        print(f"{item=}")

g = gen_with_send()
next(g)
g.send(1)
g.send(2)
g.send(42)
```

```
item=1
item=2
item=42
```

Напишем незамысловатый генератор, который в бесконечном цикле принимает значения и печатает их. Обратите внимание, что сразу после создания генератора генератор-функцией, мы сделали вызов функции next и передали туда наш генератор. Зачем? Только что созданный генератор ещё не начал своего выполнения, можно представить, что он находится на нулевой строчке своих инструкций в состоянии Created. Вызвав метод next, мы переместили его к первому yield. Ну а дальше осталось только отправить в него значение и, вуаля, — оно внутри генератора, который, прервал своё выполнение и ждал, пока в него что-то пришлют. Чувствуете намёки на асинхронность?

Кстати, ещё один способ активировать генератор, то есть выполнить его до первого yield, — это вызвать у него метод send и передать туда None.

```
15
def gen_with_send() -> Iterator[int]:
    while True:
        item = yield
        print(f"{item=}")

g = gen_with_send()
g.send(None)
g.send(1)
g.send(2)
g.send(42)
```

```
item=1
item=2
item=42
```

В коде выше не очень понятно, где генератор засыпает. Посмотрим на следующий пример:

```
33 def gen_with_send():
       item = yield
       print(f"{item=}")

   g = gen_with_send()
   g.send(None)
   g.send(1)
```

```
item=1

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

StopIteration                             Traceback (most recent

/var/folders/4v/p3ssq3k14cq3s30gh_8261qw0000gn/T/ipykernel_44435
      5 g = gen_with_send()
      6 g.send(None)
----> 7 g.send(1)
      8

StopIteration:
```

Мы инициализируем генератор и попадаем в место, где находится yield. Далее метод send передаёт что-то в генератор и по сути продолжает его выполнение, как бы вызывав next. Так как другой yield не был найден, внутри выкидывается ошибка StopIteration.

Окей, ведь просто так отправлять что-то в генератор не очень интересно, куда круче туда что-то послать и что-то оттуда получить. Как это сделать?
Очень просто!

Напишем генератор, который что-то возвращает и что-то принимает.

```
42 def gen_with_send() -> Iterator[int]:
       item = 0
       while True:
           outer_item = yield item
           print(f"{item=}")
           print(f"{outer_item=}")
           item += 1

   g = gen_with_send()
   g.send(None)
   g.send(42)
   g.send(41)
```

```
item=0
outer_item=42
item=1
outer_item=41
```

По синтаксису сразу станет понятно, что в прошлый раз yield неявно возвращал вместо вызова next значение None.
Таким образом, мы решаем проблему двух yield-мест в функции и можем нормально пользоваться send-методом.

И если вы думаете, что это все возможности yield, то как бы не так. Yield from — синтаксис, созданный для упрощения работы с генераторами внутри генераторов.

Чтобы вместо

```
48 def sub_gen():
       for i in range(5):
           yield i

   def main_gen():
       gen = sub_gen()
       for i in gen:
           yield i

   gen = main_gen()
   for i in gen:
       print(i)
```

```
0
1
2
3
4
```

,

где мы в цикле делаем yield-значения, мы просто передавали туда генератор или любой другой итерируемый объект (не забываем, что генератор именно таков).
В результате получаем что-то такое:

```
51 def sub_gen():
       for i in range(5):
           yield i

   def main_gen():
       gen = sub_gen()
       yield from gen

   gen = main_gen()
   for i in gen:
       print(i)

   0
   1
   2
   3
   4
```

И тут может показаться, что в Python добавили новый синтаксис только ради того, чтобы не писать лишний for loop. Действительно, это было бы очень странно.

```
52 def sub_gen():
       val = yield
       print(f"{val=}")


   def main_gen():
       gen = sub_gen()
       yield from gen

   gen = main_gen()
   gen.send(None)
   gen.send('From main with love')

   val='From main with love'

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

   StopIteration                             Traceback (most recent

   /var/folders/4v/p3ssq3k14cq3s30gh_8261qw0000gn/T/ipykernel_44435
        10 gen = main_gen()
        11 gen.send(None)
   ---> 12 gen.send('From main with love')
        13

   StopIteration:
```

Этот синтаксис упрощает коммуникацию между генераторами в обе стороны: так мы можем не только получать значения из-под генераторов, но и

отправлять их туда без лишних телодвижений. Как видно на примере, мы не вызываем лишний раз send-метод — всё это за нас делает Python.

Окей! Теперь мы знаем практически всё, что нужно знать о генераторах. Это необходимо для того, чтобы разобраться в таком понятии, как корутина. Ведь именно на основе генераторов реализуются корутины. И именно за счёт корутин работает асинхронность в Python. Что это такое, мы узнаем в следующем уроке. А пока выполните небольшое практическое задание.