Что такое генераторы в программировании

Они лениво раз за разом вычисляют новые значения, но не помнят, что было до этого.

Что такое генераторы в программировании

В программировании есть инструмент, который позволяет экономить память и при этом обрабатывать огромные массивы данных. Это генераторы. Мы рассмотрим работу генераторов на примере языка Python, но они есть и в других языках.

Классический подход к обработке — итераторы

Допустим, мы хотим вывести числа от 1 до 10 и для этого пишем такой код:

for i in range(1,10):

    print(i)

Это один из вариантов реализации цикла. Что делает компьютер, когда обрабатывает такое:

1. Создаст в памяти область для хранения данных.
2. Заполнит её числами от 1 до 10.
3. На каждом шаге цикла компьютер возьмёт новые данные из этой области и выведет их на экран.

При этом компьютер точно знает, какое значение у переменной i было на предыдущем шаге и будет на следующем, потому что все они хранятся в памяти.

Но что, если нам понадобится несколько переменных с диапазоном значений? Например, так:

a = range(1,100)

b = range(1000,2000)

for i in a:

    print(a[i-1] + b[i])

Когда мы запустим этот код, то увидим, что компьютер выделил большой кусок памяти для обеих переменных и что можно обратиться к отдельным ячейкам в таких диапазонах. Это удобно, когда нужно постоянно держать под рукой какие-то данные. Но если переменные со счётчиками не понадобятся, то память будет простаивать зря.

**👉 Итератор** в данном случае — это цикл, который обращается к диапазону значений и берёт по очереди оттуда данные. При этом все данные уже есть в памяти.

Итераторы хороши своей предсказуемостью, но при обработке большого потока данных могут привести к расходу памяти и неоптимальной работе программы.

Генераторы — вычисление данных «на лету»

Генераторы работают иначе: вместо того чтобы сразу хранить в памяти все данные, они их генерируют на каждом шаге и отдают в работу. Вот как выглядит цикл с генератором:

1. Цикл выполняется нужное количество раз.
2. На каждом шаге цикла генератор получает какое-то значение, отдаёт его в нужное место и забывает всё напрочь.
3. Генератор не помнит значение, которое он отдавал до этого, и не знает, что он будет отдавать на следующем шаге. Всё, что у него есть, — данные, которые нужно обработать на текущем шаге.
4. Память под работу генератора выделяется, только когда он генерирует новые данные. Пока генератор стоит или не выдаёт данные — память не выделяется.

Чаще всего генераторы используют как функции. Каждый раз, когда обращаются к такой функции-генератору, она делает так:

1. Берёт новую порцию данных из указанного ей источника.
2. Обрабатывает данные.
3. Возвращает результат.
4. Забывает про всё до следующего вызова.

Обычно функции возвращают результат своей работы с помощью команды return(), а для генераторов есть специальная команда — yield().

**Yield()** работает так же, как и return(), только функция на ней не заканчивается, а ставится на паузу. При следующем вызове генератор возьмёт новую порцию данных, и единственное, что он помнит, — на каком месте он остановился в прошлый раз. Всё остальное генератор каждый раз считает заново.

Пример из практики

Генераторы часто применяют для одноразовой обработки данных по каким-то правилам. Например, в [проекте с генератором текста на цепях Маркова](https://thecode.media/markov-text/) у нас был такой фрагмент кода:

# отправляем в переменную всё содержимое текстового файла

text = open('che.txt', encoding='utf8').read()

# разбиваем текст на отдельные слова (знаки препинания останутся рядом со своими словами)

corpus = text.split()

# делаем новую функцию-генератор, которая определит пары слов

def make\_pairs(corpus):

# перебираем все слова в корпусе, кроме последнего

for i in range(len(corpus)-1):

# генерируем новую пару и возвращаем её как результат работы функции

yield (corpus[i], corpus[i+1])

# вызываем генератор и получаем все пары слов

pairs = make\_pairs(corpus)

**PYTHON**

COPY

А вот что произошло здесь по шагам:

1. Мы открыли файл и записали всё его содержимое в переменную text.
2. С помощью встроенной функции split() мы разбили текст на отдельные слова и поместили все слова в отдельный массив. На этом этапе в массиве примерно 150 тысяч слов — для хранения такого количества данных компьютер выделил много памяти.
3. Мы пишем функцию-генератор. Каждый раз, когда к ней будут обращаться, она вернёт пару слов — текущее и следующее за ним.
4. В самом конце мы создаём новую переменную — pairs. Может показаться, что в ней сразу будут храниться все пары слов, но на самом деле это переменная-генератор. При каждом обращении к ней она вернёт новую пару слов и забудет о них.

В итоге у нас все слова хранятся в переменной corpus, а пары возвращаются «на лету» при каждом обращении к этой переменной.

👉 Главный плюс генераторов — их можно указывать в качестве диапазона в циклах. На каждом шаге цикл получает новое значение от генератора и работает уже с ним. Как только у генератора заканчиваются варианты и он останавливается — цикл тоже останавливается.

Вот как мы работаем с этой переменной дальше:

# словарь, на старте пока пустой

word\_dict = {}

# перебираем все слова попарно из нашего списка пар

for word\_1, word\_2 in pairs:

# если первое слово уже есть в словаре

if word\_1 in word\_dict.keys():

# то добавляем второе слово как возможное продолжение первого

word\_dict[word\_1].append(word\_2)

**PYTHON**

COPY

Здесь алгоритм работает так:

1. Делаем пустую переменную для словаря.
2. Запускаем цикл for и указываем переменную-генератор в качестве диапазона цикла.
3. Теперь на каждом шаге цикла он будет получать новую пару от генератора и обрабатывать её внутри цикла. При этом сами пары физически нигде не хранятся — их генератор каждый раз собирает на ходу.

❌ Если бы мы не знали про генераторы, нам бы пришлось делать отдельный массив с парами слов и выделять под него память. В нашем проекте так сделать можно, но в реальных задачах с перебором большого количества данных такой подход может съесть всю память.

И что, всё теперь нужно делать на генераторах?

Нет, нужно просто знать, что есть такая возможность — собирать данные на ходу и при этом не тратить на это память. Если вам в проекте не нужны генераторы или экономия памяти будет мизерная — работайте без них. Но если нужно обработать огромный массив данных, из которых нужна только часть, то генераторы могут реально выручить.