Итераторы и генераторы

(вольное переложение информации, найденной на просторах Интернета)

Оглавление

[Введение 1](#_Toc62560939)

[Итераторы 2](#_Toc62560940)

# Введение

Итераторы нужны, как ни странно, для того, чтобы итерировать элементы. Итератор – это специальный метод внутри класса, который позволяет итерировать элементы экземпляра этого класса по правилам, прописанным в этом методе. Итерирование мы наблюдаем в цикле for, поэтому по большому счету итераторы нужны для реализации циклов for.

Генераторы же нужны для генерации объектов. Например, для генерации списков.

Эти инструменты позволяют писать элегантный код при работе с такими математическими объектами, как бесконечные последовательности, случайные процессы, реккурентные отношения и комбинаторные структуры.

Ниже итераторы и генераторы рассматриваются более подробно, на конкретных примерах.

# Итераторы

Это объекты, которые позволяют обходить коллекции. При этом коллекции не должны обязательно существовать в памяти и быть конечными.

Итерируемый объект – это объект, в котором есть метод \_\_iter\_\_ .

Итератор – объект, в котором есть два метода: \_\_iter\_\_ и \_\_next\_\_.

Почти всегда итераторы возвращают себя из метода \_\_iter\_\_, т.к. они выступают итераторами для самих себя (но есть исключения). Зачастую у итератора нет определенной длины (по большому счету она ему и не нужна).

Стоит избегать прямого вызова методов \_\_iter\_\_ и \_\_next\_\_. Если все же надо вызвать напрямую, то можно использовать встроенные функции iter(x) (= x.\_\_iter\_\_) и next(x) (x.\_\_next\_\_).

Некоторые итерируемые объекты не являются итераторами, но используют другие объекты как итераторы. Например, list – не итератор (\_\_iter\_\_ есть, \_\_next\_\_ нет). Но итератор list относится к классу listiterator. У объектов list есть определенная длина, а у итераторов – нет.

a = [1, 2]

type(a) # list

type(iter(a)) # listiterator

it = iter(a)

next(it) # 1

next(it) # 2

next(it) # StopIteration – exception

len(a) # 2

len(it) # TypeError - sxception

Простой пример бесконечного итератора, возвращающего значения от 0 до бесконечности (аналог itertools.count):

class count\_iterator:

n = 0

def \_\_iter\_\_(self):

return self

def \_\_next\_\_(self):

y = self.n

self.n += 1

return y

Пример использования:

counter = count\_iterator()

next(counter) # 0

next(counter) # 1

next(counter) # 2

list(counter) # infinite loop

Если у объекта нет метода \_\_iter\_\_, его можно обойти, если определить метод \_\_getitem\_\_. В этом случае встроенная функция iter возвращает итератор с типом iterator, который использует \_\_getitem\_\_ для объода элементов объекта. Когда обход завершается, этот метод возвращает StopIteration или IndexError:

class SimpleList(object):

def \_\_init\_\_(self, \*items):

self.items = items

def \_\_getitem\_\_(self, i):

return self.items[i]

Пример использования:

a = SimpleList(1, 2, 3):

it = iter(a)

next(it) # 1

next(it) # 2

next(it) # 3

next(it) # StopIteration - exception

Итератор для генерации последовательности Q Хофштадтера:

Q(n) = Q(n-Q(n-1)) + Q(n-Q(n-2))

class qsequence:

def \_\_init\_\_(self):

self.s = s[:]

def \_\_next\_\_(self):

try:

q = self.s[-self.s[-1]]+self.s(-self.s[-2])

return q

except IndexError:

raise StopIteration()

def \_\_iter\_\_(self):

return self

def current\_state(self):

return self.s

Пример использования:

Q = qsequence([1, 1])

next(Q) # 2

next(Q) # 3

[next(Q) for \_\_ in range(10)]

# [3, 4, 5, 5, 6, 6, 6, 8, 8, 8]

# Генераторы

Генераторы – это итераторы, определение которых выглядит как определение функций. Или генераторы – это функции, которые используют внутри выражение yield.

Генераторы не могут возвращать значения, они могут только выдавать элементы по готовности, при этом неймспейс генератора запоминается.

Исключение StopIteration срабатывает автоматически, когда мы достигаем конца функции, поэтому его не нужно обрабатывать отдельно.

Через генератор можно реализовать бесконечную последовательность:

def count\_generator():

n = 0

while True:

yield n

n += 1

counter = count\_generator()

print(next(counter)) # 0

print(next(counter)) # 1

print(next(counter)) # 2

print(next(counter)) # 3

и последовательность Q Хофштадтера:

def qsequence\_gen(start\_list):

cur\_list = start\_list[:]

while True:

try:

q = cur\_list[-cur\_list[-1]] + cur\_list[-cur\_list[-2]]

cur\_list.append(q)

yield q

except IndexError:

return

qseq = qsequence\_gen([1, 1])

print(next(qseq)) # 2

print(next(qseq)) # 3

print(next(qseq)) # 3

print(next(qseq)) # 4

Реализуем при помощи генератора распределение Бернулли: бесконечная последовательность случайных булевых значений, вероятность True равна p, вероятность False равна q=1-p. Применяется экстрактор фон Неймана, который принимает процесс Бернулли с 0 < p < 1 как источник энтропии и возвращает чистый процесс Бернулли с p = 0.5.

import random

def bernulli\_process(p):

if p > 1.0 or p < 0.0:

raise ValueError

while True:

yield random.random() < p

def von\_neumann\_extractor(process):

while True:

x, y = next(process), next(process)

if x!= y:

yield x