Занятие 7: Итераторы и генераторы

Практикум на ЭВМ 2019/2020

Попов Артём Сергеевич
ред. Липкина Анна Львовна

МГУ имени М. В. Ломоносова, факультет ВМК, кафедра ММП

25 октября 2019 г.

Введение

Сколько памяти расходуется при выполнении?

```
>>> import numpy as np
>>> list_of_numbers = np.arange(0, 10 ** 8)
>>> sum_numbers = 0
>>> for item in list_of_numbers:
...     sum_numbers += item
```

Введение

Сколько памяти расходуется при выполнении? ($O(n)$)

```
>>> n = 10 ** 8
>>> list_of_numbers = np.arange(0, n)
>>> sum_numbers = 0
>>> for item in list_of_numbers:
...     sum_numbers += item
```

Существует реализация с памятью $O(1)$:

```
>>> item, sum_numbers = 0, 0
>>> while item < n:
...     sum_numbers += item
...     item += 1
```

Ещё одна реализация с памятью $O(1)$:

```
>>> sum_numbers = 0
>>> for item in range(0, n):
...     sum_numbers += item
```

Семантика оператора for

```
>>> for elem in elem_collection:
...     do_something(elem)
```

Концептуально такой цикл for делает следующее:

```
>>> it = iter(elem_collection) # инициализация
>>> while True:
...     try:
...         elem = next(it) # получить следующий элемент
...     except StopIteration: # если элементов больше нет
...         break
...     do_something(elem)
```

Метод __iter__

Метод __iter__ должен вернуть экземпляр класса, который будет реализовывать протокол итераций, например self.

По сути, это перегрузка встроенной обобщённой функции iter.

Примеры использования функции iter:

```
>>> my_list = [0, 1, 2]
>>> iter(my_list)
<list_iterator at 0x7f866bacdf28>
>>> my_list.__iter__()
<list_iterator at 0x7f8763166710>
>>> x = 1
>>> iter(x)
TypeError: 'int' object is not iterable
```

Метод __next__

Метод __next__ возвращает следующий по порядку элемент итератора. Если такого элемента нет, то метод должен поднять исключение StopIteration.
Считается, что если элемент поднял StopIteration, то все следующие тоже поднимут это исключение!

По сути, это перегрузка встроенной обобщённой функции next.

Примеры использования функции next:

```
>>> my_list = [0, 1, 2]
>>> it = iter(my_list)
>>> next(it)
0
>>> it.__next__() # то же самое, что next(it)
1
>>> next(my_list)
TypeError: 'list' object is not an iterator
```

Протокол итераций и итератор

Протокол итераций (протокол итераторов, iterator protocol) заключается в концепции итерирования по контейнерам с помощью методов `__iter__` и `__next__`

Итерируемое (iterable) — то, что имеет метод `__iter__`

Итератор (iterator) — то, что имеет методы `__iter__` и `__next__`, т.е. поддреживает протокол итераций

Пример написания своего итератора

```
>>> class MyFile:
...     def __init__(self, file_name):
...         self.file_name = file_name
...
...      def __iter__(self):
...         self.file = open(self.file_name, 'r')
...         return self
...
...      def __next__(self):
...         new_line = self.file.readline()
...         new_line = new_line.lower()
...         if new_line == '':
...             self.file.close()
...             raise StopIteration
...         return new_line
```

Зачем это может быть нужно?

Особенности iter и next

Другая форма вызова iter: принимает функцию и значение, вызывает функцию, пока она не вернёт это значение

```
>>> with open('my_file.txt', 'r') as input:
...     for line in iter(input.readline, ""):
...         print(line)
```

Для функции next можно вторым аргументом указать значение, которое необходимо вернуть в случае исключения:

```
>>> next(iter([]), 153)
153
```

Протокол итераций и __getitem__

Напоминание: `__getitem__` возвращает элемент по индексу либо поднимает `IndexError`, если элемента нет

Если реализован метод `__getitem__`, протокол итераций будет реализован автоматически

```
>>> class MyList:
...     def __init__(self, *args):
...         self.list = list(args)
...
...     def __getitem__(self, i):
...         return self.list[i]
...
>>> for elem in MyList(1, 2):
...     print(elem)
1
2
```

Как реализован протокол итераций исходя из примера?

Протокол итераций и __getitem__

Пример, когда итератор будет работать некорректно:

```
>>> class MyDict:
...     def __init__(self, **kwargs):
...         self.dict = dict(kwargs)
...
...     def __getitem__(self, i):
...         return self.dict[i]
...
>>> for elem in MyDict(a=1, b=2):
...     print(elem)
KeyError: 0
```

Протокол итераций и __contains__

Напоминание: `__contains__` возвращает `True`, если переданный элемент содержится в экземпляре (перегрузка `in` и `not in`)

Если класс реализует протокол итераций, метод `__contains__` реализуется автоматически «следующим образом»:

```
>>> class SomeClass:
...     # это реализуется автоматически
...     def __contains__(self, target_elem):
...         for elem in iter(self):
...             if target_elem == elem:
...                 return True
```

Пример для MyList:

```
>>> i in MyList(1, 2, 3, 4)
True
```

Генераторы

Генератор — это функция, которая использует не только оператор `return`, но и оператор `yield`

В результате выполнения оператора `yield` работа функции приостанавливается, а не прерывается, как при использовании оператора `return`

Любой генератор — итератор, обратное неверно

```
>>> def f():
...     print("Start")
...     x = 17
...     yield x
...     yield x + 1
...     print("Done")
```

Пример работы генератора

```
>>> def f():
...     print("Start")
...     x = 17
...     yield x
...     yield x + 1
...     print("Done")
```

```
>>> type(f)
<class function>
>>> gen = f()
>>> type(gen)
<class generator>
>>> next(gen)
Start
17
```

```
>>> next(gen)
18
>>> next(gen)
Done
StopIteration:
```

Пример генератора: unique

```
>>> def unique(iterable, seen=None):
...     seen = set(seen or [])
...     for item in iterable:
...         if item not in seen:
...             seen.add(item)
...             yield item
...
>>> sequence = [1, 1, 2, 3]
>>> unique(sequence)
<generator object unique at 0x7f87609ccc50>
>>> for elem in unique(sequence):
...     print(elem)
1
2
3
```

Пример генератора: chain

```
>>> def chain(*iterables):
...     for iterable in iterables:
...         for item in iterable:
...             yield item
>>> xs = range(3)
>>> ys = [42]
>>> chain(xs, ys)
<generator object chain at 0x10311d708>
>>> list(chain(xs, ys))
[0, 1, 2, 3, 42]
>>> 42 in chain(xs, ys)
True
```

Переиспользование генератора

Генераторы, как и любые итераторы, истощаются:

```
>>> def f():
...     yield 42
...
>>> gen = g()
>>> list(gen)
[42]
>>> list(gen)
[]
```

Не надо переиспользовать генераторы!

Генератор — способ задания итератора

```
>>> def my_file(file_name):
...     file_obj = open(file_name, 'r')
...     for new_line in file_obj:
...         new_line = new_line.lower()
...         if new_line != '':
...             yield new_line
...         else:
...             file_obj.close()
```

Генераторные выражения

Важно: генераторы списков, словарей множеств не являются генераторами!

Но есть похожая конструкция генераторное-выражение:

```
>>> gen = (x ** 2 for x in range(0, 5))
>>> gen
<generator object <genexpr> at 0x7f87609d7308>
>>> list(gen)
[0, 1, 4, 9, 16]
```

Оператор `yield from`

Оператор `yield from` позволяет делегировать выполнение другому генератору:

```
>>> def f():
...     yield 1
...     yield 2
...
>>> def g():
...     s = f()
...     yield from s
...
>>> list(g())
[1, 2]
```

Продвинутое использование генераторов

Генераторы не так просты как кажутся

Продвинутые методы работы с генераторами:

- Можно передать в генератор исключение и поднять его в месте, где генератор приостановил исполнение (`.throw`)
- Можно передавать в генератор свои значения (`.send`)
- Можно «закрыть» генератор, передав ему специальное исключение (`.close`)

Подробности можно узнать здесь:
https://compscicenter.ru/courses/python/2015-autumn/classes/1542/
http://dabeaz.com/coroutines/Coroutines.pdf

range

`range` возвращает итерируемый объект `range object` (не итератор!)

```
>>> my_range = range(0, 5)
>>> my_range
range(0, 5)
>>> next(my_range)
TypeError: 'range' object is not an iterator
>>> iter(my_range)
<range_iterator at 0x7f8780179240>
>>> next(iter(my_range))
0
```

`range` при любых аргументах требует константный объём памяти

В Python2 аналог `range` — `xrange`

zip

`zip` возвращает итератор

```
>>> my_list_1 = [1, 2]
>>> my_list_2 = ['a', 'b']
>>> my_tuples = zip(my_list_1, my_list_2)
>>> my_tuples
<zip at 0x7f8769928e08>
>>> iter(my_tuples)
<zip at 0x7f8769928e08>
>>> next(my_tuples)
(1, 'a')
>>> next(my_tuples)
(2, 'b')
>>> next(my_tuples)
StopIteration:
```

enumerate

`enumerate` возвращает итератор

```
>>> my_list = ['a', 'b', 'c']
>>> it = enumerate(my_list)
>>> it
<enumerate at 0x7f87609c6360>
>>> next(it)
(0, 'a')
```

Как выглядел бы enumerate, если бы мы его писали сами?

map

`map(func, sequence1, sequence2 ... )` возвращает итератор, состоящий из результатов применений функции `func` к элементам `sequence1`, `sequence2`

- `func` — функция, которая применяется к каждому элементу `sequence`
- `sequence1`, `sequence2` ... — итераторы

```
>>> it = map(lambda x: x * 2, [0, 1, 2, 3])
>>> list(it)
[0, 1, 4, 9]
>>> it = map(lambda x, y: x + y, [0, 1, 2, 3], [10, 100, 500])
>>> list(it)
[10, 101, 502]
```

filter

`filter(func, sequence)` возвращает итератор, состоящий из элементов `sequence`, для которых `func` вернула `True`

```
>>> it = filter(lambda x: x > 0, [0, -1, 2, -3, 5])
>>> list(it)
[2, 5]
```

И `filter`, и `map` можно проще записать через генераторы списков!

reduce

`reduce(func, sequence)` — кумулятивное применение функции `func` к элементам последовательности `sequence`

```
>>> from functools import reduce
...
>>> # ((1*2)*3)*4
>>> it = reduce(lambda x, y: x * y, [1, 2, 3, 4])
>>> it
24
>>> it = reduce(lambda x, y: '({}*{})'.format(str(x), str(y)),
                [1, 2, 3, 4])
(((1*2)*3)*4)
```

Соединение и повторение итераторов

Написанная нами chain есть в itertools:

```
>>> from itertools import chain
>>> it1 = iter([1,2,3])
>>> it2 = iter([4,5])
>>> list(chain(it1, it2))
[1, 2, 3, 4, 5]
```

repeat для создания итератора повторяющейся последовательности:

```
>>> from itertools import repeat
>>> b = []
>>> for i in repeat([1, 2], 4):
...     b.append(i)
>>> print (b)
[1, 2, 1, 2, 1, 2, 1, 2]
```

Бесконечные итераторы

```
from itertools import count # бесконечный range
c = []
for i in count(0, 5):
    c.append(i)
    if i > 20:
        break
print(c)
[0, 5, 10, 15, 20, 25]
```

```
>>> from itertools import cycle # бесконечный повтор
>>> d = []                      # последовательности
>>> it = cycle([1,2,3])
>>> for i, j in enumerate(it):
...     if i > 6:
...         break
...     d.append(j)
>>> print(d
[1, 2, 3, 1, 2, 3, 1]
```

Срезы и взятие частей

Срез для произвольного итератора:

```
>>> from itertools import islice
>>> e = islice(range(10), 0, 8, 2)
>>> list(e)
[0, 2, 4, 6]
```

Выбрасывать из итератора элементы, пока не найдётся элемент, удовлетворяющий условию:

```
>>> from itertools import dropwhile
>>> f = dropwhile(lambda x: x < 5, range(10))
>>> print (list(f))
[5, 6, 7, 8, 9]
```

Комбинаторные итераторы

```
>>> from itertools import permutations
>>> it = permutations("YN")
>>> print (list(it))
[('Y', 'N'), ('N', 'Y')]
```

```
>>> from itertools import combinations
>>> it = combinations("ABC", 2)
>>> print (list(it))
[('A', 'B'), ('A', 'C'), ('B', 'C')]
```

```
>>> from itertools import product
>>> it = product("AB", repeat=2)
>>> print (list(it))
[('A', 'A'), ('A', 'B'), ('B', 'A'), ('B', 'B')]
```

Итераторы в задачах с данными большого объёма

Итераторы могут быть полезны в ситуациях, когда хранить всю выборку в памяти невозможно

Алгоритм работы:

1. Сохранить данные на жёстком диске в хорошем формате
2. Написать итератор, считывающий данные неполностью и преобразующий их в вектора
3. Обучать алгоритм по батчам данных

Некоторые библиотеки поддерживают работу с итераторами из коробки (например, Gensim).

Варианты применения итераторов в data science

- Работа с огромной текстовой коллекцией
- Изменение порядка подачи объектов на обучение
- Быстрая генерация из сложных объектов более простых (например, по тексту генерируем пары предложений)

Заключение

- Итераторы позволяют не хранить одновременно в памяти все значения последовательности
- Итераторы можно задавать с помощью описания специальных методов `__iter__` и `__next__` или декларативно с помощью генераторов
- Многие операции с операторами уже реализованы в библиотеке `itertools` либо средствами `itertools` их можно реализовать проще
- Итераторы необходимо применять тогда, когда невозможно хранить весь объём данных в памяти