Глава 14. Итерируемые объекты, итераторы и генераторы.

Итерирование — одна из важнейших операций обработки данных. А если просматривается набор данных, не помещающийся целиком в память, то нужен способ выполнять её *отложенно*, т.е. по одному элементу и по запросу. Именно в этом смысл паттерна **Итератор**.

Ключевое слово **yield** (*появилось только в Py >= 2.2*) позволяет конструировать генераторы, которые работают как итераторы.

Любой генератор является итератором: генераторы используют весь интерфейс итераторов. Но итератор — в том виде, как он определён в книге "Банды четырёх", — извлекает элементы из коллекции, тогда как генератор может порождать элементы "из воздуха". Типичным примером является генератор чисел Фибоначчи — бесконечно последовательности, которую нельзя сохранить в коллекции. Однако, надо иметь в виду, что в сообществе Python слова *итератор* и *генератор* обычно употребляется как синонимы.

Класс Sentence, попытка 1: Последовательность слов.

В примере представлен класс `Sentence`, который умеет извлекать из текста слово с заданным индексом.

`sentence.py`: *объект* `Sentence` *как последовательность слов*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence:

    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)
        # Возвращает список всех не пересекающихся
        # подстрок, соответствующих регулярному выражению RE_WORD.

    def __getitem__(self, item):
        """
        self.words содержит результат .fandall, поэтому мы просто
        возвращаем слово с заданным индексом
        """
        return self.words[item]
```

```
    def __len__(self):
        """
        Что бы выполнить требования протокола последовательности,
        реализуем данный метод,- но для получения итерируемого
        объекта он не нужен.
        """
        return len(self.words)

    def __repr__(self):
        """
        По умолчанию reprlib.repr ограничивает сгенерированную
        строку 30 символами.
        >>> s = Sentence('Время жизни на репит, просто что бы закрепить.')
        >>> s
        Sentence('Время жизни ...бы закрепить.')
        >>> for word in s:
        ...     print(word)
        ...
        Время
        жизни
        на
        репит
        просто
        что
        бы
        закрепить
        >>> list(s)
        ['Время', 'жизни', 'на', 'репит', 'просто', 'что', 'бы', 'закрепить']
        """
        return 'Sentence(%s)' % reprlib.repr(self.text)


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Всякий раз когда интерпретатору нужно обойти объект `x`, он автоматически вызывает функцию `iter(x)`.

Встроенная функция `iter(x)` выполняет следующие действия.

1. Смотрим, реализует ли объект метод `__iter__`, и если да, вызывает его, что бы получить итератор.
2. Если метод `__iter__` не реализован, но реализован метод `__getitem__`, то Python создает итератор, который пытается извлекать элементы по порядку, начиная с индекса 0.
3. Если и это не получается, то возбуждает исключение — обычно с сообщением `C object is not iterable`, где *C* — класс объекта.

sentence_iter.py: *класс* Sentence*, реализован с помощью паттерна Итератор*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence():
    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        return SentenceIterator(self.words)


class SentenceIterator():

    def __init__(self, words):
        self.words = words
        self.index = 0

    def __next__(self):
        try:
            word = self.words[self.index]
        except IndexError:
            raise StopIteration()

        self.index += 1
        return word

    def __iter__(self):
        return self
```

sentence_gen.py: *класс* Sentence*, реализован с помощью паттерна Генератора*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence():
    def __init__(self, text):
        self.text = text
        self.words = RE_WORD.findall(text)

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        for word in self.words:
            yield word
        return
```

Как работает генераторная функция

Любая функция Python, в теле которой встречается слово yield, называется генераторной функцией — при вызове она возвращает объект-генератор.

sentence_gen2.py: *класс* Sentence*, реализован с помощью паттерна Генератора*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence:
    def __init__(self, text):
        """
        Ленивая реализация класса Sentence.
        Хранить список слов не нужно.
        """
        self.text = text

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        """
        finditer строит итератор, который обходит все
        соответствия текста self.text регулярному выражению
        RE_WORD, порождая объекты MatchObject.

        match.group() извлекает сопоставленный текст из
        объекта MatchObject
        """
        for match in RE_WORD.finditer(self.text):
            yield match.group()
```

Генераторные функции — замечательный способ сократить код, но генераторные выражения ещё круче.

Простые генераторные функции наподобие той, что мы использовали в предыдущем варианте класса Sentence, можно заменить *генераторным выражением*.

Можно считать что генераторное выражение — ленивая версия спискового включения: она не строит список энергично, а возвращает генератор, который лениво порождает элементы по запросу.

Генераторная функция `gen_AB` *используется сначала в списковом включении, а затем в генераторном выражении.*

```
>>> def gen_AB():
...     print('start')
...     yield 'A'
...     print('continue')
...     yield 'B'
...     print('end.')
...
>>> res1 = [x*3 for x in gen_AB()] # Списковое включение энергично обходит элементы,
порождаемые объектом-генератором, который был создан функцией gen_AB.
start
continue
end.
>>> for i in res1:
...     print('-->', i)
...
--> AAA
--> BBB
>>> res2 = (x*3 for x in gen_AB())
>>> res2
<generator object <genexpr> at 0x0000020BAC482C00>
>>> for i in res2:
...     print('-->', i)
...
start
--> AAA
continue
--> BBB
end.
```

Таким образом, генераторное выражение порождает генератор, и мы можем этим воспользоваться что бы ещё сократить размер класса Sentence.

`sentence_genexp.py`: *реализация класса* `Sentence` *с помощью генераторного выражения.*

```
import re
import reprlib

RE_WORD = re.compile(r'\w+')


class Sentence:
    def __init__(self, text):
        self.text = text

    def __repr__(self):
        return f'Sentence({reprlib.repr(self.text)})'

    def __iter__(self):
        return (match.group() for match in RE_WORD.finditer(self.text))
```

Генераторные выражения - это не более чем синтаксическая глазурь: их всегда можно заменить генераторными функциями.

Построение арифметической прогрессии

```
def aritprog_gen(begin, step, end=None):
    """
    Реализация арифметической прогрессии
    с использованием генераторной функции.
    :param begin: начальное значение.
    :param step: шаг прогрессии.
    :param end: конечное значение
    (не обязательный аргумент)

    > Тест на проверку класса возвращаемеого значения
    >>> from fractions import Fraction
    >>> b = aritprog_gen(0, Fraction(1, 3), 1)
    >>> list(b)
    [Fraction(0, 1), Fraction(1, 3), Fraction(2, 3)]
    >>> from decimal import Decimal
    >>> c = aritprog_gen(0, Decimal('.01'), .03)
    >>> list(c)
    [Decimal('0'), Decimal('0.01'), Decimal('0.02')]
    """
    result = type(begin+step)(begin)
    # Возвращает значение того же класса,
    # что и начальное begin
    forever = end is None
    index = 0
    while forever or result < end:
        yield result
        index += 1
        result = begin + step * index
```

Построение арифметической прогрессии с помощью `itertools`

Функция `itertools.count` возвращает генератор, порождающий числа. Без аргументов порождает ряд целых начиная с 0. А если задать аргументы `start` и `step`, то получится результат схожий с тем, что дает функция `aritprog_gen`

Однако `itertools.count` никогда не останавливается.

`itertools.takewhile` - порождает генератор, который потребляет другой генератор и останавливается, когда заданный предикат станет равен `False`.

```
>>> import itertools
>>> gen = itertools.takewhile(lambda n: n<3, itertools.count(1, .5))
>>> list (gen)
[1, 1.5, 2.0, 2.5]
```

`aritprog_v3.py`: *работает так же как и* `aritprog_gen.py`

```
import itertools


def aritprog_gen(begin, step, end=None):
    first = type(begin+step)(begin)
    ap_gen = itertools.count(first, step)
    if end is not None:
        ap_gen = itertools.takewhile(lambda n: n < end, ap_gen)
    return ap_gen
```

Генераторные функции в стандартной библиотеке

Table 1. Фильтрующие генераторные функции

Модуль	Функция	Описание
itertools	compress(it, selector_it)	Потребляет параллельно два итерируемых объекта; отдает элемент it, когда соответствующий selector_it принимает истинное значение.
	dropwhile(predicate, it)	Потребляет it, пропуская элементы, пока predicate принимает похожее на истину значение, а затем отдает все оставшиеся элементы (больше никаких проверок не делает)
встроенная	filter(predicate, it)	Применяет предикат к каждому элементу итерируемого объекта, отдавая элемент, если predicate(item) принимает похожее на истину значение; если predicate равен None, отдаются только элементы, принимающие похожее значение.

Модуль	Функция	Описание
itertools	`filterfalse(predicate, it)`	То же, что `filter`, но логика инвертирована: отдаются элементы, для которых предиката принимает похожее на ложь значение.
	`islice(it, stop)` или `islice(it, start, stop, step=1)`	Отдает элементы из среза `it` по аналогии с `s[:stop]` или `s[start:stop:step]`, только `it` может быть произвольным итерированным объектом, а операция ленивая.
	`takewhile(predicate, it)`	Отдает элементы, пока `predicate` принимает похожее на истину значение, затем останавливается, больше никаких проверок не делается.

Примеры фильтрующих генераторных функций

```
>>> def vowel(c):
...     return c.lower() in 'aeiou'
>>> list(filter(vowel, 'Aardvark'))
['A', 'a', 'a']
>>> import itertools
>>> list(itertools.filterfalse(vowel, 'Aardvark'))
['r', 'd', 'v', 'r', 'k']
>>> list(itertools.dropwhile(vowel, 'Aardvark'))
['r', 'd', 'v', 'a', 'r', 'k']
>>> list(itertools.takewhile(vowel, 'Aardvark'))
['A', 'a']
>>> list(itertools.compress('Aardvark', (1,0,1,1,0,1)))
['A', 'r', 'd', 'a']
>>> list(itertools.islice('Aardvark', 4))
['A', 'a', 'r', 'd']
>>> list(itertools.islice('Aardvark', 4, 7))
['v', 'a', 'r']
>>> list(itertools.islice('Aardvark', 1, 4, 7))
['a']
>>> list(itertools.islice('Aardvark', 1, 7, 2))
['a', 'd', 'a']
```

Table 2. Отображающие генераторные функции

Модуль	Функция	Описание
`itertools`	`accumulate(it, [func])`	Отдает накопленные суммы; если задана функция `func`, то отдает результат, применяя её к первой паре элементов, затем к первому результату и следующему элементу и т.д.
встроенная	`enumerate(iterable, start=0`	Отдает 2-кортежи вида `(index, item)`, где `index` начинается со значения `start`, а `item`, извлекается из `iterable`.
	`map(func, it1, [it2, ⋯, itN])`	Применяет `func` к каждому элементу `it` и отдает результат; если задано `N` итерируемых объектов, то `func` должна принимать `N` итерируемых объектов, и все итерируемые объекты обходятся параллельно.
`itertools`	`starmap(func, it)`	Применяет `func` к каждому элементу `it` и отдает результат; входной итерируемый объект должен отдавать итерируемые объекты `iit`, а `func` вызывается в виде `func(*iit)`

Примеры применения генераторной функции `itertools.accumulate`

```
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.accumulate(sample)) # Частичные суммы.
[5, 9, 11, 19, 26, 32, 35, 35, 44, 45]
>>> list(itertools.accumulate(sample, min)) # Частичные минимумы.
[5, 4, 2, 2, 2, 2, 2, 0, 0, 0]
>>> list(itertools.accumulate(sample, max)) # Частичные максимумы.
[5, 5, 5, 8, 8, 8, 8, 8, 9, 9]
>>> import operator
>>> list(itertools.accumulate(sample, operator.mul)) # Частичные произведения.
[5, 20, 40, 320, 2240, 13440, 40320, 0, 0, 0]
>>> list(itertools.accumulate(range(1, 11), operator.mul)) # Факториалы от 1! до 10!.
[1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800]
```

Примеры применения отображающих генераторных функций

```
>>> list(enumerate('XOMRKOB', 1))
[(1, 'X'), (2, 'O'), (3, 'M'), (4, 'R'), (5, 'K'), (6, 'O'), (7, 'B')]
>>> list(map(operator.mul, range(11), range(11))) # Квадраты целых чисел от 0 до 10
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> list(map(operator.mul, range(11), [2, 4, 8])) # Перемножение целых чисел из двух
параллельных итерируемых объектов; операция заканчивается, когда будет достигнут конец
более короткого объекта.
[0, 4, 16]
>>> list(map(lambda a, b: (a, b), range(11), [2, 4, 8])) # То же самое делает
встроенная функция zip
[(0, 2), (1, 4), (2, 8)]
>>> list(itertools.starmap(operator.mul, enumerate('albatroz', 1))) # Повторение
каждой буквы слова, столько раз, каков номер ее позиции
['a', 'll', 'bbb', 'aaaa', 'ttttt', 'rrrrrr', 'ooooooo', 'zzzzzzzz']
>>> sample = [5, 4, 2, 8, 7, 6, 3, 0, 9, 1]
>>> list(itertools.starmap(lambda a, b: b/a, enumerate(itertools.accumulate(sample),
1))) # Частичное среднее 
[5.0, 4.5, 3.6666666666666665, 4.75, 5.2, 5.333333333333333, 5.0, 4.375,
4.888888888888889, 4.5]
```

Модуль	Функция	Описание
itertools	`chain(it1, …, itN)`	Отдает все элементы из `it1`, затем из `it2` и т.д.
	`chain.from_iterable(it)`	Отдает все элементы из каждого итерируемого объекта, порождаемого `it`, перебирая их один за другим; `it` должен порождать итерируемые объекты, например это может быть список итерируемых объектов.
	`product(it1, …, itN, repeat=1)`	Декартово произведение: отдает N-кортежи, полученные путём комбинирования элементов из каждого входного итерируемого объекта,- так, как это делалось бы с помощью вложенных циклов `for`; аргумент `repeat` позволяет обходить входные итерируемые объекты больше одного раза.

Модуль	Функция	Описание
встроенная	`zip(it1, ⋯, itN)`	Отдает N-кортежи, построенные из элементов, которые берутся параллельно из входных итерируемых объектов; операция прекращается по исчерпанию самого короткого итерируемого элемента.
`itertools`	`zip_longest(it1, ⋯, itN, fillvalue=None)`	Отдает N-кортежи, построенные из элементов, которые берутся параллельно из входных итерируемых объектов; операция прекращается по исчерпанию самого длинного итерируемого элемента, а вместо недостающих элементов подставляется значение `fillvalue`.

Примеры применения объединяющих генераторных функций

```
>>> list(itertools.chain('ABC', range(2))) # chain обычно вызывается с 2мя и более
итерируемых объектов
['A', 'B', 'C', 0, 1]
>>> list(itertools.chain(enumerate('ABC'))) # При вызове с одним элементом, chain не
делает ничего полезного
[(0, 'A'), (1, 'B'), (2, 'C')]
>>> list(itertools.chain.from_iterable(enumerate('ABC'))) # chain.from_iterable берёт
каждый элемент из итерируемого объекта и сцепляет их в последовательность, при
условии, что каждый элемент является итерируемым объектом
[0, 'A', 1, 'B', 2, 'C']
>>> list(zip('ABC', range(5)))
[('A', 0), ('B', 1), ('C', 2)]
>>> list(zip('ABC', range(5), [10, 20, 30, 40]))
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30)]
>>> list(itertools.zip_longest('ABC', range(5), [10, 20, 30, 40]))
[('A', 0, 10), ('B', 1, 20), ('C', 2, 30), (None, 3, 40), (None, 4, None)]
```

Примеры применения генераторной функции `itertools.product`

```
>>> import itertools
>>> list(itertools.product('ABC', range(2)))
[('A', 0), ('A', 1), ('B', 0), ('B', 1), ('C', 0), ('C', 1)]
>>> suits = 'Черви Буби Кресты Пики'.split()
>>> list(itertools.product('AKQJ', suits))
[('A', 'Черви'), ('A', 'Буби'), ('A', 'Кресты'), ('A', 'Пики'), ('K', 'Черви'), ('K',
'Буби'), ('K', 'Кресты'), ('K', 'Пики'), ('Q', 'Черви'), ('Q', 'Буби'), ('Q',
'Кресты'), ('Q', 'Пики'), ('J', 'Черви'), ('J', 'Буби'), ('J', 'Кресты'), ('J',
'Пики')]
```

Table 3. Генераторные функции, расширяющие каждый входной элемент в несколько выходных

Модуль	Функция	Описание
itertools	`combinations(in, out_len)`	Отдает комбинации `out_len` элементов из элементов, отдаваемых `it`.
	`combinations_with_replacement(it, out_len)`	Отдает комбинации `out_len` элементов из элементов, отдаваемых `it`, включая комбинации с повторяющимися элементами.
	`count(start=0, step=1)`	Отдает числа, начиная с `start` с шагом `step`.
	`cycle(it)`	Отдает элементы из `it`, запоминая копию каждого, после чего отдает всю последовательность еще раз — и так до бесконечности.
	`permutations(it, out_len=None)`	Отдает перестановки `out_len` элементов из элементов, отдаваемых `it`; по-умолчанию `out_len` равно `len(list(it))`.
	`repeat(item, [times])`	Повторно отдает заданный элемент - `times` раз или бесконечно, если аргумент не задан.
	`groupby(it, key=None)`	Порождает 2-кортежи вида (`key`, `group`), где `key` — критерий группировки, а `group` — генератор, отдающий элементы группы.
	`tee(it, n=2)`	Отдает кортеж `n` генераторов, каждый из которых независимо отдает элементы входного итерируемого объекта.
встроенная	`reversed(seq)`	Отдает элементы `seq` в обратном порядке, от последнего к первому; аргумент `seq` должен быть последовательностью или иметь реализованный метод `__reversed__`.

Пример использования `itertools.groupby`

```
>>> list(itertools.groupby('AAAABBBBCCCCDDDD'))
[('A', <itertools._grouper object at 0x000001084DAE74F0>), ('B', <itertools._grouper
object at 0x000001084DAE7D30>), ('C', <itertools._grouper object at
0x000001084DAE7400>), ('D', <itertools._grouper object at 0x000001084DAE7CA0>)]
>>> for char, group in itertools.groupby('AAAABBBBCCCCDDDD'):
...     print(f'{char} --> {list(group)}')
...
A --> ['A', 'A', 'A', 'A']
B --> ['B', 'B', 'B', 'B']
C --> ['C', 'C', 'C', 'C']
D --> ['D', 'D', 'D', 'D']
>>> animals = ['утка', 'орёл', 'мышь', 'жираф', 'медведь', 'летучая мышь', 'дельфин',
'акула', 'лев']
>>> animals.sort(key=len)
>>> animals
['лев', 'утка', 'орёл', 'мышь', 'жираф', 'акула', 'медведь', 'дельфин', 'летучая
мышь']
>>> for length, group in itertools.groupby(reversed(animals), len):
...     print(f'{length} --> {list(group)}')
...
12 --> ['летучая мышь']
7 --> ['дельфин', 'медведь']
5 --> ['акула', 'жираф']
4 --> ['мышь', 'орёл', 'утка']
3 --> ['лев']
```

`yield from` — новая конструкция в Python 3.3

Вложенные циклы `for` — традиционное решение в случае, когда генераторная функция должна отдавать значения, порождаемые другим генератором.

Пример реализации сцепляющего генератора

```
>>> def chain(*iterable):
...     for it in iterable:
...         for i in it:
...             yield i
...
>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s,t))
['A', 'B', 'C', 0, 1, 2]
```

Генераторная функция `chain` дает шанс поработать каждому полученному итерируемому объекту по очереди. В документе PEP 380 — Syntax for Delegating to a Subgenerator описан новый синтаксис для решения этой задачи, он показан ниже.

```
>>> def chain(*iterables):
...     for i in iterables:
...         yield from i

>>> s = 'ABC'
>>> t = tuple(range(3))
>>> list(chain(s, t))
['A', 'B', 'C', 0, 1, 2]
```

Как видим `yield from i` полностью заменяет внутренний цикл `for`. Данная конструкция выглядит лучше, но это всего лишь *синтаксическая глазурь* .