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

Фильтрующие генераторы

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.