Итераторы

Итератор - основной объект для взаимодействия с последовательностями. Им мы последовательно перебираем объекты последовательности и взаимодействуем с ними

С некоторыми последовательностями можно проводить индексируемую итерацию. Тогда мы будем получать объекты под конкретным индексом или последовательности, ограниченные итераторами.

```
>>> r = range(1, 100, 13)
>>> r[0] # получаем конкретный объект под конкретным индексом
1
>>> r[0:2] # получаем подпоследовательность последовательности
range(1, 27, 13)
>>> r[200:] # если вышли за границы - возвращается пустая
последовательность, это корректная работа
range(105, 105, 13)
```

Можно вести неиндексируемую итерацию. Например, в цикле ***for*** просто перебираются элементы последовательности

```
>>> for i, k in enumerate("QWR"): # enumerate перебирает сразу две
последовательности - заданную и целых чисел. И возвращает два объекта -
число и объект из заданной. Работает, пока одна из последовательностей не
кончится
...     print(i, k)
...
0 Q
1 W
2 R

>>> e = enumerate("ASDfa")
>>> e
<enumerate object at 0x7f766801a570>
>>> e[3]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'enumerate' object is not subscriptable
```

Забавно получается: итераторы - неиндексируемые, но индексируемые

Основные методы для взаимодействия с итератором - ***__iter__(), __next__()***
Первый выделяет из последовательности итератор, второй позволяет непосредственно итерироваться, получая каждое следующее значение, пока последовательность не закончится и не вылетит исключение ***StopIteration***

Если объект предоставляет метод __next__() (__iter__()) он является итерируемым
Если у объекта есть метод __iter__(), то по нему можно сделать итератор

```
>>> res = map(str.lower, 'AsSert') # map - функция применяет функцию из первого аргумента ко всем элементам последовательности из второго аргумента и возвращает итерируемую последовательность из обработанных элементов
>>> next(res)
'a'
>>> next(res)
's'
>>> next(res)
's'
>>> next(res)
'e'
>>> next(res)
'r'
>>> next(res)
't'
>>> next(res)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> res = map(str.lower, 'AsSert')
>>> for r in res:
...     print(r)
...
a
s
s
e
r
t
>>> # Вызывается метод __next__ пока не будет IterationStop
```

Итератор - Одноразовая штука. Если мы прошли по какой-то последовательности итератором, то им же мы больше никуда не сможем пройти, будет постоянно возвращаться StopIteration

```
>>> for res in r:
...     print(res)
...
>>> next(r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Не все объекты итерируемы, но от них можно построить итератор

```
>>> r = range(10)
>>> next(r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object is not an iterator
>>> ite = iter(r)
>>> r
range(0, 10)
>>> ite
<range_iterator object at 0x7f7667f4fbd0>

>>> next(ite)
0
>>> next(ite)
1
>>> next(ite)
2
>>> next(ite)
3
>>> next(ite)
4
>>> next(ite)
5
>>> next(ite)
6
>>> next(ite)
7
>>> next(ite)
8
>>> next(ite)
9
>>> next(ite)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
>>> r = range(10)
>>> ite = iter(r)
>>> next(ite)
0
>>> next(ite)
1
>>> next(ite)
2
>>> next(ite)
3
>>> next(ite)
4
>>> next(ite)
5
>>> list(ite) # а тут мы сгенерировали список по итерируемой
последовательности
[6, 7, 8, 9]
```

Продолжаем экзекуции: можно также сделать итератор, если у объекта есть метод ***__getitem__()***. Вместо StopIteration тогда используется IndexError

```
>>> class C:
...     def __getitem__(self, n):
...         if n < 7:
...             return f'/{n}/'
...         else:
...             raise IndexError
...
>>> c = C()
>>> it = iter(c)
>>> next(it)
'/0/'
>>> next(it)
'/1/'
>>> next(it)
'/2/'
>>> next(it)
'/3/'
>>> next(it)
'/4/'
>>> next(it)
'/5/'
>>> next(it)
'/6/'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

```
>>> import random
>>> d = iter(lambda: random.randrange(10), 0)
>>> list(d)
[2, 2, 6, 4, 4, 5, 3, 2, 4]
>>> d = iter(lambda: random.randrange(10), 0)
>>> list(d)
[1, 8, 3, 3, 2, 6, 5, 6, 4, 7, 6, 6, 3, 9, 8, 7, 2, 4, 6, 4]
>>> d = iter(lambda: random.randrange(10), 0)
>>> list(d)
[5, 5, 2, 7, 1, 4, 2, 7]
```

тк итератор является последовательностью (вернее, описывает её), везде его вместо неё можно использовать

```
>>> a, *b = iter(lambda: random.randrange(10), 5)
>>> a
8
>>> b
[6, 9, 3, 1, 3, 9, 1, 0, 8, 4, 4, 9]

>>> a, *b = iter(lambda: random.randrange(10), 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: not enough values to unpack (expected at least 1, got 0) # Это норма, просто у нас сразу первый прилетела пятёрка, и итератор ничего не выдал

>>> a, *b = iter(lambda: random.randrange(10), 5)
>>> a
4
>>> b
[8, 4, 9, 8, 7, 6, 0, 7, 3, 6, 2, 9, 1, 9, 9, 4, 7, 6, 3, 4, 1, 7, 9, 9, 9, 4]


>>> print(*iter(lambda: random.randrange(10), 8))
2 7 0 1
>>> print(*iter(lambda: random.randrange(10), 8))
0 0 3 3 6 6 4 0 4 2 4 6 2 0 0 5 5 3 0 9 0 6 6 3 6 2 7 5 0 9 9 1 9 2 0 2 2 5 2 5 1 0 1 1 1 1 6 5 6
1 5 9 5 6 3 7
>>> print(*iter(lambda: random.randrange(10), 8))
4 0 3 6 9 7 1 2 3 9
```

Работа цикла for: создание итератора - next() - обработка StopIteration

Можно задать итерируемый объект руками - генератор обернуть круглыми скобочками

```
>>> it = (i*2+1 for i in range(10))
>>> it
<generator object <genexpr> at 0x7f76681f41e0>
>>> dir(it)
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getstate__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__lt__', '__name__', '__ne__', '__new__',
'__next__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'close', 'gi_code', 'gi_frame', 'gi_running',
'gi_suspended', 'gi_yieldfrom', 'send', 'throw']

>>> next(it)
1
>>> next(it)
3
>>> next(it)
5
>>> next(it)
7
>>> next(it)
9
>>> next(it)
11
>>> next(it)
13
>>> next(it)
15
>>> next(it)
17
>>> next(it)
19
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Поговорим про Генератор-функции. Для начала просто напишем функцию, которая у нас будет уметь обрабатывать последовательности. Мы подаём всецело последовательность в неё,она нам её выводит

```
>>> def fun(seq):
...     print(*seq)
...
>>> fun([1, 2, 3])
1 2 3
>>> fun(i*2 + 1 for i in range(10))
1 3 5 7 9 11 13 15 17 19
```

А теперь зададимся такой целью: Хочется задавать какие-то входные данные, а после, как к итератору, обращаться к функции (в любой момент времени, не обязательно всё за раз) и получать каждый раз следующее значение. При этом именно не хранить всю последовательность и её выдавать, а прям с нуля от предыдущего значения новое генерировать. (Чтобы, н-р, уметь работать с бесконечными последовательностями. Они же явно не хранятся в памяти, а просто просчитывают следующий элемент)

Такие функции называются генератор-функциями. Их отличает наличие ключевого слова ***yield***. Логика работы такой функции неочевидная, но не очень сложная: когда мы вызываем её, она отрабатывает до этого ключевого слова и, как ***return***, возвращает выражение под ***yield***. Однако на этом месте она не завершается, а как бы замораживается, после чего при последующем вызове этой функции продолжает работу со следующей после ***yield*** строчки

```
>>> def gfun(n):
...     for i in range(n):
...             yield i*2 + 1
...
>>> gfun
<function gfun at 0x7f7667f7a340>
>>> # С точки зрения питона - обычная функция
>>>
>>> fu = gfun(7)
>>> fu
<generator object gfun at 0x7f7667f60860>
>>> list(fu)
[1, 3, 5, 7, 9, 11, 13]
```

```
>>> # как оно работает
>>> def gfun(n):
...     for i in range(n):
...             print('BEFORE')
...             yield i*2 + 1
...             print('AFTER')
...
>>> fu = gfun(7)
>>> fu
<generator object gfun at 0x7f76680fac00>
>>> next(fu)
BEFORE
1
>>> next(fu)
AFTER
BEFORE
3
>>> next(fu)
AFTER
BEFORE
5
```

При “Заморозке” у генератор-функции сохраняются значения локальных параметров

```
>>> fu.gi_frame.f_locals
{'n': 7, 'i': 2}

>>> next(fu)
AFTER
BEFORE
7
>>> next(fu)
AFTER
BEFORE
9
>>> next(fu)
AFTER
BEFORE
11
>>> next(fu)
AFTER
BEFORE
13
>>> next(fu)
AFTER
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

При этом можно явно дописать return, чтобы досрочно выйти из генератора по какому-то условию, н-р

Дальше ещё немного зубодробительной штуки: ***yield from*** - вложенный итератор
По сути, мы просто рекурсивно проваливаемся в последовательность внутреннюю. У неё тоже могут быть свои ***yield***, которые будут нам наружу ретранслироваться. отработали внутреннюю часть - снова выпали во внешнюю

```
>>> def fun(n):
...     yield -1
...     yield from range(n)
...     yield -2
...
>>> list(fun(4))
[-1, 0, 1, 2, 3, -2]
```

```
>>> def it0(n):
...     for i in range(n):
...             yield i
...
>>> def fun(n):
...     yield -1
...     yield from it0(n)
...     yield -2
...
>>> list(fun(4))
[-1, 0, 1, 2, 3, -2]
```

Итератор, как мы уже выяснили, это, по сути своей, вычислимая последовательность, а значит может быть потенциально бесконечной

Ещё больше мозговыносяшек: можно передавать внутрь собственные параметры по *мере* работы генератора. Тк возвращение в генератор функцию происходит как бы через ***yield***, мы можем при возвращении передать в генератор какой-то объект, который внутри перехватить и запомнить. Происходит это с помощью метода ***send***. При первом вызове такого параметрического генератора мы как бы тоже вызываем ***send***, просто от пустого объекта ***None***. Так что первым вызовом можно делать и ***next()*** и ***send(None)***

```
>>> def bgfun():
...     res = 'Start'
...     while res:
...             res = yield f'/{res}/'
...     yield 'Finish'
...
>>> it = bgfun()
>>> next(it)
'/Start/'
>>> it.send(123)
'/123/'
>>> it.send('QQQ')
'/QQQ/'
>>> it.send('124')
'/124/'
>>> it.send(0)
'Finish'
>>> it.send(123)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> it = bgfun()
>>> it.send(None)
'/Start/'

>>> it = bgfun()
>>> it.send(23)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't send non-None value to a just-started generator
```

Кроме как отлавливать значения при работе генератора, можно также отлавливать параметры окончания работы генератора, для этого при окончании его работы должен быть прописан ***return***, который вместе со ***StopIteration*** вернёт и наш параметр

```
>>> def calc(n):
...     s = 0
...     while n > 0:
...         yield n
...         s += n
...         n //= 2
...     return s

>>> it  = calc(19)
>>> list(it) # Просто возвращаемые генератором значения. Где та самая сумма, которую мы считали?
[19, 9, 4, 2, 1]

>>> it = calc(19)
>>> next(it)
19
>>> next(it)
9
>>> next(it)
4
>>> next(it)
2
>>> next(it)
1
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration: 35 #Вот она - параметр при исключении по окончании работы итератора
```

Отлавливать значения можно не только логом об исключении, но и ручками через ***yield from***

```
>>> def runner(n):
...     res = yield from calc(n)
...     yield res
...
>>> list(runner(19))
[19, 9, 4, 2, 1, 35]
```

Самая масштабная библиотека для работы с итераторами это ***itertools***. Переберём немного её объектов

```
>>> import itertools

>>> c = itertools.count(10, 3) # count - генератор бесконечной последовательности от стартового параметра с заданным шагом
>>> c
count(10, 3)
>>> next(c)
10
>>> next(c)
13
>>> next(c)
16
>>> next(c)
19
>>> next(c)
22
>>> next(c)
25
>>> next(c)
28
>>> next(c)
31

>>> cc = itertools.cycle("QWE") # cycle - бесконечная последовательность из зацикленного итерируемого объекта
>>> next(cc)
'Q'
>>> next(cc)
'W'
>>> next(cc)
'E'
>>> next(cc)
'Q'
>>> next(cc)
'W'
>>> next(cc)
'E'
```

```
>>> r = itertools.repeat("123456") # repeat - повторяет заданный объект заданное число раз (по умолчанию бесконечно)
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'

>>> r = itertools.repeat("123456", 6)
>>> r
repeat('123456', 6)
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
'123456'
>>> next(r)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

>>> itertools.accumulate([1, 2, 3, 4, 5]) # accumulate - каждый следующий член последовательности есть заданная операция над предыдущими и текущим элементами (по умолчанию сумма)
<itertools.accumulate object at 0x7f7667f8ddc0>
>>> list(itertools.accumulate([1, 2, 3, 4, 5]))
[1, 3, 6, 10, 15]
>>> list(itertools.accumulate([1, 2, 3, 4, 5], int.__mul__)) # а здесь сделали умножение
[1, 2, 6, 24, 120]
```

```
>>> r = itertools.chain("QWE", itertools.repeat(3, 4)) # chain - объединяет итерируемые объекты в единую последовательность
>>> list(r)
['Q', 'W', 'E', 3, 3, 3, 3]

>>> s = itertools.compress('ABCDEF', [1,0,1,0,1,1]) # compress - относительно существования объектов второй последовательности (проверка на True-False) выдаёт объекты из первой
>>> list(s)
['A', 'C', 'E', 'F']

>>> f = itertools.filterfalse(lambda x: x%2, range(10)) # filterfalse - обратный встроенному методу генератор, оставляет лишь неподходящие под условие объекты последовательности
>>> list(f)
[0, 2, 4, 6, 8]

>>> d = itertools.dropwhile(lambda x: x<5, [1,4,6,4,1]) # dropwhile - выбрасывает из последовательности всё, пока не получит True на условии
>>> list(d)
[6, 4, 1]
>>> res = itertools.dropwhile(lambda x: x != "A", "DSAqwertghy")
>>> list(res)
['A', 'q', 'w', 'e', 'r', 't', 'g', 'h', 'y']

>>> res = itertools.takewhile(lambda x: x != "A", "DSAqwertghy") # аналогично, но, наоборот, оставляет всё до провала условия
>>> list(res)
['D', 'S']
>>> # takewhile(lambda x: x<5, [1,4,6,4,1]) --> 1 4
```

Неочевидный итератор ***tee*** - сделает из одного итератора несколько. ***tee*** запоминает разницу между итераторами, чтобы второй мог воспользоваться тем, что уже прошёл первый (итератор же одноразовый). Поэтому это оч тяжело по памяти

```
>>> res = itertools.tee(itertools.count())
>>> res
(<itertools._tee object at 0x7f7667f8df00>, <itertools._tee object at 0x7f7667f8e0c0>)
>>> r1, r2 = res
>>> next(r1)
0
>>> next(r1)
1
>>> next(r1)
2
>>> next(r1)
3
>>> next(r2)
0
>>> next(r2)
1
>>> next(r2)
2
>>> next(r2)
3
>>> next(r1)
4
```

```
>>> i = itertools.islice(itertools.count(10, 3), 5, 8) #islice - выдаёт кусочек итератора в границах
>>> i
<itertools.islice object at 0x7fa397e0b650>
>>> list(i)
[25, 28, 31]
```

```
>>> res = itertools.groupby("AAABBAAWWW") # groupby - создаёт последовательность пар, описывающий подпоследовательности объектов. Каждая пара - объект подпоследовательности и итератор этой самой подпоследовательности
>>> list(res)
[('A', <itertools._grouper object at 0x7f7667f4fa00>), ('B', <itertools._grouper object at 0x7f7667f80dc0>), ('A', <itertools._grouper object at 0x7f7667f814e0>), ('W', <itertools._grouper object at 0x7f7667f81060>)]
>>> [list(k) for k, g in itertools.groupby("AAABBAAWWW")]
[['A'], ['B'], ['A'], ['W']]
>>> [list(g) for k, g in itertools.groupby("AAABBAAWWW")]
[['A', 'A', 'A'], ['B', 'B'], ['A', 'A'], ['W', 'W', 'W']]
```

Комбинаторика в ***itertools***

>>> [(i, k) for i in "QWE" for k in "RTYU"] # способ перебора ручками
[('Q', 'R'), ('Q', 'T'), ('Q', 'Y'), ('Q', 'U'), ('W', 'R'), ('W', 'T'), ('W', 'Y'), ('W', 'U'), ('E', 'R'), ('E', 'T'), ('E', 'Y'), ('E', 'U')]

>>> list(itertools.product("QWE", "RTYU")) # product - то же самое декартово произведение (т.е. перебор всех со всеми) множеств
[('Q', 'R'), ('Q', 'T'), ('Q', 'Y'), ('Q', 'U'), ('W', 'R'), ('W', 'T'), ('W', 'Y'), ('W', 'U'), ('E', 'R'), ('E', 'T'), ('E', 'Y'), ('E', 'U')]

>>> list(itertools.permutations("QWRT", 2)) # permutations - Перестановки объектов без повторения с учётом позиционирования (по заданному числу объектов последовательности)
[('Q', 'W'), ('Q', 'R'), ('Q', 'T'), ('W', 'Q'), ('W', 'R'), ('W', 'T'), ('R', 'Q'), ('R', 'W'), ('R', 'T'), ('T', 'Q'), ('T', 'W'), ('T', 'R')]

>>> c = itertools.combinations('ABCD', 2) # combinations - комбинации без учёта позиционирования без повторений
>>> list(c)
[('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'C'), ('B', 'D'), ('C', 'D')]

>>> cwr = itertools.combinations_with_replacement('ABCD', 2) # combinations_with_replacement - комбинации с повторами
>>> list(cwr)
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'B'), ('B', 'C'), ('B', 'D'), ('C', 'C'), ('C', 'D'), ('D', 'D')]