Итераторы и генераторы.

for loop

Как работает цикл for?

```
a = range(5)
for i in a:
    print(i)

print("*" * 30)

# 0
# 1
# 2
# 3
# 4
```

for loop

Примерно следующим образом:

```
a = iter(a) # вместо it = a.__iter__()
# правильнее iter(a), т.к. iter кидает исключение TypeError,
# если нет __iter__, а не AttributeError
while True:
    try:
        i = next(i) # вместо i = it.__next__()
        # аналогично лучше использовать next, в него
        # же можно передать значение по-умолчанию, которое
        # нужно вернуть, если итератор кинул исключение
        # StopIteration, например, так:
        # next(it, None)
        print(i)
    except StopIteration:
        break
```

Итератор

Объекты, у которых есть метод `__iter__`, называют **iterable**.

Формально `__iter__` должен возвращать итератор (**iterator object**) — объект, реализующий магические методы `__next__` и `__iter__`.

Метод `__next__` обычно возвращает элементы итератора по одному. Если элементов больше нет, то бросает исключение `StopIteration`.

На практике итератор может не иметь метода `__iter__` и все будет работать без него, рекомендуется все-таки его реализовывать (*зачем?*).

Итерируемый стек

Пример стека, поддерживающего итерацию по нему:

```
class IterableStack:
    def __init__(self, lst):
        self.__lst = lst

    def push(self, value):
        self.__lst.append(value)

    def pop(self):
        return self.__lst.pop()

    ...

    def __iter__(self):
        return ReverseIterator(self.__lst)
```

Итерируемый стек

```
class ReverseIterator:
    def __init__(self, lst):
        self.__lst = lst
        self.position = len(lst)

    def __iter__(self):
        return self

    def __next__(self):
        self.position -= 1
        if self.position < 0:
            raise StopIteration

        return self.__lst[self.position]

container = IterableStack([1, 5, 0])
for v in container:
    ...
```

Самодостаточность

Могут ли объекты класса, для которого хотим поддержать итерацию, сами быть итераторами?

Самодостаточность

Могут ли объекты класса, для которого хотим поддержать итерацию, сами быть итераторами?

Да, могут. См. код на следующем слайде.

Самодостаточность

```
class CyclicRange:
    def __init__(self, n):
        self.pos = -1
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        if self.pos < self.n:
            self.pos += 1
            return self.pos

        self.pos = -1
        raise StopIteration
```

Самодостаточность

```
c = CyclicRange(3)
for i in c:
    print(i)

print(next(c, 'Empty c :()'))
for i in c:
    print(i)
```

Бесконечность

Поскольку значения из итератора вычисляются по одному, а не все сразу, можно написать и бесконечный итератор:

```
class InfiniteIterator:
    def __init__(self):
        self.pos = 0

    def __iter__(self):
        return self

    def __next__(self):
        self.pos += 1
        return self.pos
```

Бесконечность

```
it = iter(InfiniteIterator())

for i in range(5):
    print(next(it)) #?

# поделали что-то
print("AAA")

for i in range(5):
    print(next(it)) #?
```

Опустошение

При этом обычно итераторы конечные и не циклические, поэтому итераторы "опусташаются" (кидают `StopIteration` на любую попытку вызвать `__next__`)

```
a = range(10)
it = iter(a)
print(list(it))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(it))
# []

# но
print(list(a))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(list(a))
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Метод contains

Если объект итерируемый, то всегда возможно проверить наличие элемента в в нем, не определяя `__contains__`.

```
print(1 in IterableContainer([1, 5, 0]))
# True

# NB: будьте осторожны: не происходит никакой магии,
# просто итерируемся по контейнеру/итератору
# как следствие сложность O(n).
# По этой же причине:
it = iter(IterableStack([1, 5, 0]))
print(1 in it)
# True

print(1 in it)
# False
```

getitem

Вместо определения метода `__iter__` можно также определить `__getitem__` для индексов >= 0, for loop будет работать.

```
class Seq:
    def __init__(self, start, stop):
        ...
    def __getitem__(self, idx):
        if isinstance(idx, int) and \
                self.start + idx <=.stop:
            return self.start + idx
        raise IndexError(f"Index of {self} is wrong: {idx}")

for i in Seq(3, 7):
    print(i)
#?
```

sorted, max, min

В стандартной библиотеке есть функции, которые умеют работать с **iterable** объектами:

```
bears = [('polar', 450), ('giant panda', 100), ('brown', 80)]

# ищет максимум/минимум --
# max(iterable, *[, default=obj, key=func]) -> value
# min(--//--) -> value
max(bears, key=lambda x: x[1])
# ('polar', 450)

# сортировка, всегда возвращает список --
# sorted(iterable, /, *, key=None, reverse=False)
sorted(bears, key=lambda x: x[1])
# [('brown', 80), ('giant panda', 100), ('polar', 450)]
```

sum, any, all

```
# суммирует значения из iterable,
# start -- начальное значение
# подразумевается, что в iterable лежат только числа
# sum(iterable, start=0, /)
sum((1, 2, 3), start=-6)
# 0

# возвращает True, если для всех x из iterable: bool(x) == True
# all(iterable, /)
all([False, True])
# False

# возвращает True, если хотя бы для одного x из iterable
# bool(x) == True
# any(iterable, /)
any([False, True])
# True
```

map

```
bears = [('polar', 450), ('giant panda', 100), ('brown', 80)]

# возвращает iterable, который получается из исходного
# применением функции к каждому значению:
# map(function, iterable, ...)
list(map(lambda x: x ** 2, [0, 1, 2]))

for name in map(lambda x: x[0], bears):
    print(name, end=" ")
# polar giant panda brown

# в map можно передать функцию нескольких переменных и
несколько iterable
list(map(lambda x, y: x + y, [1, 2, 3], (2, 5))) #?
```

filter

```
bears = [('polar', 450), ('giant panda', 100), ('brown', 80)]

# возвращает iterable, который получен из исходного удалением
# всех элементов x для которых predicate(x) == False
list(filter(lambda x: x[1] >= 100, bears))
# [('polar', 450), ('giant panda', 100)]
```

zip, enumerate

```
a = ['a', 'b', 'c']
b = [25, 100]
c = [1, 1, 1, 1]

# zip принимает несколько последовательностей и возвращает
# iterable, в котором лежат кортежи;
# что будет, если последовательности разной длины?
for el_a, el_b, el_c in zip(a, b, c):
    print(el_a, el_b, el_c)

# enumerate также возвращает кортежи, но первым элементом в них
# лежит номер элемента
for idx, el in enumerate(a):
    print(idx, el)

# понятно, что enumerate выразим через zip и range. Как?
```

Генераторы

Писать итераторы долго: приходится определять целый класс.

Генераторы предоставляют синтаксис для краткой записи.

Генераторы

```
def _range(start, stop):
    """Генераторная функция _range описывает последовательность
    чисел от start до stop включительно."""
    assert start <= stop, f"start: {start} > stop: {stop}"
    while start <= stop:
        # при вызове next(_range(...)) код будет исполняться
        # до первого yield
        # yield вернет один элемент, после этого генератор
        # "засыпает" до следующего вызова  next
        yield start
        start += 1

for i in _range(5, 10):
    print(i)
#?
```

Генераторы

```
# создали генератор, он пока не был запущен
three_to_ten = _range(3, 10)
print(three_to_ten)
# <generator object _range at 0x11157aed0>

# попросили отработать до первого yield
print(next(three_to_ten))
# 3
# после чего генератор "заснул"

# снова попросили вычислиться до следующего yield:
print(next(three_to_ten))
# 4
# генератор снова "заснул"
```

Итерируемый стек

`__iter__` может быть генераторной функцией

```
class IterableStack:
    def __init__(self, lst):
        self.__lst = lst

    ...

    def __iter__(self):
        # for el in reversed(self.__lst):
        #    yield el
        # на самом деле
        yield from reversed(self.__lst)
```

Генераторы. chain

Напишем генераторную функцию, с помощью которой можно проитерироваться сразу по нескольким **iterable** объектам:

```
def poor_chain(*iterables):
    for iterable in iterables:
        for el in iterable:
            yield el

for el in poor_chain([10, 20], ("abc",)):
    print(el)
#?
```

Генераторы. chain

Напишем генераторную функцию, с помощью которой можно проитерироваться сразу по нескольким **iterable** объектам:

```
def chain(*iterables):
    for iterable in iterables:
        yield from iterable # тоже самое, но короче

for el in chain([10, 20], ("abc",)):
    print(el)
#?
```

Генераторные выражения

Еще короче можно записать генератор следующим образом:

```
gen = (x for x in range(10))
#      ^------------------^
# такое выражение называется генераторным

print(gen)
# <generator object <genexpr> at 0x11157ad50>

for i in gen:
    print(i)
```

Генераторные выражения

Генераторные выражения можно использовать для создания всевозможных встроенных коллекций

```
print([x for x in range(10)]) # list
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print({x for x in range(10)}) # set
# {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

print({x: str(x) for x in range(10)}) # dict
# тоже самое, но генератор пар
# print(dict( (x, str(x) for x in range(10)) ))
# {0: '0', 1: '1', 2: '2', 3: '3', 4: '4'}

print(tuple(x for x in range(10))) # tuple
# (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Генераторные выражения

Коллекции можно также создать с помощью * из генераторов

```
def seq():
    yield from range(10)

def pairs():
    yield from enumerate(map(str, range(5)))

print([*seq()]) # list
# [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print({*seq()})  # set
# {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

print(dict(pairs()))  # dict
# {0: '0', 1: '1', 2: '2', 3: '3', 4: '4'}
```

Модуль itertools

Модуль `itertools` содержит различные функции для работы с итераторами.

```
from itertools import count, cycle, product

# count -- бесконечный счетчик
it = count(start=1, step=10)
for i in range(10):
    print(next(it))

# cycle -- циклический итератор:
it = cycle('ABCD')
print(''.join([next(it) for i in range(10)]))
```

Модуль itertools

```
from itertools import product, permutations

# product -- декартово произведение
print([i for i in product("AA", "BB")])

# permutations -- перестановки
print([i for i in permutations("ABCD", 2)])
```

Модуль itertools

```
from itertools import dropwhile, islice, chain, takewhile, tee
# dropwhile/takewhile
# islice -- срез
print(list(islice(
    dropwhile(lambda x: x < 100, count(10, 10)),
    30)))

# chain -- итератор из нескольких
print(list(chain(product("AA", "BB"), product("AA", "BB"))))

# tee -- "раздвоить" итется посмотреть
# как tee работает?
it = takewhile(lambda x: x < 100, count(10, 10))
it1, it2 = tee(it)
print(list(it1)) #?
print(list(it1)) #?
print(list(it2)) #?
```

Модуль itertools

```
from itertools import starmap

def _starmap(function, iterable):
    for args in iterable:
        yield function(*args)

print(*_starmap(pow, [(2,5), (3,2), (10,3)]))
# почти тоже самое
# print(*starmap(pow, [(2,5), (3,2), (10,3)]))
# 32 9 1000
```

Что почитать

Рекомендуется посмотреть документацию ко всем функциям, о которых шла речь на лекции:

- built-in functions
- itertools