Компьютерная математика II

Дисциплина для студентов 1-го курса специальности «Компьютерная математика и системный анализ» и профилизации "Искусственный интеллект и математическая экономика" ММФ БГУ

Тема 5. Механизм итераций в Python

доц. Лаврова О.А., доц. Щеглова Н.Л., кафедра дифференциальных уравнений и системного анализа (ауд. 329)

март, 2025

5.1 Итерируемый объект и объект итератора

В Python существует специальный **механизм итераций**, который осуществляет итерационный проход по элементам коллекций (и не только коллекций).

Итерационный проход по элементам -- это последовательное извлечение элементов из коллекции для выполнения вычислений с применением текущего элемента коллекции.

Для реализации механизма итераций необходимо наличие двух сущностей: *итерируемого объекта* и *объекта итератора*.

Итерируемые объекты в Python -- это объекты встроенных типов, относящихся к коллекциям (`str` , `list` , `tuple` , `dict` , `set`), а также *генераторные объекты*.

Генераторные объекты называют "виртуальными последовательностями" или потоками данных, они *генерируют/создают* по запросу по одному объекту за итерацию, становясь на паузу, пока не будет запрошен следующий объект.

Файловый объект является примером генераторного объекта.

Генераторный объект является результатом выполнения *генераторного выражения* или *генераторной функции* (Раздел 5.3). Генераторный объект является также результатом выполнения встроенных функций-конструкторов `range` , `map` , `filter` , `zip` , `enumerate` (Тема 6).

In [1]:
```
range(1,2).__class__
```

Out[1]:
```
range
```

Элементами итерируемых объектов являются:

- элементы объектов коллекций (`str` , `list` , `tuple` , `dict` , `set`);

- значения, генерируемые по запросу, для генераторных объектов.

Например, для файлового объекта значением, генерируемым по запросу, является строка файла. Для функции-конструктора `range` значением, генерируемым по запросу, является целое число. И т.д.

Итерируемый объект в Python -- это любой объект, который может создать объект итератора, или просто итератор.

Объект итератора предоставляет интерфейс для доступа к элементам итерируемого объекта. Использование объекта итератора разделяет итерируемый объект и проход по итерируемому объекту.

Создание объекта итератора для итерируемого объекта осуществляется с помощью метода `__iter__`

In [2]:
```
[obj.__iter__() for obj in [str(), list(), tuple(), dict(), set([1]), range(5)]]
```

Out[2]:
```
[<str_ascii_iterator at 0x21ca6264250>,
 <list_iterator at 0x21ca6267160>,
 <tuple_iterator at 0x21ca6264df0>,
 <dict_keyiterator at 0x21ca3717880>,
 <set_iterator at 0x21ca65aa9c0>,
 <range_iterator at 0x21ca659edd0>]
```

Объект итератора для итерируемого объекта `obj` можно создать также с помощью встроенной функции `iter(obj)`

In [3]:
```
[iter(obj) for obj in [str(), list(), tuple(), dict(), set([1]), range(5)]]
```

Out[3]:
```
[<str_ascii_iterator at 0x21ca650b040>,
 <list_iterator at 0x21ca650a020>,
 <tuple_iterator at 0x21ca650b310>,
 <dict_keyiterator at 0x21ca37175b0>,
 <set_iterator at 0x21ca65ab180>,
 <range_iterator at 0x21ca659ec90>]
```

Некоторые итерируемые объекты одновременно являются и объектами итераторами. Например, файловый объект является объектом итератором

In [4]:
```
f = open("tmp.txt",'rt')
f is iter(f)
```

Out[4]:
```
True
```

In [5]:
```
r = list(range(5)); e = enumerate(range(5)); z = zip(range(5),range(5))
r is iter(r), e is iter(e), z is iter(z)
```

Out[5]:
```
(False, True, True)
```

Объект итератора в Python -- это объект, который требует реализации единственного метода `__next__()` для получения следующего элемента итерируемого объекта.

In [6]:
```
str1 = "test"
iterator = iter(str1)
[iterator.__next__() for _ in str1]
```

Out[6]:
```
['t', 'e', 's', 't']
```

Итерационный проход по элементам итерируемого объекта `obj` можно также осуществить с помощью функции `next`, аргументом которой является объект итератора для итерируемого объекта `obj`. Например, `next(iter(obj))`

In [7]:
```
rng = range(1,10,2)
iterator = iter(rng)
[next(iterator) for _ in rng]
```

Out[7]:
```
[1, 3, 5, 7, 9]
```

Объект итератора генерирует исключение `StopIteration` при попытке перехода с помощью метода `__next__` (или функции `next`) на элемент за пределами итерируемого объекта

In [8]:
```
rng = range(1,10,2)
iterator = rng.__iter__()
print([next(iterator) for _ in rng])
next(iterator)
```

```
[1, 3, 5, 7, 9]
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[8], line 4
      2 iterator = rng.__iter__()
      3 print([next(iterator) for _ in rng])
----> 4 next(iterator)

StopIteration:
```

Для итерируемого объекта, который является коллекцией, можно создать несколько независимых друг от друга объектов итераторов

In [9]:
```
obj = (1,2,3,4)
I1= iter(obj); I2 = iter(obj)
next(I1), next(I1), next(I2)
```

Out[9]:
```
(1, 2, 1)
```

Аналогичное справедливо также для функции `range`

In [10]:
```
obj = range(5)
I1= iter(obj); I2 = iter(obj)
next(I1), next(I1), next(I2)
```

Out[10]:
```
(0, 1, 0)
```

Для файлового объекта может быть определен только один объект итератор. Аналогичное справедливо также для генераторных объектов, кроме генераторного объекта, который возвращает функция `range`

In [11]:
```
obj = zip(range(3),range(3))
I1= iter(obj); I2 = iter(obj)
next(I1), next(I1), next(I2)
```

Out[11]:
```
((0, 0), (1, 1), (2, 2))
```

In [12]:
```
obj = enumerate(range(3,6))
I1= iter(obj); I2 = iter(obj)
next(I1), next(I1), next(I2)
```

Out[12]:
```
((0, 3), (1, 4), (2, 5))
```

Объект итератора предоставляет возможность для итерационного прохода по итерируемому объекту только в одну сторону.

Объект итератора не имеет методов

- для получения предыдущего элемента итерируемого объекта;
- для перевода итератора на начальное значение итерируемого объекта;
- для создания копии итератора.

Итерационный проход по элементам итерируемого объекта можно выполнять ручным и *автоматическим* способами.

Для ручного выполнения итераций используют методы `__iter__` и `__next__` или функции `iter` и `next`

In [13]:
```
str1 = "test"
iterator = iter(str1)
next(iterator), iterator.__next__(), next(iterator), next(iterator)
```

Out[13]:
```
('t', 'e', 's', 't')
```

Для *автоматического* выполнения итераций используют *итерационные инструменты*.

К итерационным инструментам относятся: цикл `for`, включения, операция проверки принадлежности `in`, присваивание последовательностей, конструкторы коллекций `str`, `list`, `tuple`, `dict`, `set`, встроенные функции `map`, `filter`, `zip`, `enumerate`, `sorted`, `sum`, `min`, `max`, `any`, `all`, генераторные функции, генераторные выражения.

Ручное выполнение итераций:

In [14]:
```
str1 = "test"
iterator = iter(str1)
next(iterator), iterator.__next__(), next(iterator), next(iterator)
```

Out[14]:
```
('t', 'e', 's', 't')
```

Автоматическое выполнение итераций:

In [15]:
```
tuple(x for x in str1)
```

Out[15]:
```
('t', 'e', 's', 't')
```

5.2 Итерационные инструменты

Итерационные инструменты в Python – это операторы, выражения и функции, создающие и использующие объект итератора для выполнения итерационных процессов на основе итерируемых объектов.

Итерационный инструмент выполняет определенные действия согласно *протоколу итерации*.

Итерационные инструменты запускают протокол итераций *автоматически*. Итерируемые объекты поддерживают протокол итераций.

Протокол итераций

Итерационный инструмент выполняет следующие действия:

1. *сначала* вызывает метод `__iter__()` итерируемого объекта для получения объекта итератора,
2. *на каждой итерации* вызывает метод `__next__()` объекта итератора,
3. *для окончания итераций* перехватывает исключение `StopIteration`, которое создает объект итератора.

К итерационным инструментам относятся: цикл `for`, включения, операция проверки принадлежности `in`, присваивание последовательностей, конструкторы коллекций `str`, `list`, `tuple`, `dict`, `set`, встроенные функции `map`, `filter`, `zip`, `enumerate`, `sorted`, `sum`, `min`, `max`, `any`, `all`, генераторные функции, генераторные выражения.

Включения считаются самым распространенным итерационным инструментом в Python

In [16]:
```
iterable = "ABCD"

[x.lower() for x in iterable], {ord(x) for x in iterable}
```

Out[16]:
```
(['a', 'b', 'c', 'd'], {65, 66, 67, 68})
```

In [17]:
```
dict1 = {1: 10, "1": "20", (1,): (30,)}
print("Итерационный проход по ключам словаря")
for x in dict1:
    print(x, end=", ")

print("\n Итерационный проход по значениям словаря")
for x in dict1.values():
    print(x, end=", ")

print("\n Итерационный проход по элементам словаря")
```

```
for i, x in dict1.items():
    print(f'{i} -> {x}', end=", ")
```

```
Итерационный проход по ключам словаря
1, 1, (1,),
 Итерационный проход по значениям словаря
10, 20, (30,),
 Итерационный проход по элементам словаря
1 -> 10, 1 -> 20, (1,) -> (30,),
```

Операция проверки принадлежности `in`, встроенные функции `any` и `all` являются итерационными инструментами

In [18]:
```
?any
```

```
Signature: any(iterable, /)
Docstring:
Return True if bool(x) is True for any x in the iterable.

If the iterable is empty, return False.
Type:      builtin_function_or_method
```

In [19]:
```
5 in range(5), any(range(5)), all(range(5))
```

Out[19]: (False, True, False)

In [20]:
```
all(x>4 for x in range(6)), any(x>4 for x in range(6))
```

Out[20]: (False, True)

Примеры использования двух и более итерационных инструментов в одном выражении

In [21]:
```
iterable = "ABCD"
sum(ord(x) for x in iterable)
```

Out[21]: 266

In [22]:
```
# проверка включения элементов второго итерируемого объекта в первый итерируемы
first_str = 'Strin'
second_str = 'tg'

contains_all = all(elem in first_str for elem in second_str)
contains_all
```

Out[22]: False

5.3 Генераторный объект

У множества итерируемых объектов существует подмножество, которое называют генераторными объектами.

Генераторный объект -- это итерируемый объект, ВСЕ элементы которого НЕ хранятся в памяти, а последовательно *генерируются*/создаются при вызове метода `__next__` объекта итератора.

Генераторный объект может быть получен тремя способами: вычислением генераторного выражения, определением и вызовом генераторной функции или вызовом встроенных функций-конструкторов, таких как `range`, `map`, `filter`, `zip`, `enumerate`.

Генераторное выражение

Генераторное выражение является обобщением спискового включения. Генераторное выражение синтаксически определяется как включение, записанное в круглых скобках.

Синтаксический шаблон генераторного выражения:

```
(expr(elem) for elem in iterable)
```

In [23]:
```
gen_obj = (x+'0' for x in "string")
gen_obj
```

Out[23]:
```
<generator object <genexpr> at 0x0000021CA687FD30>
```

Результатом выполнения генераторного выражения является генераторный объект.

In [24]:
```
for x in gen_obj:
    print(x*2)
```

```
s0s0
t0t0
r0r0
i0i0
n0n0
g0g0
```

Генераторный объект -- это итерируемый объект, который одновременно является объектом итератором. При этом не любой объект итератора является генераторным объектом. Поэтому понятие итератора является более общим, чем понятие генератора.

In [25]:
```
iter(gen_obj) is gen_obj
```

Out[25]:
```
True
```

Итерационный проход по элементам генераторного объекта можно сделать только один раз, так как объект итератора создается в единственном экземпляре

In [26]:
```
gen_obj = (x**2 for x in range(5))
I1= iter(gen_obj); I2 = iter(gen_obj)
next(I1), next(I1), next(I2), next(gen_obj)
```

Out[26]:
```
(0, 1, 4, 9)
```

Вычисление выражения `x**2` генераторного выражения `gen_obj` осуществляется только при вызове метода `__next__` вместо единовременного создания и хранения в памяти ВСЕХ элементов коллекции.

Когда генераторное выражение является единственным элементом, который необходимо дополнительно поместить в круглые скобки, то одну пару круглых скобок можно не указывать

In [27]:
```
sum(x**2 for x in range(4))
```

Out[27]: 14

In [28]:
```
import math
?math.dist
sum((x-y)**2 for x,y in zip(range(4),range(1,5)))
```

Out[28]: 4

```
Signature: math.dist(p, q, /)
Docstring:
Return the Euclidean distance between two points p and q.

The points should be specified as sequences (or iterables) of
coordinates.  Both inputs must have the same dimension.

Roughly equivalent to:
    sqrt(sum((px - qx) ** 2.0 for px, qx in zip(p, q)))
Type:      builtin_function_or_method
```

Генераторная функция

Определение генераторной функции следует тем же синтаксическим правилам, что и определение функции с помощью оператора `def`, за единственным исключением, что вместо оператора `return` в теле функции используется оператор `yield` (с версии Python 2.4) или оператор `yield from` (с версии Python 3.3) для возвращения объекта.

Генераторная функция -- это функция, которая генерирует/создает последовательность объектов с использованием оператора `yield`.

In [29]:
```
def gen(n):
    for i in range(n):
        yield i**2
```

Результатом вызова генераторной функции является генераторный объект, а не объекты, создаваемые при выполнении оператора `yield`

In [30]:
```
gen_obj = gen(10)
gen_obj
```

Out[30]: `<generator object gen at 0x0000021CA6832F60>`

Связано это с тем, что при вызове генераторной функции код тела функции не выполнется. Для выполнения кода генераторной функции необходимо дополнительно реализовать процесс итерационного прохода по элементам генераторного объекта.

У генераторных объектов методы `__iter__` и `__next__` создаются средствами самого языка, то есть автоматически. Программисту их определять не надо, что упрощает создание пользовательских типов для итерируемых объектов.

Процесс итерационного прохода по элементам генераторного объекта можно реализовать ручным и автоматическим способами

In [31]:
```
print(gen_obj.__next__(), next(gen_obj))
```

```
0 1
```

In [32]:
```
[x for x in gen_obj]
```

Out[32]:
```
[4, 9, 16, 25, 36, 49, 64, 81]
```

В случае реализации итерационного процесса с применением генераторного объекта, созданного в результате вызова генераторной функции, оператор `yield` работает следующим образом.

При вызове метода `__next__` для генераторного объекта оператор `yield` *выдает* объект-значение вызывающему коду и *приостанавливает* выполнение генераторной функции на операторе `yield` до следующей итерации. Генераторный объект встает "на паузу" до следующей итерации. При этом сохраняется состояние вызванной функции (локальная область видимости, состояние переменных, местоположение оператора `yield` и т.д.), чтобы предоставить функции возможность *возобновить* выполнение кода функции после последнего выполнения оператора `yield` при следующей итерации. На следующей итерации выполнение продолжится до очередного выполнения оператора `yield` .

In [33]:
```
def gen():
    yield 1; yield 2; yield 3

[x for x in gen()]
```

Out[33]:
```
[1, 2, 3]
```

Окончание итерационного процесса с применением генераторного объекта, созданного на основе генераторной функции, либо наступает по окончанию выполнения полного кода генераторной функции либо с выполнением оператора `return` (с версии Python 3.3)

In [34]:
```
def gen():
    yield 1; yield 2
    return 3; yield 4

[x for x in gen()]
```

Out[34]:
```
[1, 2]
```

Альтернативой использования комбинации `for` + `yield` является использование `yield from` +генераторное выражение

In [35]:
```
def gen(n):
    for i in range(n):
        yield i**2

def gen(n):
    yield from (i**2 for i in range(n))
```

Генераторные выражения и генераторные функции позволяют при одиночном вызове не возвращать всю коллекцию объектов, а возвращать по одному объекту из коллекции *по запросу*.

При выполнении генераторного выражения, а также при вызове генераторной функции *время ожидания* результата распределяется между генерациями отдельных объектов коллекции. Выполнение генераторного выражения, а также вызов генераторной функции не приводит к ожиданию генерации всей коллекции объектов. Это особенно полезно, когда генерируются коллекции больших размеров, а также когда для получения каждого возвращаемого объекта требуются длительные вычисления.

Использование генераторных выражений и генераторных функций также обеспечивает *оптимизацию расхода памяти* из-за отсутствия необходимости хранения всей коллекции объектов.

Совет: прдпочитайте генераторы коллекциям, если Вам не нужны все элементы коллекции сразу, чтобы сэкономить на потреблении памяти.

Пример 1. Бесконечный генератор чисел Фибоначчи

In [36]:
```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
```

In [37]:
```
fib_gen = fib(); fib_gen
```

Out[37]:
```
<generator object fib at 0x0000021CA6887510>
```

In [38]:
```
[next(fib_gen) for _ in range(15)]
```

Out[38]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]
```

Пример 2. Двойное итерирование по кортежу, состоящему из итерируемых объектов

In [39]:
```
iterables = ("ABC", [1,2,3], (4,5), {6:1, 7:2}, {8,9,10})

def chain(iterables):
    for iterable in iterables:
        for x in iterable:
            yield x
```

In [40]:
```
generator = chain(iterables)
generator
```

Out[40]:
```
<generator object chain at 0x0000021CA68875E0>
```

In [41]:
```
for x in generator:
    print(x, end=" ")
```

```
A B C 1 2 3 4 5 6 7 8 9 10
```

Альтернативная реализация

In [42]:
```
# old version
def chain(iterables):
    for iterable in iterables:
        for x in iterable:
            yield x

# new version
def chain(iterables):
    yield from (x for iterable in iterables for x in iterable)
```

In [43]:
```
for x in chain(iterables):
    print(x, end=" ")
```

```
A B C 1 2 3 4 5 6 7 8 9 10
```