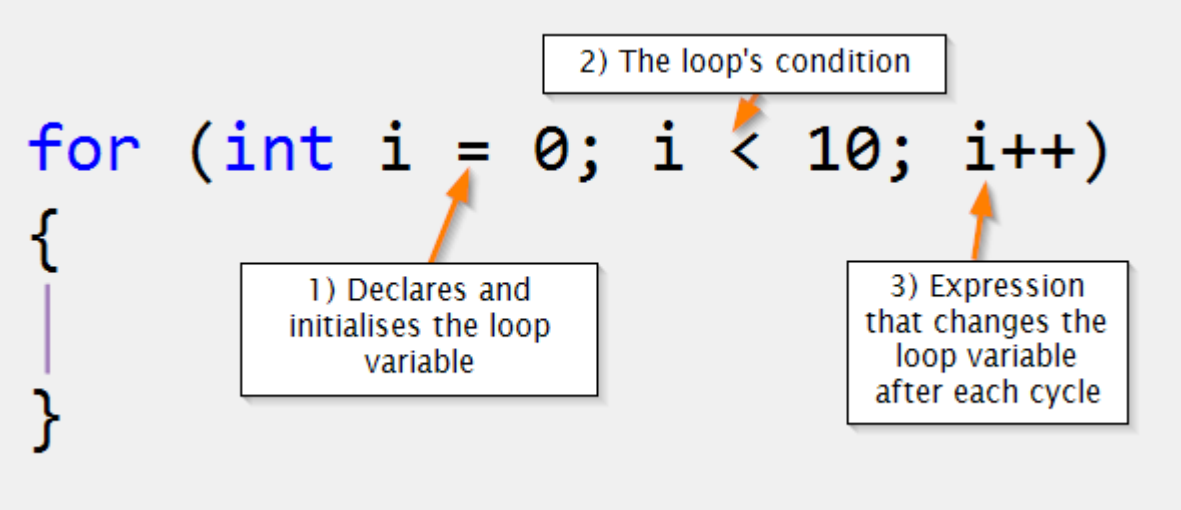
Итераторы, генераторы

цикл for в Python



for в C++

цикл for, в Python, устроен несколько иначе, чем в большинстве других языков

он больше похож на for...each, или же for...of

перебор коллекций

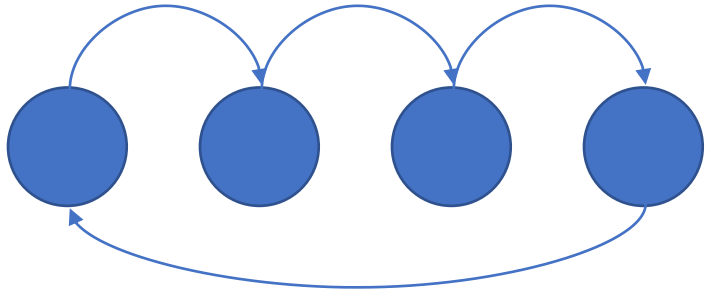
часто появляется задача перебора элементов коллекции:

- для доступа к содержимому коллекции без раскрытия их внутреннего представления

перебор коллекций

часто появляется задача перебора элементов коллекции:

- для доступа к содержимому коллекции без раскрытия их внутреннего представления
- для поддержки нескольких активных обходов одного и того же агрегированного объекта (желательно)



перебор коллекций

часто появляется задача перебора элементов коллекции:

- для доступа к содержимому коллекции без раскрытия их внутреннего представления
- для поддержки нескольких активных обходов одного и того же агрегированного объекта (желательно)
- для предоставления единообразного интерфейса с целью обхода различных агрегированных структур

перебор коллекций

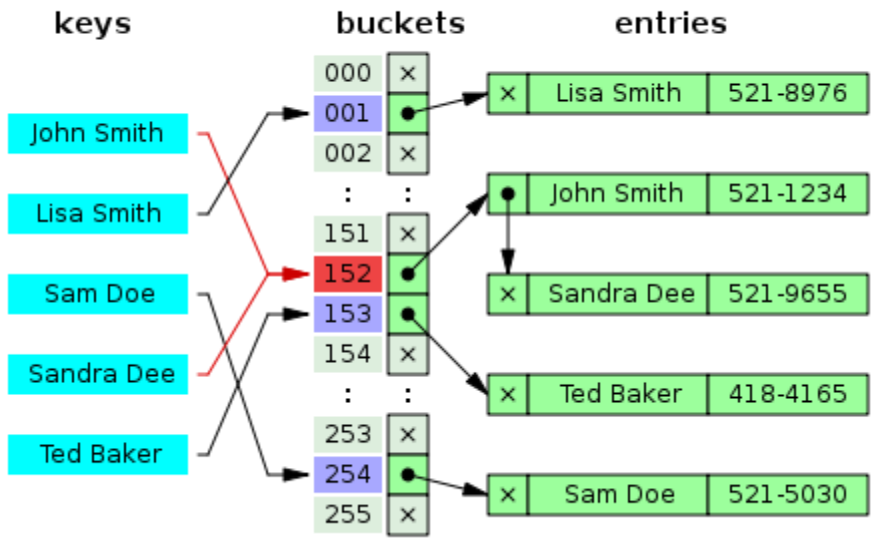
часто появляется задача перебора элементов коллекции:

- для доступа к содержимому коллекции без раскрытия их внутреннего представления
- для поддержки нескольких активных обходов одного и того же агрегированного объекта (желательно)
- для предоставления единообразного интерфейса с целью обхода различных агрегированных структур
- теоретически, структура может быть бесконечной + ленивые вычисления, память

Например, **аксиомы Пеано** и **функция следования** $S(x)$ для определения множества натуральных чисел

$$N = \{1, 2, 3, 4, 5, 6, ...\}$$

простите, что?



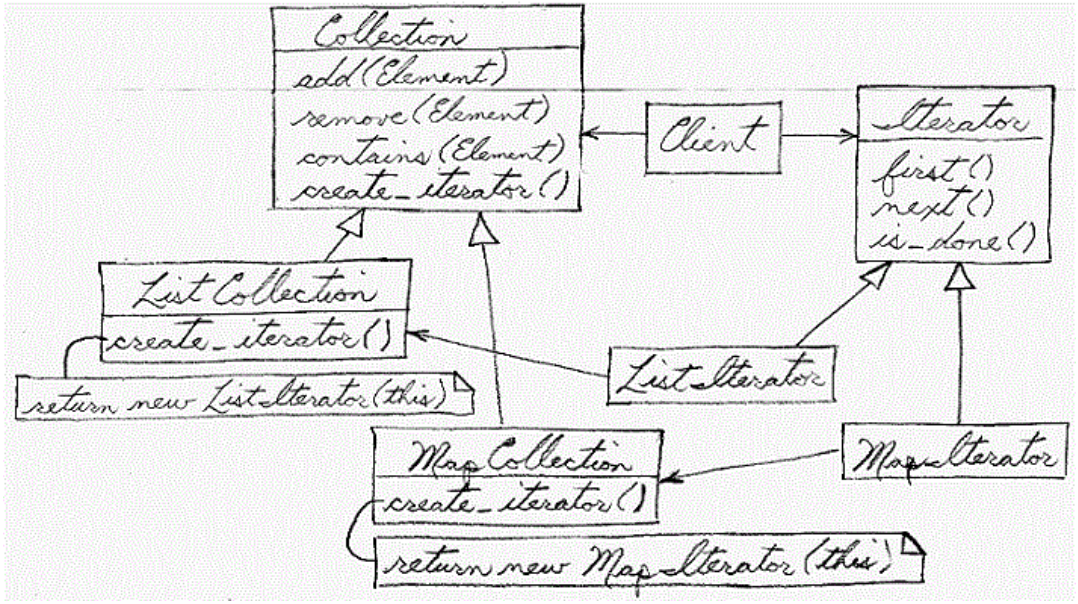
у меня есть словарь

```
>>> a_dict = {'color': 'blue', 'fruit': 'apple', 'pet': 'dog'}
>>> for item in a_dict.items():
...     print(item)
...
('color', 'blue')
('fruit', 'apple')
('pet', 'dog')
>>> for key in a_dict.keys():
...     print(key)
...
color
fruit
pet
```

я не хочу знать, как всё внутри устроено!!!

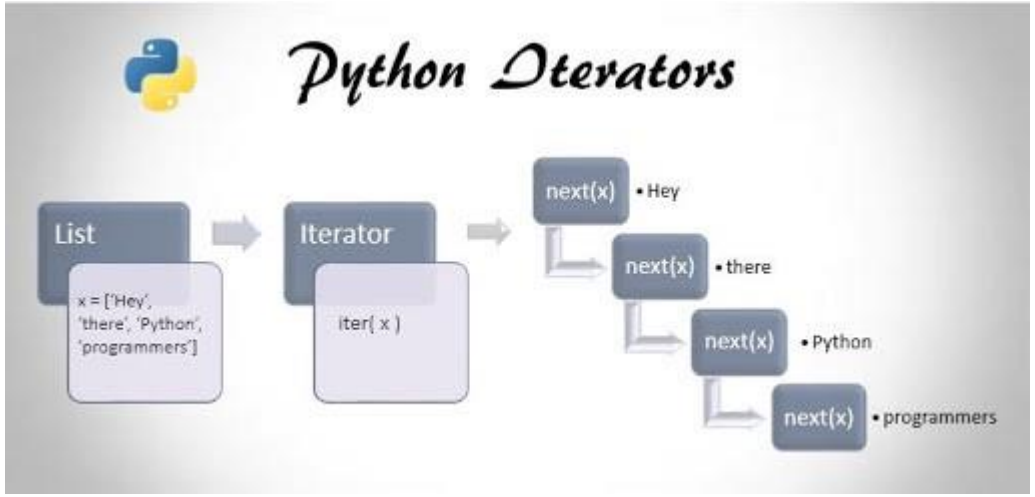
я хочу сделать for по ключам или элементам словаря!!!!

итераторы, 1995 г.



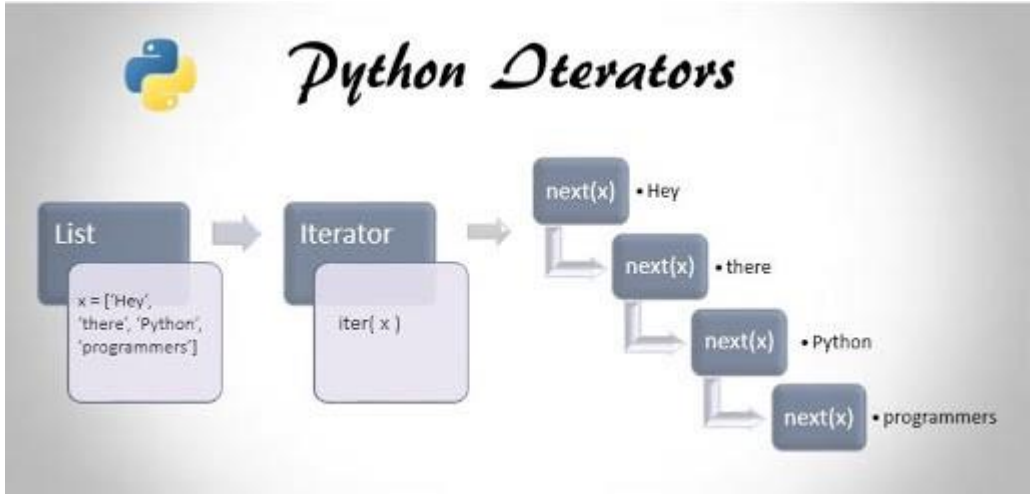
- «Банда четырёх» в программировании (англ. *Gang of Four*, сокращённо GoF) — распространённое название группы авторов (Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес), выпустивших в 1995 году известную книгу **Design Patterns** о шаблонах проектирования.

как работают?



- у итерируемого объекта должен быть метод **__iter__**, который создаст и вернёт итератор
- у итератора должен быть реализован метод **__next__**
- **__next__** будет вызываться на каждой итерации цикла **for**

как работают?



- т.е. итерируемый объект (***iterable***) — это любой объект, предоставляющий возможность поочерёдного прохода по своим элементам, а итератор (***iterator***) — это то, что выполняет реальный проход

пример __iter__ и __next__ в списках

Когда вы пишете:

```
for i in [1,2,3,4]:
    ...
```

Python неявно вызывает итератор ***__iter__***(*[1,2,3,4]*),
создавая итератор ***ListIterator***[1,2,3,4]

пример __iter__ и __next__ в списках

```
class ListCollection(collections.abc.Iterable):

    def __init__(self, collection):

        self._collection = collection


    def __iter__(self):

        return ListIterator(self._collection, -1)
```

```
class ListIterator(collections.abc.Iterator):

    def __init__(self, collection, cursor):

        self._collection = collection

        self._cursor = cursor


    def __next__(self):

        if self._cursor + 1 >= len(self._collection):

            raise StopIteration()

        self._cursor += 1

        return self._collection[self._cursor]
```

зачем (за что) мне это?

модуль ***itertools*** содержит много полезных итераторов:

- ***itertools.chain(it1, it2)*** для объединения 2х итераторов
- ***itertools.count(n)*** бесконечно выдаёт (n+1), (n+2), …
- ***itertools.cycle(l)*** бесконечно повторяет последовательность l

десятки их, изучайте!

хороши тем, что тратят мало памяти
запоминают, на каком месте остановился проход
не позволяют пройтись по ним второй раз

зачем (за что) мне это?

zip(*iterables) – берёт итерируемые объекты и выдаёт итератор, состоящий из упорядоченных n-ок

enumerate(iterable) – берёт итерируемые объекты и выдаёт итератор из пар (индекс элемента, элемент коллекции)

range(n) в Python 2 и 3 не возвращает итератор!
Возвращает итерируемый *range*-объект.

Характеристика списка или всеохватывающее описание списка

(англ. ***list comprehension***)

List comprehension

способ компактного описания операций обработки списков вида:

$$[x \mid x \in L, f(x) - \text{истинно}]$$

1985 г., язык *Miranda*:

[n | n <- [1..]; n rem 2 = 0] - список всех n, таких что n входит в [1..] и остаток от деления n на 2 равен нулю

Python: *[f(x)* ***for*** *x* ***in*** *iterator* ***if*** *condition(x)]*

List comprehension

“синтаксический сахар”, позволяющий быстро и удобно создавать списки

например, список чётных чисел – элементов списка *l*

```
>>> l = range(9)
>>> list(l)
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> comreh_list = [x for x in l if x % 2 == 0]
>>> comreh_list
[0, 2, 4, 6, 8]
```

или список их квадратов

```
>>> comreh_list_2 = [x**2 for x in l if x % 2 == 0]
>>> comreh_list_2
[0, 4, 16, 36, 64]
```

List comprehension

использование *list comprehension* – это “*Pythonic way*” и функциональный стиль программирования

Генераторы

generator expressions

используя в *list comprehension* круглые скобки, мы получим его “ленивую” реализацию – итератор, каждое значение которого будут вычисляться только при вызове функции *next*

например, на каждой итерации цикла *for*

пример

я хочу получить список вообще всех чётных чисел

```
>>> import itertools
>>> it = itertools.count(1)
>>> l = [x for x in it if x % 2 == 0]

>>> _
```

этот код, очевидно, повесит мой пк

пример

в отличие от генератора, который будет выполняться столько, сколько я захочу

```
>>> import itertools
>>> it = itertools.count(1)
>>> l = (x for x in it if x % 2 == 0)
>>> next(l)
2
>>> next(l)
4
>>> next(l)
6
>>> next(l)
8
>>> next(l)
10
>>> for i in l:
...     print(i)
...
12
14
16
18
20
```

заметьте, что *for* продолжает перебирать созданный генератором итератор и начинает не с числа 2, а с 12

синтаксический сахар для …?

list comprehension был синтаксическим сахаром для списков

generator expressions – синтаксический сахар для функций-генераторов, использующих вместо кл. слова *return* кл. слово *yield*

сравнение

list comprehension

```
L1 = [n ** 2 for n in range(12)]

L2 = []
for n in range(12):
    L2.append(n ** 2)

print(L1)
print(L2)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121]
```

generator expression

```
G1 = (n ** 2 for n in range(12))

def gen():
    for n in range(12):
        yield n ** 2

G2 = gen()
print(*G1)
print(*G2)
```

```
0 1 4 9 16 25 36 49 64 81 100 121
0 1 4 9 16 25 36 49 64 81 100 121
```

задание

1. *выведите первые 10 квадратов натуральных чисел с помощью:*
   1. *generator expression*
   2. *generator function*

      * с помощью *for* или *next*

2. с помощью *generator function* реализуйте числа Фиббоначи