Понимание итераторов в Python

https://habr.com/ru/articles/488112/

Python*
Из песочницы

Python — особенный язык в плане итераций и их реализации, в этой статье мы подробно разберём устройство итерируемых объектов и пресловутого цикла for.

Особенности, с которыми вы часто можете столкнуться в повседневной деятельности

1. Использование генератора дважды

```
>>> numbers = [1,2,3,4,5]

>>> squared_numbers = (number**2 for number in numbers)

>>> list(squared_numbers)
[1, 4, 9, 16, 25]

>>> list(squared_numbers)
[]
```

Как мы видим в этом примере, использование переменной squared_numbers дважды, дало ожидаемый результат в первом случае, и, для людей незнакомых с Python в достаточной мере, неожиданный результат во втором.

2. Проверка вхождения элемента в генератор

Возьмём всё те же переменные:

```
>>> numbers = [1,2,3,4,5]
>>> squared_numbers = (number**2 for number in numbers)
```

А теперь, дважды проверим, входит ли элемент в последовательность:

```
>>> 4 in squared_numbers
True
>>> 4 in squared_numbers
False
```

Получившийся результат также может ввести в заблуждение некоторых программистов и привести к ошибкам в коде.

3. Распаковка словаря

Для примера используем простой словарь с двумя элементами:

```
>>> fruits_amount = {'apples': 2, 'bananas': 5}
```

Распаковываем его:

```
>>> x, y = fruits_amount
```

Результат будет также неочевиден, для людей, не понимающих устройство Python, "под капотом":

```
>>> x
'apples'
>>> y
'bananas'
```

Последовательности и итерируемые объекты

По-сути, вся разница, между последовательностями и итерируемымыи объектами, заключается в том, что в последовательностях элементы упорядочены.

Так, последовательностями являются: списки, кортежи и даже строки.

```
>>> numbers = [1,2,3,4,5]
>>> letters = ('a','b','c')
>>> characters = 'habristhebestsiteever'
>>> numbers[1]
2
>>> letters[2]
'c'
>>> characters[11]
's'
>>> characters[0:4]
'habr'
```

Итерируемые объекты же, напротив, не упорядочены, но, тем не менее, могут быть использованы там, где требуется итерация: цикл for, генераторные выражения, списковые включения — как примеры.

```
# Can't be indexed
>>> unordered_numbers = {1,2,3}
>>> unordered_numbers[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: 'set' object is not subscriptable

>>> users = {'males': 23, 'females': 32}
>>> users[1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 1

# Can be used as sequence
>>> [number**2 for number in unordered_numbers]
[1, 4, 9]
>>>
>>> for user in users:
...     print(user)
...
males
females
```

Отличия цикла for в Python от других языков

Стоит отдельно остановиться на том, что цикл for, в Python, устроен несколько иначе, чем в большинстве других языков. Он больше похож на for...each, или же for...of.

Если же, мы перепишем цикл for с помощью цикла while, используя индексы, то работать такой подход будет только с последовательностями:

```
>>> list_of_numbers = [1,2,3]
>>> index = 0
>>> while index < len(list_of_numbers):
...     print(list_of_numbers[index])
...     index += 1
...
1
2
3
```

А с итерируемыми объектами, последовательностями не являющимися, не будет:

```
>>> set_of_numbers = {1,2,3}
>>> index = 0
>>> while index < len(set_of_numbers):
...     print(set_of_numbers[index])
...     index += 1
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: 'set' object is not subscriptable
```

Если же вам нужен index, то следует использовать встроенную функцию enumerate:

```
>>> set_of_numbers = {1,2,3}
>>> for index, number in enumerate(set_of_numbers):
...     print(number, index)
...
1 0
2 1
3 2
```

Цикл for использует итераторы

Как мы могли убедиться, цикл for не использует индексы. Вместо этого он использует так называемые **итераторы**.

Итераторы — это такие штуки, которые, очевидно, можно итерировать :)
Получить итератор мы можем из любого итерируемого объекта.

Для этого нужно передать итерируемый объект во встроенную функцию iter:

```
>>> set_of_numbers = {1,2,3}
>>> list_of_numbers = [1,2,3]
>>> string_of_numbers = '123'
>>>
>>> iter(set_of_numbers)
<set_iterator object at 0x7fb192fa0480>
>>> iter(list_of_numbers)
<list_iterator object at 0x7fb193030780>
>>> iter(string_of_numbers)
<str_iterator object at 0x7fb19303d320>
```

После того, как мы получили итератор, мы можем передать его встроенной функции next.

```
>>> set_of_numbers = {1,2,3}
>>>
>>> numbers_iterator = iter(set_of_numbers)
>>> next(numbers_iterator)
1
>>> next(numbers_iterator)
2
```

При каждом новом вызове, функция отдаёт один элемент. Если же в итераторе элементов больше не осталось, то функция next породит исключение StopIteration.

```
>>> next(numbers_iterator)
3
>>> next(numbers_iterator)
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
StopIteration
```

По-сути, это единственное, что мы может сделать с итератором: передать его функции next.
Как только итератор становится пустым и порождается исключение StopIteration, он становится совершенно бесполезным.

Реализация цикла for с помощью функции и цикла while

Используя полученные знания, мы можем написать цикл for, не пользуясь самим циклом for. :)

Чтобы сделать это, нам нужно:

Получить итератор из итерируемого объекта.
Вызвать функцию next.
Выполнить 'тело цикла'.
Закончить цикл, когда будет получено исключение StopIteration.

```
def for_loop(iterable, loop_body_func):
    iterator = iter(iterable)
    next_element_exist = True
    while next_element_exist:
        try:
            element_from_iterator = next(iterator)
        except StopIteration:
            next_element_exist = False
        else:
            loop_body_func(element_from_iterator)
```

Стоит заметить, что здесь мы использовали конструкцию try-else. Многие о ней не знают. Она позволяет выполнять код, если исключения не возникло, и код был выполнен успешно.

Теперь мы знакомы с **протоколом итератора**.
А, говоря простым языком — с тем, как работает итерация в Python.
Функции iter и next этот протокол формализуют. Механизм везде один и тот же. Будь то пресловутый цикл for или генераторное выражение. Даже распаковка и "звёздочка" используют **протокол итератора**:

```
coordinates = [1,2,3]
x, y, z = coordinates

numbers = [1,2,3,4,5]
a,b, *rest = numbers

print(*numbers)
```

Генераторы — это тоже итераторы

Генераторы тоже реализуют протокол итератора:

```
>>> def custom_range(number):
...     index = 0
...     while index < number:
...             yield index
...             index += 1
...
>>> range_of_four = custom_range(4)
>>> next(range_of_four)
0
>>> next(range_of_four)
1
>>> next(range_of_four)
2
>>> next(range_of_four)
3
>>> next(range_of_four)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

В случае, если мы передаём в `iter` итератор, то получаем тот же самый итератор

```
>>> numbers = [1,2,3,4,5]
>>> iter1 = iter(numbers)
>>> iter2 = iter(iter1)
>>> next(iter1)
1
>>> next(iter2)
2
>>> iter1 is iter2
True
```

Подытожим.

Итерируемый объект — это что-то, что можно итерировать.
Итератор — это сущность порождаемая функцией `iter`, с помощью которой происходит итерирование **итерируемого объекта**.

Итератор не имеет индексов и может быть использован только один раз.

Протокол итератора

Теперь формализуем протокол итератора целиком:

Чтобы получить итератор мы должны передать функции `iter` итерируемый объект.
Далее мы передаём итератор функции `next`.
Когда элементы в итераторе закончились, порождается исключение `StopIteration`.

Особенности:

Любой объект, передаваемый функции `iter` без исключения `TypeError` — итерируемый объект.
Любой объект, передаваемый функции `next` без исключения `TypeError` — итератор.
Любой объект, передаваемый функции `iter` и возвращающий сам себя — итератор.

Плюсы итераторов:

Итераторы работают "лениво" (en. **lazy**). А это значит, что они не выполняют какой-либо работы, до тех пор, пока мы их об этом не попросим.

Таким образом, мы можем **оптимизировать потребление ресурсов ОЗУ и CPU**, а так же **создавать бесконечные последовательности**.

Итераторы повсюду

Мы уже видели много итераторов в Python.
Я уже упоминал о том, что генераторы — это тоже итераторы.
Многие встроенные функции является итераторами.

Так, например, `enumerate`:

```
>>> numbers = [1,2,3]
>>> enumerate_var = enumerate(numbers)
>>> enumerate_var
<enumerate object at 0x7ff975dfdd80>
>>> next(enumerate_var)
(0, 1)
```

А так же `zip`:

```
>>> letters = ['a','b','c']
>>> z = zip(letters, numbers)
>>> z
<zip object at 0x7ff975e00588>
>>> next(z)
('a', 1)
```

И даже open:

```
>>> f = open('foo.txt')
>>> next(f)
'bar\n'
>>> next(f)
'baz\n'
>>>
```

В Python очень много итераторов, и, как уже упоминалось выше, они откладывают выполнение работы до того момента, как мы запрашиваем следующий элемент с помощью next. Так называемое, "ленивое" выполнение.

Создание собственного итератора

Так же, в некоторых случаях, может пригодится знание того, как написать свой собственный итератор и **ленивый** итерируемый объект.

В моей карьере этот пункт был ключевым, так как вопрос был задан на собеседовании, которое, как вы могли догадаться, я успешно прошёл и получил свою первую работу:)

```
class InfiniteSquaring:
"""Класс обеспечивает бесконечное последовательное возведение в квадрат заданного числа."""
    def __init__(self, initial_number):
        # Здесь хранится промежуточное значение
        self.number_to_square = initial_number

    def __next__(self):
        # Здесь мы обновляем значение и возвращаем результат
        self.number_to_square = self.number_to_square ** 2
        return self.number_to_square

    def __iter__(self):
        """Этот метод позволяет при передаче объекта функции iter возвращать самого себя, тем самым в точности реализуя протокол итератора."""
        return self


>>> squaring_of_six = InfiniteSquaring(6)
>>> next(squaring_of_six)
36
>>> next(squaring_of_six)
1296
>>> next(squaring_of_six)
1679616
>>> next(squaring_of_six)
2821109907456
>>> next(squaring_of_six)
```

```
795866110994640088439193 6
>>> # И так до бесконечности...
```

Так же:

```
>>>iter(squaring_of_six) is squaring_of_six
True
```

Таким образом мы написали **бесконечный** и **ленивый** итератор.
А это значит, что ресурсы он будет потреблять только при вызове.
Не говоря уже о том, что без собственного итератора имлементация бесконечной последовательности была бы невозможна.

А теперь вернёмся к тем особенностям, которые были изложены в начале статьи

1. Использование генератора дважды

```
>>> numbers = [1,2,3,4,5]

>>> squared_numbers = (number**2 for number in numbers)

>>> list(squared_numbers)
[1, 4, 9, 16, 25]

>>> list(squared_numbers)
[]
```

В данном примере, список будет содержать элементы только в первом случае, потому что генераторное выражение — это итератор, а итераторы, как мы уже знаем — сущности одноразовые. И при повторном использовании не будут отдавать никаких элементов.

2. Проверка вхождения элемента в генератор

```
>>> numbers = [1,2,3,4,5]
>>> squared_numbers = (number**2 for number in numbers)
```

А теперь дважды проверим, входит ли элемент в последовательность:

```
>>> 4 in squared_numbers
True
>>> 4 in squared_numbers
False
```

В данном примере, элемент будет входить в последовательность только 1 раз, по причине того, что проверка на вхождение проверяется путем перебора всех элементов

последовательности последовательно, и как только элемент обнаружен, поиск прекращается. Для наглядности приведу пример:

```
>>> 4 in squared_numbers
True
>>> list(squared_numbers)
[9, 16, 25]
>>> list(squared_numbers)
[]
```

Как мы видим, при создании списка из генераторного выражения, в нём оказываются все элементы, после искомого. При повторном же создании, вполне ожидаемо, список оказывается пуст.

3. Распаковка словаря

При использовании в цикле for, словарь будет отдавать ключи:

```
>>> fruits_amount = {'apples': 2, 'bananas': 5}

>>> for fruit_name in fruits_amount:
...     print(fruit_name)
...
apples
bananas
```

Так как распаковка опирается на тот же протокол итератора, то и в переменных оказываются именно ключи:

```
>>> x, y = fruits_amount
>>> x
'apples'
>>> y
'bananas'
```

Выводы

Последовательности — итерируемые объекты, но не все итерируемые объекты — последовательности.

Итераторы — самая простая форма итерируемых объектов в Python.

Любой итерируемый объект реализует протокол итератора. Понимание этого протокола — ключ к пониманию любых итераций в Python.

Теги:
python

iteration
generators
sequences
Хабы:
Python