1 Итераторы в Python

Неформально, 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* (или т.н. Dunder* метод .__*next*__()). При каждом новом вызове, функция отдаёт один элемент. Если же в итераторе элементов больше не осталось, то функция next породит исключение *StopIteration*.

```
set_of_numbers = {1,2,3}

numbers_iterator = iter(set_of_numbers)
next(numbers_iterator)
# 1
next(numbers_iterator)
# 2
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 можем сделать с итератором: передать его функции *next*. Как только итератор становится пустым ("exhausted") порождается исключение *StopIteration*, и он становится совершенно бесполезным.

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.

Можем возникнуть вопрос. Как тогда мы можем несколько раз подряд запускать цикл for?(можно вспомнить как мы реализовывали цикл for через цикл while и итераторы, посмотрев в jupyter ноутбуке с пары или на кусок кода чуть ниже). Дело в том, что под капотом каждый раз создаётся новый итератор с помощью *iter*.

Ниже привидена реализация цикла for через цикл while (как устроен for под капотом):

```
years_iterator = iter(years)
running = True

while running:
    try:
        year = next(years_iterator)
        print(year)
    except:
        running = False
```

1.1 Протокол итератора

Теперь формализуем протокол итератора целиком:

1. Чтобы получить итератор мы должны передать функции *iter* итерируемый объект. Итерируемый объект - это буквально объект, имеющий соответствующий dunder метод .$\_\_iter\_\_$(), который аналогичен функции *iter*.

2. Далее мы передаём итератор функции *next* (или вызываем dunder метод .$\_\_next\_\_$() от итератора).

3. Когда элементы в итераторе закончились, порождается исключение *StopIteration*.

Ещё раз обозначим особенности:

1. Любой объект, передаваемый функции *iter* без исключения (т.е. без ошибки) $TypeError$ — итерируемый объект.
2. Любой объект, передаваемый функции *next* без исключения $TypeError$ — итератор.
3. Любой объект, передаваемый функции *iter* и возвращающий сам себя — итератор.

Зачем же нужны итераторы?
Плюсы итераторов:

1. Итераторы работают "лениво"(lazy). А это значит, что они не выполняют какой-либо работы, до тех пор, пока мы их об этом не попросим.
2. Таким образом, мы можем оптимизировать потребление ресурсов ОЗУ и CPU, а так же создавать бесконечные последовательности.

Мы уже видели много итераторов и генераторов в Python. Я уже упоминал о том, что генераторы — это тоже итераторы, но с ещё одной особенностью: к последовательности применяется какая-то функция. В Python любой генератор является итератором (но обратное неверно). Про генераторы подробнее мы поговорим позже.

Многие встроенные функции является итераторами. Примеры встроенных итераторов/генераторов:

1. функция range для создания последовательностей
2. функция enumerate для создания последовательности кортежей из индексов и значений
3. generator expression, например (x * x for x in range(10))
4. функция map для применения функции к последовательности
5. функция open для работы с файлами

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.

1.2 Dunder методы

В Python dunder методы - это методы, которые позволяют экземплярам класса взаимодействовать со встроенными (built-in) функциями и операторами языка. Слово "dunder"происходит от "double underscore потому что имена методов dunder начинаются и заканчиваются двумя подчеркиваниями, например $\_\_str\_\_$ или $\_\_add\_\_$.

Что такое класс и экземпляр класса мы поговорим позже, пока можете под этим понимать самый обыкновенный объект в питоне. Например, переменная $my_friends = $['Егор', 'Дима', 'Максим'] это экземпляр класса список, т.е. конкретная реализация списка. И у этого экземпляра класса (то бишь конкретная реализация списка или для простоты просто список) есть dunder метод $.\_\_iter\_\_()$, который позволит создать соответствующий итератор.

Как правило, dunder методы не вызываются программистом напрямую, что создает впечатление, будто они вызываются по волшебству. Именно поэтому методы dunder иногда называют «магическими методами».

Однако методы Dunder не вызываются магически. Они просто вызываются языком неявно, обычно внутри других методов, в определенные моменты времени, которые четко определены и зависят от конкретного dunder метода.

Стоит запомнить, что Dunder методы чаще всего являются техническими методами, которые созданы не для того, чтобы пользователь взаимодействовал с ними напрямую. Тем не менее, он всегда это может сделать, особенно если ему необходимо достать что-то "из глубины"объекта.

1.3 Источники вдохновления:

1. https://habr.com/ru/articles/488112/
2. https://mathspp.com/blog/pydonts/dunder-methods