Итераторы , генераторы и декораторы

Итераторы и итерируемые объекты

`list`, `tuple`, `dict` и `sets` — это все итерируемые объекты. Они являются итерируемыми контейнерами, из которых вы можете получить итератор. Все эти объекты имеют метод `__iter__()`, который используется для получения итератора.В Python есть встроенные функции iter() и next(), которые соответственно вызывают методы `__iter__()` и `__next__()` объектов, переданных в качестве аргумента

In [30]:
```
a = [1, 2]
print(type(a))
b = a.__iter__()
print(b.__next__())
print(b.__next__())
```

```
<class 'list'>
1
2
```

У итерируемого объекта нет метода `__next__()`, который используется при итерации:

In [9]:
```
a.__next__()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-9-d34d2a8e0899> in <module>
----> 1 a.__next__()

AttributeError: 'list' object has no attribute '__next__'
```

Метод `__next__()` исчерпанного итератора вызывает исключение StopIteration.

In [19]:
```
print(b.__next__())
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
<ipython-input-19-cb2e1653435a> in <module>
----> 1 print(b.__next__())

StopIteration:
```

Внутренний механизм цикла `for` сначала вызывает метод **iter()** объекта. Так что, если передан итерируемый объект, создается итератор. После этого применяется метод **next()** до тех пор, пока не будет возбуждено исключение StopIteration.

Поскольку метод **iter()** итератора возвращает сам итератор, то после перебора циклом `for` объект исчерпывается. То есть получить данные из итератора можно только один раз. В случае с коллекциями это не так. Здесь создается другой объект - итератор. Он, а не итерируемый объект, отдается на обработку циклу `for`.

In [22]:
```
a = range(2)
b = iter(a)

print(type(a))
for i in a:
    print(i)

print(type(b))
for i in b:
    print(i)
```

```
<class 'range'>
0
1
<class 'range_iterator'>
0
1
```

Итерируемый объект — это любой объект, от которого встроенная функция iter() может получить итератор.

Итератор в python — это любой объект, реализующий метод `__next__` без аргументов, который должен вернуть следующий элемент или ошибку StopIteration. Также он реализует метод `__iter__` и поэтому сам является итерируемым объектом.

In [34]:
```
class MyNumbers:
    def __iter__(self):
        self.a = 1
        return self

    def __next__(self):
        if self.a <= 5:
            x = self.a
            self.a += 1
            return x
        else:
            raise StopIteration

myclass = MyNumbers()
myiter = iter(myclass)
for x in myiter:
    print(x)
```

```
1
2
3
4
5
```

Генераторы

Генератор в Python — это языковая конструкция, которую можно реализовать двумя способами: как функция с ключевым словом `yield` или как генераторное выражение. В результате вызова функции или вычисления выражения, получаем объект-генератор. В объекте-генераторе определены методы `__next__` и `__iter__`, то есть реализован протокол итератора, с этой точки зрения, в Python любой генератор является итератором.

Простой генератор не используя объект-генератор

In [76]:
```
class FibonacciGenerator:
    def __init__(self):
        self.prev = 0
        self.cur = 1

    def __next__(self):
        result = self.prev
        self.prev, self.cur = self.cur, self.prev + self.cur
        return result

    def __iter__(self):
        return self

for i in FibonacciGenerator():
    print(i)
    if i > 5:
        break
```

```
0
1
1
2
3
5
8
```

Но используя ключевое слово `yield` можно сильно упростить реализацию:

In [77]:
```
def fibonacci():
    prev, cur = 0, 1
    while True:
        yield prev
        prev, cur = cur, prev + cur

for i in fibonacci():
    print(i)
    if i > 5:
        break
```

```
0
1
1
2
3
5
8
```

Любая функция в Python, в теле которой встречается ключевое слово `yield`, называется генераторной функцией — при вызове она возвращает объект-генератор. Объект-генератор реализует интерфейс итератора, соответственно с этим объектом можно работать, как с любым другим итерируемым объектом.

In [80]:
```
f=fibonacci()
print(next(f))
print(next(f))
print(next(f))
print(next(f))
```

```
0
1
1
2
```

Генераторное выражение (generator expression)

Генераторное выражение это синтаксически более короткий способ создать генератор, не определяя и не вызывая функцию. А так как это выражение, то у него есть и ряд ограничений.В основном удобно использовать для генерации коллекций, их несложных преобразований и применений на них условий.

In [90]:
```
a=(i for i in range(10))
b=[i for i in range(10)]

print(type(a))
print(type(b))

print(next(a))
print(next(a))
print(next(a))
```

```
<class 'generator'>
<class 'list'>
0
1
2
```

Декораторы

Декоратор — это функция, которая позволяет обернуть другую функцию для расширения её функциональности без непосредственного изменения её кода.

Здесь `decorator_function()` является функцией-декоратором. Как вы могли заметить, она является функцией высшего порядка, так как принимает функцию в качестве аргумента, а также возвращает функцию. Внутри `decorator_function()` мы определили другую функцию, обёртку, которая обёртывает функцию-аргумент и затем изменяет её поведение. Декоратор возвращает эту обёртку.

In [114]:
```
def decorator_function(func):
    def wrapper():
        print('Функция-обёртка!')
        print('Оборачиваемая функция: {}'.format(func))
        print('Выполняем обёрнутую функцию...')
        func()
        print('Выходим из обёртки')
    return wrapper
```

Иными словами, выражение `@decorator_function` вызывает `decorator_function()` с `hello_world` в качестве аргумента и присваивает имени `hello_world` возвращаемую функцию. Вместо `hello_world = decorator_function(hello_world)`

In [116]:
```
@decorator_function
def hello_world():
    print('Hello world!')

hello_world()
```

```
Функция-обёртка!
Оборачиваемая функция: <function hello_world at 0x04EAC540>
Выполняем обёрнутую функцию...
Hello world!
Выходим из обёртки
```

Декоратор, замеряющий время выполнения функции, которая делает GET-запрос к главной странице Google. Чтобы измерить скорость, мы сначала сохраняем время перед выполнением обёрнутой функции, выполняем её, снова сохраняем текущее время и вычитаем из него начальное.

In [113]:
```
def benchmark(func):
    import time

    def wrapper():
        start = time.time()
        func()
        end = time.time()
        print('[*] Время выполнения: {} секунд.'.format(end-start))
    return wrapper

@benchmark
def fetch_webpage():
    import requests
    webpage = requests.get('https://google.com')

fetch_webpage()
```

```
[*] Время выполнения: 0.7383291721343994 секунд.
```

In [ ]:
```
Декораторы с аргументами
```

In [134]:
```
def benchmark(iters):
    def actual_decorator(func):
        import time

        def wrapper(*args, **kwargs):
            total = 0
            for i in range(iters):
                start = time.time()
                return_value = func(*args, **kwargs)
                end = time.time()
                total = total + (end-start)
            print('[*] Среднее время выполнения: {} секунд.'.format(total/iters))


        return wrapper
    return actual_decorator


@benchmark(iters=10)
def fetch_webpage(url):
    import requests
    webpage = requests.get(url)
    return webpage.text

webpage = fetch_webpage('https://vk.com')
print(webpage)
```

```
[*] Среднее время выполнения: 1.1918861389160156 секунд.
None
```

Несколько декораторов для одной функции.

In [132]:
```
def bread(func):
    def wrapper():
        print("хлеб")
        func()
        print("хлеб")
    return wrapper

def ingredients(func):
    def wrapper():
        print("#помидорыя")
        func()
        print("~салат~")
    return wrapper

@bread
@ingredients
def sandwich(food="ветчина"):
    print(food)

sandwich()
```

```
хлеб
#помидорыя
ветчина
~салат~
хлеб
```