Итераторы, генераторы, декораторы

Итераторы

- Итератор - объект, имеющий метод __next__ и реализующий корректное поведение при его использовании.
- Метод __next__ должен иметь следующую семантику: вернуть текущий объект, а следующий сделать текущим.
- Если текущего объекта нет, то вызвать исключение StopIteration.

In [16]:

```
class SomeIter(object):

    def __iter__(self):
        return self

    def __next__(self):
        raise StopIteration()

print([val for val in SomeIter()])
```

```
[]
```

In [17]:

```
class ContainerIter(object):
    def __init__(self, cont):
        self.cont = cont

    def __next__(self):
        raise StopIteration()

class Container(object):
    def __iter__(self):
        return ContainerIter(self)

print([val for val in Container()])
```

```
[]
```

Цикл for и итерирование

- for вызывает функцию iter(obj) от переданного ему объекта после ключевого слова in.
- В результате вызова должен быть получен объект итератор.
- Далее for использует next(), пока не будет вызвано исключение StopIteration.
- Чтобы добавить в свой класс возможность итерирования по нему следует определить магический метод __iter__ так, чтобы он возвращал итератор.
- Таким образом, мы теперь можем называть объект iterable, когда в нём дана возможность получения итератора и итерирования.
- Кроме этого, объект ещё iterable, когда у него есть __getitem__ способный принимать последовательные индексы начиная с 0 и бросающий IndexError при окончании промежутка (так у строк).

Почитать про итераторы:

- PEP про итераторы - https://www.python.org/dev/peps/pep-0234/

Генераторы

- Генератор - простой механизм создания итераторов.
- Выглядит как обычная функция, но вместо return используется ключевое слово yield.
- __iter__ и next() создаются для генератора автоматически.
- При каждом вызове next() генератор продолжает выполняться с момента остановки (первый раз - начало функции, остальные - последний yield).
- Промежуточное состояние сохраняется автоматически между вызовами.
- После окончания выполнения (внутри функции дошли до конца) автоматически выбрасывается StopIteration.

In [18]:

```
def counter():
    val = 1
    for i in range(10):
        yield val
        val = val + 1

print(type(counter), type(counter()))
print(dir(counter()))
```

```
print([val for val in counter()])
```

```
<class 'function'> <class 'generator'>
['__class__', '__del__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getatt
ribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__lt__',
'__name__', '__ne__', '__new__', '__next__', '__qualname__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'close', 'gi_code', 'gi_frame', 'gi_running',
'gi_yieldfrom', 'send', 'throw']
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Генераторные выражения

- Записываются полностью как list comprehensions, только вместо квадратных скобок - круглые.
- Генераторное выражение возвращает генератор.
- Используется более короткая, но и более ограниченная форма записи.

In [19]:

```
gen = (val for val in [1, 2, 3])
print(gen)
print([val for val in gen])
print([val for val in gen])
```

```
<generator object <genexpr> at 0x7f078d600580>
[1, 2, 3]
[]
```

Iterator vs Iterable vs Generator

Декораторы

- Декоратор - способ задания изменения поведения некоторой функции.
- По сути, представляет собой функцию, которая принимает и возвращает функции.
- Задаются просто созданием подходящей функции.
- Применяются записью перед телом функции после символа @.

In [20]:

```
def multiply_result(func):

    def multiplier(*args, **kwargs):
        return -1 * func(*args, **kwargs)

    return multiplier

@multiply_result
def get_random_value():
    return 4

def get_one_more_random_value():
    return 4

print(get_random_value())
print(get_one_more_random_value())
```

```
-4
4
```

Декорирование через @ - это так называемый синтаксический сахар.

Задекорировать можно и другим способом:

In [21]:

```
def get_five():
    return 5

print(get_five())

get_five = multiply_result(get_five)

print(get_five())
```

```
5
-5
```

Некоторые встроенные декораторы

- @classmethod - делает из функции т.н. классовый метод.
- @staticmethod - делает из функции статический метод класса.

In [22]:

```
class SomeClass(object):

    @classmethod
    def print_name(cls):
        print(cls.__name__)

class ChildClass(SomeClass):
    pass

SomeClass.print_name()
ChildClass.print_name()
```

```
SomeClass
ChildClass
```

In [23]:

```
class SomeOtherClass(object):

    @staticmethod
    def print_hello():
        print("Hello!")

SomeOtherClass.print_hello()
```

```
Hello!
```

Параметризированный декоратор

In [24]:

```
def parameterized_decorator(name):
    print("Hello from parameterized decorator")

    def my_decorator(f):
        print(f"Hello from main decorator code ! My name is {name}")

        def wrapped(*args, **kwargs):
            print("Hello from wrapped func")
            return f(*args, **kwargs)
        return wrapped
    return my_decorator

@parameterized_decorator("abacaba")
def mul(x, y):
    return x * y

print(mul(3, 4))
```

```
Hello from parameterized decorator
Hello from main decorator code ! My name is abacaba
Hello from wrapped func
12
```

Можно декорировать как функции, так и декораторы (да-да, они тоже функции)

In [27]:

```
def parameterized_decorator(decorator):
    print("Hello")
    def decorator_maker(*args, **kwargs):
        def wrapper(func):
            return decorator(func, *args, **kwargs)
        return wrapper
    return decorator_maker
```

In [34]:

```
@parameterized_decorator
def decorated_decorator(f, *args, **kwargs):
    print(f"Decorator arguments: {args}, {kwargs}")
    def wrapper(*func_args, **func_kwargs):
        print(f"Function arguments: {func_args}, {func_kwargs}")
        return f(*func_args, **func_kwargs)
    return wrapper

@decorated_decorator("Abacaba")
```

```
def sum(x, y):
    return x + y

print(sum(0, y=42))
print("#" * 42)
print("Another one call")
print(sum(0, 42))
```

```
Hello
Decorator arguments: ('Abacaba',), {}
Function arguments: (0,), {'y': 42}
42
##########################################
Another one call
Function arguments: (0, 42), {}
42
```

Примеры использования

- таймеры выполнения
- логгер вызова функций и их параметров
- кеш для вызова функций

Что почитать и изучить

- https://www.python.org/dev/peps/pep-0318/
- http://thecodeship.com/patterns/guide-to-python-function-decorators/
- https://habr.com/ru/post/141411/
- https://habr.com/ru/post/141501/
- functools.wraps