PYTHON

Лекция 11

ПЛАН ЛЕКЦИИ

- Генераторные выражения
- Декораторы

ГЕНЕРАТОРНЫЕ ВЫРАЖЕНИЯ

- Есть списочные выражения
- Есть множественные и словарные
- А кортежных - нет
- Потому что круглые скобки отдали генераторным

ПРИМЕР

```
1 data = (v for v in range(10000000000000000000000000000))
2 data = (v for v in data if v % 7)
3 data = (v >> 2 for v in data)
4 for _ in range(10):
5     print(next(data))
```

ХОРОШИЙ СТИЛЬ

- Хороший стиль - выражать логику в терминах пребразований над последовательностями
- (numpy - отдельная история)
- Если последовательности небольшие - лучше использовать списочные выражения
- Если есть риск, что большие или бесконечные - генераторные выражения

ОСОБЫЙ СИНТАКСИС

- Есть функции, принимающие коллекции
- Примеры: `all, any, map, filter`
- Они могут принимать ленивые коллекции, в том числе генераторы
- Формально надо писать одну пару скобок для обозначения вызова
- И еще одну - для обозначения ленивого генератора
- Можно обойтись одной парой

ПРИМЕР

```
 1 def take(data, n):
 2     data_iter = iter(data)
 3     for _ in range(n):
 4         yield next(data_iter)
 5
 6 max_word_size = max(len(w) for w in input().split())
 7
 8 with open('data.txt') as f:
 9     numbered = enumerate(s.strip() for s in f)
10     selected = (index for index, w in numbered
11                 if len(w.split() > max_word_size))
12     head = take(selected, 10)
13     print(' '.join(str(v) for v in head))
```

РАБОТА С БОЛЬШИМИ ФАЙЛАМИ

- Можно открыть большой файл
- И использовать его в ленивом генераторе
- Или в генераторном выражении
- Но у файла как у источника есть особенность
- Его надо закрыть

ВАРИАНТЫ

- Вариант 1: уверены, что дочитаем до конца
- Ленивость нужна, чтобы не зачитывать все в память
- И чтобы оптимизировать операции над коллекцией
- Отличное решение - функция с `yield`

ПРИМЕР

```
1 def lazy_read(fname):
2     with f = open(fname):
3         while True:
4             s = f.readline()
5             if not s:
6                 return
7
8
9 # ....................................
```

ПРИМЕР

```
# ....................................


data = lazy_read(fname)
data = (v.strip() for v in data)
data = map(int, data)
data = (v * 2 for v in data % 10 == 4)
for e in data:
    print(e)

# и тут файл закрывается
```

ВАРИАНТЫ

- Вариант 2: можем рано решить, что хватит
- Не хотим дочитывать до конца, но файл закрыть
- Есть механизм передачи значения в генератор
- Метод `send`, значение выйдет как результат `yield`

ПРИМЕР

```
 1 def lazy_lines(name):
 2     with open(name, 'rt') as f:
 3         while True:
 4             line = f.readline()
 5             if not line:
 6                 return
 7             cmd = yield line
 8             print('cmd', cmd)
 9             if cmd == 'stop':
10                 return
11
12 # ........................
```

ПРИМЕР

```
 1 # ........................
 2
 3 f = lazy_lines('_11.py')
 4 mapped = map(str.strip, f)
 5 print(next(mapped))
 6 print(next(mapped))
 7 print(next(mapped))
 8 print(next(mapped))
 9
10 f.send('stop')
```

CLOSE

- `send` - изначально для универсального взаимодействия с генератором
- Для закрытия есть более специфичный инструмент - `close`
- Можно вызвать и породить в генераторе исключение `GeneratorExit`
- В самом генераторе обработать исключение
- Сильно нежелательно его подавлять

ПРИМЕР

```
 1 def lazy_lines(name):
 2     try:
 3         with open(name, 'rt') as f:
 4             while True:
 5                 line = f.readline()
 6                 if not line:
 7                     return
 8                 yield line
 9     finally:
10         print('done')
11
12 # ..................................
```

ПРИМЕР

```
1 # ................................
2
3 f = lazy_lines('_11.py')
4 mapped = (v.strip() for v in f)
5
6 for i, v in enumerate(mapped):
7     print(next(mapped))
8     if i == 3:
9         mapped.close()
```

НЮАНСЫ

- `send` и `close` определены для генераторов
- Они не входят в контракт итераторов
- Их нет у генераторного выражения
- По сути - это правильно
- Если нужно управлять обернутым генератором - нужно хранить ссылку на него

НЮАНСЫ

- Итераторы на файлах ставят два противоречивых требования
- Файлы надо не оставлять открытыми
- На момент чтения из итератора на файле файл должен быть открыт
- Если это какой-то производный итератор - тоже
- Об этом надо думать и решать индивидуально

ДЕКОРАТОРЫ

- Общая идея: модификация функций или классов
- Сам декоратор - это функция, преобразующая другую функцию или класс
- Иногда декоратором можем быть callable-объект
- Это помогает хранить состояние

БАЗОВЫЙ USE CASE

- Бывают функции, которые вызываются не очень часто
- И факт их вызова является важным событием
- И мы хотим его как-то фиксировать
- Например, логировать
- Возможно, с переданными параметрами

ПРИМЕР

```python
def track(f):
    def wrapper(*args, **kwargs):
        print("before call to", f.__name__)
        result = f(*args, **kwargs)
        print("after call to", f.__name__)
        return result
    return wrapper

@track
def f1(a):
    return a + 1

# ..........................
```

ПРИМЕР

```
 1 # ........................
 2
 3 @track
 4 def f2(a, b=0):
 5     return a + b
 6
 7 print(f1(5))
 8 print(f2(7, 10))
 9 print(f2(7, b=5))
10 print(f2(11))
```

ПРОБЛЕМА

```
 1 def track(f):
 2     def wrapper(*args, **kwargs):
 3         print("before call to", f.__name__)
 4         result = f(*args, **kwargs)
 5         print("after call to", f.__name__)
 6         return result
 7     return wrapper
 8
 9 @track
10 def f1(a):
11     return a + 1
12
13 # ..............................
```

ПРОБЛЕМА

```
1 # ..............................
2 def f2(a, b=0):
3     return a + b
4
5 @track
6 def f3(a, b=0):
7     return a + b
8
9 # ..............................
```

ПРОБЛЕМА

```
1 # ..............................
2
3 print(f1)
4 print(f2)
5 print(f3)
6 print()
7 print(f1.__name__)
8 print(f2.__name__)
9 print(f3.__name__)
```

ГРУБОЕ РЕШЕНИЕ

```
 1 def track(f):
 2     def wrapper(*args, **kwargs):
 3         print("before call to", f.__name__)
 4         result = f(*args, **kwargs)
 5         print("after call to", f.__name__)
 6         return result
 7     wrapper.__name__ = f.__name__
 8     return wrapper
 9
10 # ..............................
```

ГРУБОЕ РЕШЕНИЕ

```python
# ..............................

@track
def f1(a):
    return a + 1

def f2(a, b=0):
    return a + b

@track
def f3(a, b=0):
    return a + b

# ..............................
```

ГРУБОЕ РЕШЕНИЕ

```
1 # ..............................
2
3 print(f1)
4 print(f2)
5 print(f3)
6 print()
7 print(f1.__name__)
8 print(f2.__name__)
9 print(f3.__name__)
```

КАК ЛУЧШЕ

- Надо тоньше
- Еще есть несколько полей
- И с новыми версиями могут появиться новые нюансы
- Логика корректировки свойств инкапсулирована в библиотечный декоратор

ПРАВИЛЬНОЕ РЕШЕНИЕ

```python
import functools

def track(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):
        print("before call to", f.__name__)
        result = f(*args, **kwargs)
        print("after call to", f.__name__)
        return result
    return wrapper

# ..........................
```

ПРАВИЛЬНОЕ РЕШЕНИЕ

```
# ..........................

@track
def f1(a):
    return a + 1

def f2(a, b=0):
    return a + b

@track
def f3(a, b=0):
    return a + b

# ..........................
```

ПРАВИЛЬНОЕ РЕШЕНИЕ

```
# ..........................

print(f1)
print(f2)
print(f3)
print()
print(f1.__name__)
print(f2.__name__)
print(f3.__name__)
```

ПАРАМЕТРИЗОВАННЫЙ ДЕКОРАТОР

- `functools.wraps` - пример параметризованного декоратора
- Многие декораторы можно осмысленно параметризовать
- На первый взгляд - не должно быть сложностей
- Ведь декоратор - функция

ПРОБЛЕМА

- Нам нужно передавать декорируемую функцию
- И параметры декоратора
- Декораторы могут быть разными
- И схемы передачи параметров - тоже

ПРОБЛЕМА

- Где-то захочется только чисто именованные
- Где-то - только чисто позиционные
- Где-то произвольный набор позиционных и/или именованных
- Непонятно - как передавать функцию, чтобы работало во всех случаях
- И отличать декорируемую функцию от параметров декорирования

РЕШЕНИЕ

- Решение - в еще одном уровне косвенности
- Декоратор со скобками ожидает функции, которая принимает параметры декоратора
- И возвращает функцию, которая принимает декорируемую функцию
- И возвращает обновленную функцию

ПРАВИЛЬНОЕ РЕШЕНИЕ

```python
import functools

def track(with_params=False, with_result=False):
    def decorator(f):
        @functools.wraps(f)
        def wrapper(*args, **kwargs):
            print("before call to", f.__name__)
            if with_params:
                print("positional params", args)
                print("keywords params", kwargs)
            result = f(*args, **kwargs)
            print("after call to", f.__name__)
# ..............
```

ПРАВИЛЬНОЕ РЕШЕНИЕ

```
1 # ..............
2
3             if with_result:
4                 print("result", result)
5             return result
6         return wrapper
7     return decorator
8
9 # ..............
```

ПРАВИЛЬНОЕ РЕШЕНИЕ

```
1  # ..............
2
3  @track(with_params=True)
4  def f1(a):
5      return a + 1
6
7  @track(with_result=True)
8  def f3(a, b=0):
9      return a + b
10
11
12 f1(1)
13 f1(a=2)
14 f3(2, 3)
```

ДЕКОРАТОР С СОСТОЯНИЕМ

- Пример: хотим замерять время работы функции
- Пишем декоратор, обрамляем вызов
- Время померили, но нужна статистика на многих вызовах
- Как и где хранить ?

ДЕКОРАТОР С СОСТОЯНИЕМ

- Вариант 1: используем функцию как объект
- Заводим свое свойство
- Имя обрамляем в __
- Используем какой-то фрагмент с претензией на уникальность
- И базовые соображения про безопасность и т.п.

ПРАВИЛЬНОЕ РЕШЕНИЕ

```
 1 def count_calls(f):
 2     @functools.wraps(f)
 3     def wrapper(*args, **kwargs):
 4         wrapper.\
 5          __io_github_sreznick_count_calls_n_calls__ += 1
 6         result = f(*args, **kwargs)
 7         wrapper.
 8        __io_github_sreznick_count_calls_n_returns__ += 1
 9         return result
10 # ................
```

ПРАВИЛЬНОЕ РЕШЕНИЕ

```
# ..............
    wrapper.\
      __io_github_sreznick_count_calls_n_calls__ = 0
    wrapper.
      __io_github_sreznick_count_calls_n_returns__ = 0
    return wrapper


@count_calls
def recip(v):
    return 1 / v

# ..............
```

ПРАВИЛЬНОЕ РЕШЕНИЕ

```
# ..............

try:
    print(recip(55))
    print(recip(5))
    print(recip(0))
except:
    pass

print(recip.__io_github_sreznick_count_calls_n_calls__,
      recip.__io_github_sreznick_count_calls_n_returns__)
```

ДЕКОРАТОР С СОСТОЯНИЕМ

- Вариант 1: через механизм callable object
- (Двойное применение)
- Выглядит - как будто специальный языковой механизм
- Реально - уместное применение универсального механизма

РАЗБЕРЕМ ДЕТАЛЬНЕЕ

- Интерпретатор видит нечто с @ перед определением функции
- Он воспринимает это как декоратор
- Который надо вызвать как функцию
- Ключевое слово - "как"

РАЗБЕРЕМ ДЕТАЛЬНЕЕ

- Это может быть callable object
- А любой класс - callable object
- Который создает объект, передает свои параметры в метод `__init__`
- И возвращает созданный объект

РАЗБЕРЕМ ДЕТАЛЬНЕЕ

- Можно указать класс в качестве декоратора
- Тогда будет создан объект, вызван конструктор
- Конструктору переданы параметры в зависимости от типа декоратора
- Если конструктор не сломается, то полученный объект будет привязан к имени функции

ПЕРВЫЙ ШАГ

```
 1 class C:
 2     def __init__(self, *args, **kwargs):
 3         print('ctor', self, args, kwargs)
 4
 5 @C
 6 def f1(v):
 7     ...
 8
 9 @C(1, 'hello', name='vasya')
10 def f2(v):
11     ...
12
13 print(f1)
14 print(f2)
```

РАЗБЕРЕМ

- Применение без скобок завершается успешно
- Потому что "функцию" мы пока что не вызвали
- А как вызовем - сломаемся
- Применение со скобками ломается
- Потому что интерпретатор пытается вызвать то, что он считает функцией - преобразователя декорируемой

БЕЗ ПАРАМЕТРОВ

- Чтобы полученная "функция" вызвалась, надо сделать объект callable
- То есть реализовать метод `__call__`
- Чтобы он вызвал декорируемую функцию, передал ей свои параметры и вернул ее результат
- И сделал нужные сопутствующие действия

РАЗВИВАЕМ

```
 1 class C:
 2     def __init__(self, func):
 3         self._func = func
 4         self._n_calls = 0
 5         self._n_returns = 0
 6     def __call__(self, *args, **kwargs):
 7         self._n_calls += 1
 8         self._func(*args, **kwargs)
 9         self._n_returns += 1
10
11 # ..............................
```

РАЗВИВАЕМ

```
 1 # ..........................
 2
 3 @C
 4 def f1(v):
 5     return v * v
 6
 7 f1(5)
 8 f1(10)
 9 print(f1._n_calls)
10 print(f1._n_returns)
11
12 print(f1)
13 print(f1.__name__) # рецидив старой проблемы
14                    # - в более тяжелой форме
```

УТОЧНЯЕМ

```
 1 import functools
 2
 3 class C:
 4     def __init__(self, func):
 5         self._func = func
 6         self._n_calls = 0
 7         self._n_returns = 0
 8         functools.update_wrapper(self, func)
 9     def __call__(self, *args, **kwargs):
10         self._n_calls += 1
11         self._func(*args, **kwargs)
12         self._n_returns += 1
13 # ........................
```

УТОЧНЯЕМ

```
 1 # ........................
 2
 3 @C
 4 def f1(v):
 5     return v * v
 6
 7 f1(5)
 8 f1(10)
 9 print(f1._n_calls)
10 print(f1._n_returns)
11
12 print(f1)
13 print(f1.__name__) # рецидив старой проблемы
14                    # - в более тяжелой форме
```

С ПАРАМЕТРАМИ

- Вариант 1: реализовать `__call__`, который примет функцию
- И вернет другой `callable`
- Возможно, другого класса
- Громоздко и не факт, что нужно

С ПАРАМЕТРАМИ

- Вариант 2: параметры декоратора принимать в функции
- В функции создавать шаблон класса-декоратора
- И заполнять его свойства класса значениями параметров декорации
- И возвращать это класс (по совместительству callable объект)

РАЗВИВАЕМ

```
1 def suppress_exc(exc_classes=(Exception,)):
2
3     class Suppressor:
4         def __init__(self, func):
5             self._func = func
6             self._exc_classes = exc_classes
7 # ........................
```

РАЗВИВАЕМ

```
# ........................
        def __call__(self, *args, **kwargs):
            try:
                self._func(*args, **kwargs)
            except BaseException as exc:
                if any(isinstance(exc, ec)
                       for ec in exc_classes):
                    print("thrown", exc)
                else:
                    raise

    return Suppressor
# ........................
```

РАЗВИВАЕМ

```
1 # ........................
2 @suppress_exc(exc_classes=(ArithmeticError,))
3 def recip(v):
4     return 1 / v
5
6 @suppress_exc(exc_classes=(LookupError,))
7 def recip2(v):
8     return 1 / v
9 # ........................
```

РАЗВИВАЕМ

```
1 # ........................
2
3 print(10)
4 recip(10)
5 print(0)
6 recip(0)
7
8 print(0)
9 recip2(0)
```

НЕСКОЛЬКО ДЕКОРАТОРОВ

- Декораторов может быть несколько
- Применяются "снизу вверх"
- Цепочка преобразований
- На это надо рассчитывать

ПРИМЕР

```
 1 def deco1(func):
 2     print('run deco1')
 3     def wrapper(*args, **kwargs):
 4         print('deco1, before')
 5         result = func(*args, **kwargs)
 6         print('deco1, after')
 7         return result
 8 
 9     return wrapper
10 # ..........................
```

ПРИМЕР

```
# ..........................
def deco2(func):
    print('run deco2')
    def wrapper(*args, **kwargs):
        print('deco2, before')
        result = func(*args, **kwargs)
        print('deco2, after')
        return result
    return wrapper
# ..........................
```

ПРИМЕР

```
1 # ..........................
2 @deco1
3 @deco2
4 def f(v):
5     return v * v
6
7 f(5)
```

РАЗВИВАЕМ

```
1 class C:
2     def __init__(self, func):
3         self._func = func
4         self._n_calls = 0
5         self._n_returns = 0
6     def __call__(self, *args, **kwargs):
7         self._n_calls += 1
8         self._func(*args, **kwargs)
9         self._n_returns += 1
```

РАЗВИВАЕМ

```
 1 # ....................
 2 def deco(f):
 3     def wrapper(*args, **kwargs):
 4         return f(*args, **kwargs)
 5     return wrapper
 6
 7 @deco
 8 @C
 9 def f1(v):
10    return v * v
11 # ....................
```

РАЗВИВАЕМ

```
1 # ....................
2
3 f1(5)
4 f1(10)
5 print(f1._n_calls)
6 print(f1._n_returns)
```

ДЕКОРАТОРЫ И КЛАССЫ

- Можно декорировать методы - как обычные функции
- Понятия "объявление переменных класса" нет - тут декорировать нечего
- Можно декорировать класс
- Но тут появляется двусмысленность

ДЕКОРАТОРЫ И КЛАССЫ

- Формально можно применить декоратор, рассчитанный на функцию, к классу
- И он класс как callable объект заменит на функцию
- Мы задекорируем конструирование объекта
- И в простых случаях такие объекты даже будут работать
- Можно будет создавать объекты и через объекты работать с классом

ДЕКОРАТОРЫ И КЛАССЫ

- Но нельзя будет обращаться к классу напрямую
- Если хотим декорировать конструктор - подумать над вариантом декорирования `__init__`

ДЕКОРАТОРЫ И КЛАССЫ

- Но может быть так, что хотим декорировать класс
- Вариант 1 - вернуть то, что есть, изменив нужные свойства полученного объекта
- Вариант 2 - создать новый класс-объект на основе полученного